# Specifying and Detecting Relevant Changes in Programs

Yijun Yu*, Thein Than Tun* and Bashar Nuseibeh* †
* The Open University
Milton Keynes, UK
Email: {t.t.tun, y.yu, b.nuseibeh}@open.ac.uk
† Lero, Irish Software Engineering Research Centre
Limerick, Ireland
Email: bashar.nuseibeh@lero.ie

***Abstract*—Software developers are primarily interested in the changes in code that are relevant to their current tasks, therefore not all changes to evolving software are of the same importance. However, most existing `diff` tools notify developers more changes than they wish to see. In this paper, we propose an automated technique to specify and detect only those relevant changes in programs in order to avoid looking at other trivial changes. 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 the non-trivial or meaningful differences automatically. Our tool has been evaluated on a benchmark of programs to demonstrate its improved precision comparing 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 meaningful to different purposes. For example, changing the indentations of statements does not necessarily alter the meanings or semantics expressed by a program. Nonetheless it could lead to false alarms to any revision control system as text-based difference comparison algorithms are typically used (e.g., the **diff** utility in Unix [1]). Although an indentation is not meaningful to the executation semantics of C/Java programs, it can be very important to other programming languages such as Python. Still can it be meaningful to C/Java developers who care about pretty-prints for the sake of code reviews. Another example is the API evolution: thanks to the widely adopted information hiding and modular design principle [2], users of object-oriented programming libraries are encouraged to neglect any changes behind the API. Therefore detecting changes to the API of software components becomes meaningful [3]. On the other hand, providers of the API need to pay attention to most changes inside the API implementation.

Given that a change considered as meaningful for one purpose may be trivial to another, how can one specify the types of changes to be detected for this given purpose? Furthermore, how can such a specification be used for an 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 programming or modeling language such as UML class diagrams [6], dependency graphs [7], or Verilog [8]. However, few aims 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 "normalisation" of grammatically correct source programs. The normalised results are always valid in a language for the specific purpose, possibly refined from the source language. We show that such normalisations can be specified simply as annotations on the original "production rules" [9], while specific needs of further normalisation can be accommodated by user-defined function. Each type of elementary normalisation corresponds to an elementary kind of terms in the production rules of the source programming 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, which separates meaningful changes from the trivial 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/CSharp/Python), as well as several graphical 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. Also we applied `mct` to detect these meaningful changes in the CVS repositories of two medium-sized open-source projects.

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 running example, Section III explains the approach we adopt to bootstrap the normalisation

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

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.

## 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`

```
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`

```
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 + "!");
```

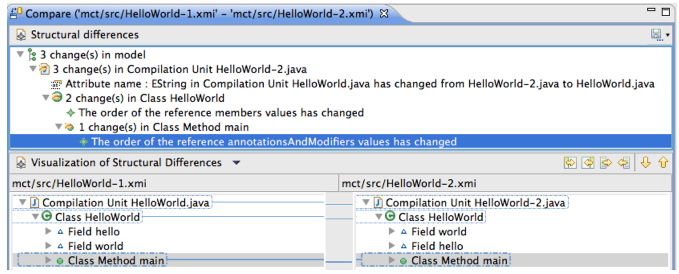


Fig. 1. The differences found by EMFCompare on the two EMF models

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 `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.

- *Step 1. Specification*: the programmer defines a number of annotations to the given grammar of the programs;
- *Step 2. Detection*: the tool generates two sets of transformations (normalisation, clone-removal) from the specification in Step 1 and applies these transformations to the two source programs to report the meaningful changes.

Figure 2 illustrates the workflow of a typical use case where the thin arrow indicates the manual specification step for the developer to annotate the given grammar; and the thick arrows indicate the automated transformation steps, for the mct system to generate the grammar refinement and transformation rules to detect meaningful changes from the programs in the CVS repository.

## III. Specifying Relevant 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

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

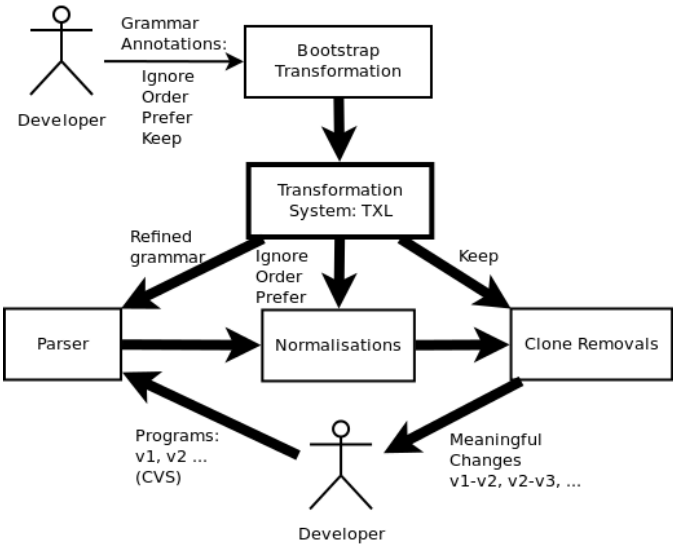


Fig. 2. Specifying and detecting meaningful changes, an overview

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 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 $N \leftarrow (T_1? \dots T_n*)[| \dots]$ 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:* **Composibility of syntax validty.** 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:* **Compobility of terminable normalisation** 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.

*Definition 4:* **Elementary structural clone removals.** Any entity in the grammar can be marked as possible clones such that a clone removal transformation can remove any duplicated occurrance.

In principle any parameterised AST-based clone detector could be used for this purpose. To illustrate in this paper we show the simpliest exact clones.

Table I summarises the four elementary annotations to the meta-grammar for the normalisation/clone removals. Although they are by no means complete, we found that they are sufficient for many practical meaningful change detection tasks.

### A. 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].

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
```

[3]http://txl.ca

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

```
include "java.grm"
redefine program
    [package_declaration] [opt package_declaration]
end define
redefine package_declaration
    [opt package_header kept]
    [repeat import_declaration ignored]
    [repeat type_declaration kept ordered]
end define
define class_or_interface_body
    '{ [NL][IN]
       [repeat class_body_declaration kept ordered ignored
           when Private] [EX]
    '} [opt ';] [NL][NL]
end define
redefine method_declaration
    [NL] [repeat modifier ordered by Descending] [opt
         generic_parameter] [type_specifier] [
         method_declarator] [opt throws] [method_body
         preferred]
end define
redefine method_body
       [opt annotation_default] '; [NL][NL]
    |  [block]          [NL][NL]
end define
...
function Private A  [class_body_declaration]
    match  [class_or_interface_body] B  [
         class_or_interface_body]
    construct M  [modifier *] _  [^ A]
    construct PublicModifiers [modifier*] 'public '
         protected
    where not M [contains each PublicModifiers]
end function
rule Descending B [modifier]
 match [modifier] A [modifier]
 construct SA [stringlit] _ [quote A]
 construct SB [stringlit] _ [quote B]
 where SA [< SB]
end rule
```

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 which will be ignored by the parser, but will be inserted by the unparser to pretty print the transformed code. NL, IN or EX are respectively for new line, increasing and decreasing indention levels. Therefore the output has two lines per import declaration.

Lines 377-379 define the method declaration as zero to many modifiers (as listed in Lines 254-267), plus an optional generic parameter, a type specifier, a method declarator, optional throws exception declarations and the 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.

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*] |

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.

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. First of all, the top level rule is modified from a singleton to one or optionally two instances of the programs (Lines 2-4). The reason for this is to allow the clone detection to work on the concatenated programs being compared to remove those inter-program clones such that the kept elements are all about different elements. The annotations "kept" are appended to the terms such as "package_header" (Line 6), "type_declaration" (Line 8), "class_body_declaration" (Line 12). These instruct a clone detector to compare these three types of entities for possible clones. Although the technique is similar, there is also a fundamental difference between general-purpose clone detection and cross-program clone-removal here. The purpose is not to show the clones, instead the opposite: those non-clones are the differences to be detected. When a single program is provided as the source, of course, the transformations will then degenerate into just selecting the parts of the program in which changes are embedded.

However, the ordering of elements or appearance of ignore-able details can get in the way of meaingful change detections. Therefore the 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 29-34) can be 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 the members ascendingly is the same as descendingly as long as the same criterion is applied to all source programs.

The Ignore annotation (ignored), on the other hand, will replace the optional or repeated terms by empty. The terms `import_declaration` at Line 7, `class_body_declaration` at Line 12, are examples. In particular, the Ignore annotation to `class_body_declaration` is conditional, it uses a user-defined function from Lines 23-28 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. Our default implementation will transform any occurrence of other alternatives into the first one listed by the production rule. For example, when the `method_body` at Line 16 is annotated by preferred, the production rule from Lines 18-21 are used to transform any block into a semicolon because it is the preferred alternative. Note that for this to work, users need to modify the production rule of `method_body` in the original grammar Lines 407-410 to swap the two alternatives, which is perfectly doable without modifying the semantics of the Java5 grammar.

### 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
```

```
7   ... [opt kept] [opt orderedBy] [opt ignoredWhen] [opt
        preferredWith]
8 end define
9 define kept 'kept end define
10 define orderedBy 'ordered [opt byFunction] end define
11 define byFunction 'by [id] end define
12 define ignoredWhen 'ignored [opt whenFunction] end define
13 define whenFunction 'when [id] end define
14 define preferred 'preferred 'with [literal+] end define
```

The second part of the implementation is a specification of the normalisation transformations. Limited by space, here we only show a simplified Listing 8 for the Order extension. It generates rules for eliminating `ordered` annotations for the generated grammar to be recognizable by TXL at runtime, and for producing rules for ordering the terms parsed as `[repeat X]`.

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
```

In brief, the above extension of Java5 grammar has 8 annotations added by the user, plus 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 `mct` 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.

When the same transformation is applied to the concatenated inputs of both programs, the clone removal step runs to correctly remove all the clones according to the terms annotated by "kept". As a result, their is no longer anything left after the clone removal, 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. `cat HelloWorld.java HelloWorld-2.java > HelloWorld-2.pair`
`mct HelloWorld-2.pair java.Txl`

## IV. Performance evaluation

To evaluate the `mct` tool, we experimented on three evolving programs, namely `gmf`, `jhotdraw` in Java and `Uart16650` in Verilog. These benchmark 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 relevant changes 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 `mct` tool is implemented as `mct.Txl` as 19 additional rules that transform the 7 extended terms. The Java API normalisation tool is implemented by redefining 21 terms using 7 Keep, 17 Order, 1 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 4 Keep, 4 Order and 1 Ignore, which generates 11 additional transformation rules. On a laptop running Mac OSX 10.6.7, with 2.66GHz Intel i7 CPU, 4GB DDR 1067MHz memory, the automated generation of these normalisation rules takes no more than 0.02 seconds. The programming language grammars in TXL (L) and the generated normalisation transformation rules in TXL (M) will be used in the remaining experiments.

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 | +544 | +7 | +19 | |
| java5.Txl (L) | Java 5 grammar | 976 | 168 | 1 | |
| java.norm | annotation | +123 | +21 | +1 | |
| java.Txl (M) | result | +721 | +6 | +47 | 0.02s |
| v.Txl (L) | Verilog grammar | 233 | 37 | 1 | |
| v.norm | annotation | +45 | +9 | 0 | |
| verilog.Txl (M) | result | +191 | +2 | +11 | 0.01s |

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

| Program<br>$\Delta$ = diff | File<br>Commit | LOC(J)<br>$\Delta$ File(J)<br>$\Delta$ LOC(J) | LOC(L)<br>$\Delta$ File(L)<br>$\Delta$ LOC(L) | LOC(M)<br>$\Delta$ File(M)<br>$\Delta$ LOC(M) |
|---|---|---|---|---|
| uart16650 | 12 | 51,601 | 28,805 | 28,582 |
| (diff) | 128 | 62 | 53 | 52 |
| | | 1,864 | 879 | 1,001 |
| (ldiff) | 128 | 62 | 53 | 52 |
| | | 1, 552 | 694 | 816 |
| jhotdraw | 1,297 | 317,487 | 210,421 | 39,164 |
| (diff) | 1,590 | 1,264 | 1,107 | 612 |
| | | 29,087 | 21,479 | 3,051 |
| gmf | 8,809 | 8,981,629 | 4,614,668 | 512,108 |
| (diff) | 17,499 | 14,211 | 14, 323 | 9,188 |
| | | 924,148 | 566,597 | 25,093 |

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 normalised ones (L) rather than the original code (J).

Table V lists the size metrics of these programs. The metric 'File(J)' is the number of files stored in the repository; 'Commit' is the number of commits or revisions from the CVS repositories; 'LOC(X)' is the number of accumulated lines of code of all the revisions; '$\Delta$ LOC(X)' is the number of accumulated lines of changes detected by the `diff` utility; and '$\Delta$ FIle(X)' is the number of their RCS revisions that has been found different from their previous revision according to `diff`.

All the size metrics show that `UArt16650` $<<$ `JHotDraw` $<<$ `GMF`, roughly by a magnitude of 10. 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 Java API and Verilog Module Interface extraction (M) has a much larger size reduction effect. The absolute size of the normalised code LOC(M) is almost 10 times smaller than LOC(L), and $\Delta$ LOC(M) is also much smaller than the counterparts. As a result, in all the three examples, fewer file-

TABLE IV
TIME PERFORMANCE OF DIFF TOOLS

| Program | Time(J) | Time(L) | Time(M) | Time(ldiff) |
|---|---|---|---|---|
| `uart16650` | 0.2 | 1.78 | 182.6 | 18.5 |
| `jhotdraw` | 21.7 | 43.6 | 123.4 | 633.5 |
| `gmf` | 105.5 | 1038.7 | 23954.3 | |

TABLE V
PERFORMANCE OF CLONE REMOVAL VERSUS `diff`

| **Program** | $\|\Delta\|$(M) using `diff` | $\|\Delta\|$ (M) using `mct` |
|---|---|---|
| `uart16650` | 1,001 | |
| `jhotdraw` | 3,051 | |
| `gmf` | 25,093 | |

level changes are found by `diff`.

The time it took for the experiment is shown in Table IV, in the units of second. "Time(J)" is the time it took to compute `diff` between pairs of consequent revisions of the original programs; "Time(L)" is the time it took to first parse the programs using TXL (removing whitespaces/comments), then perform `diff` on the results; "Time(M)" is the time it took to parse the program, normalise them and perform `diff` on the results. Among these three, `Time(M)` is the longest: on average 1.36 seconds per revision. However, it is much faster than `ldiff`, which takes on average 39 seconds per revision.

## V. RELATED WORK

Xing and Stroulia [6] propose an approach to recover UML models from Java code, and compares the structural changes in the class diagrams corresponding to the differences between the two designs. Their approach is specific to UML. It uses similarity metrics for names and structures in order to determine various types of changes made to them. Since UML models such as class diagrams, statecharts, etc., can be specified in domain specific languages such as textUML, it makes it possible to apply our tool on these diagrams to detect such changes similarly.

Apiwattanapong et al [13] 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 Java-specific features, such as the exception hierarchies. The tool is therefore not made for specifying changes on other programming languages.

Brunet et at [14] defines some challenges in model mangements including the operations *merge, match, diff, split* and *slice*, as well as the properties that need to be preserved by these operations. These operations and properties are independent of models and modelling languages. In comparison, our *normalisation* steps are more specific to *slice* operations because that it is a transformation performed from model to model based on slicing criteria. And the *clone removal* step is more specific to *diff* operations that transform two models to a difference model. Our emphasis is on ensuring the transformed programs (or models) valid against the original language grammar, so that it is easier for programmers to compare the differences to the original programs.

Beyer et al [15] 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.

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. On the other hand, our tool present all the choices for the user to decide whether a change in constant should be ignored or not as a user-defined function.

Kawaguchi et al [16] 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. On the other hand, such dynamic structures are typically unspecified in the source programs. If it is possible to refine the program structures by annotating with dynamic input/output models, then it is possible to specify such changes as user-defined functions.

Duley et al [8] present VDiff for differencing non-sequential, "position independent" Verilog programs. Their algorithm first extracts the abstract syntax trees of the programs, and match 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 annotationed terms.

Loh and Kim [17] present the LSDiff tool which automatically identifies structural changes as logic rules.

## VI. CONCLUSIONS AND FUTURE WORK

Scalability

Meta-changes: Changes to the Transformations

We believe there is no need for infinite meta-levels. One example is 'KM3' or 'MOF', although in principle there is always a possibility to ask for meta-level changes, typically the language is less likely to change than the program.

How to make use of 'iChange' for runtime adaptation?

The deployed system also need to adapt dynamically to the changes in its environment at runtime. [18] [**?**] [10] [19] [6] [20] [21] [**?**] [14] [**?**] [4]

## 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] 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

[14] 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

[15] 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.

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

[17] A. Loh and M. Kim, "Lsdiff: a program differencing tool to identify systematic structural differences," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 263–266. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810348

[18] 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

[19] 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

[20] 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.

[21] 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

[22] 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.