Meeting the User's Intention in Programming by Demonstration Systems

Michael Sassin
Siemens Rolm Communications
Santa Clara, CA 95052
email: Michael.Sassin@siemensrolm.com

Siegfried Bocionek
Siemens AG
Medical Engineering, GT 7
D-91050 Erlangen, Germany

1 Introduction

A *Programming by Demonstration* or *PbD* system allows its users to construct procedural knowledge or programs on an abstraction level highly above traditional programming languages. The acquired procedural knowledge can be used to automate repetitive tasks and to enable users to concentrate their efforts on the creative part of their work. The major application areas are interactive systems as graphics editors [Maulsby and Witten, 1993; Sassin, 1994], text editors [Mo and Witten, 1993], hypertext systems [Cypher, 1993], spreadsheets [Spenke, 1993], personal assistant agents [Bocionek, 1995], and robotics [Münch *et al.*, 1994].

While the use of classical or visual programming languages require an extensive training to enable users to describe procedures, PbD systems support the acquisition of procedural knowledge for interactive systems *while* users are "manually" solving a task in utilizing the given user interface of such a system. Ideally, the PbD system acquires a new program, e.g. to extract subject lines of email [Cypher, 1993], by *observing* and *analyzing* a sequence of actions performed at the user interface of the interactive system, in this particular case a combination of a mail tool and hypertext system. Since the user knows the handling of this interactive system already, no training would be necessary to acquire a new program.

Nevertheless, a PbD system with such qualities is hard to develop. One of its major problems is that single actions and action sequences can be interpreted in various ways since there is often not enough information about the user's goals or intention available. Thus, the PbD system has to consider many possible hypotheses spanning a large search space.

Although several machine learning algorithms are available that generalize a concept based on one or more examples, PbD researchers have been quite disappointed by the standard ML algorithms [Witten, 1995]. On the one hand, inductive ML algorithms require a large number of examples that cannot be provided by the demonstrating person. On the other hand, deductive algorithms need a strong domain theory that cannot usually be defined for the interactive system beforehand or limits the flexibility of the PbD system by doing so [Sassin and Bocionek, 1993]. Therefore, most PbD applications consist of various pragmatic assumptions, heuristics, and — as we propose in this paper — different ways of user interaction.

Additionally, the ML algorithms are often not suited for the analysis of action sequences and the derivation of procedures with loops and branches [MacDonald, 1991]. New methods have to be developed that enable a PbD system to derive program structures that are compatible with the user's intention. In order to find and verify such programs with the user's intention, new methods of visualization and interaction have to be provided that go beyond just showing program code described in a program language [Myers, 1988] or anticipation of actions [Cypher, 1993; Maulsby and Witten, 1993]. The user wants to know what was derived to really trust the derived program.

Since the environment of a PbD system is interactive by its nature, users are available *during* as well as *after* a demonstration sequence has been completed. Therefore, the derivation power of a PbD system can be enhanced if the user interacts with the system as a *teacher*. For example, often an action can be interpreted in a large number of possible ways in a particular situation. To solve such ambiguities we propose system-initiated dialogs which let the user choose amongst possible alternatives. In our graphics editor ProDeGE+ hypothesis selection panels list possible interpretations sorted by some plausibility criteria and enable the user to verify or change the automatically derived hypothesis according to the user's intention. User assistance is also necessary to derive the correct program structure. Our PbD system visualizes program structures with a *two-dimensional history board* and enables the user to adapt it to the program he or she had in mind by user-initiated interaction.

In the following chapter we will discuss the architecture of our PbD system and in particular our ideas for integrating a variety of interaction mechanisms. The architecture is applied to our PbD graphics editor ProDeGE+ [Sassin, 1994] that will be described in chapter 3. Chapter 4 will summarize our results and outline future research topics.

2 PbD Architecture

2.1 Teacher-Student Metaphor

The development of our PbD system started with the understanding that a lecture of a human teacher consists of more than just a demonstration of the solution of particular problems based on a few examples. A good teacher provides students with further information that enables them to learn the particular task much faster. Typical approaches for this are the *focusing* of the student's attention on a special aspect of the solution and the *explanation* of actions while it is demonstrated. Additionally, the student might want to ask the teacher for clarification to ensure that the learned concept is matching the teacher's intention.

We concluded from this observations that a PbD system should implement this *teacher-student metaphor* to overcome the difficulties of existing PbD systems as well as the limitations of ML algorithms. Our PbD system assumes that the

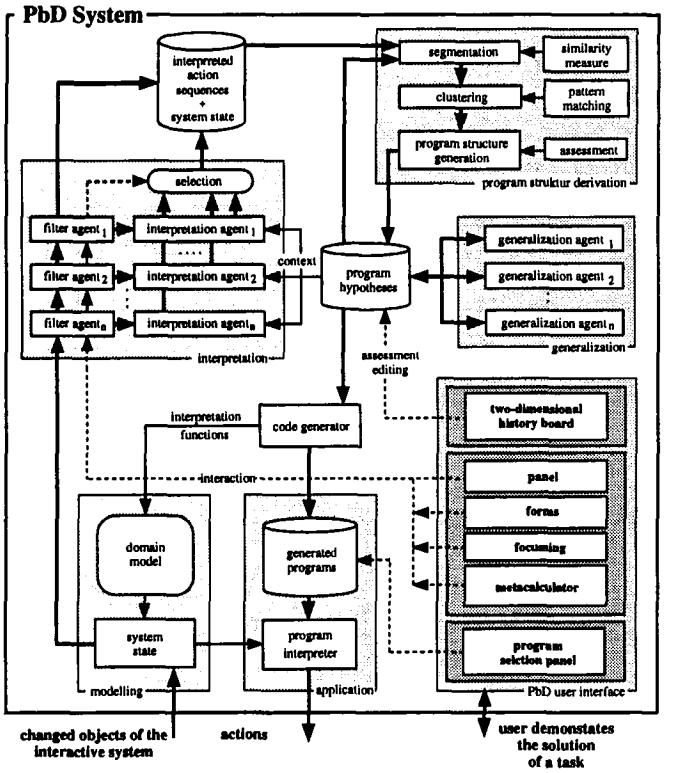

Figure 1: general PbD architecture

user is the "teacher" who demonstrates the intended program to the PbD system which takes the place of the "student".

Since the transfer of the procedural knowledge from the teacher to the student is dependent on both the performance of the teacher and the student, a PbD system cannot simply substitute the lack of the teacher's ability to teach nor is the user able to transfer knowledge if the derivation algorithms of the PbD system are unsuited or insufficient for this challenge. Therefore, we had to develop a PbD system that

- introduces new interaction methods that support user- as well as system-initiated dialogs to enhance automatic derivation algorithms for procedural knowledge,
- combines ML learning algorithms, heuristics, and new interaction methods,
- develops fast and better algorithms for the analysis and interpretation of multiple action sequences, that derive user-intended program structures, and
- creates a general framework that is applicable to several interactive systems as office software.

2.2 Architecture

The PbD system (see fig. 1) supports — with a combination of automatic methods and dialog-based approaches [Bocionek and Sassin, 1993] — the derivation of programs for the automation of complex tasks. Its architecture is derived from the general architecture for PbD [Sassin and Bocionek, 1993]. The user creates a generalized program with loops and branches by demonstrating *several* action sequences that solve one particular task. The begin and the end of each sequence and names for the programs are indicated by the user.

While the user is demonstrating an action sequence the changes of the *system state* is monitored (see fig. 1, bottom, left) and used for the interpretation. The system state is based

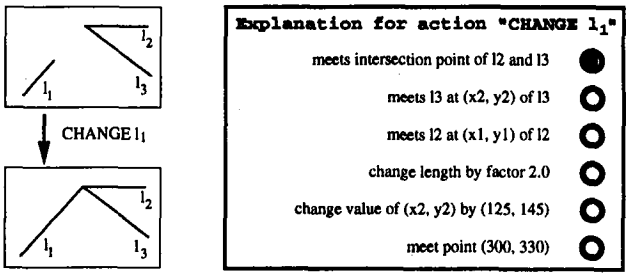

Figure 2: interpretation hypotheses are automatically derived and can be verified and changed by the user

on a *model* that describes the particular interactive system connected with the PbD system. A model consists of *property and relation functions* [Sassin and Bocionek, 1993] that describe the properties and the relations of the objects represented in the interactive system.

After an action is performed the PbD system sends the detected changes of the system state to a *filter agent* (see fig. 1, top, left). Each filter agent selects the *relevant* changes and parts of the system state and sends them to its associated *interpretation agent*. Additional hints of the user as eye-tracking, speech, gestures or simple pointing might help the filter agent to be even more successful in *focusing* this information.

Each interpretation agent infers — based on the context of the action and the information it receives from its filter agent — several *assessed interpretation hypotheses* that are presented on a *panel* (see fig. 2) in the order of assessment. If the selected interpretation is identical with the user-intended interpretation of the action, the user can continue with the demonstration of the action sequence. Otherwise, the user selects the correct interpretation at the panel. In the case that an interpretation is not displayed at the panel, the user can review additional hypotheses or construct an interpretation based on the interaction with *forms* and the *metacalculator* which allows the combination of properties and relations.

The agent concept enables the designer of a PbD system to adapt it to the interactive systems that will be connected to it. Some agents could be designed so that they derive interpretations based on their general knowledge of interactive systems using general inference algorithms, while others might be particularly designed to exploit knowledge about a special interactive system or the behavior of a specific group of users. Since each agent contains a learning component, new knowledge will be acquired during the demonstration sessions so that the agents' performance will enhance over time and reduce the amount of the interactions with the user.

After an action sequence is completely demonstrated, the interpreted action sequence is passed to the generator of the program structure (see fig. 1, top, right). Here, new program structures are derived that generalize the sequences or existing program structures which are based on previously demonstrated action sequences are extended by the new information in the latest action sequence (see fig. 3).

In the first step, the interpreted action sequence is *segmented*. This is done by identifying similar subsequences of the action sequence which indicate that they are caused by the same substructure of a program. The segmentation algorithm is based on *approximate matching* algorithms [Myers and Miller, 1989] that find optimal alignments between two sequences or between a sequence and a regular expression over an arbitrary alphabet. Since the program structure can be interpreted as a finite automaton and every regular expression

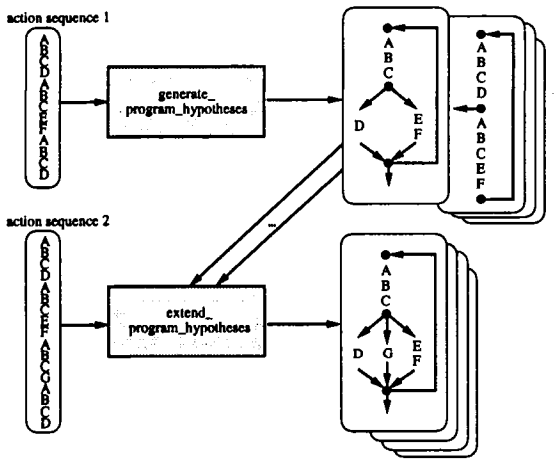

Figure 3: The regularities in the action sequence 1 are used to derive several possible program structures. The action sequence 2 is used to extend, transform, or reject this program structures.

can be transformed in an finite automaton this algorithm is able to find optimal parses between a sequence and a program structure. In the second step, specialized *pattern matching* algorithms will detect the boundaries of repetitive structures. In the third step, the program structures are *constructed*, *transformed*, *extended*, or *rejected* based on the repetitive subsequences that were found in the interpreted action sequences. See more details of the algorithms in [Sassin, 1996].

Since several program structures can be induced, each program structure is assessed by an informational measure related to the minimal description length [Rissanen, 1985] that in average assigns the user-intended structures a high rating. The measure combines several heuristics as *Occam's razor*, the induction principle and probability arguments.

To pinpoint the user-intended structure, the PbD system visualizes the program structure with the highest assessment at the *two-dimensional history board* as shown in fig. 4. It enables users to recognize regular patterns in the action sequence and the structure that is derived from it. Similar parts of the action sequences, represented as icons, are grouped underneath each other if they are part of the same loop. If the user's intention is different from the structure that is actually visualized, the user will be able to create or delete loops and branches, manipulate the start of such structures or add or delete iterations of loops or branches. These changes are initiated by selecting icons, dragging them to different locations at the two-dimensional history board and dropping them there. The interaction is used to select or generate program structures that meet the demonstrated action sequences and the additional interactions at the two-dimensional history board. The program structures that do not fulfill the hints given by the user are eliminated so that this approach guarantees the derivation of the user-intended program structure.

After the correct program structure is derived, the PbD system induces the data flow of the best-assessed program and determines the conditions of loops and branches. Each task is performed by *generalization agents* (see fig. 1, middle, right) that contain specialized inference mechanisms. The data flow is determined by a heuristics that identifies the repetitive use of objects in the action sequence to derive hypotheses. Conditions for branches and loops are derived by ML algorithms as ID3 or PRISM based on positive and negative examples that are extracted from the system states associated with the

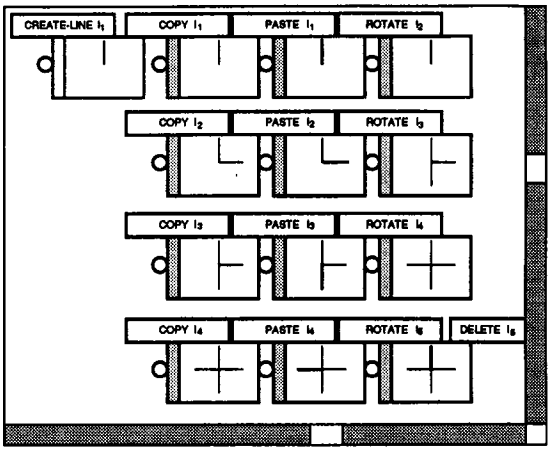

Figure 4: Two-dimensional history board displays a loop that is inferred from the action sequence $s_1 s_2 s_3 s_4$ with s_i:=COPY l_i PASTE ROTATE l_i.

demonstrated action sequences.

Since the ML algorithms often do not have a sufficient number of examples, their result might be incomplete or wrong. Therefore, the user can visualize the inferred data flow and the conditions and adapt them with simple interactions.

3 Application

To verify our ideas we combined our PbD system with a basic graphics editor resulting in the programming by demonstration graphics editor ProDeGE+ and defined an appropriate model for ProDeGE+ that included relations like "`meet`(obj_1, obj_2)" and "`contains`(obj_1, obj_2)"[1] to express the important properties and relations between the graphics objects.

As an example, we chose the following application: Often, users wish to extend two-dimensional objects in the graphics editor to three-dimensional objects. Since normal graphics editors do not include functions, that automatically perform this task, the user has to perform the whole work manually. With ProDeGE+, however, the user is able to acquire the procedural knowledge to solve this task by demonstrating two action sequences (see fig. 5 and 6). After the demonstration and the derivation of the program, the PbD system provides the user with a new functionality at the user interface of the PbD system that automatically performs the task as demonstrated.

Based on the polygon g defined by the lines l_1 to l_7 and the help line h (see fig. 5a), that describe in which way the polygon has to be extended, the user will demonstrate the first action sequence:

In the first step, a copy h' of the help line h is produced by the actions SELECT, COPY, and PASTE. Afterwards, this copy is moved by the action MOVE(h') to the end of the line l_1. The interpretation of this action is `meet-begin-end`(h', l_1) which expresses that the begin of h' has to meet the end of l_1. In the next step, the action CREATE-LINE creates the line l that connects h and h'. This is expressed by the interpretations `meet-begin-end`(l, h) and `meet-end-end`(l, h') which are derived by the interpretation agents so that no additional interactions are necessary.

Afterwards, the extension of the polygon is analogously continued (see fig. 5b). Only in two exceptional cases other

[1] "`meet`(obj_1, obj_2)" indicates that the object obj_1 meets the object obj_2. "`contains`(obj_1, obj_2)" is a predicate that decides if object obj_1 is surrounded by the contour of the object obj_2.

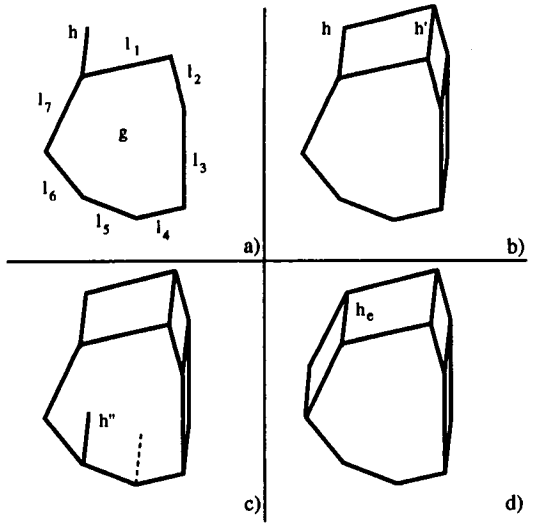

Figure 5: The first action sequence demonstrates the basic program structure for the desired program.

actions are performed:

- The first exception occurs when the end of the connecting line l meets the end of the help line h (see fig. 5d). In this case the demonstration ends.
- The second exception occurs when a copy h'' of h lies inside the polygon (see fig. 5c). In this case, the line h'' will be moved — by the action MOVE — to the end of the line k that h'' meets at its beginning.

The PbD system detects the repetition of the subsequence SELECT, COPY, PASTE, MOVE and generates a loop for it. The alternative handling of the line h'' (two MOVE actions) is mapped onto an branch that is within this loop (see fig. 7).

After the derivation of program structures, the best-assessed one is visualized on the two-dimensional history board (see fig. 8). The icons of the action sequences SELECT, COPY, PASTE, MOVE are grouped underneath each other to indicate that they are part of the same loop. These four actions are followed by the action CREATE-LINE or two MOVE actions that are inducing a branch within the loop. The branch is indicated by dashed lines surrounding the icons that are part of the branch. Additionally, the beginning of each branch is indicated by a number that is in front of the first icon of the branch. Only those branches with the same numbers are mapped to the same branch of the program structure. Since the two MOVE actions infer another nested loop they are grouped underneath each other. In this case the visualization meets the user's intention so that no additional interactions are necessary.

The second action sequence (see fig. 6) demonstrates the handling of another exception. If the polygon that has to be extended is non-convex (see fig. 6a), a line h' will be neither inside nor outside the polygon but cross it (see fig. 6b). In this case, the line h' has to be shortened so that the impression of a hidden line can be provided. Therefore, the user introduces a point p by the action CREATE-POINT that indicates where the two lines are meeting each other (see fig. 6c). This is indicated by the interpretation X(p):=first(k) and Y(p):=second(k). The value k represents the coordinates of the point where the two lines are intersecting. This value is calculated by the relation function `intersect`(h', l) that returns the coordinates of the intersection point of two lines. Afterwards, the line is

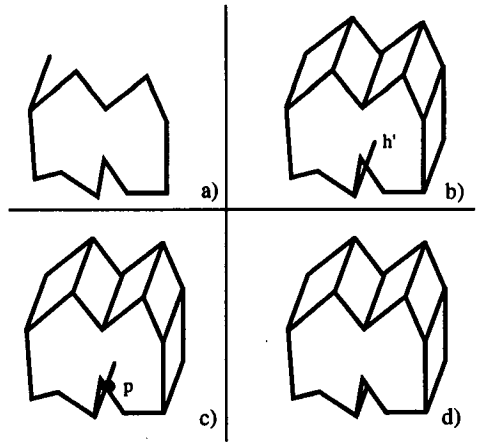

Figure 6: The second action sequence introduces the exception handling for the extension of a non-convex polygon.

program: 3D-extension(ep_1, ep_2)

parameter ep_1: polygon
parameter ep_2: line

$X1$:=UNGROUP(ep_1)
$X2$:= ep_2
$X4$:= *nil*
while $\exists X3$: `meet-begin-begin`($X2, X3$)$\wedge$
  $\neg$`meet`($X4, ep_2$) **do**
  SELECT($X2$)
  COPY
  $X4$:=PASTE
  MOVE($X4$), `meet-begin-end`($X4, X3$)
  if `intersect`($X4, ep_1$) **then**
    / this branch was introduced by 2nd example */*
    $X5$:=`intersect`($X4, ep_1$)
    $X6$:=CREATE-POINT, X($X6$) :=first($X5$),
      Y($X6$) :=second($X5$)
    CHANGE($X4$), `meet-end`($X4, X6$)
    DELETE($X7$)
  elsif $\neg$`contains`($X4, ep_1$) **then**
    $X5$:=CREATE-LINE, `meet-begin-end`($X5, X2$),
      `meet-end-end`($X5, X4$)
  elsif `contains`($X4, ep_1$) **then**
    while $\exists X3$: `meet-begin-begin`($X4, X3$) **do**
      MOVE($X4$), `meet-begin-end`($X4, X3$)
  $X2$:= $X4$

Figure 7: derived program for the extension of a polygon

changed by CHANGE so that the end of h is meeting p. This is indicated by the interpretation `meet-end`(h', p).

After the demonstration of each action sequence, the generalization agents derive generalizations based on the system states of the action sequences and the best-assessed action sequence. While this agents are successful in inducing the correct data flow, they do not find all correct conditions. Therefore, the user has to assist the PbD system deriving the end condition $\neg$`meet`($X4, ep_2$) of the program (see fig. 7) and the condition `intersect`($X4, ep_1$) for the exception introduced by the second action sequence.

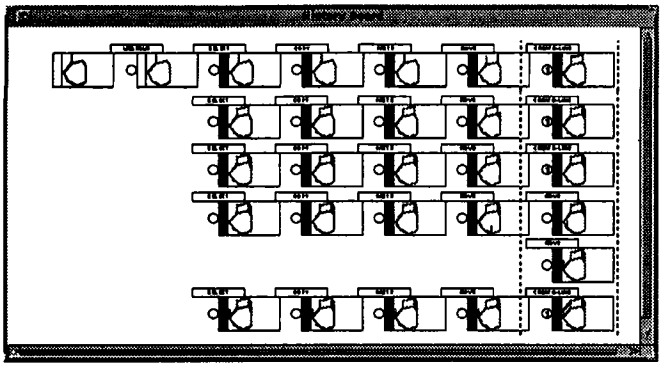

Figure 8: visualization at the 3-dimensional history board of the first action sequence for the extension of a polygon

4 Conclusion and Future Research

Based on Programming by Demonstration in combination with interactions (Dialog-Based Learning) users without special programming skills are enabled to extend the functionality of the graphics editor ProDeGE+. In comparison with other work, this PbD system is more successful in deriving complex programs based on a few action sequences that describe the solution of a task. Both, the automatic and the interactive derivation methods provided by the PbD system enable the user to produce the user-intended programs which can be reused to solve similar tasks from then on.

The new *PbD architecture* provides a general framework for the combination of ML algorithms, automatic derivation methods for *program structures* and *assessment*, and a set of very strong *interaction techniques*. The *agent concept* for the derivation of interpretations and the generalization of program structures supports the adaptation of the PbD system to the interactive system it is combined with, and gives the option to distribute the PbD system on several computers or processors. The generation method for program structures is – unlike those of other PbD systems – the first one that is based on advanced matching algorithms with a sound similarity and assessment measure. Both introduce a strong bias for the system that is not dependent on the individual interactive system but only on the similarity of interpreted action subsequences.

The PbD architecture supports the *teacher-student metaphor* that implements several interaction methods helpful to communicate information and hints between the user and the PbD system. Similar to a human teacher, the user is able to help the PbD system interpreting actions by visualizing the inferred interpretations at a *panel* and selecting the *user-intended interpretation* if it was not selected automatically. The *two-dimensional history board* visualizes the inferred program structure of the program and enables the user to give hints from which the user-intended program structure is derived.

In the future, additional work has to be done to reduce the amount of interaction between the user and ProDeGE+. This can be achieved by enhancing the performance of agents used in the PbD system. To do so, better suited inference and learning components of the agents have to be developed as well as a better exchange of information between agents.

Additionally, this PbD architecture should be extended so that it enables the derivation of helpful programs without or with smaller amount of interactions that define the begin and end of action sequences. This would result in an even easier way of acquiring procedural knowledge that can be used to automate repetitive tasks and extend interactive software.

References

[Bocionek and Sassin, 1993] S. Bocionek and M. Sassin. Dialog-Based Learning (DBL) for Adaptive Interface Agents and Programming-by-Demonstration Systems. Technical Report CMU-CS-93-175, Carnegie Mellon University, July 1993.

[Bocionek, 1995] S. Bocionek. Agent systems that negotiate and learn. *Int. J. Hum.-Comp. Studies*, 42:265–88, 1995.

[Cypher and Halbert, 1993] A. Cypher and D.C. Halbert, editors. *Watch what I do: Programming by demonstration.* MIT Press, 1993.

[Cypher, 1993] A. Cypher. EAGER: Programming repetitive tasks by demonstration. In [Cypher and Halbert, 1993], pages 271–290.

[MacDonald, 1991] B.A. MacDonald. Instructable systems. *Knowledge Acquisition*, (3):381–420, 1991.

[Maulsby and Witten, 1993] D.L. Maulsby and I.H. Witten. Metamouse: An instructable agent for programming by demonstration. In [Cypher and Halbert, 1993], pp 156–82.

[Mo and Witten, 1993] D.H. Mo and I.H. Witten. TELS: Learning text editing tasks from examples. In [Cypher and Halbert, 1993] pages 180–203.

[Münch *et al.*, 1994] S. Münch, J. Kreuziger, M. Kaiser, and R. Dillmann. Robot Programming by Demonstration (RPD) — Using Machine Learning and User Interaction Methods for the Development of Easy and Comfortable Robot Programming Systems. In *Proc. of the ISIR '94, Hannover, Germany*, April 1994.

[Myers and Miller, 1989] E.W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

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

[Rissanen, 1985] J. Rissanen. Minimum description length principle. In E. S. Kotz and N. L. Johnson, editors, *Encycl. of Statistical Sciences*, vol. 5, pages 523–527. 1985.

[Sassin and Bocionek, 1993] M. Sassin and S. Bocionek. Programming by Demonstration (PbD): The Basis for Auto-Customizing Workstation Software. In A. H. Rubenstein and H. Schwärtzel, editors, *Intelligent Workstations for Professionals*, pages 199–220. Springer-Verlag, 1993.

[Sassin, 1994] M. Sassin. Creating user-intended programs with programming by demonstration. In *1994 IEEE/CS Symposium on Visual Languages*, pp 153–60, Oct. 1994.

[Sassin, 1996] M. Sassin. *Architektur, Interaktionsmethoden und Segmentierungsalgorithmen für Programmieren-durch-Vormachen-Systeme. PhD thesis,* University of Karlsruhe, Department of CS, 1996. to appear.

[Spenke, 1993] M. Spenke. *Perplex: Ein graphisch-interaktives System zur Unterstützung der logischen Programmierung.* R. Oldenbourg Verlag, 1993.

[Witten, 1995] I.H. Witten. PbD systems: When will they ever learn? In *Workshop Proc. at the 12th Int. Conf. on Machine Learning*, pages 1–9, July 1995.