Imitation: An Alternative to Generalization in Programming by Demonstration Systems

Technical Report UW-CSE-98-08-06

Amir Michail
University of Washington
amir@cs.washington.edu

http://www.cs.washington.edu/homes/amir/Opsis.html

Abstract

In this paper, we propose a new mechanism, imitation, *which can be used in a programming by demonstration system as an alternative to generalization. A system using imitation always asks the user for help when a new case arises in an algorithm. At this point, the user may demonstrate the steps for this case or tell the system: (1) that this case is similar to another one already demonstrated and (2) in what way it is similar. In this way, the user can use imitation to generate code for the new case based on the one already demonstrated. Generalization is not required at any point while programming using this technique. We demonstrate our imitation method using Opsis, a system we built to teach binary search tree algorithms.*

1 Introduction

Existing programming by demonstration systems involve generalization in one form or another. For example, consider demonstrating an algorithm with several cases that are similar in some ways. By using generalization, we can abstract out the similarities and avoid having to demonstrate each of these cases separately. How this would be done varies widely among programming by demonstration systems.

David C. Smith's Pygmalion [11, 12], the first programming by demonstration system, requires users to specify the program in full generality — so they must anticipate and generalize similar cases of an algorithm ahead of time to avoid repetition in the demonstration.

Henry Lieberman's Tinker [5, 6] allows users to demonstrate one case at a time and will ask them to provide a predicate to distinguish one case from another. In this way, users never have to anticipate cases (by providing conditionals ahead of time); they only have to react to the current situation. However, this feature doesn't reduce the number of cases to be demonstrated.

Daniel Halbert's SmallStar [3] allows the user to demonstrate only one of the cases. Demonstrating the one case generates a straight-line program which users generalize *after the fact* by manually adding set iteration and conditionals. The generalized code can then be applied to other similar cases.

Brad A. Myers' Peridot [10, 9] uses inferencing to generalize a demonstration of one case so that it can be applied to other similar cases. To reduce incorrect generalizations, Peridot asks questions to aid its inferencing process.

Whether generalization is done a priori or a posteriori, with or without the aid of inferencing, we encounters two major problems:

1. it requires a more abstract graphical vocabulary than the cases themselves; and

2. it requires the user to abstract and generalize a

case so that the demonstration applies to other cases also.

Concerning problem (1), David J. Kurlander writes in [4], p. 168:

> "If the number of generalizations known by the system is large, then the graphical vocabulary must also be large. Unless the same graphical conventions are used by the system during normal editing, the user would need to learn a new visual language in order to define macros."

For this reason, many systems such as Kurlander's Chimera [4] and Halbert's SmallStar [3] depict generalizations by using textual annotations along with the particular case demonstrated by the user. However, such a solution still requires the user to interpret a more complicated textual/graphical vocabulary.

Concerning problem (2), David C. Smith writes in [12], p. 32:

> "...on the one hand, it (Pygmalion) attempts to make programming accessible to a wider class of users; on the other hand, it relies on the kind of planning which only experienced programmers are good at."

Abstracting and generalizing similar cases requires experience and planning. Demonstrating each case separately (without generalization) is easier conceptually but repetitive and laborious.

In this paper, we propose a new mechanism, *imitation*, which is an alternative to generalization that alleviates these two major problems. A programming by demonstration system using imitation always asks the user for help when a new case arises. At this point, the user may demonstrate the steps for this case or tell the system: (1) that this case is similar to another one already demonstrated and (2) in what way it is similar. In this way, the user can use imitation to generate code for the new case based on the one already demonstrated. *Generalization is not required at any point while programming using this technique.* We demonstrate our imitation method using Opsis [7], a system we built to teach binary search tree algorithms.

The remainder of the paper is organized as follows: Section 2 presents a typical user interaction using the imitation technique; Section 3 describes the implementation; Section 4 surveys related work; and Section 5 concludes with a summary and future research directions.

2 User Interaction

In this section, we demonstrate our imitation method using Opsis, a programming by demonstration system we built to teach binary search tree algorithms. In particular, we show how imitation can be useful in demonstrating the double rotations in the AVL tree insertion algorithm [1].

2.1 AVL Tree Insertion

An AVL tree is a binary tree in which each node has left and right (possibly empty) subtrees whose height differ by at most one. The *balance* of a node is the height of its right subtree minus the height of its left subtree. Thus, each node has balance -1, 0, or +1. (The balance is stored explicitly in each node. See Figure 1, (5).) AVL trees are attractive because: (1) every AVL tree with n nodes has height $O(\log n)$ (so lookups take $O(\log n)$ time); and (2) a node can be added or deleted from an AVL tree with n nodes in time $O(\log n)$, while preserving the AVL property.

The AVL tree insertion algorithm works by: (1) inserting the node as a new leaf x (following the standard binary tree insertion method); (2) updating the balances of ancestors of x, in order of increasing distance from x; and (3) if an ancestor's balance becomes ± 2, then, depending upon the situation, a single or double rotation is performed to restore the AVL tree property.

(In Figure 1, (1), the node with key "T" has been inserted as a leaf and the balance numbers of the ancestors with keys "W" and "R" have been updated appropriately. Since the node with key "R" now has a balance of +2, a double rotation is performed. First, a right rotation about the node with key "W" yields (2). Second, a left rotation about the node with key "R" yields (3). Finally, an update of the balance numbers of the nodes with keys "R" and "W" yields (4) and (5), respectively.)

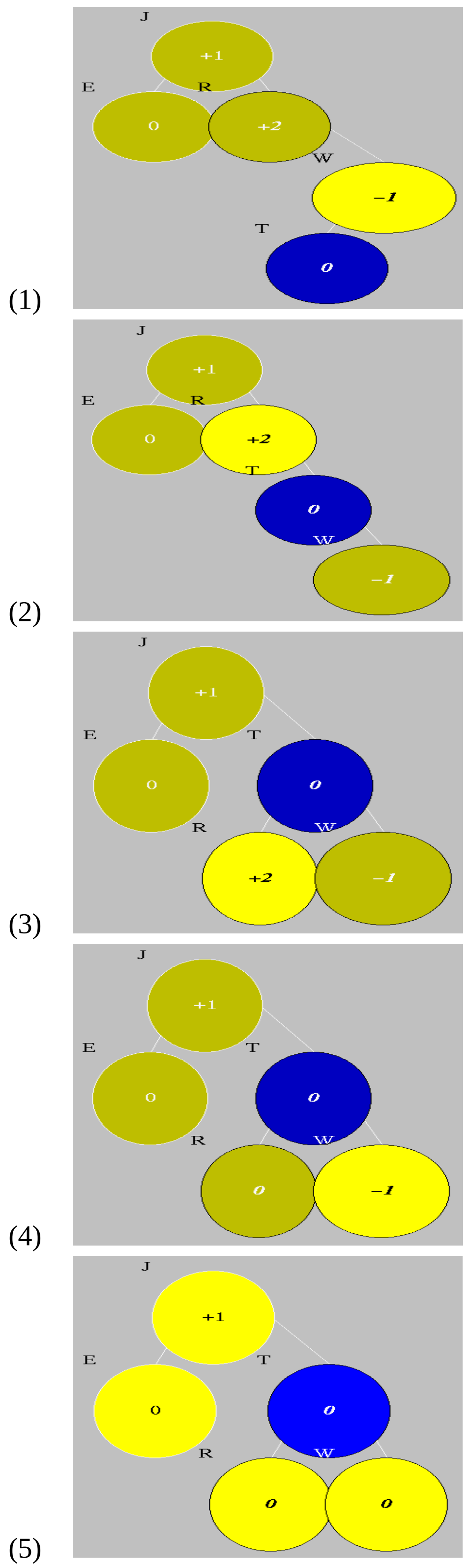

Figure 1. A double rotation in the AVL tree insertion algorithm.

To illustrate a typical user interaction using imitation, we consider how a student might demonstrate the double rotations performed in the AVL tree insertion algorithm using Opsis. The student does this by manipulating abstract trees, each of which represents a set of binary trees with particular properties. (Opsis uses programming by abstract demonstration [2].) During such a demonstration, the student would encounter six abstract cases in which a double rotation is required.

(See Figure 2. A black node denotes the newly inserted node while a black subtree indicates a subtree containing this node. The case in Figure 2 (a) corresponds to the example in Figure 1 and can be demonstrated using Opsis as shown in Figure 3. Also, observe that case (a) and its mirror image involve the newly inserted node in the double rotation while cases (b) and (c) and their mirror images do not as they occur higher up in the tree.)

If we were to generalize one of these double rotation cases so that it applies to the other five, then we would encounter the two problems mentioned in Section 1. In particular, such a generalization would require more abstract graphical notation (especially to take into account the mirror image cases!) and/or textual annotations. Moreover, it is unlikely that the student would realize that there are indeed six double rotation cases without considerable "planning ahead".

Of course, demonstrating the double rotation and updating the node balance numbers appropriately six times is rather laborious — particularly since the six cases are handled similarly. Instead, we show how a student can demonstrate the double rotation steps once (as shown in Figure 3 for case (a) in Figure 2) and use imitation to generate the steps for the other five cases. In particular, we describe two imitation mechanisms available in Opsis that make this possible: *imitate mirror image* and *imitate selected state*.

2.2 Imitate Mirror Image

Many algorithms involve some form of symmetry. In particular, binary tree algorithms tend to have cases which are mirror images of other cases. Indeed, the steps required for the double rotation cases (a), (b), and (c) can be "mirrored" to obtain those steps for the respective mirror images of those cases (and vice versa).

When presented with a case to demonstrate, the stu-

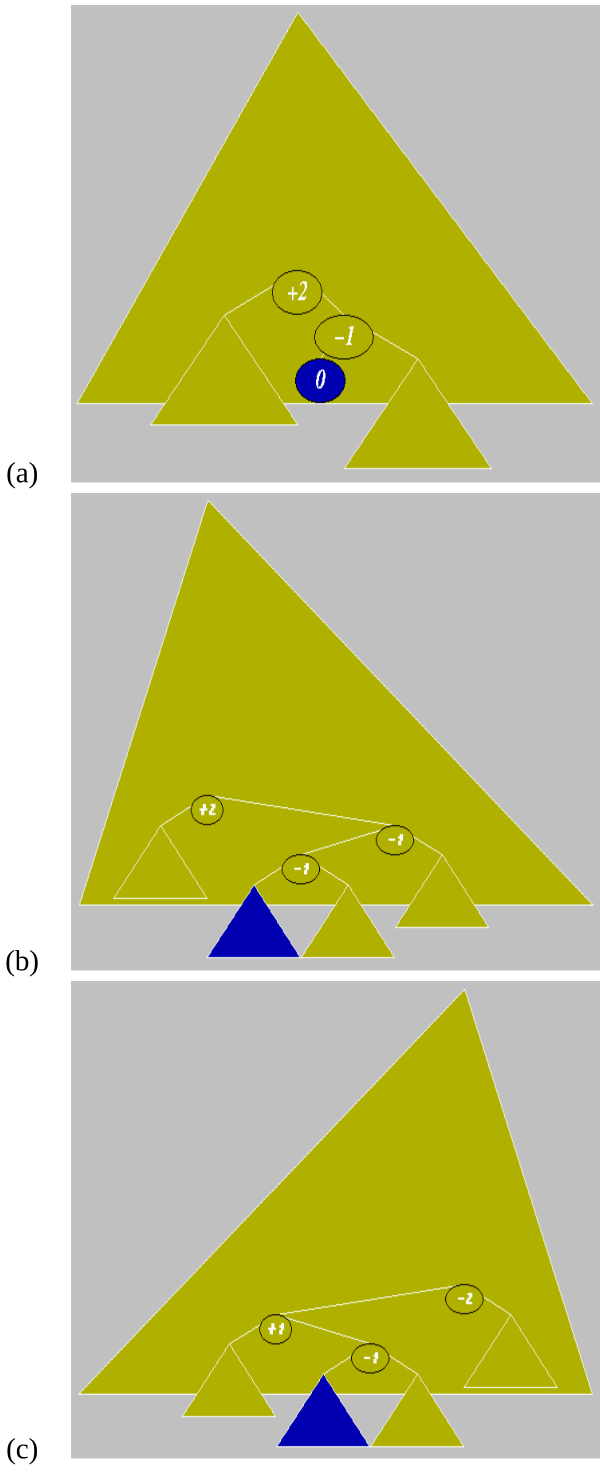

Figure 2. The three cases shown above and their mirror images (with balance numbers negated) all require a double rotation in the AVL tree insertion algorithm.

dent may instead instruct Opsis to imitate the steps for the mirror image of that case — if it was demonstrated earlier — but with the steps mirrored accordingly.

(For example, the student can demonstrate the steps for Figure 2 (a) as shown in Figure 3 and then tell Opsis to generate the steps automatically for the mirror image of that case. The imitation mechanism automatically swaps left and right rotations steps and negates the balance numbers in the update steps.)

Observe that the imitate mirror image mechanism is robust and predictable. Moreover, no inferencing is required. A student who uses this mechanism must invoke it manually and must be aware of the symmetry in the algorithm. In domains with several kinds of symmetry (eg., linear algebra), the user would choose one of several imitate methods (eg., imitate the steps for the image obtained by reflecting the current image along the diagonal, x-axis, y-axis, etc...).

2.3 Imitate Selected State

Using the imitate mirror image mechanism alone, we can reduce the number of double rotation cases that the user has to demonstrate from six to three. By also using the imitate selected state mechanism, we can reduce this number even further to just one case. We describe how this can be done in what follows.

The imitate selected state mechanism requires two steps: (1) the user selects a case by choosing its initial state; and (2) the user moves to another state and invokes the mechanism to perform "similar" steps in this case as is done starting at the selected state.

The two cases need not match exactly (in which case no imitation is required!). Consequently, we use inferencing to determine the parameters of the commands "replayed" from the old case in the new case. These parameters include the particular nodes and fragments of the tree that a command is invoked on (eg., the node whose balance number is to be updated) as well as the results of the command (eg., the new balance number to be stored at that node).

As with other inferencing systems, such as Peridot [10, 9], Opsis may ask the user questions throughout the inferencing process. In particular, it asks the user to confirm or modify the information in update commands (eg., those updating balance numbers in an AVL tree algorithm).

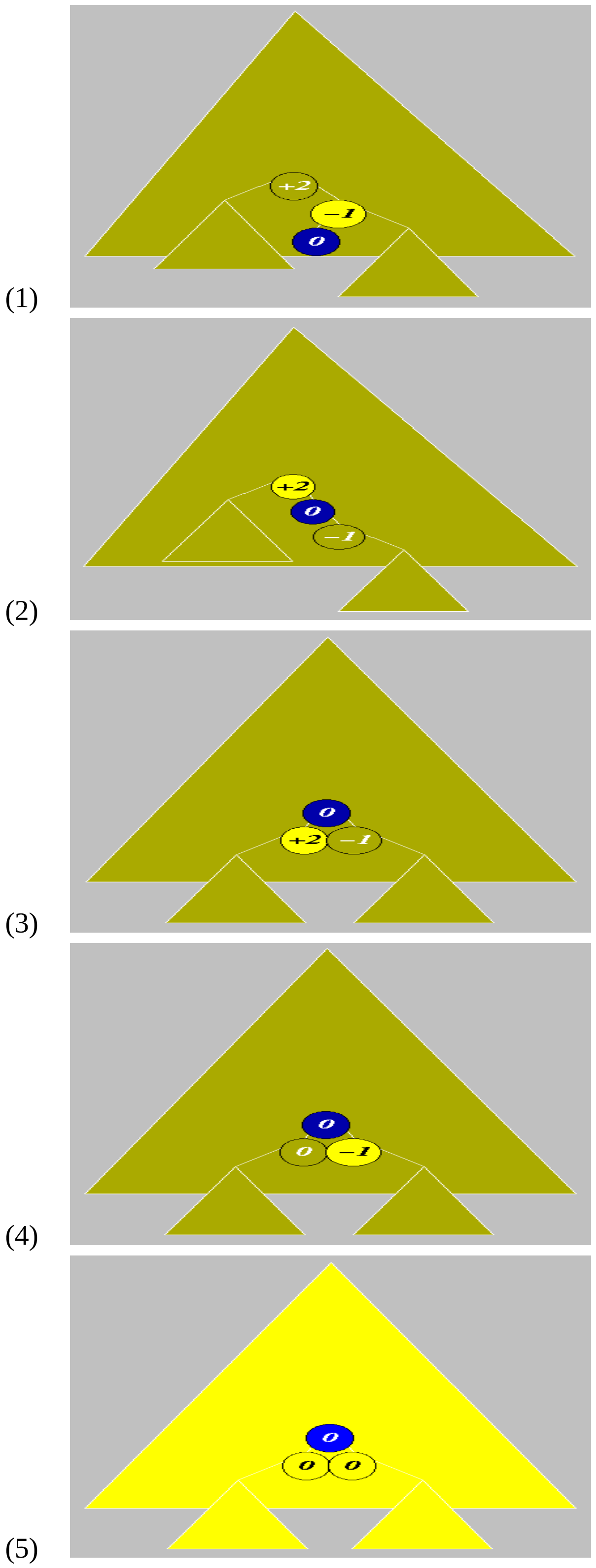

Figure 3. The demonstration done by a user in Opsis to implement the double rotation case illustrated in Figure 1.

(Returning to our example, we can use this imitation mechanism to perform similar steps for the case in Figure 2 (b) as those done for case (a). In this process, the user can tell the system that the update performed in step (4) in Figure 3 that changes -1 to 0 for case (a) should now change -1 to +1 for case (b). Observe that the double rotation for case (a) involves a node with balance 0 before the double rotation that also ends up with balance 0 after, so an update is not required. This is no longer true for case (b) since that node has balance -1 before the double rotation but 0 after. Consequently, the steps generated by this mechanism are not quite correct. The user can simply modify the newly generated steps by adding a step that updates the balance number to 0 after the updates for the other two nodes involved in the double rotation).

Imitating a selected state may not generate exactly the right steps in the new case. For this reason, we may want to modify, insert, or delete command(s) in the middle of some sequence of steps. (For example, we might want to update the balance number to 0 *before* the other two updates after the double rotation.) Such editing changes can also be useful in correcting steps demonstrated by the user manually. To perform such changes, we can again use the imitate selected state mechanism! Modifying command(s) can be done by removing and inserting new commands so it suffices to consider the latter two editing operations.

To insert command(s) in the middle of a command sequence, one performs the following steps: go to the state in which the command(s) are to be inserted (before the command on that state); select that state; remove the current command on that state (so the state is now final); perform the new command(s); and imitate the selected state.

To remove command(s) in the middle of a command sequence, one performs the following steps: go to the state following the last one with a command to be removed; select that state; go to the first state with a command to be removed; remove the command (so the state becomes final); and imitate the selected state.

Using both imitate mirror image and imitate selected state, one way to handle the six double rotation cases for AVL tree insertion is to: (1) demonstrate the steps for case (a); (2) invoke imitate mirror image to generate steps for the mirror image of case (a); (3) select case (a) and invoke imitate selected state to generate steps for case (b) and perform the correction described above; (4) invoke imitate mirror image to generate steps for the mirror image of case (b); (5) select the mirror image of case (b) and invoke imitate selected state to generate steps for case (c) (with no additional correction required this time as the steps imitated include three balance updates); and (6) invoke imitate mirror image to generate steps for the mirror image of case (c).

Unlike the imitate mirror image mechanism, the imitate selected state mechanism uses inferencing and may yield unexpected results. However, this is mitigated by (1) asking the user to confirm or modify some command parameter(s) throughout the process; and (2) allowing the user to correct errors (again using the imitate selected state mechanism). Finally, observe that whereas other systems use inferencing to aid generalization, Opsis uses inferencing to aid imitation.

3 Implementation

3.1 State Model

We assume that the computation is represented as a *state graph*, which is a directed graph whose nodes represent states and whose directed edges represent transitions from one state to another. Computation starts at the *initial state* (which has no predecessors) and ends at one of the *final states* (which have no successors). Each non-final state has a *command* that transforms the state to one of its successors. A cycle in the state graph indicates a loop while a state with more than one successor indicates a conditional command at that state.

Generally speaking, our imitation mechanisms imitate a subgraph of the state graph. This subgraph may include loops and conditionals.

3.2 Imitate Mirror Image

The procedure imitateMirrorImage is shown in Figure 4 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). We assume that states s and a are mirror images of each other. We invoke a command on state a when s has a command leading to some successor(s) but a does not. In that case, we use the command in the source s, modified appropriately for the mirror image, in state a. For each successor s' of s, we determine the newly generated mirror image a' of s' (which is a successor of a) and we then invoke the procedure recursively on s' and a'.

```
// s: source state
// a: add state
// (s is always mirror image of a)
void imitateMirrorImage(s, a) {
  // stopping condition
  if (s.command==null or
      a.command!=null) {
    return;
  }
  // add ``mirrored'' command
  a.command=mirrorCommand(s.command);
  a.command.execute();
  // follow successor(s)
  for (i=1; i<=s.noSucc; i++){
    j=mirrorSucc(s.succ[i]);
    imitateMirrorImage(s.succ[i],
                       a.succ[j]);
  }
}
```

Figure 4. Code for imitateMirrorImage procedure.

Let G be the state graph before invoking imitateMirrorImage on states s and a. Let X be the set of non-final states reachable from s in G (including s if s is not a final state) whose mirror images are final states or do not exist in G. For each state $x \in X$, the algorithm "mirrors" x's command and adds the result to the mirror image $\overline{x}$ of x (after adding $\overline{x}$ if necessary).

To determine the running time, it suffices to consider the number of times a command is added during the traversal. (This follows because the number of successors of any state is bounded by a small constant in Opsis so we can ignore those calls made on successors which return without adding a command since this only requires a constant amount of work.) Since we check that a doesn't already have a command before adding one to a, it follows that we only add a command to a particular state a once. Since the algorithm

[1]The code in Opsis is more complicated because several different states may have exactly the same visual representation.

```
//s: source state in
//   copied state graph
//a: add state in
//   original state graph
void imitateSelectedState(s,a){
  s.visited=true;
  if ( s.command == null or
       a.command != null ) {
    return;
  }
  a.command=s.command;
  determineSelectedFragments(s,a);
  a.command.execute();
  for (i=1; i <= s.noSucc; i++ ) {
    if (not s.succ[i].visited) {
      imitateSelectedState(
        s.succ[i],a.succ[i]);
    }
  }
}
```

Figure 5. Code for imitateSelectedState procedure.

only adds commands to the mirror images of the states in X, it follows that the running time is $O(|X|)$.

3.3 Imitate Selected State

As explained in Section 2.3, the user first selects a state to imitate. This selection actually makes a copy of the part of the state graph that contains states reachable from the selected state (including the selected 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 G and the one copied from the selection G'. This copying ensures that: (1) exactly those states selected will be used in the imitation process (even after manual changes to these states after selection but before imitation); and (2) the imitation process itself will not affect the states being imitated.

The procedure imitateSelectedState is shown in Figure 5 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 s has a command leading to some successor(s) but a does not. 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(s, a) to determine which fragments to select in the tree of the add state before invoking the command from s. (If the command is not applicable, then an error will result and the imitation process will halt.) 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 s' and a'. Since we never visit a state in the copied state graph G' more than once, the algorithm is guaranteed to terminate in time linear in the size of G'.

The determineSelectedFragments is a domain dependent heuristic. Our implementation in Opsis is rather complex and allows students not only to imitate similar cases but also to make non-trivial editing changes to the state graph. We refer the interested reader to [8] for a description of the Opsis implementation of this procedure.

4 Related Work

Our imitation mechanism is most closely related to the editable graphical histories of Chimera [4]. Editable graphical histories depict important events in the history of the application by using a sequence of panels. A user may make changes to the history sequence, in place, by edits to the panels themselves. Any such changes are propagated automatically to the present state. Moreover, the user can also select panels in the history and "replay" the commands verbatim in another context.

However, our imitation technique improves upon the editable graphical histories in several ways. Whereas Chimera maintains a *linear* history, we allow the user to construct a state graph. In this way, our imitation algorithms can imitate not just a simple sequence of commands but a subgraph with conditionals and loops.

Unlike Chimera which replays commands verbatim, our imitation mechanism is more flexible. In particular, we provide more than one way to replay commands in another context. For example, in Opsis we provide two mechanisms, imitate mirror image and imitate selected state. Also, recall that the imitate selected state mechanism asks the user to confirm or change updates to node information so even these commands need not be replayed verbatim. Of course, in other domains (such as linear algebra), there may be many varieties of imitation mechanisms. As far as we know, replaying commands in one of several distinct ways is unique to our approach.

Chimera (and SmallStar [3]) actually provide a macro definition mechanism that does handle conditionals and loops. However, this mechanism requires two steps: a demonstration step followed by a generalization step. The demonstration pass is a simple sequence of commands while the generalization pass allows the addition of conditionals and loops. Once a macro is defined in this way, it can be applied to other cases. Our approach avoids the generalization step and provides a simpler scheme in which both the demonstration and imitation mechanisms allow conditionals and loops.

Finally, we should mention that much of the work on inferencing (such as in Peridot [10, 9]) can also be used in our imitation method. However, it is preferable to provide several varieties of imitate methods rather than fewer more unpredictable ones that rely heavily on inferencing. In this way, the user would make his intent more explicit to the system by selecting the appropriate imitation mechanism.

5 Conclusions and Future Work

In this paper, we have proposed a new mechanism, imitation, which can be used in a programming by demonstration system as an alternative to generalization. By using imitation, a user can tell the system: (1) that a case is similar to another one already demonstrated; and (2) in what way it is similar. In this way, the user can use imitation to generate code for the new case based on the one already demonstrated. Imitation is particularly useful in domains with some symmetry (such as with data structure and linear algebra algorithms).

Since generalization is not required at any point in this process, we alleviate two major problems with existing programming by demonstration systems: (1) we avoid having to use a more abstract graphical vocabulary than the cases themselves (or to resort to using textual annotations); and (2) we don't require the user to abstract and generalize a case so that the demonstration applies to other cases also.

However, imitation does require domain analysis to determine the kinds of imitate procedures that would be useful. Also, we have introduced a potential maintenance problem. Since generalization never occurs, modifications to one case will not automatically propagate to another similar case unless the user explicitly performs another imitate to update that case.

For future research, it would be interesting to explore ways to keep track of imitations that have been performed and warn the user when changes are made to a case that was used to generate code for another case. Perhaps the system can also propagate these changes automatically at the request of the user.

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 Alan Borning, David Notkin, and 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] G. M. Adel'son-Vel'skii and E. M. Landis. An Algorithm for the Organization of Information. *Soviet Math. Doklady*, 3:1259–1262, 1962.

[2] G. A. Curry. Programming by Abstract Demonstration. Technical Report 78-03-02, University of Washington, 1978. PhD thesis.

[3] D. C. Halbert. Programming by Example. Technical Report OSD-T8402, Office Systems Division, Xerox Corporation, 1984. Reprint of Berkeley Computer Science PhD thesis.

[4] D. J. Kurlander. *Graphical Editing by Example*. PhD thesis, Columbia University, July 1993.

[5] H. Lieberman. Seeing What Your Programs are Doing. *International Journal of Man-Machine Studies*, 21:311–331, 1984.

[6] H. Lieberman. An Example Based Environment for Beginning Programmers. *Instructional Science*, 14:277–292, 1986.

[7] A. Michail. Teaching Binary Tree Algorithms through Visual Programming. In *Symposium on Visual Languages*, pages 38–45. IEEE, September 1996.

[8] A. Michail. Visual Programming without Procedures. Technical Report UW-CSE-97-05-02, University of Washington, 1997.

[9] B. A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, 1988.

[10] B. A. Myers and W. Buxton. Creating Highly Interactive and Graphical User Interfaces by Demonstration. *Computer Graphics*, 20(4):249–268, August 1986.

[11] D. C. Smith. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Birkhauser, Verlag, Basel, Stuttgart, 1977. Reprint of 1975 Stanford Computer Science PhD thesis.

[12] D. C. Smith. Pygmalion: An Executable Electronic Blackboard. In A. Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 19–47. MIT Press, 1993.