Specifying and Detecting Meaningful Changes

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 that are relevant to their current tasks, therefore not all changes to an 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 meaningful changes in programs. Using our elementary extensions to program language grammars, developers can specify, with limited effort, what types of changes are meaningful. The algorithms for generating the normalisation and clone detection transformations distill the meaningful differences automatically. Our tool has been evaluated on a benchmark of programs to compare with similar 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). 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 principle, 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. 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 meaningless to another, how can one specify the types of changes that need to be detected for this given purpose? Furthermore, how can such a specification be used for an automatic detection? Most change detection tools are good at either reporting *all* changes in programs through general purpose diff algorithms, or at finding out certain or all changes that are *specific* to one particular programming or modeling language. 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 the grammatically correct source programs. The normalised results are always valid in language for the specific purpose, possibly refined from the source language. We show that such normalisations can be specified as as simple as annotations on the original grammar rules, while specific needs of the normalisation can be further accommodated by user-defined transformations. Each type of elementary normalisation corresponds to an elementary kind of production rules in the grammars. Once such annotations are specified, a fully automated meta-transformation can turn them into a composition of transformations that operate directly on the source programs. The composed transformation separates meaningful changes from the meaningless ones.

These specifications, including the meta-transformations, are all written as a few generic modifications to the meta-grammar[1] of the TXL transformation systems [1]. 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 repository 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. Section III explains the approach we adopt to bootstrap the normalisation transformations needed in the implementation of the tool. Section **??** presents the results of a number of experiments in using the tool, and comparing the performance with existing `diff` tools. Section **??** compares the conceptual differences in the design of existing approaches, and indicates some limitations of our approach. Section **??** concludes the findings.

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

II. Motivating Examples

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 reports these changes as 1 deletion and 1 modification of a big chuck in Listing 3. Applying a more advanced algorithm `ldiff` [2] 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-1.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 HelloWorld2.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-1.java HelloWorld-2.java`

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

Applying a structured diff algorithm [2] to the EMF models corresponding to the abstract syntax structures of the two example Java programs[3]), it reports 3 changes: one concerns the renamed compilation units, one concerns the class HelloWorld that about "the order of the reference members", and

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

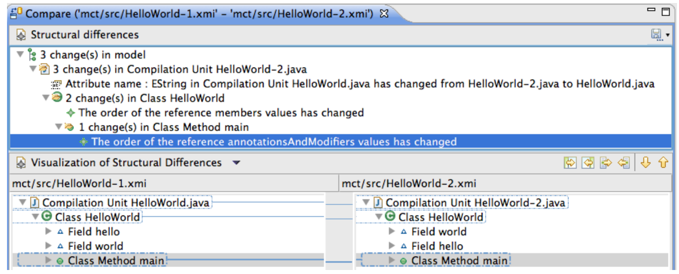

Fig. 1. The differences found by the EMFCompare on the two EMF models

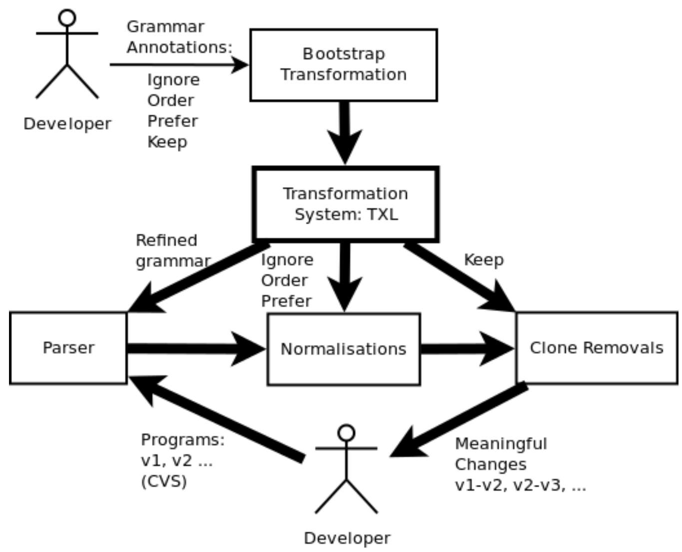

Fig. 2. Specifying and detecting meaningful changes, an overview

the final one concerns the method "main" about "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 detect 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. Transformation*: the tool generates two sets of transformations (normalisation, clone-removal) from the specification in Step 1 and apply 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. Bootstrapping Transformations

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

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

As discussed earlier, the exact meaning for 'meaningful equivalent' in the Definition 1 is intentionally left open for user to define by the normalisation function, because it depends on the purpose of the analysis. 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 $N \leftarrow (T_1? \ldots T_n*)[| \ldots]$ where A is a non-terminal, and T_i is the i-th term (which could be either terminal or non-terminal), and optionally the rule could contain more than one alternative sequence patterns. Every optional (denoted by '?') term can be *ignored* if with or without the value makes no difference to the developer; elements of the repeated (denoted by '*') term can be *ordered* if the ordering of the values are not important to the developer; and the whole element that matches with one mandatory alternative ('|') can be *preferred* to a simpler values if a difference between the alternative rules are not significant to the developers.

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

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

Of course, sometimes the validity of the normalised programs is not a concern if the purpose of checking meaningful changes is not to obtain a compilable program. In those cases, one can relax the requirement about the preferred rule for alternative sequences '|'.

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

After each program revision is normalised, the next task is to detect the meaningful 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 [**?**]. 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 remove 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

TABLE I
BASIC ANNOTATION RULES TO THE 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]	[method_body *preferred*]
Keep	Any non-terminal term	[. . . kept]	[class_body_declaration *kept*]

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, 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.

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.

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
```

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
```

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 <u>ordered</u> 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 7. 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.

Listing 7. `cat -n problem.rules2.grammar`

```
1  ...
2  define problem_description
3     [indent] [repeat E+ ordered by Small] [dedent]
4  end define
5  rule Small B [E]
6   match [E] A [E]
7   construct SA [stringlit] _ [quote A]
8   construct SB [stringlit] _ [quote B]
9   where SA [< SB]
10 end rule
```

Of course, the user may choose to ignore certain information to further abstract the normalised structure, e.g., as indicated in Listing 8, the `details` can be ignored by using `ignore` at the end of the optional part `[opt details]`.

Listing 8. `cat -n problem.rules3.grammar`

```
define E
   [NL] [name]
   [NL] [opt type]
   [opt details ignore]
end define
```

Note that using the above extensions after `opt`, `repeat` and `list` parts, the normalised programs will still be valid for the original syntax.

B. 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 9 shows the extension to the existing `typeSpec` rule and the addition of rules `orderedBy` and `ignored`.

Listing 9. `cat -n grm1.grm`

```
include "grm.grm"
// The extension of the Txl grammar
keys
 ... 'ordered 'by 'ignored
end keys
define typeSpec
 [opt typeModifier]
 [typeid]
 [opt typeRepeater]
 [opt orderedBy]
 [opt ignored]
end define
define ignored
 'ignored
end define
define orderedBy
 'ordered [opt byField]
end define
define byField
 'by [id]
end define
```

The second part of the implementation is a specification of the normalisation transformations, simplified in Listing 10: we removed the very similar rules for eliminating `ignore` annotations, and for producing rules from the `[list X orderedBy]` annotations because they are very similar to that of eliminating the `orderedBy` annotations, and to that of producting rules for the `[repeat X orderedBy]`, respectively.

TXL programs can be understood top-down from the back. Lines 50-71 specify how to generate the transformation rules `Rules` on the fly by checking every `defineStatement` in the TXL grammar such as those definitions in Listings **??** to 8. For each occurrence of `[repeat X ordered by F]`, the transformation in Lines 12-35 is invoked to generate a rule such as those instantiated in Lines 26-31. These rules have unique name because there names are constructed uniquely from the names of the `defineStatement` and `X`. By the end of the main transformation, the rule in Lines 2-10 are applied to eliminate the extended annotations introduced earlier by the rules in the Listing 9.

Listing 10. `cat -n grm.Txl`

```
include "grm1.grm"
rule typeSpec_eliminateOrderedBy
 replace * [typeSpec] T [typeSpec]
 deconstruct T
   M [opt typeModifier] I [typeid]
   R [opt typeRepeater] O [orderedBy]
 deconstruct O 'ordered B [opt byField]
 construct T1 [typeSpec] M I R
 by T1
end rule
% similar rule of typeSpec_eliminateIgnored
function typeSpec_repeat_byField DS [defineStatement] T [typeSpec]
 import Rules [statement*]
 import RuleIDs [id*]
 replace [statement*] _ [statement*]
 deconstruct DS 'define TID [typeid] TYPE [literalOrType*]
   REST [barLiteralsAndTypes*] 'end 'define
 deconstruct T 'repeat I [typeid] R [opt typeRepeater] O [opt orderedBy]
 deconstruct O 'ordered B [opt byField]
 deconstruct B 'by F [id]
 construct StrID [id] _ [quote TID]
 deconstruct I TypeID [id]
 construct ID [id] 'normalise_list
 construct ruleID [id] ID [_ StrID] [_ TypeID]
 construct S [statement*]
  'rule ruleID
   'replace '[ 'repeat I ']
     'N1 '[ I '] 'N2 '[ I '] 'Rest '[ 'repeat I ']
   'where 'N1 '[ F 'N2 ']
   'by 'N2 'N1 'Rest
  'end 'rule
 export Rules Rules [. S]
 export RuleIDs RuleIDs [. ruleID]
 by S
end function
function DS_replace DS [defineStatement]
 replace [statement*] S0 [statement*]
 construct T [typeSpec*] _ [^ DS]
 construct S1 [statement*] _ [typeSpec_repeat DS each T]
 construct S2 [statement*] _ [typeSpec_repeat_byField DS each T]
 construct S3 [statement*] _ [typeSpec_ignore DS each T]
 construct S [statement*] S0 [. S1] [. S2] [. S3]
 by S
end function
function id_to_type ID [id]
 replace [literalOrExpression*] L [literalOrExpression*]
 construct T [literalOrExpression*] '[ ID ']
 by L [. T]
end function
function main
 replace [program] P [program]
 export Rules [statement*] _
 export RuleIDs [id*] _
 construct DS [defineStatement*] _ [^ P]
 construct S [statement*] _ [DS_replace each DS]
 import Rules
 import RuleIDs
 deconstruct P S0 [statement*]
 construct ID [id*] RuleIDs [print]
 construct PL [literalOrExpression*] 'Prg
 construct PL2 [literalOrExpression*] _ [id_to_type each RuleIDs]
 construct L [literalOrExpression*] _ [. PL] [. PL2]
 construct REPLACE [replacement] L
 construct MAIN [statement]
   'function 'main 'replace '[ 'program ']
       'Prg '[ 'program '] 'by REPLACE
   'end 'function
 construct P1 [program] S0 [. Rules] [. MAIN ]
 by P1 [typeSpec_eliminateIgnored]
       [typeSpec_eliminateOrderedBy]
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 by Listings **??** to 8, a concrete normalisation transformation is produced in the original syntax of TXL, as shown in Listing 11. Lines 1-10 are the same as the original rules in the Listing **??** because of the elimination rules. The user-defined comparison rule is retained as lines 11-16. The lines 17-21 and lines 22-28 are respectively generated from the context of the two `orderedBy` annotations from Listing **??**. Lines 17-21 uses the user-defined comparison rule because the `orderedBy` has explicited specified the name of the rule `Small`, lines 22-28 on the other hands use the default string comparison rule using the TXL's builtin rule $>$. Another minor difference is that Lines 17-21 are for arrays `repeat` whilst Lines 22-28 are for comma separated lists. Both of these generated rules `normalise_repeat_problem_description_E` and `normalise_list_details_phenomena` are used by the generated `main` rule to produce the normalised program `Prg`.

Listing 11. `cat -n problem.Txl`

```
1  define problem_description
2    [indent] [repeat E +] [dedent]
3  end define
4  define E
5    [name] [opt type] [opt details] [opt ':] [opt stringlit]
6  end define
7  define details
8    '{ [indent] [list phenomena] [NL]
9    [dedent] '}
10 end define
11 rule Small B [E]
12    match [E] A [E]
13    construct SA [stringlit] _ [quote A]
14    construct SB [stringlit] _ [quote B]
15    where SA [< SB]
16 end rule
17 rule normalise_repeat_problem_description_E
18    replace [repeat E] N1 [E] N2 [E] Rest [repeat E]
19    where N1 [Small N2]
20    by N2 N1 Rest
21 end rule
22 rule normalise_list_details_phenomena
23    replace [list phenomena] N1 [phenomena], N2 [phenomena], Rest [list phenomena]
24    construct T1 [stringlit] _ [ quote N1 ]
25    construct T2 [stringlit] _ [ quote N2 ]
26    where T1 [> T2]
27    by N2, N1, Rest
28 end rule
29 function main
30    replace [program] Prg [ program ]
31    by Prg [ normalise_repeat_problem_description_E ]
32           [ normalise_list_details_phenomena ]
33 end function
```

In brief, the problem frames grammar has got 3 annotations inserted by the user, plus 1 additional user-defined string comparison rule for sorting the nodes in inverse alphabatical order. Similarly, we have annotated 11 repeat/list patterns in the Java5 TXL grammar `java.grammar` without introducing any user-defined ordering rules to accept the ascending alphabatical order by default. These 11 `ordered` annotations already make a big difference for detecting meaningful changes.

D. The normalised programs

From Listing **??**, applying the transformation in Listing 11, the normalised program is shown in Listing 12 where the elements are descending alphabetically, while the phenomena are ascending alphabetically.

Listing 12. `cat -n CommandedBehaviour.n1.problem`

```
problem: CommandedBehaviour
    RB "Commanded Behavior"
    CM M "Control Machine"
    CM — CD {
        event Behaviour1,
        event Behaviour2
    } "b"
    CD C "Controlled Domain"
    CD <~ RB {
        event Command1,
        event Command2
    } : "a"
```

Alternatively when the `[opt details ignored]` is specified, Listing 13 shows the resulting abstraction where the details are ommitted.

Listing 13. `cat -n CommandedBehaviour.n2.problem`

```
problem: CommandedBehaviour
    RB "Commanded Behavior"
    CM M "Control Machine"
    CM — CD "b"
    CD C "Controlled Domain"
    CD <~ RB : "a"
```

As long as the same normalisation is used, two programs with meaningfully changes will be detected while the opposite will not.

Applying the same generic `mct` transformation to the two Java programs in Listings 1 and 3 are now normalised into the same program in Listing 14. Both `hello` and `world` members are ordered after the `main` method by the alphabetical ordering; `public` and `static` are also ordered in the same way. These normalisation would no longer differentiate the variations in the Listings 1 and 3.

Listing 14. `txl HelloWorld.java java.grammar`
`txl HelloWorld2.java java.grammar`

```
public class HelloWorld {

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

    static String hello = "Hello";
    static String world = "world";
}
```

IV. Evaluation

In this section, we aim to evaluate the proposed `mct` tool for the efficiency and scalability to normalise programs. Table III lists the grammars with the number of meaningful annotations as well.

To evaluate the efficiency of `mct`, we take the CVS repository of `org.eclipse.gmf` modeling project, fetched on April 15, 2011. First, we checkout every single revision of every RCS file with the extension of `,v`. Then we compare every consequent files by the `diff`, `ldiff` and `mct` commands. If the differences are non-empty, we count the number of revisions, number of non-empty differences and the time it took to compute the results.

TABLE II
SIZE OF THE FULL GRAMMAR EXTENDED

Grammar	description	LOC	+LOC
txl.grm	TXL meta-grammar	408	15
java.grm	Java 5	979	11
problem.grm	problem frames	82	5

TABLE III
PERFORMANCE OF CHANGE DETCTION

GMF	cmts	Rev.	Hunks	Time
`diff`	with	17,521	93,254	
`+ diff`	w/o	15,116	41,212	
`ldiff`	with			
`txl + ldiff`	w/o			
`mct + diff`	w/o			
`mct + ldiff`	w/o			

V. Related Work

A. Grammarware

[4]

B. Transformation systems

'TXL' [cordy02]

C. Bi- Directional Synchronisation

'UnQL+'

D. Model- Driven Development

'Kermeta', 'ATL'

E. Requirements Traceability

Information Retrieval

F. Change Management

'CVS', 'Subversion'
'Git'

G. Fine- grained Change Management

'Molhado'
Incremental IR

H. Invariant Traceability

RE05, ICSM08, ASE08

I. Model Diff

Xing and Stroulia [5] propose an approach to recover UML models from java code, and compares them producing a tree of structural changes, which reports the differences between the two design versions. Their approach is specific to UML. It uses similarity metrics for names and structures in order to determine various changes made to them. (This paper discusses various types of change operations, the correctness of their tool for those operations. So we can follow their example)

Apiwattanapong et al [6] present a graph-based algorithm for differencing object-orienetd programs. Since their approach and the tool JDiff is geared towards Java, there is explicit support Java-specific features, such as the exception hierarchy.

There are several differencing tool working at the semantic level. Jackson and Ladd [7] uses 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 the semantics but not the dependency graph, such as the changes in constants.

Kawaguchi et al [8] 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.

Brunet et at [9] defines some chanllenges 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.

Duley et al [**?**] 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.

Loh and Kim [**?**], [**?**] 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. [4] [**?**] [1] [10] [5] [11] [12] [**?**] [9] [**?**] [2]

References

[1] J. Cordy, "Txl resources, http://www.txl.ca/nresources.html."

[2] 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

[3] 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.

[4] 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

[5] 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

[6] 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

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

[9] 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

[10] 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

[11] 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.

[12] 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