Specifying and detecting meaningful changes in programs

Conference Item

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 differencing tools, such as `diff`, notify developers of more changes than they wish to see. In this paper, we propose a technique to specify and automatically detect only those changes in programs deemed meaningful, or relevant, to a particular development task. Using four elementary annotations on the grammar of any programming language, namely *Ignore*, *Order*, *Prefer* and *Scope*, developers can specify, with limited effort, the type of change they wish to detect. Our algorithms use these annotations to transform the input programs into a normalised form, and to remove clones across different normalised programs in order to detect non-trivial and relevant differences. We evaluate our tool on a benchmark of programs to demonstrate its improved precision compared to other differencing approaches.**

I. Introduction

Nothing endures but change.

Heraclitus (c.535 BC—475 BC)

This philosophy is true in many software development projects. However, not all changes are equally relevant to developers engaged in different development tasks. For example, changing the indentation of statements in a Java program does not necessarily alter its execution semantics. Nonetheless, most revision control systems, typically using text-based differencing tools such as the `diff` utility in Unix [1], do not discard changes to the indentation of such programs. Although indentation is not meaningful to the execution semantics of Java programs, it can be very important for the execution of Python programs. Furthermore, for those who are concerned with pretty-prints of Java programs, indentation is important. Another example is API evolution [2]: users of object-oriented programming libraries are encouraged to use the API instead of its implementation, to adhere to the information hiding principle [3]. As a result, some developers may wish to identify only those changes made to the API, whilst others may want to determine 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 effective either at reporting all changes through general purpose differencing algorithms on programs represented as line-separated text and structured models [1], [4], [5] or 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 specifications [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 source programs. Two programs that are not different due to orderings, abstractions and preferences are normalised into the same program, using three basic normalising transformations *Order*, *Ignore* and *Prefer* respectively. 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 type of transformation to a 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 of the `TXL` transformation system [10], that is, the grammar of all `TXL` grammars. Therefore, it is applicable to any source language specifiable by `TXL`, which currently supports several general-purpose programming languages such as C, Java, C#, and Python as well as several modelling languages such as XML, XMI, GXL. An evaluation of our *Meaningful Change Tool* (hereafter `mct`) on the `CVS` repositories of two medium-sized software projects and one small hardware project shows that (i) few annotations are needed to specify typical meaningful changes, such as changes made to the API, and (ii) the tool is scalable.

The remainder of the paper is organised as follows: Section II introduces a running example illustrating the need for detecting meaningful change. Section III explains our approach to generate from meta-grammar specifications those normalisation transformations needed to detect meaningful changes. Section IV presents the results of a number of experiments in using the tool, and compares the performance with existing differencing tools. Section V compares existing approaches with ours, highlighting some of our limitations. Section VI concludes the paper.

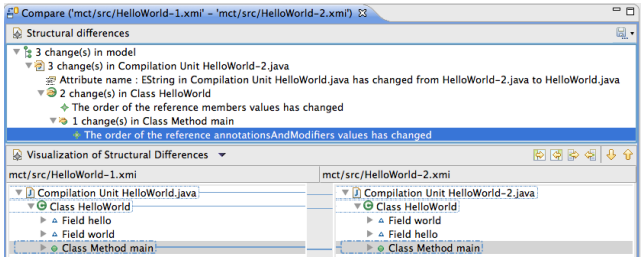

Fig. 1. The differences found by `EMFCompare` on the two EMF models

II. A Motivating Example

The essence of meaningful change can be illustrated using a simple Java program in Listing 1. After some changes, the execution semantics in Listing 2 remain the same. The Unix `diff` utility [1] reports these changes, ignoring white spaces, as one deletion and one modification shown in Listing 3. A more advanced algorithm `ldiff` [4] reports these changes, again ignoring white spaces, as two insertions, one deletion and two modifications shown in Listing 4.

Listing 1. `cat -n HelloWorld.java`

```
1 public class HelloWorld
2 {
3  static private String hello = "Hello";
4  private static String world = "world";
5  static public void main(String args[]) {
6   System.out.println(hello + ", " + world + "!");
7  }
8 }
```

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

```
1 public class HelloWorld
2 {
3  private static String world = "world";
4
5  static private String hello = "Hello";
6  public  static void main(String args[]) {
7   System.out.println  (hello + ", "
8      + world + "!");
9  }
10 }
```

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

```
1 3d2
2 <  static private String hello = "Hello";
3 5,6c4,8
4 <  static public void main(String args[]) {
5 <   System.out.println(hello + ", " + world + "!");
6 ---
7 >
8 >  static private String hello = "Hello";
9 >  public  static void main(String args[]) {
10 >   System.out.println  (hello + ", "
11 >      + world + "!");
```

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

```
1 3,3d2
2 <  static private String hello = "Hello";
3 4a4,5
4 >
5 >  static private String hello = "Hello";
6 5,5c6,6
7 <  static public void main(String args[]) {
8 ---
9 >  public  static void main(String args[]) {
10 6,6c7,7
11 <   System.out.println(hello + ", " + world + "!");
12 ---
13 >   System.out.println  (hello + ", "
14 6a8,8
15 >      + world + "!");
```

The `EMFCompare` tool (http://eclipse.org/emf/compare) is applied to the corresponding EMF models of the two Java examples parsed by the `JaMoPP` Java parser (http://www.jamopp.org), reporting three changes: 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".

If only the differences in the execution semantics are meaningful to a programmer, none of the changes identified in this example are relevant: just as adding a newline or some white spaces would not change the syntax of the program, swapping the keywords `public` and `static` in the declaration of the `main` method makes no semantic difference.

III. The MCT System

Our approach to finding meaningful change between two versions of a program has two steps shown in Figure 2:

- *Step 1. Specification*: A developer specifies annotations onto the grammar terms of programs, see dotted arrows in Figure 2;
- *Step 2. Detection*: The `mct` tool generates a refined parsing 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 any meaningful change to the developer. See solid arrows in Figure 2.

In a typical work flow the specification step is done manually by the developer, whilst the detection step is done automatically by the `mct` system to find meaningful changes of programs in revisions stored in the repository.

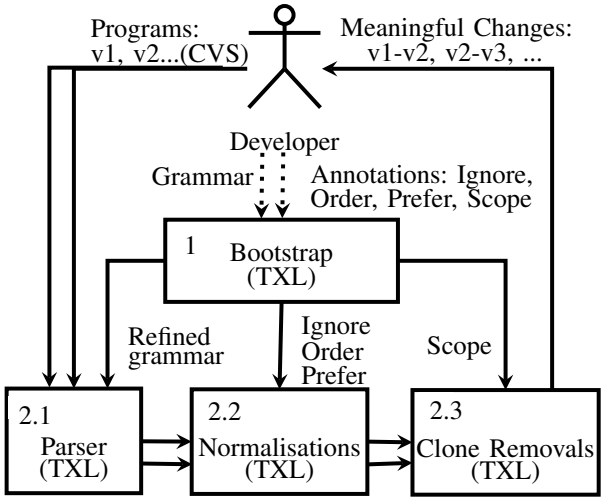

Fig. 2. Specifying and detecting meaningful changes using `mct`

TABLE I
BASIC ANNOTATIONS TO THE TERMS IN A TXL GRAMMAR

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

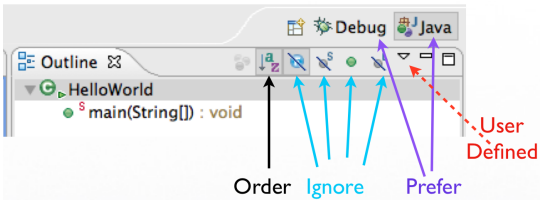

Fig. 3. The *outline* view of the Eclipse IDE helps programmers to select meaningful information interactively while reviewing a Java class. Typically, they care about the ordering (e.g., the "a-z" button), and may ignore certain details (e.g., the"hide fields", "hide non-public members", "hide static members" buttons), and express a preference (e.g., "Java" or " Debug" perspective). Other filtering options can be expressed by user-defined dialogues.

A. Specifying Meaningful Changes

We first define a few requirements for specifying *meaningful changes* through the concept of normalisation transformations.

Definition 1: **Equivalence classes by normalisation**. 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.

The exact meaning for 'meaningful equivalent' in the Definition 1 is intentionally left open to the user who defines the normalisation function appropriate to the tasks at hand.

This definition only 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 program. After the normalisation transformation is defined, the detection of the meaningful changes becomes a comparison of two normalised programs.

According to Definition 1, however, a normalisation transformation can be a recursive function that does not terminate when applied. For example, adding white spaces can be regarded as a normalisation transformation but it would lead to infinite results when the transformation is applied recursively. Therefore, a useful normalisation transformation needs to be *terminable* so that no further transformation is necessary when a fixed point is reached.

Definition 2: **Fixed points of terminable normalisation.** A fixed point of normalisation N is a program P such that $N(P) = N^0(P) = P$. A normalisation transformation *terminates* upon its fixed point. The normalisation is *terminable* if any input program P can converge to a fixed point: $N^{i+1}(P) = N^i(P)$, where $i \geq 0$ is a finite number. Every fixed point in $FP(N)$ is the *representative element* for a class of meaningfully equivalent programs according to the normalisation N.

By this definition, the *identity* transformation is trivially terminable. There are non-trivial terminable normalisations. For example, ordering the modifiers (i.e., "private", "static") of all class members is a terminable transformation, because the number of inverse order of the modifiers is finite, so is the number of class members.

The following property guarantees that a composition of any two terminable normalisation transformations is still terminable, for a subset of fixed points.

Property 1: **Composability of terminable normalisations** If two normalisations N_1 and N_2 are terminable according to Definitions 1 and 2, then the functional composition $(N_1 \oplus N_2)(P) = N_2(N_1(P)) = P$ is also a terminable normalisation, and $FP(N_1 \oplus N_2) \subseteq FP(N_1) \cap FP(N_2)$.

For example, ignoring the method bodies in all classes is a terminable normalisation composable with reordering the method modifiers. A program with method bodies ignored may not have the modifiers sorted, and vice versa. Only fewer programs with both the method bodies ignored *and* the modifiers ordered are the fixed points that terminate the composed normalisation transformation. In general, the more normalisation transformations are composed, the fewer fixed points can be distinguished.

In order to support most programming languages, at the level of the meta-grammar we aim to generate normalisation transformations by composing three elementary terminable normalisations derived from the production rules of grammars: *Ignore*, *Order* and *Prefer*. For every term of a production rule, one can check whether any one of these basic types of terminable normalisations is applicable to provide what programmers like to compare. For example, traditionally, Java programmers get help from an IDE such as Eclipse to obtain meaningful information in an "outline" view, as shown in Figure 3. The abstraction presented in the outline view can be further customised by clicking at the right buttons by the programmer. Our elementary normalisation operations already cover all the most commonly used meaningful selection buttons of the Eclipse IDE.

Definition 3: **Elementary normalisations: *Ignore*, *Order* and *Prefer*.** Let a production rule be $N \leftarrow (\ldots T_i[?|*] \ldots)| \ldots$ where N 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 pattern. Every optional (denoted by '?') term can be *ignored* if, with or without the value, it makes no difference to the developer; elements of a repeated (denoted by '*') term can be *ordered* if the ordering of the values is not important to the developer; and the whole element N matches a mandatory alternative can be *preferred* to another alternative (denoted by '|'), if the

difference between these two alternatives are not significant to the developer.

Table I gives an example of each of these elementary transformations: by ignoring the members when they are private; by sorting the member declarations in ascending order; and by preferring the semicolon over the detailed method bodies.

These elementary normalisations can be used to preserve the conformance to the syntax of the source programming language, if one does not apply *Ignore* to mandatory terms. Since different programming languages have different semantics, users may wish to preserve the semantics by avoiding certain normalisations. For example, one would not reorder the statements in normalisation to avoid breaking the execution semantics of Java. Whether these syntax or semantics-preserving transformations are desirable really depends on the developer's task.

Property 2: **Syntax conformance.** The normalised programs generated by the three elementary transformations in Table I are valid programs in the *source* programming language.

Of course, syntax conformance of the normalised programs is not an issue when the purpose of checking meaningful changes is not to obtain a compilable program. One may choose to define the target grammar to be incompatible with the source language. The elementary normalisations can also be further customised. The unconditional ignored rule for an optional term '?' can be associated with a conditional check: in the API extraction example, one would remove a declaration if and only if it does not have 'public' or 'protected' modifiers. Similarly, the ordered rule for repeating terms '*' can be customised to ascending or descending orders, and the ordering criteria can be associated with certain foreign keys.

B. Detecting Meaningful Changes

After each program revision is normalised, the next task is to detect the non-trivial changes. A simple method is to apply an existing `diff` algorithm on the normalised results. However, this may not detect the exact and subtle differences smaller than one line of code. The method we adopted is to apply clone detection [11] in order to take advantage of knowing the meaningful structures independent of the line boundaries. As long as the normalised entities are the same, a clone detector could locate them. On the other hand, if the two entities are similar but not exactly the same, a meaningful context of the difference may need to be shown. After removing the clones across a pair of programs, their differences become evident. Note that we do not check intra-program clones as our purpose is not to detect clones, but only to *reuse* existing clone detection techniques for the sake of differencing. In principle any parametrised AST-based clone detector could be used for this purpose. To illustrate this in this paper, we use the simplest exact clone detection.

Not all language constructs at all levels of granularity should be considered as clones either. For example, index variables of a for-loop are not apparently so meaningful to detect as a clone because it makes little sense to scope the comparison at this level of abstraction. To be able to specify the scope of language construct that needs to be considered as clones to remove, another kind of annotation is introduced to non-terminals.

Property 3: **Scoping the clone detection.** Any optional or repeat term in the production rule can be marked as a possible scope for clone detection such that removing the inter-program clones at this level does not change the conformance of the source programming language. Instances of any mandatory term in the grammar can be marked as possible clones if required, whilst the target programming language may violate the syntax rules of the source programming language.

The last row of Table I summarises the scoping annotation for clone removals.

C. Running Example: Specifying Normalisations for the Java API

To illustrate the features of our method, we use the example of the Java 5 grammar provided by the TXL website (http://txl.ca), containing 970 lines of code. TXL is a functional programming language in which a transformation is defined by either a non-recursive function or a recursive rule [10]. Instead of using pseudo code to illustrate the algorithms, in this paper, some of the exact declarative rules are used in order that the work can be easily reproduced.

Listing 5. `cat -n java.annotated.grm`

```
1 include "java.grm"
2 annotate package_declaration [opt package_header scoped]
3 annotate package_declaration [repeat import_declaration
      ignored]
4 annotate package_declaration [repeat type_declaration
      scoped ordered]
5 annotate class_or_interface_body [repeat
      class_body_declaration scoped ordered ignored when
      Private]
6 annotate method_declaration [repeat modifier ordered by
      Descending]
7 annotate method_declaration [method_body preferred with
      ';]
8 % ...
9 function Private A  [class_body_declaration]
10    match  [class_or_interface_body] B  [
          class_or_interface_body]
11    construct M  [modifier *] _  [^ A]
12    construct PublicModifiers [modifier*] 'public '
          protected
13    where not M [contains each PublicModifiers]
14 end function
15 rule Descending B [modifier]
16  match [modifier] A [modifier]
17  construct SA [stringlit] _ [quote A]
18  construct SB [stringlit] _ [quote B]
19  where SA [< SB]
20 end rule
```

Line 1 includes the original Java 5 grammar. The `annotate` rules in Table I are used or composed in some of the term extensions. Here we explain the rationale behind these extensions. The annotations "scoped" are appended to the terms such as "package_header" (Line 2), "type_declaration" (Line 4), "class_body_declaration" (Line 5). These instruct a clone detector to compare these three types of entities for possible clones. Although the technique is similar to 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 ignorable details can get in the way of meaningful change detections. Therefore several normalisation transformations are required to be applied before the change detection step. Since one does not care about 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` is unimportant to the Java semantics. However, the default behaviour of a 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 ordered at the end of the `[repeat modifier]` term.

Furthermore, if one would like to normalise the elements in descending order, a user-defined rule `Descending` (Lines 15-20) is added in Listing 5. This is just to illustrate how easy it is to customise 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 in ascending order is the same as ordering them in descending order.

The Ignore annotations (ignored), on the other hand, will replace the optional or repeated terms with `[empty]`. The terms `import_declaration` at Line 3 and `class_body_declaration` at Line 5, are examples. In particular, the Ignore annotation to `class_body_declaration` is conditional, it uses a user-defined function from Lines 9-14 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 the annotated term `[import_declaration ignored]` does. Since such terms are unconditionally ignored, it is unnecessary to compose them with the Scope annotation as other siblings 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 annotations (preferred) are appended to the terms that have more than one alternative expansion. 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 7 is annotated by `preferred ';`, this will lead to a transformation to turn any block into a semicolon.

D. Brief Implementation of the `mct`

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 meta-grammar (given in the file `txl.grm`). Listing 6 shows the extension to the existing `typeSpec` rule and the addition of four annotation rules `orderedBy`, `ignoredWhen`, `preferred` and `scoped`, corresponding respectively to the three elementary normalisations and the scoping rules.

Listing 6. `cat -n norm.grm`

```
1 include "txl.grm"
2 % The extension of the Txl grammar
3 keys
4   ... 'scoped 'ordered 'by 'ignored 'when 'preferred '
        with 'annotate
5 end keys
6 define typeSpec
7   ... [opt scoped] [opt orderedBy] [opt ignoredWhen] [opt
        preferredWith]
8 end define
9 define scoped 'scoped 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
15 redefine statement ... | [MCT_annotate] end define
16 define MCT_annotate 'annotate [id] '[ [typeSpec] '] [NL]
     end define
```

The second part of the implementation is a specification of the normalisation transformations.

For brevity, we only discuss the extension for the Order transformation, shown in Listing 7. The transformation generates the rules for eliminating `ordered` annotations so that it is recognisable by TXL at runtime, and for producing the rules for ordering the parsed terms. Lines 44-64 specify how to generate the transformation rules denoted by the variable `Rules` on the fly by checking every `defineStatement` in the TXL grammar such as those in Listings 5. 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 `defineStatement` 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 7. `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
```

E. 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 5, a concrete normalisation transformation is produced in the original syntax of TXL, as shown in Listing 8. Lines 2-9 are generated from the `[repeat modifier ordered by Descending]` annotations from the Line 6 of Listing 5, using the user-defined comparison function `Descending`, which was listed in Lines 15-20 in Listing 5.

Listing 8. `cat -n java.Txl`

```
1  include "java.grm"
2  rule normalise_list_method_declaration_modifier
3      replace  [repeat modifier]
4          N1  [modifier]  N2  [modifier]  Rest  [repeat
               modifier]
5      where
6          N1  [Descending  N2]
7      by
8          N2  N1  Rest
9  end rule
10 function main
11     replace  [program]
12         Prg  [ program ]
13     by
14         Prg  [ normalise_list_method_declaration_modifier
               ]
15 end function
```

F. 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 produces the normalised results that are identical, as shown in Listing 9. 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 in ascending order 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 *"scoped"* annotated terms, because one usually would not remove duplication of low-level term such as identifiers.

Listing 9. `mct HelloWorld.java HelloWorld-2.java`

```
1 public class HelloWorld {
2
3     public static void main (String args []);
4 }
5 public class HelloWorld {
6
7     public static void main (String args []);
8 }
```

As a result, there is no longer anything left, leaving the output empty as shown in Listing 10.

Listing 10. `mct -diff HelloWorld.java HelloWorld-2.java`

IV. Experimental Results

The `mct` tool has been applied to three benchmark evolving programs in the public domain: (i) GMF is a model-driven code generator for Eclipse Graph Editors, (ii) JHotDraw is a GUI framework for technical and structured Graphics, and (iii) OpenCores Uart16650 is a specification of FIFO queue for hardware, which was used for the study of Verilog Diff [8].

A. Specifying Meaningful Changes on Java and Verilog

The meaningful changes we want to detect are differences in the APIs of these programs in different programming languages. Table II lists the sizes 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 Scope, 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 define 6 new refined terms. The Verilog annotations include 1 Scope, 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.

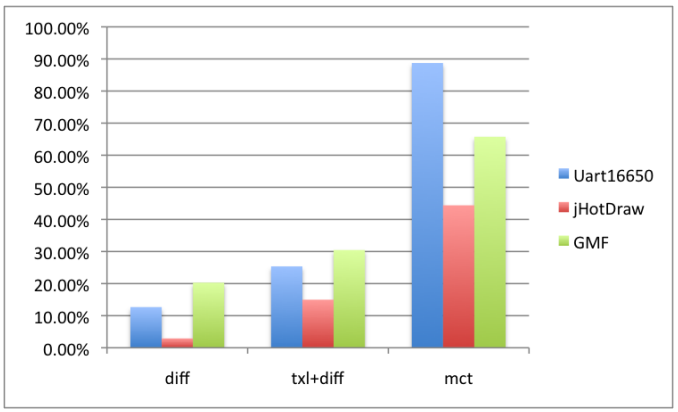

Fig. 4. Contrast the effectiveness of change detection at the file level as the percentage of reduced changed files

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

Grammar	Description	LOC	Terms	Rules
txl.grm	meta-grammar	408	58	1
mct.Txl	Implementation	1081(+673)	65(+7)	29(+28)
java5.Txl (L)	Java 5 grammar	976	168	1
java.norm	annotation	1108(+132)	189(+21)	2(+1)
java.Txl (M)	result	1837(+729)	194(+5)	45(+43)
v.Txl (L)	Verilog grammar	233	37	1
verilog2.norm	annotation	254(+21)	41(+4)	1(+0)
verilog2.Txl (M)	result	469(+215)	46(+5)	8(+7)

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

Program Δ = diff	File Commit	LOC(J) Δ LOC(J)	LOC(L) Δ LOC(L)	LOC(M) 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

B. Detecting Meaningful Changes on Three Projects

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 Duley et al [8]. Let 'J' stand for the original code, 'L' stand for the comment-less code and 'M' stand for the normalised code. The Java parsers obtained from the TXL website are already designed to remove all the comments and white spaces in the Java 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' is the number of accumulated lines of code of all the revisions; 'Δ LOC' is the number of accumulated lines of changes detected by the `diff`/`ldiff` utilities.

All the size metrics show that `UArt16650` << `JHotDraw` << `GMF`, roughly by a magnitude of 10. The GMF CVS repository of accumulated lines of code has close to 9 million lines of code. Taking out white spaces/comments does help reduce the size by almost half, indicating that the three open-source programs were all well-commented. The performance

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%)

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 applied, where `mct` has the most effective result. The ratios are compared in Figure 4.

The time, in seconds, it took for the experiment is shown in Table V. "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 for the `diff`/`ldiff` measurements in generating the program. 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, from the slowest on the left to the fastest on the right. We only compare the scalability of five configurations for detecting changes, using a combination of 4 tools, `diff`, `ldiff`, `txl` and `mct`. Figure 5 plots the data in this table.

C. Threats to Validity

Although some of our experiments were large scale, we should not over-generalise our findings. The GMF project

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

is model-driven, and an unknown portion of CVS commits attributes to the code generations, thus it is 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 slightly different measurements. Finally, we did not compare the interactive supervised diff results as some of the related

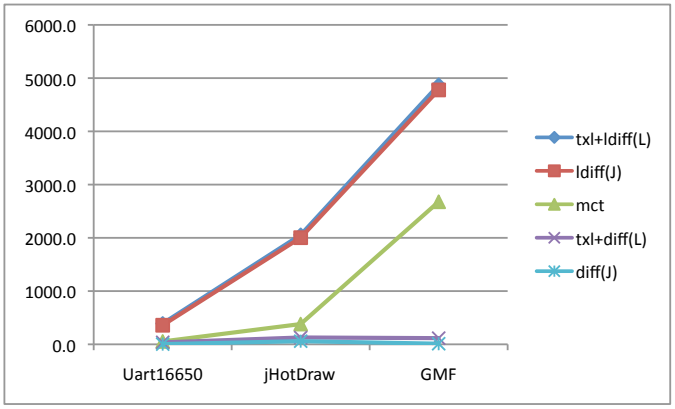

Fig. 5. Contrast the scalability of the experimented command line tools

work does because such supervisions require a large amount of time on the code base, and truthful interpretation from the developers that is intrusive to the development projects. As a trade-off, our command line tool `mct` could substitute the underlying `diff` algorithms for the interactive use cases.

V. RELATED WORK

General problem. Brunet et at [12] define the challenges of model managements in terms of *merge, match, diff, split* and *slice* operations and the properties that should be preserved 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 [13] are also related to our work in the sense that they 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 to compare the differences with 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 [14] 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[1]. 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 [15] present a graph-based algorithm for differencing API of Java 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 as language-independent as ours. In addition, we can also specify the meaningful changes to extract UML models and to isolate user-specified "generated NOT" elements from model-driven software development with only a few additional semantic rules in TXL. These are beyond the scope of `JDiff`.

Beyer et al [16] 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 *visualisation* 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 for checking the differences among ordered structures such as statements or parameter lists.

Fluri et al [17] describe an approach to find differences on abstract syntax trees. The criterion for measuring the

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

size of differences is the length of minimal number of editing operations such as *insert, delete, move, update*, as well as 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 considers 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. Several differencing tools work at the semantic level which may be complementary to ours. Jackson and Ladd [7] use dependency between input and output variables of a procedure as a way to detect certain changes. The dependency 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 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 [18] present a static semantic diff tool called `SymDiff`, which uses the notion of partial/conditional equivalence where two versions of a program are 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 sub-trees in the AST's 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 [19] present a view that the evolution history of models can be traced into modelling elements and visualise the change metrics. When large amounts 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 [20] present a promising way to use change detection for recommendation systems, especially for the framework evolution. Recently, Kawrykow and Robillard [21] demonstrate the `DiffCat` tool to classify both CVS and SVN change sets of Java programs for non-essential changes such that those induced by cosmetic refactoring operations can be ignored. `DiffCat` aims to *classify* the changes as input data that might be non-essential for automated techniques, instead of defining for Java programmers which changes are essential or meaningful. On the other hand, our approach can be used to define a suitable normalisation for each of these classified change types. For example, a local variable renaming could be normalised by renaming any local variable to the surrounding context of its declaration. Therefore it is possible to combine the two approaches further for classifying the different types of meaningful changes.

Kim and Notkin [22] present the `LSdiff` tool to automatically identify structural changes. Similar to us, there are no predefined change types in `LSdiff`, though the abstraction of `LSdiff` is at the level of code element and structural dependencies. While both `LSdiff` and `mct` share a common goal of helping programmers to understand changes at a high level, instead of focusing on identifying and summarizing systematic structural differences, right now our tool focuses on programmer-provided annotations on the production rules to filter out meaningless changes.

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 that "normalise" the structures in order to remove irrelevant changes when evolving programs are compared. To detect an advanced refactoring, especially those happened to the API's (see Dig et al[2]), a user-defined function works on a term of appropriate scope (e.g., class_declaration, or method_body), depending on the 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].

VI. Conclusions and Future Work

In this paper, we have presented a declarative approach to specify changes in programs that are relevant to different programming tasks. In particular, we have demonstrated its use in extracting the changes at the API levels for evolving Java programs and hardware specifications. We have shown that four elementary annotations can account for all light-weight syntactical normalisations by default, and can be customised for more advanced semantic changes through user-defined functions. Our declarative specification approach already works on several programming and domain-specific

languages. Our automated tool has been implemented on top of a generic transformation system, `TXL`, and the results of detecting relevant differences in the three programs were evaluated and compared to 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. The tool currently supports a substantial list of programming languages inherited from http://txl.ca, as well as several requirements modelling languages including i* [26], Problem Frames [27] and argumentation [28], [29].

Our implementation of the clone removal is currently limited to exact clones. We are planning to integrate a near-miss clones detector such as `NiCAD` by Roy and Cordy [11] to ignore more structural changes. Recently, Kim et al [30] proposed the `MeCC` tool to detect semantic clones (i.e. Types 3/4 according to Roy et al [11]) by *statically* analysing the equivalent classes of memory footprints upon the exit of every C/C++ procedure. Their approach adopts a specific technique to guarantee a fixed point can be reached by the normalisations implicitly defined by the checked equivalence classes. Without an explicit user-defined function, our approach cannot be applied to these cases. Therefore, we plan to specify such normalisations explicitly.

Recent work on requirements monitoring [31], [32] indicates that it is important to detect meaningful changes at run-time. In a generic graph-based approach, high-level meaningful changes are expressed declaratively as *change patterns* [33]. We are investigating how to extend our meta-annotation approach to generate such change patterns for detecting *dynamic* meaningful changes at the runtime.

ACKNOWLEDGEMENT

The work is partly 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 programme at Lero. The authors would like to thank Charles B. Haley and Michael A. Jackson 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. 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.

[3] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053–1058, December 1972.

[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.

[5] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, "Closing the gap between modelling and Java," in *2nd International Conference on Software Language Engineering*, 2009, pp. 374–383.

[6] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing," in *20th IEEE/ACM Conference on Automated Software Engineering*, 2005, pp. 54–65.

[7] D. Jackson and D. A. Ladd, "Semantic diff: A tool for summarizing the effects of modifications," in *International Conference on Software Maintenance*. IEEE Computer Society, 1994, pp. 243–252.

[8] A. Duley, C. Spandikow, and M. Kim, "A program differencing algorithm for Verilog HDL," in *IEEE/ACM international conference on Automated Software Engineering*, 2010, pp. 477–486.

[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 *International Conference on Program Comprehension*. IEEE Computer Society, 2008, pp. 172–181.

[12] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh, "A manifesto for model merging," in *International Workshop on Global Integrated Model Management*. ACM, 2006, pp. 5–12.

[13] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and merging of statecharts specifications," in *International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 54–64.

[14] M. Schmidt and T. Gloetzner, "Constructing difference tools for models using the SiDiff framework," in *30th International Conference on Software Engineering*. ACM, 2008, pp. 947–948.

[15] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *19th IEEE international conference on Automated Software Engineering*. IEEE Computer Society, 2004, pp. 2–13.

[16] 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.

[17] 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.

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

[19] S. Wenzel and U. Kelter, "Analyzing model evolution," in *30th International Conference on Software Engineering*. ACM, 2008, pp. 831–834.

[20] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *30th International Conference on Software Engineering*. ACM, 2008, pp. 481–490.

[21] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories." in *31st International Conference on Software Engineering*. ACM, 2011, pp. 351–360.

[22] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *29th International Conference on Software Engineering*. ACM, 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.

[24] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 126–139, February 2004.

[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.

[26] E. S. K. Yu, "Towards modeling and reasoning support for early-phase requirements engineering," in *International Conference on Requirements Engineering*. IEEE Computer Society, 1997, pp. 226–235.

[27] M. Jackson, *Problem frames: analysing and structuring software development problems*. Addison-Wesley, 2001.

[28] C. B. Haley, R. C. Laney, J. D. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Trans. Software Eng.*, vol. 34, no. 1, pp. 133–153, 2008.

[29] Y. Yu, T. T. Tun, A. Tedeschi, V. Nunes Leal Franqueira, and B. Nuseibeh, "OpenArgue: Supporting argumentation to evolve secure software systems," in *19th IEEE International Requirements Engineering Conference*. IEEE Computer Society, August 2011.

[30] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: memory comparison-based clone detector," in *33rd International Conference on Software Engineering*. ACM, 2011, pp. 301–310.

[31] M. Salifu, Y. Yu, and B. Nuseibeh, "Specifying monitoring and switching problems in context," in *15th IEEE International Requirements Engineering Conference*, 2007, pp. 211–220.

[32] Y. Wang, S. A. McIlraith, Y. Yu, and J. Mylopoulos, "Monitoring and diagnosing software requirements," *Autom. Softw. Eng.*, vol. 16, no. 1, pp. 3–35, 2009.

[33] G. Bergmann, I. Ráth, G. Varró, and D. Varró, "Change-driven model transformations. change (in) the rule to rule the change." *Software and Systems Modeling*, pp. 1–31, 2011.