

Towards Learning to Detect Meaningful Changes in Software

Yijun Yu*, Arosha Bandara*, 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 concerned with particular changes that are relevant to their current tasks: not all changes to evolving software are equally important. Specified at the language-level, we have developed an automated technique to detect only those changes that are deemed meaningful, or relevant, to a particular development task [1]. In practice, however, it is realised that programmers are not always familiar with the production rules of a programming language. Rather, they may prefer to specify the meaningful changes using concrete program examples. In this position paper, we are proposing an inductive learning procedure that involves the programmers in constructing such language-level specifications through examples. Using the efficiently generated meaningful changes detector, programmers are presented with quicker feedback for adjusting the learnt specifications. An illustrative example is used to show how such an inductive learning procedure might be applied.

I. INTRODUCTION

Changes are the norm for many software development projects, yet 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.

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. Recently we have developed an automated technique to extract meaningful parts of a program according to a programmer-defined specification [1]. The tool is based on TXL, a grammar-based source to source transformation systems developed by Cordy [4]. To create the parser for the concrete syntax of a programming language using TXL, programmers only need to specify a list of tokens and keys and a set of production rules. As a result, TXL can parse a program of the language, as well as pretty-print (unparse) the program in the source form automatically.

Our extension to TXL is a refinement of the syntax of production rules to derive a set of meta-transformations that *normalise* the parsed program before it is pretty-printed. The

normalisation is a composition of elementary modification operations on individual terms in the production rules, which is guaranteed to be terminable and is resulting in a fixed point by recursively applying the normalisations. Moreover, we also extend the annotation refinement to derive meta-transformations for detecting and removing cross-program clones, such that the structural differences of the two programs are evident.

Figure 1 presents an overview of this approach in finding meaningful change between two versions of a program as two steps:

- *Step 1. Specification:* A developer specifies annotations onto the grammar terms of programs, see dotted arrows in Figure 1;
- *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 1.

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.

Even though the annotations are very simple, however, it is found that not all programmers are familiar with the

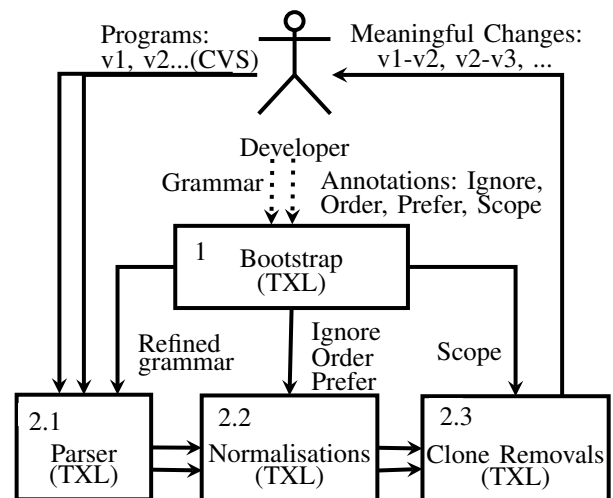


Fig. 1. Specifying and detecting meaningful changes using `mct`

annotations because the production rules are usually abstract. If the programmers are not concerned with the language design, they may incline to indicate only the pairs of concrete program changes that are meaningful. For better usability of the *mct* tool, it is therefore possible for us to consider an inductive learning procedure that may find out the specification of meaningful changes according to given concrete program examples by searching among a space of alternatives at the grammar level. In other words, a programmer in Figure 1 is also responsible for training the system with a set of meaningful or meaningless changes, and an additional learning component derives the grammar and annotations from the search space we will explain in the next section.

II. THE SPECIFICATION SEARCH SPACE

Before defining the specification search space, we first use an example to illustrate the typical usage of *mct* and explain what a typical specification for *mct* looks like.

Given the grammar of a programming language, say Java (Line 1), a developer first annotates its production rules, as shown in Listing 1.

Every clause like *annotate X [Y mods]* introduces one modification to an existing production rule with *X* being the left-hand-side (LHS), *[Y mods]* being the term *Y* modified by the basic modification operations *mods* at the right-hand-side (RHS) of the production rule. For example, according to Line 2 in the Listing 1, the term `import_declaration` appears on the RHS of the production rule of `package_declaration` is modified by the `ignored` operator. That is to say, for a concrete Java program, all the `import` declaration statements will be ignored by the generated normalisation. Similarly, the `ordered` operator is used to generate a transformation that reorders all the type declarations according to the next *annotate* rule (Line 3). In Line 7, the *preferred with* `';` operator leads to a transformation that modify all the method bodies by a semicolon, thus the generated transformation will ignore the implementation details of the methods. In Line 5 and 6, the two *ordered* operators respectively normalise the transformed class to have alphabetically ordered modifiers per method signature, or alphabetically ordered member declarations. As indicated in Line 5, the basic modification operators are composable, making it possible to both ignore certain method declarations of private members, and order the remaining declarations in the resulting APIs. Of course, not all the annotations are specified on the syntactical level. The two user-defined semantics functions such as `Descending` ordering and `Private` member selection need to be provided by the programmers.

Last but not the least, the *scoped* operator appeared in the specification tells the *mct* tool to perform clone removal at the right levels of abstraction. Line 4 indicates that if two classes are exactly the same after the normalisation step, then we can remove them from the results as there is no meaningful changes to them by definition. Line 5 says that if two methods are exactly the same, then we can also remove them to show those methods that are meaningfully changed on the APIs.

Listing 1. `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

```

All the modification operators of the annotation rules currently supported by the *mct* tool are listed in Table I.

As a result of applying the transformation specified above, the following two Java programs are considered to be the same, without meaningful changes on the APIs.

Listing 2. `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 3. `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 }

```

The *mct* tool will now show the following output since all the cross-program clones on the normalised programs are now marked:

Listing 4. `mct HelloWorld.java HelloWorld-2.java`

```

1 <<<<< 1 "public class HelloWorld {public static void main (String args [])};"
2 >>>>> 1 "public class HelloWorld {public static void main (String args [])};"

```

In this case, both APIs are exact clones according to the clone detectors. Therefore, their source forms are printed as one pair of equivalent strings. On the other hand, if these two Java programs are not considered as equivalent, then the annotation rules require to be changed in order to reflect what the programmers really want.

TABLE I
BASIC ANNOTATIONS TO THE TERMS IN A TXL GRAMMAR

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

In summary, the specification search space consists of the alternative modification operators per term in the production rules of a programming language, which are finite; plus infinitely possible number of user defined functions which can be restricted to common recurring patterns such as “conditional ignoring” and “ascending/descending ordering”, etc.

Therefore, it is best not to treat the learning as a fully automated process. Instead, programmers’ interaction shall be supported by an interactive learning process given that the heuristics can be accumulated and reused within the development projects.

III. MACHINE LEARNING TECHNIQUES

At a high-level, the problem we are trying to address can be considered to be one of automatically identifying the changes in a given fragment of code as meaningful (or not), based on some example changes that have been correctly classified by the software developer. In other words, this is an example of a supervised learning problem.

One approach to deriving such a classification scheme would be to use statistical learning techniques, such as those used for data mining [5], to map the changes detected by traditional text-based “diff” tools into the the categories of meaningful and non-meaningful. However, this approach has the disadvantage of producing a classification scheme that is opaque to the developer, with no scope for manual adjustment of the rules used to identify meaningful changes.

Therefore, we propose using an Inductive Logic Programming (ILP) approach [6] to automating the derivation of the production rule annotations used by the `mct` tool. ILP is a machine learning technique that uses positive and negative examples (E^+/E^-) to derive a set of logical hypotheses (H). The particular feature of the derived hypotheses is that, when combined with some background knowledge (B), they logically entail all the positive examples ($H \cup B \vdash E^+$) but none of the negative ones. The scope for the terms used by in the derived hypotheses can be specified using a special set of rules called the *language bias*. In the context of our work, the language bias would be derived from the production rules of the programming language(s) being used by the developer and the TXL grammar annotations being used to isolate the changes. A particular challenge of using ILP to represent the annotated production rules used by our approach to meaningful change detection is the complexity of the production rule language. For example, the production rules have recursive definitions and may even require the use of negation, both of which have been a challenge for many approaches to

ILP [7], [8], [9], [10]. However, recent advances in inductive logic programming, particularly the approach used by the Top-directed Abductive Learning (TAL) tool developed by Corapi et al. [11], have overcome many of these challenges and is therefore a good candidate technique for our problem.

IV. PROPOSED INDUCTIVE LEARNING PROCEDURE

Our proposal for the use of an inductive learning procedure to revise the production rule annotations used by the `mct` tool is illustrated by Figure 2. Initially, the base language grammar (G) is used together with a generic set of annotated production rules (P) to generate a set of changes in the program source code (C) that have been classified as meaningful (mc) and not meaningful ($\neg mc$). The developer then has an opportunity to reclassify the changes as needed to produce example sets of meaningful (mc') and meaningless ($\neg mc'$) changes. These are provided as the set of positive and negative examples to the inductive logic programming tool, `ilp`, which derives a revised set of annotated production rules, P' . These new production rules are used to produce new sets of changes, and the training process will continue until the developer is satisfied with the output of the `mct` tool. In this way, the ILP approach has the advantage of being able to directly derive the set of annotated production rules that can be used by the `mct` tool. Additionally, this means the output of the learning process can be manually revised by the developer if needed.

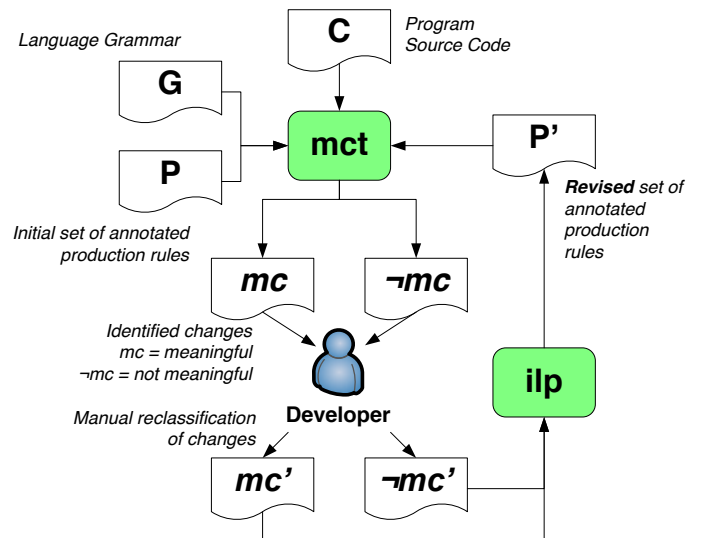


Fig. 2. Learning the meaningful changes from developers using `mct` and an inductive learning procedure

Using the previously defined the space of specifications of meaningful changes, a naive exhaustive inductive learning procedure would be as follows:

- 1) for each term in every production rule in a programming language, check (a) if it is an optional or a repetitive term then the “ignored” modification operator can be added; (b) if it is only a repetitive term, then the “ordered” modification operator can be added; (c) if it is only a mandatory term, then the “preferred” modification operator can be added. (d) In all cases, a “scoped” modification operator can be added to indicate possible abstraction level for clone removals;
- 2) generate a change detector based on that configuration of the specification language;
- 3) for all the pairs of program examples that are given as meaningful changes, test if the selected configuration leads to a change detector that finds the change; and for all the pairs of program examples that are given as meaningless changes, test if the selected configuration leads to a change detector that finds no change. If there is any violation to these two conditions, go back to step (2) to generate another change detector.

For this procedure, the number of change detectors generated can be very large. For the Java grammar with 435 terms in 162 production rules, the total number of configurations with only the 4 basic modification operators would be 2^{1740} , not to mention the infinite number of user-defined semantic functions. Therefore we envision the best way to induce the annotation rules is based on heuristics. Here is one possible procedure for further investigations.

- Use traditional text-based “diff” tools on the pairs of meaningfully change programs to detect a region of the changes on both programs. These regions can hint on the *scoped* modification operator because the clones that can removed must be scoped inside these regions;
- Use traditional diff tools on the pairs of meaninglessly changed programs to detect a region of changes on both programs. The minimal term that can cover this region, be it a class or a method body, or a type of statement, can be tagged as *ignored* modification operators;
- Similarly, if the meaningless change concerns the re-ordered elements that otherwise are the same, one can annotate the corresponding term as *ordered* to avoid such changes;

After applying these steps, if there are still unclassified changes, then one can ask the programmer directly what is the condition for selecting or unselecting the changes as meaningful changes. If such conditions can be expressed by a given conditional ignoring pattern, such as the *Private* function used in our Java APIs example, then we can specify the rules accordingly. These heuristics could be encoded as part of the background theory for the ILP tool. This would modify step (1) of the naive learning approach such that the search space is greatly reduced. Whether it is effective, we aim to assess in the near future through controlled experiments.

At the MALETS workshop, we hope to learn from the participants about the types of existing machine learning (clas-

sification) algorithms can be used to help our programmers in classifying meaningful changes. Also it would be interesting to learn how to collect such classifications from the programmers effectively. In the longer term, a benchmark of meaningful changes in programs or other design documents should be collected to help improve the effectiveness of the learning tools for detecting meaningful changes.

V. CONCLUSIONS

In this position paper we propose a research agenda that aims to apply well-established machine learning techniques, such as inductive logic programming, to the research problem of detecting meaningful changes in software. The end goal of this research would be to develop tools that eliminate the distraction of developers receiving a constant stream of change notifications from the source control system, allowing them to concentrate on the important change events. This has the potential to both improve the productivity of developers while at the same time improving the overall quality of the software being written. Additionally, by building annotated data sets of changes in software, we hope to provide machine learning researchers with an additional resource for testing and enhancing the algorithms for learning to detect meaningful changes.

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] Y. Yu, T. T. Tun, and B. Nuseibeh, “Specifying and detecting meaningful changes in programs,” in *26th IEEE/ACM Conference on Automated Software Engineering*, 2011, accepted.
- [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] J. Cordy, “The TXL source transformation language,” *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.
- [5] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Burlington, MA: Morgan Kaufmann, 2011.
- [6] N. Lavrac and S. Dzeroski, *Inductive Logic Programming: Techniques and Applications*. New York: Ellis Horwood, 1994.
- [7] J. Quinlan, “Learning first-order definitions of functions,” *Arxiv preprint cs/9610102*, 1996.
- [8] B. Richards and R. Mooney, “Automated refinement of first-order horn-clause domain theories,” *Machine Learning*, vol. 19, no. 2, pp. 95–131, 1995.
- [9] O. Ray, “Nonmonotonic abductive inductive learning,” *Journal of Applied Logic*, vol. 7, no. 3, pp. 329–340, 2009.
- [10] S. Muggleton, J. Santos, and A. Tamaddoni-Nezhad, “Toplog: Ilp using a logic program declarative bias,” *Logic Programming*, pp. 687–692, 2008.
- [11] D. Corapi, A. Russo, and E. Lupu, “Inductive logic programming as abductive search,” in *Technical Communications of the 26th International Conference on Logic Programming*, vol. 7, Dagstuhl, Germany, 2010, pp. 54–63.