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 [4]. 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.

Listing 1. `cat -n HelloWorld.java`

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

It is still the same program even after a programmer modifies it into Listing 2.

Listing 2. `cat -n HelloWorld2.java`

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

Traditional comparison tool such as the Unix `diff` utility reports a number of trivial changes (Listing 3), e.g., an insertion of the declaration of the `world` string (the Lines 3-4 chunk of HelloWorld2.java) and the replacement of the chunk of HelloWorld.java (Lines 4-6) with the chunk of HelloWorld2.java (Lines 6-8).

Listing 3. `diff -w HelloWorld2.java HelloWorld.java`

```
3,4d2
<  static String world = "world"; // ending
<
6,8c4,6
<  public  static void main(String args[]) {
<   System.out.println   (hello + ", "
<      + world + "!");
---
>  static String world = "world"; // ending
>  static public void main(String args[]) {
>   System.out.println(hello + ", " + world + "!");
```

Applying a more advanced line-diffing algorithm `ldiff` [3] to this example, one can see that 5 smaller hunks of changes are still reported as 2 insertions, 1 deletions and 2 modifications, even though they are now smaller for programmers to check.

Listing 4. `ldiff.pl -w -o diff HelloWorld2.java HelloWorld.java`

```
3,4d2
<  static String world = "world"; // ending
<
5a4,4
>  static String world = "world"; // ending
6,6c5,5
<  public  static void main(String args[]) {
---
>  static public void main(String args[]) {
7,7c6,6
<   System.out.println   (hello + ", "
---
>   System.out.println(hello + ", " + world + "!");
8,8d6
<      + world + "!");
```

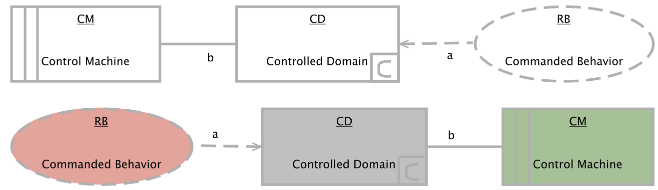

Fig. 1. Equivalent problem frame diagrams: commanded behaviour

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.

A. Graphs and serialisation/abstraction

Graph models have similar problems, if not worse. To illustrate this point, consider the two simple diagrams drawn by an Eclipse-based graphical modeling tool in Figure 1[2]. Although the diagram below has changed the diagram above it by placing the ellipse node (Requirement) on the left rather than on the right, and by filling different colors into the shapes, the topology of the graph and the content of the corresponding EMF model remain the same, according to the domain-specific language defined in Problem Frames [7].

The underlying `HashMap` in the modeling tool, however, could serialise the modeling elements by different (and rather random) orderings, making it more difficult for a simple diff tool to detect meaningful changes to their EMF models. The EMF model can be saved as an equivalent model in corresponding domain-specific (textual modeling) language (DSL) such as those specified using the Eclipse Xtext framework [3] (e.g., List 5).

Listing 5. `cat -n CommandedBehaviour.problem`

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

Besides the whitespace problems, in this example, the phenomena are not alphabetically ordered, nor do the domain nodes. One can imagine easily that swapping two nodes or two phenomena in this serialised model could lead to false alarms which in fact do not require attention by the designers. Comparing two models at the EMF level using a tool such as `EMFcompare` [4], one could avoid such false alarms[5].

[2]Most UML modeling tools based on Eclipse also use the same underlying model-driven technology to create the editor plugins for editing and saving EMF/GMF models such as class diagrams.

[3]http://eclipse.org/xtext

[4]http://eclipse.org/emfcompare

[5]TO CHECK

Moreover, analysts/designers do not always want to view every detail of the modeling elements. For example, the phenomena of the domain nodes in a problem diagram, or the exact declaration of the class methods in a class diagram, are not viewed when the analyst/designers is concentrating on the structural relationship between the larger entities (domain interfaces or class hiearchies). As such, an abstraction transformation is often required before one compare meaningful structures rather than the meaningless details that should be hidden from the view. Of course, programmers may sometimes want to compare the details of the exact declarations of method, while still would wish to ignore the detail implementation of the method bodies. Therefore for the same model (e.g., graph) it is helpful to allow an abstraction as the preprocessing step for a meaningful comparison.

III. Bootstrapping Normalisation

Before explaining the theory and the implemenations of our transformations, we first define *normalisation transformations* and their rationale.

Definition 1: **Normalisation (normalising transformation).** A program P is said to be `normalised` into a program $N(P)$ if any program P' such that $P' \neq P \wedge N(P') = N(P)$ is *meaningfully equivalent* to P. In other words, $N(P)$ is the representative element of the equivalent class to which P belongs, and P' has no meaningful changes to P.

As discussed earlier, the exact meaning for 'meaningful equivalent' in the Definition 1 is intentionally left open or undefined because it depends on the purpose of the analysis. Even so, the definition is still useful because it provides the general criteria for determining whether a transformation is a suitable normalisation once it is clear what is meaningfully equivalent to the users. Once the normalisation transformation is defined, the detection of the meaningful changes becomes comparing the two normalised programs. However, in principle there are infinite possible normalisation transformations for some equivalent classes. For example, adding any number of whitespaces can be regarded as normalisation transformations. According to Definition 2, it is not possible to have infinite number of normalisations because the programs are of finite size.

Definition 2: **Terminable Normalisation.** A normalisation N is terminable if $N(P)$ is strictly smaller than P in size.

Although not mandatory, for pragmatic reasons it is preferable to have the outputs of normalisation readily compared by reusing line-based diff tools.

Definition 3: **Diff-friendly Normalisation.** A normalisation N is diff-friendly, if any line in the normalised program $N(P)$ has at most one meaningful changes.

Given that a program is expressed in programming languages consist of production rules [**?**], our normalisation tool `mct` needs to satisfy the following requirements, in order to handle those examples motivated in the previous section:

- R1 Ignore the trivial differences in optional or repetitive *terminals* in the production rules;
- R2 Ignore certain non-terminals in the rules according to the needs of further abstraction;
- R3 Ignore the ordering of the unordered collections by ordering them sequentially, such that two collections of the same set of elements are the same sequences;
- R4 Carry out the normalisation transformations on the fly, without any user intervention once the grammar rules are extended using the annotations for [R1], [R2] and [R3].
- R5 (optional) Occupy at east one line per non-terminal;
- R6 (optional) Make sure the normalised program is still a valid program in the original grammar.

The requirement [R1] is sufficient to ignore all unnecessary non-terminals in the output program. Such normalisations help focus the comparison more on the abstract syntax rather than on the concrete syntax, because it is the abstract syntax that carries the meaning of the representation while the concrete syntax (reflected by the non-terminals) are merely the auxiliary tokens to parse. The default *unparse* functionality of TXL [4] can already satisfy the requirement of removing extra whitespaces. To remove the extra non-terminals, we must introduce the "ignore" attribute to the type specification, such as `[opt 'terminal 'ignore]` . Requirement [R2] is similar to that of [R1], but now it is the non-terminals that are to be ignored for the abstraction purpose.

Requirement [R3] is useful especially when users knows when a list or an array of literals (tokens and non-terminals) is in fact unordered (e.g., the type modifiers in Java, the phenomena in problem frames), therefore combinatory numbers of possible differences can be removed by ordering the elements in the same way. By default, we can use the serialised string of the literal and sort them in ascending ordering. Of course, such ordering can be made more flexible by allowing users to specify the appropriate ordering key/transformations. Examples of this extension are `[ repeat X ordered ]` where the non-terminal X will be ordered in the normalised program; or `[list X ordered by Y]` where the non-terminal X will be ordered by the comparison rule specified by `Y(X)`. In other words, users can choose to order the elements in descending order, or by the ordering of a particular key field.

Satisfying the requirement [R4] would allow the transformation from the program to a 'simplified' grammar to be generated on the fly, appending additional transformation rules by either ignoring or ordering the literals that have been extended. The implementation of [R4] is done through the technique of *bootstrapping*, that is, to reflectively annotate certain literals in the TXL meta-grammar using the `ignore` and `ordered by` extensions given that the TXL grammar itself is expressed in TXL as a meta-grammar. By processing each annotation in the context of the production rules, it produces a context-aware rule for the transformation on-the-fly. Then the annotations are stripped off by default, which produces a pure-TXL grammar without the annotations, while additional rules based on those removed annotations are combined together. This combined grammar specification is then used to parse the programs in the original language and produces the normalised programs.

Optionally, requirement [R5] can be enforced by inserting a new line directive [NL] to the end of each non-terminal literal in the production rules. Sometimes the normalised program does not have to be a valid program in the same grammar, because removing the details may make the output no longer a valid program, therefore [R6] is the default preference applied if the user would like to preserve the program syntax as well. For example, the abstracted program is still in valid Java syntax by retaining the { } braces while hiding the body of a Java method.

A. A Running Example

To illustrate the application of the approach laid out in the previous section, here we use the example of the problem frames syntax to illustrate the point. Listing 6 selects to show three production rules of the original problem frames grammar. Lines 1-3 define a `problem_description` as an array of elements (`E`); Lines 5-7 define each element to have an optional description of `details`; and Lines 9-13 define the details by a list of comma separated `phenomena` inside curly braces.

Listing 6. `cat -n problem.rules0.grm`

```
1  define problem_description
2     [indent] [repeat E+] [dedent]
3  end define
4
5  define E
6     [NL] [name] [opt type] [opt details]
7  end define
8
9  define details
10    '{ [indent]
11    [list phenomena]
12    [NL] [dedent] '}
13 end define
```

Since one does not care whether an element is before another element or not, the array of `E` is unordered. Similarly, the ordering of the phenomena list is unimportant to the meaning of the problem frames language. To specify the normalisation, one only has to insert the ordered at the end of the `[repeat E+]` and `[list phenomena]` respectively, as shown in Listing 7.

Listing 7. `cat -n problem.rules1.grammar`

```
define problem_description
   [indent] [repeat E+ ordered] [dedent]
end define
...
define details
   '{ [indent]
   [list phenomena ordered]
   [NL] [dedent] '}
end define
```

Furthermore, if one would like to normalise the elements by the descending order, a user-defined rule `Small` can be added in Listing 8. 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 8. `cat -n problem.rules2.grammar`

```
...
define problem_description
   [indent] [repeat E+ ordered by Small] [dedent]
end define
rule Small B [E]
 match [E] A [E]
 construct SA [stringlit] _ [quote A]
 construct SB [stringlit] _ [quote B]
 where SA [< SB]
end rule
```

Of course, the user may choose to ignore certain information to further abstract the normalised structure, e.g., as indicated in Listing 9, the `details` can be ignored by using `ignore` at the end of the optional part `[opt details]`.

Listing 9. `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 10 shows the extension to the existing `typeSpec` rule and the addition of rules `orderedBy` and `ignored`.

Listing 10. `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 11: 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 7 to 9. 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 10.

Listing 11. `cat -n grm.Txl`

```
1  include "grm1.grm"
2  rule typeSpec_eliminateOrderedBy
3   replace * [typeSpec] T [typeSpec]
4   deconstruct T
5     M [opt typeModifier] I [typeid]
6     R [opt typeRepeater] O [orderedBy]
7   deconstruct O 'ordered B [opt byField]
8   construct T1 [typeSpec] M I R
9   by T1
10 end rule
11 % similar rule of typeSpec_eliminateIgnored
12 function typeSpec_repeat_byField DS [defineStatement] T [
       typeSpec]
13  import Rules [statement*]
14  import RuleIDs [id*]
15  replace [statement*] _ [statement*]
16  deconstruct DS 'define TID [typeid] TYPE [literalOrType*]
17    REST [barLiteralsAndTypes*] 'end 'define
18  deconstruct T 'repeat I [typeid] R [opt typeRepeater] O [
        opt orderedBy]
19  deconstruct O 'ordered B [opt byField]
20  deconstruct B 'by F [id]
21  construct StrID [id] _ [quote TID]
22  deconstruct I TypeID [id]
23  construct ID [id] 'normalise_list
24  construct ruleID [id] ID [_ StrID] [_ TypeID]
25  construct S [statement*]
26   'rule ruleID
27    'replace '[ 'repeat I ']
28      'N1 '[ I '] 'N2 '[ I '] 'Rest '[ 'repeat I ']
29    'where 'N1 '[ F 'N2 ']
30    'by 'N2 'N1 'Rest
31   'end 'rule
32  export Rules Rules [. S]
33  export RuleIDs RuleIDs [. ruleID]
34  by S
35 end function
36 function DS_replace DS [defineStatement]
37  replace [statement*] S0 [statement*]
38  construct T [typeSpec*] _ [^ DS]
39  construct S1 [statement*] _ [typeSpec_repeat DS each T]
40  construct S2 [statement*] _ [typeSpec_repeat_byField DS
        each T]
41  construct S3 [statement*] _ [typeSpec_ignore DS each T]
42  construct S [statement*] S0 [. S1] [. S2] [. S3]
43  by S
44 end function
45 function id_to_type ID [id]
46  replace [literalOrExpression*] L [literalOrExpression*]
47  construct T [literalOrExpression*] '[ ID ']
48  by L [. T]
49 end function
50 function main
51  replace [program] P [program]
52  export Rules [statement*] _
53  export RuleIDs [id*] _
54  construct DS [defineStatement*] _ [^ P]
55  construct S [statement*] _ [DS_replace each DS]
56  import Rules
57  import RuleIDs
58  deconstruct P S0 [statement*]
59  construct ID [id*] RuleIDs [print]
60  construct PL [literalOrExpression*] 'Prg
61  construct PL2 [literalOrExpression*] _ [id_to_type each
        RuleIDs]
62  construct L [literalOrExpression*] _ [. PL] [. PL2]
63  construct REPLACE [replacement] L
64  construct MAIN [statement]
65    'function 'main 'replace '[ 'program ']
66        'Prg '[ 'program '] 'by REPLACE
67    'end 'function
68  construct P1 [program] S0 [. Rules] [. MAIN]
69  by P1 [typeSpec_eliminateIgnored]
70        [typeSpec_eliminateOrderedBy]
71 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 7 to 9, a concrete normalisation transformation is produced in the original syntax of TXL, as shown in Listing 12. Lines 1-10 are the same as the original rules in the Listing 6 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 7. 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 12. `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 5, applying the transformation in Listing 12, the normalised program is shown in Listing 13 where the elements are descending alphabetically, while the phenomena are ascending alphabetically.

Listing 13. `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 14 shows the resulting abstraction where the details are ommitted.

Listing 14. `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 15. 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 15. `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 II lists the grammars with the number of meaningful annotations as well.

TABLE I
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 II
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			

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.

V. Related work

A. Grammarware

[9]

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 [12] 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 [1] 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 [6] 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 [2] 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 topdown. 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. [9] [**?**] [4] [11] [12] [5] [10] [**?**] [2] [**?**] [3]

References

[1] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[2] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A manifesto for model merging. In *Proceedings of the 2006 international workshop on Global integrated model management*, GaMMa ’06, pages 5–12, New York, NY, USA, 2006. ACM.

[3] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Tracking your changes: A language-independent approach. *IEEE Softw.*, 26:50–57, January 2009.

[4] James Cordy. Txl resources, http://www.txl.ca/nresources.html.

[5] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33:725–743, 2007.

[6] Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, ICSM ’94, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.

[7] M. Jackson. *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley/ACM Press, 2001.

[8] Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebelo. Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft, October 2010.

[9] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14:331–380, July 2005.

[10] Maik Schmidt and Tilman Gloetzner. Constructing difference tools for models using the sidiff framework. In *Companion of the 30th international conference on Software engineering*, ICSE Companion ’08, pages 947–948, New York, NY, USA, 2008. ACM.

[11] Sven Wenzel and Udo Kelter. Analyzing model evolution. In *Proceedings of the 30th international conference on Software engineering*, ICSE ’08, pages 831–834, New York, NY, USA, 2008. ACM.

[12] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE ’05, pages 54–65, New York, NY, USA, 2005. ACM.