Visual Programming without Procedures

Technical Report UW-CSE-97-05-02

Amir Michail

Department of Computer Science and Engineering
University of Washington, Box 352350
Seattle, Washington, 98195
amir@cs.washington.edu

Abstract

In this paper, we motivate the idea of visual programming without procedures or functions. If a visual program contains several similar cases, then the user need only implement one case and use built-in domain sensitive algorithms that "imitate" the implemented case to automatically produce visual code for the similar cases. This work contributes to visual programming research in two ways. First, it eliminates the need for the user to structure visual code. Second, it reduces the complexity of the visualizations thus making the visual code more apparent. We demonstrate our method using Opsis, a system we built to teach binary tree algorithms.

1 Introduction

Advances in programming language design have often come about by increasing the amount of abstraction, parameterization, and qualification (i.e., local definitions).[5] In particular, abstraction allows programmers to name and define meaningful syntactic classes such as expressions and commands by using functions and procedures, respectively. These syntactic classes may also have parameters and can admit local definitions. Abstraction and parameterization facilitate code reuse while local definitions allow encapsulation. (In this paper, we restrict our discussion to procedures although they apply to other meaningful syntactic classes as well.)

Procedures are not necessarily desired in a visual programming language. This is because the visual language must provide visualizations not only for the domain elements but also for the procedure name, parameters, and local definitions. By eliminating (or reducing) the use of procedures we obtain simpler visual notations that are more readily learned.

A programmer who uses procedures also gives structure to his code. Although this is desirable in a large system, it may not be necessary for smaller visual programs. In particular, one might want the user to concentrate on the task at hand rather than on how to structure the visual code for that task. For example, in a system we developed for teaching binary tree algorithms[4], we want students to concentrate on the algorithms themselves rather than worry about what procedures to define.

Yet, eliminating procedures reintroduces the problems of code reuse and encapsulation. For small to medium-sized visual programs, encapsulation is not essential but code reuse usually is. We certainly do not want the user to repeatedly implement similar cases of an algorithm.

In this paper, we show how visual programming can be done without procedures. To alleviate the code reuse problem, we show how domain sensitive methods can be used to automatically "imitate" a section of visual code in a new context. As an added benefit, the same methods can be used to make changes to existing visual code in non-trivial ways. We demonstrate our method using Opsis, a system we built to teach binary tree algorithms.[4]

In typical visual programming languages, the user first determines *which* procedures to define and how *general* each one should be. A more general procedure will have greater logical and visual complexity. In contrast, using our methods, the user first codes a specific case of the algorithm and then can decide later if such code is necessary in another similar case.

Finally, we stress that we do not advocate eliminating procedures entirely in all visual languages. Rather, we provide an alternative mechanism that may be suited for some small to medium-sized visual programs where one wants the user to concentrate on the task to be done without the distraction and mental burden of structuring code and understanding the more complex visualizations. Moreover, our approach can be used in a hybrid manner where code reuse is accomplished through a combination of procedures and automated generation of code for similar cases.

The remainder of the paper is organized as follows: Section 2 surveys previous work done on related subjects; Section 3 presents the visual programming model; Section 4 describes our techniques for automatically "imitating" a piece of visual code in a new context; and Section 5 provides a summary and future research directions.

2 Past Work

Software engineering researchers have long recognized the importance of code reuse. But code reuse through procedures, functions, classes, and modules, induces structure in the code, which can make it harder to change and adapt the code to new (but similar) applications. This can happen if the associated structure is not suitable for the new application although many of the concepts coded in the old system still apply in the new system. This problem can be more acute with object-oriented programming as the inheritance hierarchy for one application may need substantial changes for a slightly different application.[3, 6]

Consequently, researchers have started to look into ways in which code can be written with less structure. Many of these techniques involve having the programmer write the code in less structured fragments which are then combined to produce a particular application. Although such programs still have structure, it is imposed at a higher level — a form of meta-abstraction. The phase in which the relatively unstructured fragments are combined into a structured whole may be automated to some extent. (For our purposes, unstructured code is important not so much because it makes change easier but because it requires simpler visualizations and is easier to code.)

As an example, Liberherr has developed a system for adaptive object-oriented programming using graph-based customization.[3] Adaptive object-oriented programming facilitates expressing elements that are essential to an application while not making commitments on the particular class structure of the application. A (compatible) class structure for a particular application can be automatically integrated with this code to produce the resultant system.

As another example, VanHilst and Notkin have developed a method to program in a relatively unstructured manner using class templates.[6] Each class template specifies some behavior or aspect of an object. To produce the resultant object, one combines the behaviors (i.e., class templates) through inheritance. In this way, structure is isolated to a small piece of code while the remaining code contains relatively unstructured pieces.

In our approach, we make use of unstructured code but skip the structure inducing step at the end. In other words, we do not even use meta-abstraction. Instead of adding structure at the end, we have algorithms that automatically imitate sections of visual code in a new context.

The idea of making automated changes to software is not new. Johnson and Feather have developed a library of meaning-changing transformations that can be applied to formal specifications.[2] For example, one might make a specification more abstract or add new constructs. In their work, they address the problem of determining how a change in a specifications affects other parts of the specification. In this paper, we also address this problem in the context of visual code.

Recently, Griswold and Notkin have developed a system for making changes in code so that the remainder of the program is modified accordingly to preserve the semantics of the computation.[1] For example, one can inline a function in the source and variables in the function will be renamed according to context. Conversely, one can extract a piece of code and form a function automatically.

Our approach will be a combination of using unstructured code throughout and also using domain sensitive "imitate" functions which allow the user to make automatic changes to the visual code (though not necessarily ones that preserve semantics). As far as we know, these ideas have not been discussed in the visual programming literature although they have been considered recently in software engineering as noted above.

3 Visual Programming Model

For our computation model, we use an *abstract state*, which is an abstract visual diagram that represents a set of concrete states (e.g., a set of binary trees). Two abstract states are *identical* only if their respective abstract visual diagrams match exactly. However, we do allow two states with exactly the same abstract visual diagrams to not be identical if the user so desires. This is allowed because we do not assume that the abstract state completely defines the state of the program at that point. Finally, it is possible for two abstract states to have different diagrams but still represent the same set of concrete states. In the remainder of the paper, we shall use the word "state" whenever we mean "abstract state".

3.1 State Graph

We model a program as a *state graph*, which is a directed graph whose nodes represent states and whose directed edges represent *operations* to transform one state to some other(s). Computation starts at the *initial state* and ends at one of the *final states*. (In Figure 1, the initial state is (1) and the final states are (2a), (4c), (6), and (8). Also, observe that the operation shown (textually) below each tree diagram is invoked on the selected fragments, which are denoted by dashed lines.)

3.2 Visual Language

We demonstrate our techniques using a binary tree visual language that we designed for teaching.

Before we proceed, we need to introduce the notion of a *fragment*: fragment is a (possibly empty) connected subgraph of a binary tree. It is similar to a subtree in that it consists of a root node and descendents of that node. Unlike a subtree, a fragment need not include all descendents of its root. (As an example, in state (4c), we have a fragment with a node as a child, which in turn has two subtrees as children.)

A node or fragment is shaded lightly (heavily) to indicate keys less than (resp. greater than) a particular key K. A fragment may also be shaded lightly on the left and heavily on the right. This indicates that the fragment may have some nodes with keys less than K and some nodes with keys greater than K *but* that it doesn't have a node with key K. The vertical boundary between the light and dark halves indicates a path of nodes, $(x_1, x_2, \ldots, x_j)$, such that:

1. the key at each x_i is not K; and

2. x_i is the left (resp. right) child of its parent x_{i-1} if and only if K is less (greater) than the key at x_{i-1}.

Intuitively, this path is the path followed if we search the binary tree looking for key K. Consequently, we call it a *search path*. (Again, see state (4c) for an example.)

3.3 A Running Example

As a running example throughout this paper, we consider the splay algorithm used to maintain splay trees. No knowledge is required of the reader about splay trees nor their analyses; the splay algorithm visual code just happens to be ideal for demonstrating our techniques.

Roughly speaking, a splay operation consists of a search for a key K in a binary search tree followed by "bubbling up" (via rotations) of either the node with key K if it exists or its inorder successor or predecessor if it doesn't.

We demonstrate the binary tree visual language by explaining the visual code for the initial search in the splay operation. It is not essential for the reader to completely understand the visual code to see how we will manipulate it later on.

The code proceeds as follows. In state (1), we "expand" the tree. If the tree is not empty, this leads to state (2b) in which the root node is revealed. Otherwise, the expansion leads to state (2a) which depicts an empty tree.

In state (2b), we prepare to search for key K in the tree by adding an initially empty fragment above the currently selected subtree. As the new fragment is initially empty, we can require that the node shown be on the search path. Upon adding the empty fragment, we now obtain the loop invariant for the search as shown in state (3).

We now compare the visible node's key with K. The comparison yields one of three states (4a), (4b), (4c) depending on whether the node's key was less than, greater than, or equal to K, respectively.

If the node's key is equal to K, then we have found the node and we are done. Otherwise, we are in state (4a) or (4b). At this point, there is still a subtree which might contain a node with key K. So, we expand this subtree. If it is empty, then K is not in the tree and we have found its inorder predecessor (as denoted in state (5a) with key K^-) or its inorder successor (as denoted in state (7a) with key K^+). If the subtree is not empty (as in state (5b) and (7b)), we collapse the shaded node and fragments as they do not contain K; doing so yields the loop invariant in state (3). In this way, we have formed a loop by matching the loop invariant.

Looking ahead, we also add empty subtrees to states (5a) and (7a) to form a loop invariant which we will use to construct a loop for "bubbling up" a node. The final states of the search are (2a), (4c), (6), and (8).

4 Programming without Procedures

We have implemented the binary tree visual language described in a system named Opsis. We have used Opsis to experiment with various ways to support convenient programming without procedures. These methods are mostly domain independent except for the heuristics described in Section 4.5.

4.1 User Interface

A screenshot of the Opsis system is shown in Figure 2. Editing occurs on the current state which dominates much of the display. On the right, a sequence of states show a history of the computation that leads to the current state. Observe the arrow in the state history list: this arrow indicates that the operation on the state at its tail of the arrow yields the state at its head (thus indicating a loop). At the bottom, the final states of the computations are shown (i.e., these are states at which the computation terminates).

The user interface has been designed to give the user a useful view of the state graph. As the state graph is not hierarchical, it would be easy to become lost in the myriad of states. But in restricting the view in the manner described,

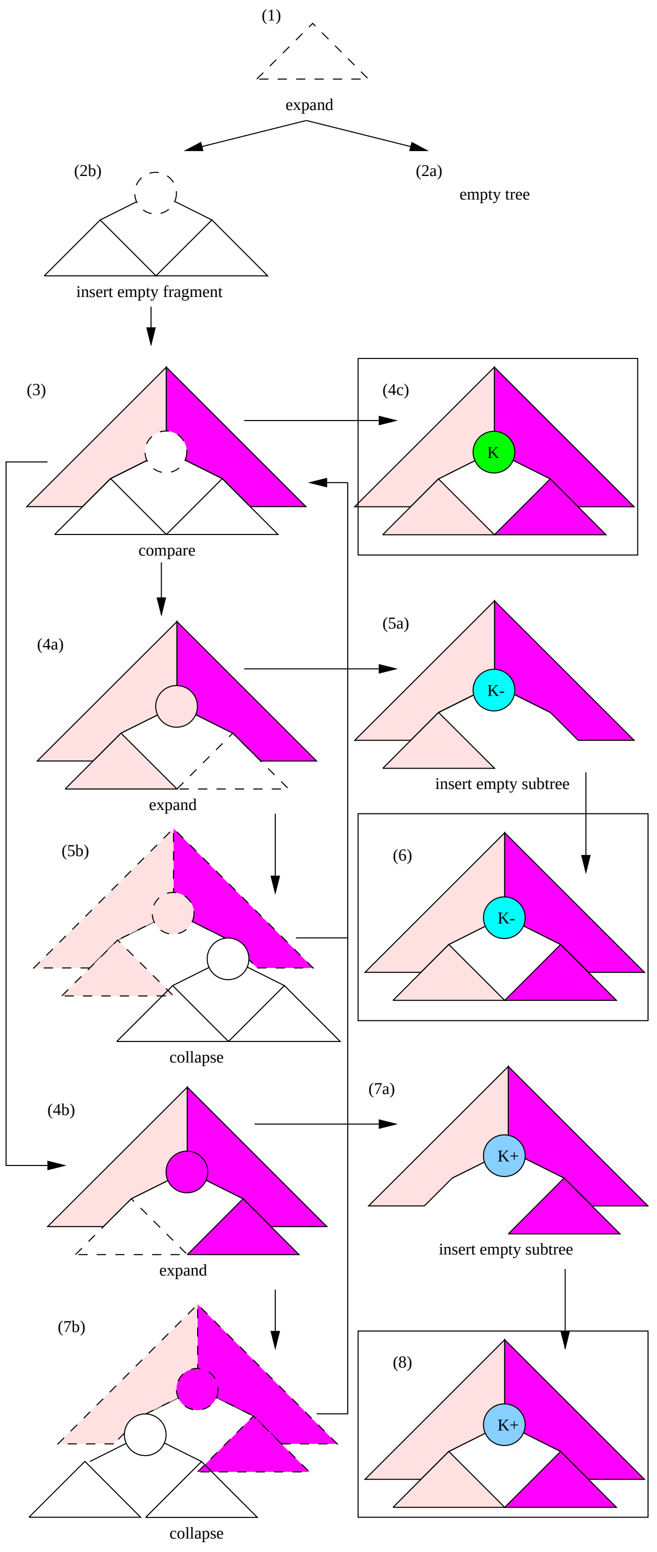

Figure 1. Visual code for initial splay search.

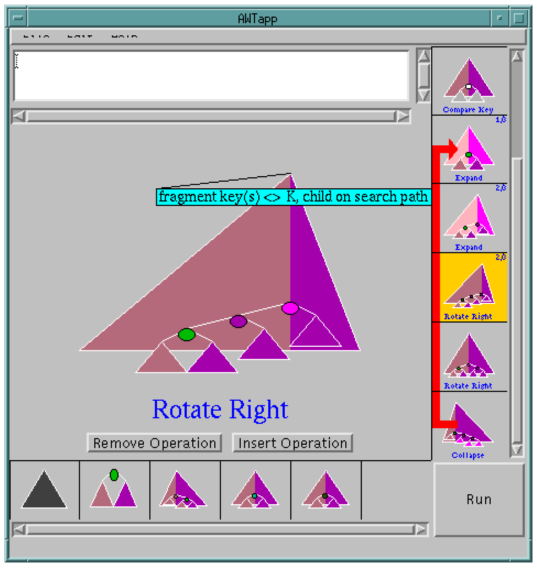

Figure 2. The splay algorithm being developed in the Opsis system.

with the history and final state lists, we have reduced the complexity of the state graph as observed by the user.

.

4.2 User Interaction

A user implements an algorithm by starting out with the initial state and repeatedly invoking operations on one of the final states at each point. This process continues until the only remaining final states in the program represent valid results of the algorithm being defined.

In particular, a user creates a state graph by concentrating on one state at a time. The user selects by mouse some tree node(s) and/or fragment(s) in the *current* state and invokes an operation on that selection. Invoking an operation causes the creation of transitions from the current state to one or more other resultant states. One of these resultant states (chosen arbitrarily if there is more than one) replaces the current state on the screen.

4.3 Imitate Mirror Image

In this section, we show how the user can implement algorithms with symmetrical code. The user need only implement code for one case and then tell the system to generate code for the mirror image case. (Of course, the notion of symmetry is domain dependent; some domains may have many possible symmetries. Our technique can also be applied with other symmetries.)

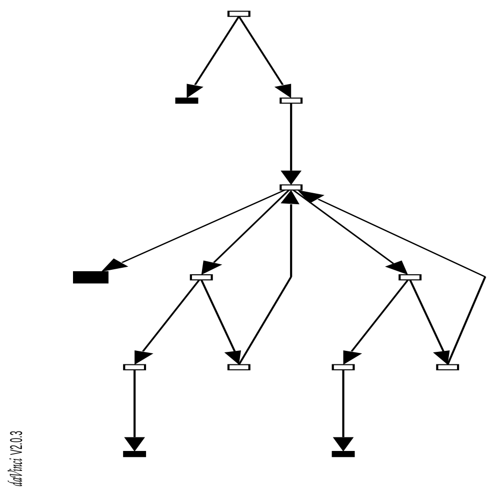

Figure 3. **State graph for initial splay search.**

In Figure 1, we have shown the initial search in the splay operation. The same state graph is also shown in Figure 3, though this time we omit the visual contents of the states to save space. We show final states in black. One of these final states is distinguished (by being larger) and corresponds to state (4c) in Figure 1; this state depicts the case where we have found the key K we were looking for. In Figures 2 and 4, we have some of the code that bubbles up the node with key K. The code is incomplete because we have only handled the case with the node with key K being the left child of its parent; the mirror image case where it is the right child of its parent is not coded yet (as evident from the third final state in Figure 2 and the distinguished final state in Figure 4).

The user could code the other case manually but this would be wasted effort as the two cases are mirror images of each other. Instead, Opsis allows the user to select the state for the uncoded case and simply invoke the "imitate mirror image" command. Opsis then looks for the mirror image state and performs a traversal of the state graph so as to generate the new mirror states for the mirror image case. The result is shown in Figure 6.

The procedure **imitateMirrorImage** is shown in Figure 5 and works as follows.[1] Parameter s is the source state and parameter a is the add state (i.e., where successors will be produced). States s and a are assumed to be mirror images of each other. We add a command to state a when: a doesn't already have a command; a and s are distinct states (i.e., not symmetrical); and s is not a final state. In that case, we use the command in the source s, modified appropriately

[1]The code in Opsis is more complicated because several different states may have exactly the same visual representation. For clarity, we assume here that any new state with the same visual representation as an existing one must match that state.

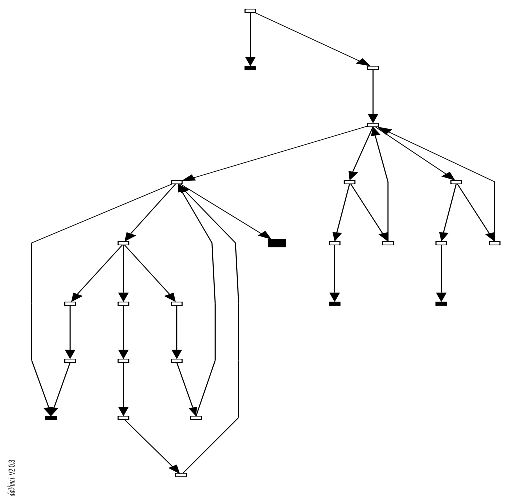

Figure 4. **State graph with partially complete bubbling code for node with key K.**

```
;; s: source state
;; a: add state
;; s is always mirror image of a
proc imitateMirrorImage(s, a)
  s.visited=true
  if a.noSucc>0 or s==a
    return
  endif
  if s.command==null
    a.command=null
    return
  endif
  a.command=mirrorCommand(s.command)
  a.command.execute()
  for i=1 to s.noSucc
    if not s.succ[i].visited
      j=mirrorSucc(s.succ[i])
      imitateMirrorImage(s.succ[i],
                         a.succ[j])

    endif
  endfor i
endproc imitateMirrorImage
```

Figure 5. Code for imitateMirrorImage procedure.

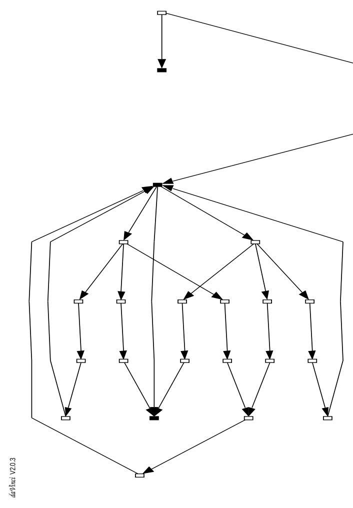

Figure 6. **State graph with complete bubbling code for node with key K.**

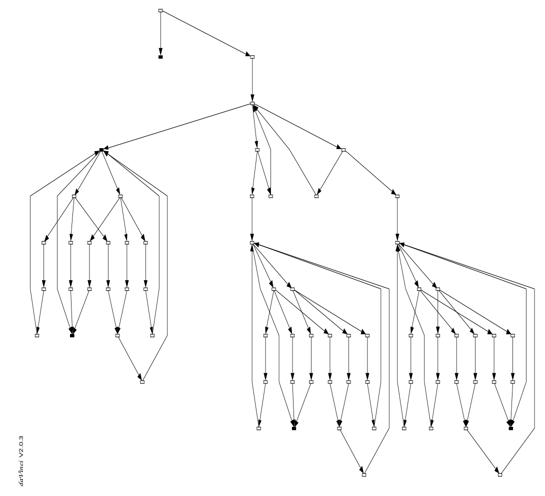

Figure 7. **State graph for complete splay algorithm.**

for the mirror image, in state a. For each successor s' of s not already visited, we determine the newly generated mirror image a' of s' (which is a successor of a) and we then invoke the procedure recursively on a' and s'.

4.4 Imitate Selected State

In this section, we show how the user can generate code for similar cases in an algorithm. The user need only implement code for one case, and then select a source state which the system can then "imitate" in a different setting.

After invoking the "imitate mirror image" in the previous section, we have completed the algorithm to bubble up the node with key K to the root of the tree. This addition to the state graph in Figure 4 leads to the state graph in Figure 6.

However, we still have to code the cases in which we find the inorder successor and inorder predecessor nodes shown in states (6) and (8), respectively, in Figure 1. These two states are distinguished in the state graph in Figure 6. Fortunately, the operations to handle these two cases are identical to those for bubbling up the node with key K. Consequently, we simply select state (4c) in Figure 1 (which is the distinguished state in Figure 3), and then we invoke "imitate selected state" on each of the two distinguished states in Figure 6. The result is shown in Figure 7.

In general, the states being imitated need not match structurally (modulo node and fragment properties) as in the case just described. If the tree structures are slightly different, we use heuristics to select the probable intended fragments in the tree before invoking the operation. We consider this in more detail in Section 4.5.

As explained, the user first selects a source state to imitate. This selection actually makes a copy of the part of the state graph that contains states reachable from the source state (including the source state). Unlike, the "imitate mirror image" traversal which operates on one state graph, the "imitate selected state" traversal operates on two state graphs: the original and the one copied from the selection. (This copying allows us to use "imitate selected state" for editing and not just generation of new states; we explore this issue in Section 4.6.)

The procedure **imitateSelectedState** is shown in Figure 8 and works as follows. Parameter s is the source state and parameter a is the add state (i.e., where successors will be produced). We add a command to state a when: a doesn't already have a command; a and s are distinct states; and s is not a final state. In that case, we use the command in the source s in state a. However, as the trees in states s and a need not be identical structurally, we call a heuristic procedure **determineSelectedFragments**(a, s) to determine which fragments to select in the tree of the add state before invoking the command from s. Finally, for each successor s' of s not already visited, we determine the matching newly generated state a' (which is a successor of a) and we then invoke the procedure recursively on a' and s'.

4.5 Domain Dependent Heuristics

The "imitate selected state" traversal makes use of the **determineSelectedFragments** procedure shown in Figure 9**.** Given a source state and an add state, the procedure **determineSelectedFragments** selects fragments and nodes in the add state based on the tree in the source state. This selection is done prior to invoking the command obtained from the source state.

As the two trees need not be the same structurally, the procedure **determineSelectedFragments** is a domain dependent heuristic. Our implementation of this procedure is motivated by the idea that "imitate selected state" can be useful not only for generating new states but also for modifying state graphs in non-trivial ways. For example, say we forgot to insert an empty fragment to form a loop invariant in Figure 1, (2b), (3). Suppose we realized this after performing the comparison in state (3) and the expand operations in state (4a) and (4b). We would like to insert the "insert empty fragment" operation just before the comparison and have the following compare and expand commands replayed in the new context. The implementation of **determineSelectedFragments** is flexible enough to handle such a situation. (It is possible that this heuristic does not give the desired result; one could alleviate this by also providing semi-automated "imitate" procedures that ask questions of the user.)

In particular, **determineSelectedFragments** works by first trying to unify the two trees as is. If the unification succeeds, then selection is performed by calling the **selectFragments** procedure. (We will explain how procedures **unify** and **selectFragments** work in a moment.)

If unification fails, we attempt to add exactly one new fragment above an existing node or fragment in the source tree. (This allows us to handle the editing example just described.) If unification succeeds, we call **selectFragments** and we are done. If no addition of a fragment to the source leads to successful unification, then we attempt to add exactly one new fragment above an existing node or fragment in the add tree. Again, if unification succeeds, we call **selectFragments** and we are done. Otherwise, no unification has been successful but we still call **selectFragments** as a last resort.

Now, let's look at the **unify** procedure. Let T_s and T_a denote the trees in the source and add state, respectively. Let T' be the *maximal* tree such that the following properties hold:

- T' is a connected subgraph of both T_s and T_a and includes the root of each tree; and
- all nodes and fragments in T' are of the same "structure" as their counterparts in T_s and T_a (modulo the particular properties indicated by shading); that is, nodes must match nodes and fragments must match fragments.

The two trees T_s and T_a unify if

- if a leaf fragment (that is not a node) f' in T' corresponds to f_s in T_s and f_a in T_a, then f_s and f_a both have a child or they both do not have a child; and
- if a leaf node n' in T' corresponds to n_s in T_s and n_a in T_a, then at least one of n_s and n_a has no left child and at least one of n_s and n_a has no right child.

```
;; s: source state in
;;    copied state graph
;; a: add state in
;;    original state graph
proc imitateSelectedState(s,a)
  s.visited=true
  if a.noSucc>0 or s==a
    return
  endif
  if s.command==null then
    a.command=null
    return
  endif
  a.command=s.command
  determineSelectedFragments(a,s)
  a.command.execute();
  for i=1 to s.noSucc do
    if not s.succ[i].visited
      imitateSelectedState(
        s.succ[i],a.succ[i])
    endif
  endfor i
endproc imitateSelectedState
```

Figure 8. **Code for imitateSelectedState procedure.**

Intuitively speaking, T_s and T_a match in that part that is T' with the remainder of T_s and T_a satisfying the restrictions above. Although the definition of whether two trees unify appears rather arbitrary, we found it leads to a good heuristic for selecting nodes and fragments before invoking the command from the source state. In particular, these rules work well for correcting loop invariant errors and in generating similar code for well known balanced binary tree algorithms.

Finally, we look at the **selectFragments** procedure. Basically, this procedure looks at the parts of T_s and T_a that match as T' (where these parts are identical structurally) and selects the nodes and fragments in T_a that are selected in T_s.

4.6 Editing in State Graphs

As hinted at earlier, the "imitate selected state" operation not only allows us to automatically code similar cases, but it also allows us to make changes in the visual code. Essentially, this is done by modifying the code at some point and replaying the sequence of commands that followed before the modification was made.

For example, to insert command(s), one performs the following steps: go to the state in which the command is to be inserted (before the command on that state); select that state; remove the current command on that state (so the state

```
proc determineSelectedFragments(s,a)
  if unify(s,a)
    selectFragments(s,a)
    return
  else
    for each fragment f in s.tree
      s.tree=insertFragment(f)
      if unify(s, a)
        selectFragments(s,a)
        s.tree=removeFragment(f)
        return
      endif
      s.tree=removeFragment(f)
    endfor
    for each fragment f in a.tree
      a.tree=insertFragment(f)
      if unify(s, a)
        selectFragments(s,a)
        a.tree=removeFragment(f)
        return
      endif
      a.tree=removeFragment(f)
    endfor
  endif
  selectFragments(a, s)
endproc determineSelectedFragments
```

Figure 9. Code for determineSelectedFragments procedure.

is now final); perform the new command(s); and imitate the selected state.

Recall that selecting a state copies the portion of the state graph reachable from that state. To see why this is done, consider again the example where we forgot to insert an empty fragment before the comparison to form a loop invariant in Figure 1, (2b), (3). To fix the problem, we select the state with the comparison, remove the compare operation, add the insert empty fragment operation, and imitate the selected state. However, had we not made a copy of the relevant states, we would be imitating the selected state with the insert empty operation and not the compare operation. In essence, our "imitate selected state" works like a "copy and paste" combination.

It is also possible to remove command(s) in the middle of a command sequence. One performs the following steps: go to the state following the last one whose command will be removed; select that state; go to the first state whose command will be removed; remove the command (so the state becomes final); and imitate the selected state.

5 Conclusion

In this paper, we have shown how visual programming can be done without procedures. Essential to our approach is the idea of using automated "imitate" algorithms for code reuse. This approach helps end-users in two ways: it eliminates the need for the user to structure visual code; and it reduces the complexity of the visualizations thus making the visual code more apparent. We have demonstrated our method using Opsis, a system we built to teach binary tree algorithms.

For future research, it would be interesting to come up with a more elegant unify procedure. Perhaps the unify procedure can mutate on-the-fly with the aid of AI learning techniques. In this way, the "imitate selected state" operation can be used to correct common errors that a particular user makes. In addition, the operation can also adapt to the algorithm being designed thus making generation of states more reliable. Finally, we believe that similar techniques might be useful for viewing unstructured visual code. In particular, one can impose structure on the code for the purpose of understanding the visual program though this structure is not inherent in the code.

6 Acknowledgments

I would like to thank Steve Tanimoto for careful reading of this paper, helpful suggestions, and encouragement throughout this research. I would also like to thank David Notkin for interesting discussions about programming using relatively unstructured code. Finally, I would like to thank Ashraf Michail for helpful comments on this paper.

Funding for this research was provided by the Natural Sciences and Engineering Research Council of Canada.

References

[1] W. G. Griswold and D. Notkin. Architectural tradeoffs for a meaning-preserving program restructuring tool. *IEEE Transactions on Software Engineering*, 21(4):275–287, April 1995.

[2] W. L. Johnson and M. Feather. Building an evolution transformation library. In *International Conference on Software Engineering*, pages 238–248, 1996.

[3] K. J. Lieberherr, I. Silva-Lepe, and C. Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, May 1994.

[4] A. Michail. Teaching binary tree algorithms through visual programming. In *Symposium on Visual Languages*, pages 38–45. IEEE, September 1996.

[5] D. A. Schmidt. *The Structure of Typed Programming Languages*. The MIT Press, 1994.

[6] M. VanHilst. Using C++ templates to implement role-based designs. In *International Symposium on Object Technologies for Advanced Software*, 1996.