CLONING IN MAX/MSP PATCHES

Nicolas Gold, Jens Krinke, Mark Harman
Department of Computer Science
University College London

David Binkley
Computer Science Department
Loyola University in Maryland

ABSTRACT

Max/MSP is widely used for developing applications in music and art yet less attention has been given to supporting developers working in this language than for more traditional languages such as Java. Technologies such as code-completion, reuse support, and refactoring may be helpful but are largely unexplored. Such methods rely on detecting similarities between language elements. This paper presents a method for detecting similarities between Max/MSP patches (and sub-patches) based on clone detection techniques. The method has been implemented and a proof-of-concept evaluation has been undertaken by applying it to the set of Max tutorial patches supplied with Max/MSP 5. The results show that significant cloning takes place both within and outwith an individual patch and that, as clone constraints are relaxed, the number of clone pairs increases.

1. INTRODUCTION

Support for Max/MSP programming (outside of the environment itself) is limited in comparison to other more traditional programming languages. There are opportunities for content-based patch-library management and browsing to support developers when building their own patches, and code completion (or suggestion) of commonly-used sub-patches. In order to achieve these, a method for detecting similarity among patches is required.

Since Max/MSP patches are, in effect, source code, techniques that work well in identifying similarities in traditional general purpose programming languages may have much to offer in addressing this problem. One such technique is *clone detection* [1,9]. Clone detection is used to locate and identify sections of program source code that are duplicates (or near-duplicates) of each other (a *clone-pair*). Clones typically arise from the common cut-and-paste activities of software engineers when developing and maintaining systems [9]. A variety of successful automated clone-detection techniques and tools have been developed for text-based programming languages (see Roy et al. [9] for a recent survey). Non-textual dataflow-oriented programming languages have received less attention, with the work of Deißenböck et al. [3] and Pham et al. [8] on clone detection in Simulink [6] models being the notable exceptions. These techniques do not apply directly to Max/MSP patches because spatial position has no semantic meaning in Simulink whereas in Max/MSP, the order in which messages are sent is determined by the relative spatial positions of objects in a patch. Information about layout (rightly dismissed as irrelevant by Deißenböck et al. [3]) is thus important and must be considered by a clone detection method. Max/MSP is not unique in relying on layout attributes to convey semantics (other examples include ProGraph, see the work of Karam et al. [5]).

This paper presents a method for automatic clone detection on Max/MSP patches that accounts for the layout semantics. A proof-of-concept implementation is used to evaluate the approach on the patches supplied as part of the Max tutorials.

2. CLONE DETECTION

Code reuse through copy-and-paste is a common software development activity [9]. Clone detection methods need to be able to find code fragments that are similar without knowing in advance which fragments might be repeated [9]. In order to undertake this task, it is important to establish what is meant by similarity; in other words, if two fragments are said to be clones, what does this mean? A classification of textual clone types has emerged [1,9] but this does not apply directly to languages such as Max/MSP since it refers to elements of textual languages such as types. More recently, Gold et al. defined a classification scheme for graphical languages such as Simulink and Max/MSP [4]. Adapting the existing textual-language definition, they define a dataflow-language clone as "two (sub)graphs that are similar with respect to some defined similarity measure."[4]. This definition is similar to that of Pham et al. [8] and Deißenböck et al. [3] who define clones more formally for their specific contexts. Gold et al. [4] go on to define four types of clones:

- DF0: Exactly-copied code fragments.
- DF1: Exactly-copied code fragments except for non semantics-affecting variations in layout and variations in comments.
- DF2: Exactly-copied code fragments except for non semantics-affecting variations in layout, variations in comments, and changes to literal values.
- DF3: Code fragments with modifications allowing additions, deletions, changes to connections, and free movement of objects.

Note that 'exactly-copied' does not actually require the clones to be created by a copy operation, it is only necessary that the clone *can* be created by a copy operation (and subsequent modifications). It should also be noted that type DF3 is sufficiently broad as to allow any fragment to be a clone of any other (this is also a problem in the textual classification). Consequently, this paper considers only types DF0 to DF2. Following the Simulink-oriented methods [3,8], clones must be disjoint (sub)graphs (clones of textual-language programs are often allowed to overlap).

3. ALGORITHM FOR MAX/MSP CLONE DETECTION

This section outlines the algorithm for Max/MSP clone detection. A Max/MSP patch consists of *boxes*, representing messages and the operations that generate, use and modify them, and *lines*, that represent the flow of data between boxes. Lines are attached to *ports* on the boxes.

3.1. Preprocessing

The first stage involves preprocessing the collection of patches to be analysed. The (JSON-format) source file for each patch is parsed and each box given a unique identifier. The following information is retained:, *Max id, object class, patching rectangle text, number of inlets, number of outlets*, and the *patching rectangle position* and *size*. Nested patches are not parsed recursively but simply stored as a top-level "p" object with the above information. Patch-lines are then parsed and recorded using the unique identifiers assigned to their boxes. This preprocessing approach is similar to that used by Deißenböck et al. [3] and Pham et al. [8] except that positional information about the patching rectangles is also retained.

3.2. Clone Candidate Generation

For each patch-line extracted from the patch, the patch-line itself, and all possible paths reachable from it are stored as candidate clone fragments. If a cycle is found (i.e. following the sequence of boxes and lines leads back to the starting box), the candidate fragment is generated only as far as the start of the second cycle. Thus all edges in the patch are considered but only once in any one path. This set of candidates is subsequently used as a pool against which to compare the candidates. Port numbering of line connections is preserved.

3.3. Clone Detection

Each member, *m*, of the candidate path pool is compared to every same-sized member of the pool (excluding *m* and any path that overlaps *m*). Clone validity is assessed using the following criteria:

- DF2: the pair is a DF2 clone if the types of the objects contained with the candidate path match those in the pool path, the line being considered connects to the same port number on those objects, and the relative position of the boxes at each end of a candidate patch-line on the path is the same as that of the corresponding boxes in the pool (e.g., both source boxes may be above and to the left of their destination boxes).
- DF1: the pair is a DF1 clone if it is a DF2 clone and the literal values contained within the corresponding patching rectangles are the same.
- DF0: the pair is a DF0 clone if it is a DF1 clone and the absolute positional difference of the source and destination boxes of each pair in the candidate path matches that of the pool path being considered.

The algorithm may therefore find that a clone pair can be classified as one, two, or all three types.

4. EVALUATION

4.1. Implementation

The algorithm described in section 3 has been implemented. As it stands the implementation constrains the depth of path that can be generated as described in section 3.2 to avoid running out of memory. In addition, the current implementation only finds clones occurring on paths that descend linearly from their root box. For example, if box A connects to B, C and D on ports 0, 1 and 2 respectively, and box E connects similarly to F on port 0 and G on port 2, a clone relationship would be observed as both (A→B,E→F) and as (A→D,E→G). A post processing step that unifies such cases into A→B,D and E→F,G is presently unimplemented. The current implementation will therefore find all cloned linear paths (unifying linear paths would be likely to find fewer larger clones).

4.2. Experimental Configuration

The implementation was used to evaluate the algorithm when applied to the set of Max patches supplied as part of the Max/MSP distribution [2]. The corpus of patches was first pre-processed to extract all sub-patches into separate files (thus treating each as an independent conceptual entity). In total, this forms a corpus of 68 patches of varying complexity and purpose comprising 2155 boxes and 2102 single lines. The levels of cloning detected are discussed in the next section. The implementation was executed using a maximum clone size of 10 boxes (thus 9 edges).

5. RESULTS

5.1. Clone Pairs Found

The number of clone pairs found is shown in Table 1.

Clone Type	Pairs Found in 6615 Total Paths
DF0	559 (9%)
DF1	1501 (23%)
DF2	5696 (86%)

Table 1. Clone pairs found and proportions by type

As may be expected, more relaxed clone criteria produce greater numbers of clones.

5.2. Body of Code Involved in Clone Relationships

In addition to the overall number of clones, it is interesting to consider the volume of code that is participating in cloning relationships. Figure 1 shows the proportion of code elements that are found in clone relationships. Each code element (a line or a box) is counted only once, regardless of the number of clone pairs it participates in.

It is interesting to note that more than two-thirds of the code elements in the patch set are found in some cloning relationship under the most relaxed criteria (DF2). This indicates the presence of many common structures in the patches being analysed. A surprisingly large number of code elements are involved under the DF1 criteria also (over one-third).

5.3. Clone-Pair Size

The distribution of clone pair sizes is shown in Table 2. As may be expected most clones are found at size two (copies of single boxes are not considered to be clones; thus the smallest possible size is two).

Furthermore, as the clone size increases, the number of clone pairs found decreases. Similarly, as the criteria become more restrictive, the number of clone pairs found decreases.

5.4. Clone Distribution

In addition to the overall summary of cloning presented above, it is useful to consider the distribution of clones across the files. Figure 2 shows this data, indicating, for example that 42 patches have no DF0 clones found within them but 35 have DF0 clones found in other files. It is interesting to note that as the clone criteria become more relaxed, in general, the proportion of fragments cloned outside of their originating file increases. Not surprisingly, the overall number of clones also increases. In the context of a potential application in supporting composers using Max/MSP, this would suggest that DF2 (the most relaxed criteria) clones might be the most helpful kind to present since other types of the cloned instances of a patch currently under development would likely be visible to the composer anyway as they would be in the same file.

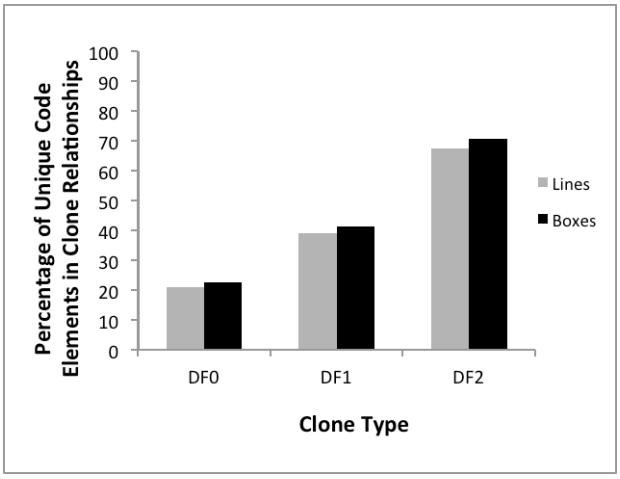

Figure 1. The proportion of unique code elements participating in cloning relationships.

Size\Clone Type	DF0	DF1	DF2
2	298	964	4468
3	146	335	871
4	83	133	245
5	18	43	67
6	9	15	28
7	4	10	10
8	1	1	7

Table 2. Distribution of clone pair sizes.

6. RELATED WORK

Most previous work on clone detection has applied to textual programming languages. This is surveyed by Bellon et al. [1] and Cordy et al. [9]. Clone detection methods for visual dataflow languages have thus far addressed only Simulink models. The approach of Deißenböck et al. [3] is based on graph theory. Simulink models are converted to directed multigraphs and clones are found when two isomorphic (with respect to the node labels) subgraphs are identified. They adopt a heuristic approach to minimize the computational cost. Pham et al. [8] improved on this approach, defining two algorithms for clone identification. The first, eScan, matches clones exactly using canonical labeling of the graph to efficiently compare clone candidates. The second, aScan, uses a vector representation of the graph and computes an edit-distance similarity measure to find approximate clones.

Both methods differ from that presented here. Neither accounts for the layout of the graphical elements and its associated semantics. This limits their applicability to Max/MSP since isomorphism is not sufficient to capture all semantic information available in a patch. Both methods are more efficient than the relatively naïve algorithm presented here but the principles of their approaches could be adopted by future work. In particular, the candidate lists used here are not dissimilar to the clone lattice adopted by Pham et al.

7. CONCLUSIONS AND FUTURE WORK

This paper has presented an approach for finding clones in Max/MSP patches. Although the maintenance of Max/MSP patches is a less significant problem than for traditional software systems, the levels of cloning are somewhat surprising. There are many applications of Max/MSP clone information. It could be used to facilitate content-based browsing of large patch collections based on the occurrence of particular elements. For example, a part-patch query might be issued that expresses something of interest to the user and in return, various patches containing clones of that part-query could be displayed. Clone information might also be used to support composition. For example, a tool could be created that displays other patches sharing similar structures to the one being created.

Future work will include the development of tools that highlight sharing within and between patches, and that can offer "code completion" suggestions based on the content of other patches. In addition, the algorithm could be made more efficient, and the remaining unimplemented step included, to provide clone-pair merging for building the largest possible clone fragments from the linear clone pairs generated by the current method. Initial consideration of this idea indicates that merging fragments may actually be as complex as clone detection itself but this will require further analysis. Finally, it would be valuable to evaluate a broader range of patches.

8. REFERENCES

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo: "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591, Sept. 2007.

[2] Cycling74, *Max/MSP*, http://www.cycling74.com

[3] F. Deißenböck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, S. Teuchert: "Clone detection in automotive model-based development," *Proceedings of the 30th International Conference on Software Engineering*, pp. 603–612, 2008.

[4] N.E. Gold, J. Krinke, M. Harman, D. Binkley: "Issues in Clone Classification for Dataflow Languages," *Proceedings of the 4th International Workshop on Software Clones (IWSC '10)*. ACM, New York, pp. 83-84, 2010.

[5] M. R. Karam, T. J. Smedley, S. M. Dascalu: "Unit-level test adequacy criteria for visual dataflow languages and a testing methodology," *ACM Transactions on Software Engineering and Methodology,* Vol. 18, No. 1, pp. 1–40, 2008.

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

[7] Pd, http://puredata.info

[8] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, T. N. Nguyen: "Complete and accurate clone detection in graph-based models," Proceedings of the 31st International Conference on Software Engineering, pp. 276–286, 2009.

[9] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495, 2009.

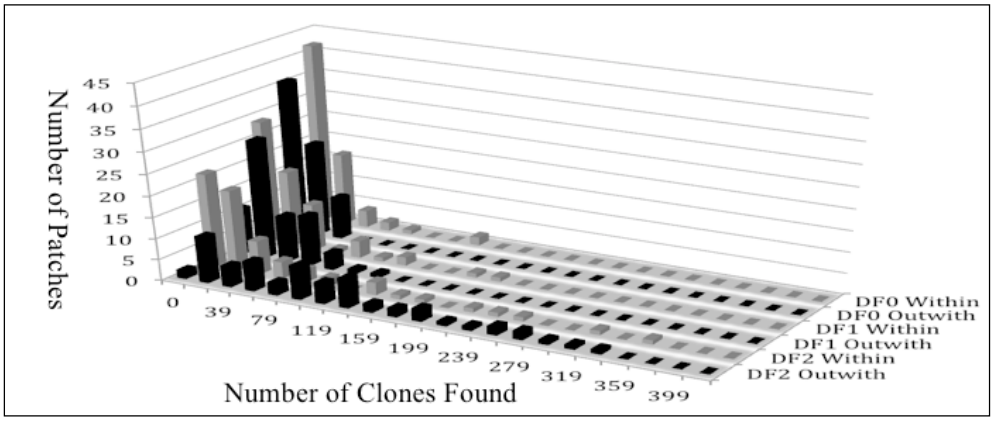

Figure 2: Distribution of clone types/locations, and number of clones by number of patches containing them.