CoCloRep: A DSL for Code Clones

Robert Tairas[1], Shih-Hsi Liu[2,1], Frédéric Jouault[3,1], Jeff Gray[1]

[1] University of Alabama at Birmingham, Birmingham, AL 35294
{tairasr, gray}@cis.uab.edu

[2] California State University, Fresno, Fresno, CA 93740
shliu@csufresno.edu

[3] ATLAS group (INRIA & LINA), University of Nantes, France
frederic.jouault@univ-nantes.fr

Abstract. Code clones are pieces of code that are duplicated in multiple locations in the source code of a software application. The existence of these clones and the availability of clone detection tools to find them lend to the need for techniques to analyze the clones in an effort to enhance the quality of the source code. This paper describes an investigation into the development of CoCloRep, a Domain-Specific Language (DSL) for the representation of code clones. The benefits of Model-Driven Engineering (MDE) in this context are observed through the use of model transformations on the clones that are represented by instances of this DSL. These transformations provide a means to analyze the clones. The DSL is developed in the AMMA platform (ATLAS Model Management Architecture), which is a modeling tool that provides features through its own DSLs to specify metamodels and to perform transformations on models.

Keywords: Clone Detection, Clone Analysis, Domain-Specific Languages, Model-Driven Engineering

1 Introduction

Code clones represent sections of code that are duplicates of each other and are scattered throughout a software application. A decade ago, Baker documented the existence of clones in large applications as discovered by an automated clone detection tool [1]. Since then, new advances in clone detection capabilities and corresponding tools have emerged each year [7]. Duplicate code is considered a “bad smell” in the refactoring world [4], where clones can pose problems during the maintenance stage of software development, because a modification of one clone copy could require the modification of all of its clones. The elimination of clones through refactoring activities may lessen the challenges associated with maintaining clones. Clones come in many shapes and forms and the removal of all clones is not always possible, and in some situations may not be advisable. In fact, a study has shown that some clones are actually “good” for an application [11]. However, the possibility of discovering clones that can be refactored may assist programmers

during the maintenance phase by reducing the amount of code that must be maintained.

In order to determine refactoring opportunities for clones, analysis of the clones must be performed. The results of clone detection tools provide information about sets of clones that have been automatically detected in a program. Most tools provide a textual listing and visual representation of clones to allow programmers to analyze the results manually. A listing of various clone detection tools and techniques can be found at [14]. A tool by Higo et al. goes a step further and provides a metric system that can compute values for clone sets (or a group of similar clones) to determine refactoring opportunities on the clones [6].

The purpose of this paper is to describe an investigation into the development of a Domain-Specific Language (DSL) for code clones called CoCloRep (Code Clones Representation). The paper provides an initial assessment to determine the benefits of Model-Driven Engineering (MDE) in the context of clone analysis. Instances of CoCloRep represent software clones and their explicit representation as model instances. Through the transformations of these models, information about the clones can be retrieved and analyzed. Two assumptions are made related to the input of CoCloRep. The first assumption is the existence of a clone detection tool that can detect all three types of clones or a combination of them. These types are described in the next section. The second assumption is the ability to generate an instance that conforms to the grammar of the DSL from the results of a clone detection tool.

The paper is organized as follows. The next section presents the language of CoCloRep used to represent clones in Java code. Section 3 provides an implementation overview of CoCloRep in the AMMA platform. Some observations are made in Section 4, with Section 5 listing related work. Section 6 gives a discussion of future work and a conclusion.

2 Language Description

Although no standard definition of code clones exists, CoCloRep is designed to represent the three types of clones used in evaluating various clone detection techniques in [3]. The three types are: exact matches, exact matches with parameterized variables, and near exact matches. An exact match refers to code fragments that are verbatim copies of each other. A parameterized match allows variable names to be different, but still contains the same code sequence. Near exact matches are clones that differ slightly from each other. This may include the existence or absence of a few lines, but the overall source code sections or blocks of code generally match each other.

The following example illustrates near exact matching clones with parameterized variables. This example will be used to describe the language elements of the CoCloRep DSL. The sequence of statements in *Clone 1* and *Clone 2* are the same with the exception of line 3 in *Clone 1* making the clones near exact matches. In addition, each clone uses two different sets of variables for the same code sequence. That is, variables *f* and *g* in lines 1, 2, and 4 for *Clone 1*, and variables *p* and *q* in lines 1-3 for *Clone 2*. These variables are the parameterized variables.

An example of two clone instances appears below:

Clone 1:

```
1: int g;
2: int f = g + 3;
3: i = i + 1;
4: c = f + m;
```

Clone 2:

```
1: int q;
2: int p = q + 3;
3: c = p + m;
```

CoCloRep uses two elements to represent clones called *clone instances* and *clone groups*. Once the clones have been represented, an additional element called a *clone command* is used to perform actions on the clone instances and groups. The three elements are described further below.

Clone Instances. Each clone that is detected is represented by a clone instance. These instances in turn are associated with a clone group. In addition to the clone group association, parameterized variables are "passed" to the clone group using a function parameter-like syntax. Also, source code that is distinct to a specific clone instance is included in the instance body.

In the example code below, instance *r* is declared for *Clone 1* in lines 1-5 and instance *s* is declared for *Clone 2* in line 7. These lines include the associations to the clone group *cg* and the two sets of actual variables that are "passed" to *cg*. The extra statement in *Clone 1* is represented in lines 2-4 inside the *t* element, which is associated with a box defined in *cg*.

An example of two clone instances, named *r* and *s*, appears below:

```
1: instance r = cg(f, g) {
2:   t {
3:     i = i + 1;
4:   }
5: };
6:
7: instance s = cg(p, q);
```

Clone Groups. Clone groups represent the common properties shared among related clone instances. Properties that are distinct are handled through "placeholder" variables and "boxes" for additional code. To accommodate parameterized matching clones, a clone group can contain placeholder variables that can be swapped with the actual variables associated to a specific clone. In the clone group, these variables are prefixed with a '$.' To represent near exact matches, the clone group can contain boxes that can be filled by code that is not found in all the clones of the clone group.

An example of a clone group called *cg*, appears below:

```
1: clone cg($a, $b) {
2:   int $b;
3:   int $a = $b + 3;
4:   {{ t }}
5:   c = $a + m;
6: }
```

In the declaration of the clone group *cg* above, lines 2-5 contain the source code associated with *cg*. Two placeholder variables, (*$a* and *$b*), are defined in line 1 and are used inside the source code of this clone group. These variables are associated with the actual variable names that are passed from the clone instance declarations. A box denoted by double curly brackets in line 4 represents additional sections of code that do not occur in all clone instances of the group.

Clone Commands. Once the clones have been represented through clone instances and groups, they can be "probed" for analysis purposes. In this context, *clone commands* are used to perform a specific activity on the clones. One command that has been implemented is the "variables" command. Executing this command on a clone group will provide information about the variables associated with the group. In the context of refactoring, information about the existence of variables that are both defined inside and outside a clone is needed to determine appropriate refactoring strategies [6]. For example, it is easier to perform the *Extract Method* [4] refactoring task if all variables represented by a clone group are also declared within the clone group. The *variables* command separates the variables in a clone group into different categories. Variables can be declared inside the clone group, declared outside the group and assigned a new value or used only in an expression. The output of the command "variables cg" is given below. The placeholder variables *a* and *b* are declared inside *cg*. The variables *c* and *i* are variables not declared in *cg*, but are assigned new values in *cg*. Note that variable *i* is only found in clone instance *r*. Variable *m* is an externally declared variable that does not change in *cg*.

Variable information using the *variables* command (original layout simplified) is shown below:

```
1: Variable information for clone group cg
2: Declared variables:
3:   b
4:   a
5: Outside assigned variables:
6:   c
7:   i (in instance r)
8: Outside non-assigned variables:
9:   m
```

An *expand* command is also available, which "expands" a clone instance and generates the original code that a clone instance represents. This command was a command initially developed to test the process of obtaining information about clones in this context. It would not be used in the analysis of clones, per se.

3 Implementation Overview

3.1 Supporting Tool: The AMMA Platform

CoCloRep is developed using components of the AMMA platform (ATLAS Model Management Architecture) [13], which provides two DSLs: KM3 and TCS that are respectively used for the specification of the abstract and concrete syntaxes of CoCloRep. Another DSL (called ATL) is used to generate output from the commands described in the previous section (i.e., *variables* and *expand*) through model transformations.

An overview of the process is displayed in Figure 1. The vertical dashed lines separate different "technical spaces." A *technical space* consists of a collection of models and the tools that can manage these models [13]. In the graph, the technical spaces separate the elements between the grammar (EBNF) technology and the modeling technology. Both the source (CoCloRep) and target (output of commands) are in the EBNF technical space. The main process is done in the MDE technical space, which requires projections from and to the EBNF technical space, which in AMMA are called *injection* and *extraction*, respectively. The elements in the graph are also separated by a horizontal solid line into three levels: M1, M2, and M3. Following the MDE paradigm, M1 represents terminal models, which conform to their respective metamodels in M2. M3 consists of meta-metamodels (one per technical space), which conform to themselves.

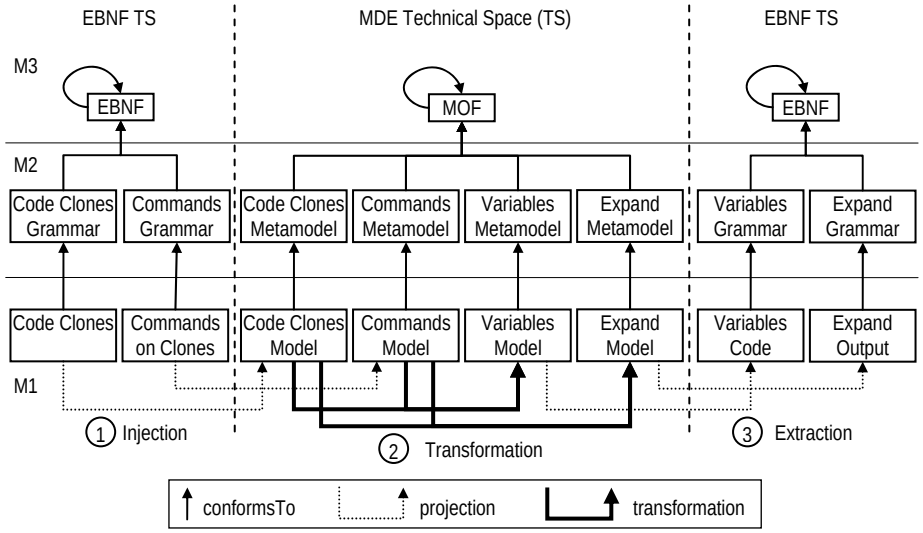

Fig. 1. Model transformation process applied to clone representation

3.2 Metamodel Specification

The EBNF technical space on the left of Figure 1 consists of the code clones and the commands to be performed on them. These two are separated to allow different command sequences to be generated from one representation of clones. In order for processing to be performed in the modeling space, both the clone representation and the commands are "injected" into the MDE technical space (step 1 of Figure 1, denoted by directed dotted lines). The injection produces models that conform to the *Code Clones* and *Commands* metamodels. In AMMA, these metamodels are specified in KM3 [9] and correspond to the abstract syntax. The concrete syntax is specified using TCS [8]. The code snippets below provide an example of these specifications.

A snippet of the abstract syntax of CoCloRep (written in KM3) follows:

```
1: class CloneGroup extends LocatedElement {
2:   attribute cloneName : String;
3:   reference parameters[1-*] container : Variable;
4:   reference statements[*] container : Statement;
5: }
```

The *CloneGroup* element consists of a clone group name (line 2), a set of parameters (line 3), and a set of statements (line 4). The clone group name is of primitive type *String*. The parameters consist of *Variable* elements that represent placeholder variables in the clone group. The statements consist of *Statement* elements that represent the statements associated to the clone. The specification of the *Variable* and *Statement* elements are not shown.

A snippet of the concrete syntax of CoCloRep (written in TCS) is shown below:

```
1: template CloneInstance context addToContext
2:  : "instance" instanceName "=" cloneName{refersTo = cloneName}
3:    "(" arguments{separator = ","} ")"
4:    (isDefined (boxes) ? "{" boxes{separator = ","} "}" ) ";";
```

The *CloneInstance* template defines the concrete syntax for a clone instance. The clone group associated to a clone instance is linked in line 2 through a reference to its clone group name. The existence of boxes in a clone instance is determined in line 4. This allows instances to contain zero or more boxes.

3.3 Model Transformations

The heart of the process is the model transformations that produce output based on the commands that are given on the clone representation. The output of the two types of commands described earlier (*variables* and *expand*) are also specified and implemented using KM3 and TCS. The transformations are between *Code Clones* and *Commands* models as the source models and *Variables* or *Expand* models as the target models (step 2 of Figure 1, denoted by directed solid lines). The model transformations are defined using another DSL of AMMA called ATL [10]. A set of rules that define the transformation between model elements of the source and target

models provide the structure of the transformation as a whole. Code snippets of the model transformation of the *expand* command are given below.

A snippet of a transformation rule for the *expand* command follows:

```
1: lazy rule AssignStat {
2:   from s : Clones!AssignStat
3:     to t : Expand!AssignStat (
4:               variable <- thisModule.processVar(s.variable),
5:               initExp <- thisModule.processExp(s.initExp)
6:            )
7: }
```

The above code snippet contains a rule for the transformation of *AssignStat* elements. This transformation rule is very simple, where the variable and expression associated to an assignment statement are transformed from the source *Code Clones* model to the target *Expand* model (lines 4-5). However, placeholder variables must be replaced by their corresponding actual variables. This is done by processing the variables and expression first through the helper functions called *processVar* and *processExp*. After the transformation is complete, the output of the commands is displayed by extracting the models back into the EBNF technical space (step 3 of Figure 1, denoted by directed dotted lines).

4 Observations

CoCloRep enables clones to be processed in the modeling space through specified model transformations. Processing only in the Grammarware [12] space (staying solely in the EBNF technical space) would also require a representation of clones. One possible representation is the Abstract Syntax Tree (AST) of the program. Subtrees of the AST that represent the clones would be retrieved and processed for analysis. The representation in CoCloRep can be considered as a modified AST that can reduce the number of duplicate branches of the clones in the original AST. This requires the syntax of the original programming language to be part of CoCloRep, and thus must be specified in the metamodel of CoCloRep. In other words, clone representation of a group of clones in CoCloRep is unified into one representation, but this requires the grammar level specifications of a language to be included in the modeling level specifications as well. The boundaries between the two levels are blurry and therefore it has to be ensured that complexities of the lower level do not hinder or slow down the process at the higher level.

The primary mechanisms of MDE are the transformations that occur between models. An ideal transformation considers each element from the source model and transforms it into an element in the target model. This is usually not the case as two models may differ greatly. Such a case can be seen in the transformations described above. If *expand* commands were given to five instances of a clone group, then the target model would contain five copies of the source model. The *variables* command is even more awkward, because it provides variable information, which is just a small part of the source model. Nevertheless, both commands have been implemented using transformations and they demonstrate the feasibility of processing clone

representations to obtain specific information about the clones at the modeling level. Other developed commands related to the analysis of clones may not exhibit such characteristics and may feel more natural in these types of transformations. Furthermore, these commands are intermediary parts of an ultimate goal of the refactoring of clones, which fits nicely in the MDE paradigm, because a source model can be viewed as the original source code that is then transformed to generate a target model representing the refactored source code.

5 Related Work

A generic model for clones is introduced by Giesecke in [5]. This model is based on a different definition of clone sets where a set is based on all matching pairs between clones rather than one group that represents all similar clones. Giesecke's model currently only describes or represents clones. Analysis is delegated to external means, whereas in this paper analysis through the MDE paradigm is one of the main purposes in addition to representing the clones. It is worth noting that the model proposed by Giesecke does strive to separate clone description from clone detection to allow different detection algorithms to use one generic model.

Analysis of clones with the goal of refactoring in the Grammarware space can be seen in [2, 6]. As has been observed, additional comparisons are needed to determine the benefits of work in the modeling level as compared to existing approaches.

6 Future Work and Conclusion

This paper has described CoCloRep, a DSL that can represent code clones and perform commands on the representation to obtain information for the analysis of clones. Source code representing clones are elevated into models in an effort to utilize model transformations to assist in clone analysis tasks. Two commands are introduced on clone models as an initial demonstration of clone analysis at the modeling level. Future extensions will be developed to provide further evidence on how clones represented in CoCloRep can be manipulated.

The commands on clones described in this paper are initial implementations. These commands provide an evaluation of the feasibility of processing clones at the modeling level. Future work will include the investigation of additional commands on the clone representation, such as the analysis of clones with respect to determining refactoring opportunities and comparisons with techniques indigenous to Grammarware. Currently, the representation of clones is limited to a small subset of Java. Future work will also include expanding this to allow representation of more complex clones. A project web site is available at:
http://www.cis.uab.edu/tairasr/coclorep.

Acknowledgments. This project is supported by NSF grant CPA-0702764 and the OpenEmbeDD project.

References

1. Baker, B.: On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Working Conference on Reverse Engineering*, Toronto, Canada, July 1995, pp. 86-95 (1995)
2. Balazinska, M., Merlo, E., Dagenais, M., Lagüe, B., Kontogiannis, K.: Advanced Clone-Analysis to Support Object-Oriented System Refactoring. In *Proceedings of the Working Conference on Reverse Engineering*, Brisbane, Australia, November 2000, pp. 98-107 (2000)
3. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33 (9), pp. 577-591 (2007)
4. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, MA (1999)
5. Giesecke, S.: Generic Modeling of Code Clones. In *Proceedings of Duplication, Redundancy, and Similarity in Software*, *Internationales Begegnungs- und Forschungszentrum für Informatik Schloss Dagstuhl*, *D-06301*, Saarbrücken, Germany, July 2006 (2006)
6. Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: ARIES: Refactoring Support Environment Based on Code Clone Analysis. In *Proceedings of the International Conference on Software Engineering and Applications*, Cambridge, MA, November 2004, pp. 222-229 (2004)
7. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the International Conference on Software Engineering*, Minneapolis, MN, May 2007, pp. 96-105 (2007)
8. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, Portland, OR, October 2006, pp. 249-254 (2006)
9. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In *Proceedings of International Conference on Formal Methods for Open Object-Based Distributed Systems*, LNCS 4037, Bologna, Italy, June 2006, pp. 171-185 (2006)
10. Jouault, F., Kurtev, I.: Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, LNCS 3844, Montego Bay, Jamaica, October 2005, pp. 128-138 (2006)
11. Kapser, C., Godfrey, M.: "Cloning Considered Harmful" Considered Harmful. In *Proceedings of the Working Conference on Reverse Engineering*, Benevento, Italy, October 2006, pp. 19-28 (2006)
12. Klint, P., Lämmel, R., Verhoef, C.: Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology*, 14 (3), pp. 331-380 (2005)
13. Kurtev, I., Bézivin, J., Jouault, F., Valduriez P.: Model-based DSL Frameworks. In the *Companion to the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, OR, October 2006, pp. 602-615 (2006)
14. Tairas, R.: Bibliography of Clone Detection Literature, http://www.cis.uab.edu/tairasr/clones/literature