Issues in Clone Classification for Dataflow Languages

Nicolas Gold, Jens Krinke, Mark Harman
King's College London
Centre for Research on Evolution, Search and Testing (CREST)
{nicolas.gold,jens.krinke,mark.harman}@kcl.ac.uk

David Binkley
Loyola University Maryland
Baltimore, MD, USA
binkley@cs.loyola.edu

ABSTRACT

While clone detection and classification research for textual source code is well-established, clones in visual dataflow languages have only recently received attention. The accepted existing clone classification framework does not adequately capture the nature of clones in the latter kind of programs. In this article, we propose a new classification framework for clone types that may be found in dataflow programs. It parallels the scheme for textual languages but accounts for the differences in syntax and semantics present in graphical languages.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Standardization

Keywords

Clone Detection, Clone Classification

1. INTRODUCTION

Code cloning is widespread in software development and because of the perceived problems that it raises for software maintenance, many methods have been developed to automatically detect clones in traditional text-based programming languages [1,9]. Non-textual dataflow-oriented programming languages have received less attention, the work of Deißenböck et al. [3] and Pham et al. [7] being the notable exceptions. Their work produced two approaches (embodied in the tools CloneDetective and ModelCD respectively) for finding clones in Simulink [6] models.

The development of clone detection methods has been helpfully informed by the formation of a classification scheme for clone types [1] that allows the easy description of a clone detection method in terms of the type of clones that a method can detect. When considering a new clone detection method, this allows one to answer the question *what kind of clones are detected?*, with respect to a particular approach classified within the framework. When considering graphical data-flow programming languages, one finds that the existing classification does not adequately capture all the necessary qualities needed to define a clone (e.g. named variables do not generally exist in dataflow languages thus clone types allowing variable renaming have no meaning). This paper contributes a classification scheme for clones in graphical dataflow languages designed to provide a descriptive capability similar to that already in existence for textual programming languages.

2. CLASSIFYING TEXTUAL CLONES

The existing framework of four clone types for textual programming languages is succinctly summarized by Bellon et al. [1] and Roy et al. [9] and described in more detail by Roy and Cordy [8]. Types 1-3 capture notions of textual similarity and Type 4 captures functional similarity [8]. These authors define the clone types as:

Type 1: Code is copied exactly except for variation in whitespace, layout, and comments.

Type 2: Code is syntactically identical except for variations in identifiers, literals, types, and variations permitted under Type 1.

Type 3: Code is copied but can be further modified by changed, added, or removed statements, in addition to variations allowed under Type 2.

Type 4: Code fragments undertake the same computation but using different syntax.

Note that the types classify clones but do not define what a clone is. This is usually defined to mean that two source code fragments are clones if they are similar with respect to some defined similarity measure.

3. CLASSIFYING GRAPHICAL CLONES

In this section, we highlight some of the problems of applying the existing classification scheme to dataflow programs and define new clone-type descriptions. We ground our synthesis in the existing classification, the work of Pham et al. [7] and Deißenböck et al. [3] on Simulink clone detection, and our own early experiments using Max/MSP [4]. Simulink is widely used in automotive systems development [3] and Max/MSP [2] is widely used for digital music applications. The inclusion of Max/MSP is motivated by our ongoing work [4] on finding music similarity through clone detection methods and the language poses additional challenges to those of Simulink. Note that we do not consider Type 4 clones.

Before actually classifying the clones, we have to define what a clone is in dataflow languages. This is a straightforward adaption of the usual definition for textual clones: Two (sub)graphs are clones if the two graphs are similar with respect to some defined similarity measure.

We start by noting that the textual classification does not include a clone type for code fragments generated by a straightforward copy-and-paste with no reformatting. Since this notion could be more important in graphical clones, we introduce a Type DF0 clone (an equivalent Type 0 could extend the textual classification if desired) thus:

Type DF0: Exactly-copied code fragments.

Type 1 clones are produced by copy-and-paste. Any subsequent reformatting must leave the token order unchanged even if the visual presentation is different. In the context of a visual dataflow language such as Simulink we can maintain this notion of copy-and-paste (and can also ignore comments) but must reconsider the meanings of "token ordering" and whitespace variation. The functionality of Simulink models depends only on the link structure and functions of the blocks used [3]. Dataflow is thus governed entirely by the connections between the blocks; whitespace variation (a rearrangement of block positions) will have no effect on the execution order of the model.

A definition of a dataflow clone-type, equivalent to a Type 1 textual clone, would simply require an exact copy of the original program allowing variation in whitespace, layout, and comments: in other words, an almost direct translation of the original definition. However, this would be too liberal to capture the "copy-and-paste" concept of Type 1 clones if applied to languages like Max/MSP or ProGraph because for these languages, the editing engine determines the execution order [5]. In Max/MSP, the order in which messages are sent is determined by the relative spatial positions of the objects in a patch. Therefore information about layout, rightly dismissed by Deißenböck et al. as irrelevant for Simulink [3], is semantically meaningful and must be captured in a Type 1 clone definition for visual dataflow languages. It is however, only the *relative* position of the objects that is important. This reasoning leads to the following definition:

Type DF1: Exactly-copied code fragments except for non semantics-affecting variations in layout and variations in comments.

The definition of a clone type equivalent to textual Type 2 clones is easier due to the lack of variables in visual dataflow languages. The concepts of renaming identifiers and types are not relevant and thus the only remaining possible variation is in literal values. This leads to the following definition:

Type DF2: Exactly-copied code fragments except for non semantics-affecting variations in layout, variations in comments, and changes to literal values.

A Type 3 clone for visual dataflow languages is defined in terms of the variations that can be made in such languages (e.g. changes to connections, addition/deletion of objects). More formally:

Type DF3: Code fragments with modifications allowing additions, deletions, changes to connections, and free movement of objects.

4. DISCUSSION

The framework defined above offers a clone-type classification for visual dataflow languages. Using it we can now classify the Deißenböck et al. [3] approach as capable of detecting Type DF0, DF1 and DF2 clones and Pham et al.'s approaches [7] as capable of detecting Types DF0, DF1 and DF2 (eScan algorithm) and Types DF0, DF1, DF2, and DF3 (aScan algorithm).

Although the above framework strongly echoes the structure of the textual clone classification, it is appropriate to consider whether this is the best approach. It may be better, for example, to remove the layout-related semantic-invariance condition from DF2. DF1 would be left requiring a strict copy of the original fragment (in semantic terms) and DF2 would then require layout-related semantics-preserving fragment but where layout and/or literals could change. The approaches of Deißenböck et al. [3] and Pham et al. [7] would still be classified as capable of detecting both Types DF1 and DF2 since their methods do not appear to consider literal values and operate on a language that attaches no significance to layout position. Methods operating on Max/MSP however could then be separated into those that preserve the execution semantics and those that do not. This approach would offer a finer-grained separation of methods and clone-types but is less easily compared to the textual clone classification and therefore potentially less intuitive when considering clone detection more broadly. It may also be helpful to limit the amount of modification that can be made for a Type 3 clone to be classified as such. Such changes might be considered for the textual classification scheme also. An alternative formulation of the definitions might use notions of isomorphism with respect to graph labeling so that Types DF1 and DF2 would require isomorphic fragments (rather than "copied fragments"). This kind of isomorphism is used by both Deißenböck et al. [3] and Pham et al. [7] in their clone detection methods and offers conceptually stronger, and perhaps more precise, definitions that could be applied to Simulink and Max/MSP under an appropriate labeling approach. However, this dependency on a specific way of labeling may overly restrict the definitions for the general case.

5. CONCLUSIONS

This paper has proposed a classification framework for clone types in visual dataflow languages. It is sufficiently general to capture clone types in the languages considered but specific enough to allow the differentiation of clones and clone detection methods.

6. REFERENCES

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, Sept. 2007.

[2] Cycling74. Max/MSP. `http://cycling74.com/products/maxmspjitter/`.

[3] F. Deißenböck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proc. 30th Intl. Conf. on Software Engineering*, pages 603–612, 2008.

[4] N. E. Gold, J. Krinke, M. Harman, and D. Binkley. Clone detection for Max/MSP patch libraries (poster). Digital Music Research Network Workshop (DMRN+4), London, 2009.

[5] M. R. Karam, T. J. Smedley, and S. M. Dascalu. Unit-level test adequacy criteria for visual dataflow languages and a testing methodology. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–40, 2008.

[6] Mathworks. Simulink. `http://www.mathworks.co.uk/products/simulink/`.

[7] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proc. 31st Intl. Conf. on Software Engineering*, pages 276–286, 2009.

[8] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 2007-541, School of Computing, Queen's University at Kingston, Sept. 2007.

[9] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.