Programming by Demonstration: An Inductive Learning Formulation*

Tessa A. Lau and **Daniel S. Weld**
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350
October 7, 1998
{tlau, weld}@cs.washington.edu

ABSTRACT

Although Programming by Demonstration (PBD) has the potential to improve the productivity of unsophisticated users, previous PBD systems have used brittle, heuristic, domain-specific approaches to execution-trace generalization. In this paper we define two application-independent methods for performing generalization that are based on well-understood machine learning technology. TGEN_{VS} uses *version-space generalization*, and $\text{TGEN}_{\text{FOIL}}$ is based on the FOIL *inductive logic programming* algorithm. We analyze each method both theoretically and empirically, arguing that TGEN_{VS} has lower sample complexity, but $\text{TGEN}_{\text{FOIL}}$ can learn a much more interesting class of programs.

Keywords

Programming by Demonstration, machine learning, inductive logic programming, version spaces

INTRODUCTION

Computer users are largely unable to customize mass-produced applications to fit their individual tasks. This problem of end-user customization has been addressed in several ways.

- Adjustable preferences and defaults are simple to use, but limited to those options considered important by the application designer.

- Applications that support macros allow users to record a fixed sequence of actions and later replay this sequence using a shortcut such as a mouse click or a keypress. Macros require little sophistication to create — the user simply records a normal interaction with the application; however, macros have limited utility if the repetitive task has minor variations.

- Scripting languages allow sophisticated users to write programs to control the application. While this type of customization allows a user the most control, it also requires programming experience and knowledge of the potentially complex scripting language and application interface.

Programming by Demonstration (PBD) [4] aims to combine the simple interface of macros with the expressiveness of a scripting language. Like a macro recorder, PBD allows users to construct a program by simply performing actions in the user interface with which they are already familiar. Unlike macros, however, the learned programs can contain control structures such as iteration, conditionals, and abstraction.

A complete PBD system consists of two major parts. The *trace generalizer* constructs a programmatic representation of the user's actions by generalizing from an execution trace to a program capable of producing that trace. The *interaction manager* explains the resulting program to the user and gains authorization before executing the program on the user's behalf.

Example 1: Suppose a user performed the following sequence of actions using an email application:

Goto Message #0, whose sender is Corey
Goto Message #1, whose sender is Oren
Goto Message #2, whose sender is Dan
Forward the current message to Tessa
Goto Message #3, whose sender is Corey
Goto Message #4, whose sender is Dan
Forward the current message to Tessa
Goto Message #5, whose sender is Corey
Goto Message #6, whose sender is Corey

*Our research was greatly improved by discussions with and comments from Greg Badros, Alan Borning, Adam Carlson, William Cohen, Pedro Domingos, Michael Ernst, Oren Etzioni, Neal Lesh, Alon Levy, and Mike Perkowitz. This research was funded by Office of Naval Research Grant N00014-98-1-0147, by National Science Foundation Grant IRI-9303461, and by ARPA / Rome Labs grant F30602-95-1-0024.

Given this execution trace, a PBD trace generalizer might conjecture that the user is "forwarding to Tessa all email messages sent from Dan," and output the following procedure:

```
For each message m in Mailbox:
      Goto(m)
      If Sender(m) is Dan:
            Then Forward(m, Tessa)
```

The PBD system's interaction manager would describe the procedure to the user, accept any user-specified refinements to the program, offer to execute the code, highlight the resulting effects, and implement an undo facility.

This paper focuses on the problem of PBD trace generalization. Previous PBD systems have used brittle, heuristic approaches. In contrast, we define two robust, *application-independent* methods for performing generalization based on well-understood machine learning technology:

- Version-space generalization (TGEN_{VS})
- Inductive logic programming ($\text{TGEN}_{\text{FOIL}}$)

We theoretically and empirically analyze these two implemented systems, offering a rational reconstruction of aspects of the Eager [3] PBD system. TGEN_{VS} can learn with fewer examples, but $\text{TGEN}_{\text{FOIL}}$ can learn a much more interesting class of programs, such as programs with variable-length loop iterations. For example, $\text{TGEN}_{\text{FOIL}}$ quickly learns a program equivalent to the one shown above from the training examples shown.

PRIOR WORK

A complete PBD system is composed of *trace generalizer* and *interaction manager* components. Generalizing from an execution trace to a program involves (among other things) determining the length of an iterative loop, noticing regular changes in each iteration of such a loop, and recognizing conditional constructs. The interaction manager, on the other hand, explains the generalization to the user and obtains authorization for program execution. Although prior PBD systems have made significant contributions to both of these components, our work is restricted to trace generalization so we focus on that aspect in this discussion.

Cypher [4] describes many PBD systems. The Eager [3] system detects and automates repetitive actions in Hypercard applications. Each user action is compared against the action history using a pattern-matching scheme. Once the system detects a repetitive sequence, it highlights its prediction of the next action, and offers to automate the remainder of the sequence. Peridot [15] allows programmers to construct user interfaces by drawing them interactively. It uses a set of condition-action rules to determine when to infer constraints over widget placement and iterations over sequences. Cima [12] learns text editing commands by example; it relies on user hints, rather than multiple examples, to disambiguate possible programs. Chimera [6] generalizes constraints between graphical objects from multiple snapshots and provides a macro-by-example facility for creating macros in a graphical editing domain. Tinker [10] supports programming Lisp by demonstration, but performs no inference, instead relying on the user to disambiguate the examples.

These PBD systems all rely on heuristic rules to describe when a generalization is appropriate. These heuristics make PBD systems (like the rule-based expert systems on which they are modelled) brittle, laborious to construct, and difficult to extend. For instance, Eager is hardcoded to recognize a fixed set of sequences over which a user may iterate, such as the email messages in a mailbox or the days of the week. Peridot uses a set of domain-dependent condition-action rules to infer constraints and loops. Unfortunately, these fragile heuristics must be hand-crafted for each domain. In contrast, we advocate using established machine learning techniques to perform robust trace generalization. As we demonstrate in the remainder of this paper, our approach also yields low sample complexity — TGEN_{VS} and $\text{TGEN}_{\text{FOIL}}$ are capable of generalizing from a small number of training examples.

Others have considered applying domain-independent algorithms to PBD. Witten *et al* [27] identify shortcomings in current machine learning approaches when applied to PBD, such as an inability to take advantage of domain knowledge or user hints. Paynter sketches out a general-purpose PBD framework using machine learning [18]. Nevill-Manning and Witten [16, 17] describe a linear-time algorithm for detecting hierarchical structure in sequences by generalizing a grammar from repeated subsequences in a single example. Their algorithm is elegantly simple, but it is not obvious how to apply background knowledge to bias the generated grammars.

Plan recognition [21, 5] typically requires a large library of possible plans which are checked against user actions. This work may be viewed as inductive learning with a very strong bias (the plan library). Bauer [1] presents algorithms for constructing plan libraries from a corpus of logged user traces and optional action models. Weida and Litman [25] use terminological subsumption with temporal constraint networks to match execution sequences with library plans. Lesh and Etzioni [8, 9] adapt the version space algorithm to perform goal recognition with an implicit plan-library representation which is constructed by reasoning about action preconditions and effects.

Our work is similar to previous machine learning efforts at developing self-customizing software. Schlimmer and Hermens [24] describe a note-taking system that predicts what the user is going to write and facilitates completion; the system works by learning a finite state machine (FSM) modeling note syntax and decision tree classifiers for each FSM state. Maes and Kozierok [11] use nearest-neighbor learning (adjusted by user feedback) to predict the user's next action from the action most recently executed, but in contrast to our work there is no attempt to learn loops.

PBD AS AN INDUCTIVE LEARNING PROBLEM

By watching a user's actions, a PBD system must induce the abstract procedure that the human is executing. In this section we consider two different ways of encoding PBD trace generalization as an inductive learning problem.

- In the *version-space* encoding, each example is an iteration of a loop (*i.e.*, an ordered sequence of actions), and the system generalizes from multiple iterations.
- In the *inductive logic programming* encoding, examples are relational descriptions of individual user actions, and the system learns a Datalog program — an ordered set of first-order rules that predict the next action to be executed.

In the remainder of this section, we elaborate on these encodings and discuss their relative advantages.

Version Space Encoding

Version space algorithms [13] use a general-to-specific ordering among possible hypotheses to maintain a compact representation (the set of maximally specific and the set of maximally general hypotheses) of the *version space* — the set of expressible hypotheses which are consistent with the training data to date. The intuition underlying this algorithm is simple: positive examples result in changes (generalizations) to the set of maximally specific hypotheses while negative examples cause the set of maximally general hypotheses to be refined (made more specific). Since we are only interested in the most specific hypotheses consistent with the training examples, we only need the FIND-S half [14, p. 29] of the full algorithm.

TGEN_{VS} uses version spaces[1] to perform trace generalization by considering each loop iteration to be a positive example, encoded as a feature vector with a column for each command, argument, and (optionally) argument attributes. We adopt a simple conjunctive hypothesis language with "?" wildcards [13]; the semantics of this representation is simple — a hypothesis represents the set of examples (loop iterations) that match the pattern. For example, the hypothesis

Cmd1	Arg1	Cmd2	Arg2
Copy	Subj	Paste	?

matches any two-action sequence whose first command copies the subject into the clipboard and whose second action pastes the clipboard into *some* file. A hypothesis with k wildcards corresponds to a PBD procedure with a k-nested loop. FIND-S requires a general-to-specific order, so we say that hypothesis h_1 is equal to or more general than h_2 if and only if for each conjunct c_i in h_2 the corresponding conjunct in h_1 either is equal to c_i or is the wildcard symbol.

Example 2: Consider the following simple execution trace adapted from the Eager paper [3] in which a horizontal line separates the loop iterations.

Goto message #1, whose sender is "Dan"
Copy its subject to the clipboard
Paste from the clipboard into "File1"
Goto message #2, whose sender is "Tessa"
Copy its subject to the clipboard
Paste from the clipboard into "File1"

In TGEN_{VS} the previous trace is encoded as shown below. We have chosen to represent only a single attribute of Goto's message argument: the message's sender. This choice is critical since an omitted attribute (*e.g.*, length or date) will never be considered during generalization. This choice is independent of the learning algorithm and need only be done once per application; in addition, it's simpler and less restrictive than hand-coding a procedure.

Act1	Arg1	Sender	Act2	Arg2	Act3	Arg3
Goto	1	Dan	Copy	Subj	Paste	File1
Goto	2	Tessa	Copy	Subj	Paste	File1

Using this conjunctive hypothesis language, TGEN_{VS} quickly processes the two examples shown above and returns

Act1	Arg1	Sender	Act2	Arg2	Act3	Arg3
Goto	?	?	Copy	Subj	Paste	File1

as the most specific consistent hypothesis. In other words, TGEN_{VS} predicts that the user is trying to copy the subjects of all messages (regardless of sender or message number) into File1.

Our version space approach has both strengths and weaknesses. On the plus side, the algorithm is simple, fast, and can generalize from two examples. This low sample complexity is due to the very strong bias afforded by a conjunctive hypothesis language.

This strong bias is also responsible for a major limitation. The hypothesis language is incapable of expressing a program that saves messages from Dan in one file and messages from anyone else in another.

Example 3: Given the following training data:

Act1	Arg1	Sender	Act2	Arg2	Act3	Arg3
Goto	1	Dan	Copy	Subj	Paste	File1
Goto	2	Tessa	Copy	Subj	Paste	File2
Goto	3	Dan	Copy	Subj	Paste	File1
Goto	4	Oren	Copy	Subj	Paste	File2

[1] We use Mooney's Common Lisp version-space implementation, available at `ftp://ftp.cs.utexas.edu/pub/mooney/ml-code/`.

the TGEN_{VS} system produces

Act1	Arg1	Sender	Act2	Arg2	Act3	Arg3
Goto	?	?	Copy	Subj	Paste	?

which corresponds to an unintuitive loop in which the user saves each message (regardless of sender) to *all* files. Note that the hypothesis language is incapable of expressing a program that saves messages from Dan in one file and messages from all others in another.

One might sidestep this problem by developing a more expressive hypothesis language with limited disjunction, but there is a more serious problem: it cannot handle variable-length loop iterations, such as the one in example 1. While the feature-vector approach is an improvement over macros in that it handles a restricted class of conditional loops, it breaks down if the individual loop-iterations have a different number of actions, because the one-to-one correspondence between actions in the iterations is lost. To address this problem, we next consider a much more expressive hypothesis language: recursive Datalog programs.

Inductive Logic Programming (FOIL)

An ILP learner, such as FOIL [22], is a first-order extension of a decision rule learner. It takes as input a set of relational training examples classified into those that are examples of a target concept (the positive examples) and those that are not. From these examples the algorithm learns an ordered set of Horn clauses that describe the training examples in terms of the input relations.

The learner employs a sequential covering algorithm which creates a rule to explain a subset of the positive training examples, then removes these examples and relearns on the remainder, until all positive examples have been covered.

Each rule is grown from general to specific. The learner starts out with the empty rule (a Horn clause with the target relation as its head and an empty body), and then adds conjunctive literals to the rule until a stopping criterion is met. At each step, the learner performs a search for the best literal, based on an information gain heuristic. The literals considered in the search include instantiations of relations in the domain, equality over variables in the rule, and negations of any of the above; the best of these literals is added to the rule. Once the rule covers only positive training examples, it is considered complete.

In summary, the algorithm can be viewed as a hill-climbing search over the space of conjuncts of literals, based on an information gain heuristic.

FOIL Formulation of PBD

The encoding of user actions for FOIL differs from the feature vector encoding in four ways:

1. A predicate calculus ontology, instead of feature vectors, encodes data about training examples.

2. Examples correspond to individual actions instead of loop iterations. This circumvents the problem with variable-length loops, and enables a simpler user interface in which the user need not manually identify separate iterations, but it makes the learning problem harder.

3. We encode a simple model of time so the learner can utilize the order in which actions are executed when generalizing. The feature vector encoding cannot express this information.

4. The closed world assumption (CWA) [23] provides negative training examples — if the second command is "Save message to file1," then the second command is *not* everything else. (It's impossible to exploit these negative examples without a notion of time, because the user might eventually execute any action.)

ActionType: act1, act2, ..., actN.
CommandType: goto, copy, paste.
ArgumentType: arg1, arg2, ..., argN.
TextType: subject.
FileType: file1, file2.
MesgType: mesg1, mesg2, ..., mesgM.
SenderType: dan, oren, tessa, corey.

Command(ActionType, CommandType)
Argument(ActionType, ArgumentType)
Before(ActionType, ActionType)
InSeq(MesgType, MesgType)
TextOfArg(ArgumentType, TextType)
FileOfArg(ArgumentType, FileType)
MesgOfArg(ArgumentType, MesgType)
SenderOfMesg(MesgType, SenderType)
CurrentMesg(ActionType, MesgType)
PrevCommand(ActionType, CommandType)

Figure 1: (Email ontology) Types and relations used in describing the PBD learning problem. Target relations are in **bold** and logically redundant relations (added to reduce sample complexity) are in *italics*.

We illustrate our $\text{TGEN}_{\text{FOIL}}$ algorithm in terms of a hypothetical email application. Background information about a user's past interactions with the email application is encoded in terms of a predicate calculus ontology, "Email," whose types and primitive relations are shown in Figure 1. This ontology is based on *actions*, which are the operations performed by the user in the interface (*e.g.*, going to a message, copying a subject to the clipboard, or pasting from the clipboard into a file). Each action is composed of an *command* (*e.g.*, goto, copy, or paste) and an *argument* that represents the argument of that command (*e.g.*, the 57th message in a mailbox, the subject of the current message, File1). Messages are structured objects; *e.g.*, mes-

Action	Command	Arg	Message	Text	File	Sender	CurrentMesg	PrevCommand
a1	goto	arg1	mesg1			dan	null	null
a2	copy	arg2		subject			mesg1	goto
a3	paste	arg3			file1		mesg1	copy
a4	goto	arg4	mesg2			tessa	mesg1	paste
a5	copy	arg2		subject			mesg2	goto
a6	paste	arg3			file1		mesg2	copy
a7	goto	arg6	mesg3			oren	mesg2	paste
a8	copy	arg2		subject			mesg3	goto
a9	paste	arg3			file1		mesg3	copy

Figure 2: Compact display of training examples in the Email ontology. Each line in the table represents a single positive training example. The last two columns are redundant attributes which speed learning.

sages have a sender attribute that describes who sent the message. The `Before` relation encodes background knowledge about the ordering of actions; `Before(`e_i`,` e_j`)` holds if and only if action e_j is the immediate successor of action e_i. The `InSeq` relation expresses background knowledge about the ordering of messages in a sequence and facilitates recognition of loops which iterate over message sequences. A full PBD application would also include sequencing relations for other common sequences (*e.g.*, days of the week).

User actions are converted into a set of literals in this ontology. For example, if the user's first move is to Goto message #1 whose sender is Dan, this action is represented by adding `command(act1, goto)`, `argument(act1, arg1)`, `mesgofarg(arg1, mesg1)`, and `senderofmesg(mesg1, dan)` to the background theory. In addition, the closed world assumption is used to add negative literals, ensuring that (for example) `act1` has only one command and one argument.

TGEN$_{\text{FOIL}}$ learns two ordered sets of Horn clauses which respectively predict the user's next command and the argument to that command.

For example, Figure 2 shows nine positive training examples in a compact form. It represents the first three iterations of a loop in which the user is copying the subject of successive messages (regardless of sender) and pasting them into file1. Given these training examples and their CWA complements, TGEN$_{\text{FOIL}}$ outputs the following program:

```
command(A,goto) :- prevcommand(A,paste).
command(A,copy) :- prevcommand(A,goto).
command(A,paste) :- prevcommand(A,copy).
command(A,goto) :- prevcommand(A,null).
```

These rules allow a PBD system to predict the user's next command as follows: if the command of the last action was a `paste` then she will next execute a `goto`. If she last executed a `goto`, then the next command will be a `copy`, and so on. The fourth rule is only useful for classifying the user's first command and has no predictive value — it could be pruned during postprocessing.[2]

[2]Alternatively, we could modify the FOIL learning algorithm so that it does not attempt to classify `act1`, but simply uses it when classifying subsequent actions.

TGEN$_{\text{FOIL}}$ learns the following rules for `argument` given the same training data:

```
argument(A,R):-prevcommand(A,goto), textofarg(R,T).
argument(A,R):-prevcommand(A,copy), fileofarg(R,F).
argument(A,R):-currentmesg(A,M), prevcommand(A,
               paste), mesgofarg(R,N), inseq(M,N).
argument(A,R):-currentmesg(A,null),
               mesgofarg(R,mesg1).
```

The first rule states that action A has argument R if the previous command was a `goto` and R has *some* associated text T (the learner has exploited the fact that "Subject" is the only known text attribute). The second rule is similar: it says that if the previous command was a `copy` then the argument to the next command will be an argument with a `fileofarg` attribute, *i.e.* `file1`. The third rule predicts the argument to a `goto` command. If the previous command was a `paste` then the argument will be an argument, R, whose message, N, follows directly after (in sequence with) the current message, M. The fourth rule is akin to the last `command` rule above; it is only generated to classify the argument of action `act1`, has no predictive value, and could be pruned.

In this simple example, TGEN$_{\text{FOIL}}$ requires negligible CPU time and only nine positive examples, which corresponds to three iterations of the user's loop.

Conditional Loops

To test the power of our inductive logic programming framework, we show how it is able to learn some loops involving conditional constructs.

The first conditional loop revisits example 3, first described in the section on version spaces. It is similar to the previous task (example 2) of copying all message subjects into a file, but in this case the user copies the subjects of messages from Dan to `File1`, and copies the subjects of all other messages to `File2`. Using the Email ontology, TGEN$_{\text{FOIL}}$ learns Horn clauses, which are equivalent to the following pseudocode, in 4 iterations (12 positive training examples):

```
For each message m in Mailbox:
     Goto(m)
     Copy(Subject)
     If Sender(m) is Dan:
         Paste(File1)
     else:
         Paste(File2)
```

Example 1 (first presented in the introduction) is a conditional, variable length loop where one path through the loop contains two actions, and the other path only contains one action. Using the Email ontology, TGEN$_{\text{FOIL}}$ learns Horn clauses equivalent to the pseudocode presented in the introduction in as few as 9 training examples (seven iterations) and 0.3 CPU seconds on a Pentium 90.

The next section describes several experiments that shed further light on the power of theTGEN$_{\text{VS}}$ and TGEN$_{\text{FOIL}}$ trace generalization methods.

EXPERIMENTAL RESULTS

In this section we report the results of two experiments. First, we compare the speed and sample complexity of TGEN$_{\text{VS}}$ and TGEN$_{\text{FOIL}}$ on a small set of email-domain examples. Next, we measure the effect on TGEN$_{\text{FOIL}}$ sample complexity of several different ontologies for representing the email domain.

Comparison of Techniques

Figure 3 compares TGEN$_{\text{VS}}$ and TGEN$_{\text{FOIL}}$ on the three examples described in the paper. Like Eager, TGEN$_{\text{VS}}$ learns programs only for example 2, since the remaining examples involve conditional constructs. Although TGEN$_{\text{FOIL}}$ requires an additional iteration to correctly learn this program, it is able to learn conditional programs not expressible in TGEN$_{\text{VS}}$. These results show that our machine learning approach does as well as or almost as well as Eager's heuristic pattern-matching-based approach (which required 2 iterations) for example 2.

Ontological Comparison

The choice of an ontology has a very significant impact on learnability. In the course of developing TGEN$_{\text{FOIL}}$ we experimented with a number of different ontologies before converging to the Email ontology. In this section we compare the sample size and CPU times during induction of rules for `argument` in example 2 from the previous section using the following ontologies:

- Email is the standard ontology, defined in the previous section.

- Email– is the same as Email except the redundant `currentmesg` and `prevcommand` relations are removed. Eliminating these relations decreases the branching factor of the space searched by FOIL, but it increases the path length for a solution (and hence increases the length of the resulting rules).

- Email3– adds to Email– a redundant `before3(actionType, actionType)` relation, where `before3` holds for two actions a_i and a_j if and only if $j = i + 3$. Assuming that the iterative loop is composed of exactly three actions, `before3` should allow the learner to induce the desired rules with a shallower search (at the cost of increased branching factor). Our motivation for Email3– is as follows. Since `command` and `argument` are learned independently in Email, one can imagine a PBD system learning the (easier) `command` rules first, and then adding a new length-dependent relation to help speed induction of `argument`.

- The "Combined Action" ontology replaces `command` and `argument` with a new target relation `action(actionType, commandType, argumentType)`, which relates an action to both a command and an argument. It is motivated by the observation that the `command` and `argument` relations in Email– are separate and hence learned independently; perhaps this makes the problem harder.

- The "Big Relation" ontology replaces `command` and `argument` with the target relation `action(actionType, commandType, mesgType, textType, fileType, senderType)`. The ontology was designed as an attempt to reduce the number of conjuncts in each rule. Instead of learning `command` and `argument` in terms of a number of small selector relations, the system is given examples of (and must learn) this single target relation.

Figure 4 shows the results of this experiment. The Email ontology required the fewest training examples

	VS		FOIL	
	# actions	Time	# actions	Time
Example 1	–	–	9	0.3
Example 2	6	0.01	9	0.2
Example 3	–	–	12	0.6

Figure 3: Comparison of TGEN$_{\text{VS}}$ and TGEN$_{\text{FOIL}}$. The second column is the number of actions required to learn the correct program. Examples 2 and 3 contain three actions per iteration, while example 1 has a variable-length loop. CPU times are in seconds on a Pentium 90. TGEN$_{\text{VS}}$ receives each iteration as a single training example. TGEN$_{\text{VS}}$ is not capable of learning correct programs for either examples 1 or 3, both of which involve conditionals.

	# Actions	CPU Time
Email	9	0.1
Email–	30	1.1
Email3–	9	0.1
Combined Action	84	28.2
Big Relation	>99	6918.0

Figure 4: Comparison of sample size and CPU times for different ontologies. The "# Actions" column reports the number of training examples required to learn the correct program; each iteration of the program is three steps long. $\text{TGEN}_{\text{FOIL}}$ was unable to learn the correct program for the Big Relation ontology in fewer than 100 training examples and several hours. Times are reported using Quinlan's FOIL on a Pentium 90 running Linux and are measured in seconds.

to learn, but Email3– was comparable. The loop-independent `currentmesg` and `prevcommand` give as much benefit as `before3` which hints about loop length. $\text{TGEN}_{\text{FOIL}}$ was unable to learn the correct program for the Big Relation ontology in under 100 training examples and several hours. This is unsurprising since the branching factor of the search grows exponentially with the arity of the target predicate; see Pazzani and Kibler [19] for an analysis of FOIL's complexity.

FUTURE WORK

Our work raises a number of research questions. The primary concern for our $\text{TGEN}_{\text{FOIL}}$ system is whether it will scale as the domain becomes more complex and as program complexity increases. We can see several ways to address the issue of scalability: prioritizing literals in order to direct the heuristic search down more promising paths; increasing the search bias by using FOCL [20] to give the learner background information about previous tasks or likely loop lengths; increasing bias by encoding command-argument agreement constraints as a grammar and learning with GRENDEL [2]; or asking the user to disambiguate between several equally likely (to the learner) alternatives.

Other machine learning algorithms may prove better suited to PBD than either version spaces or inductive logic programming. Explanation-based generalization might use models of action preconditions and effects to learn from a single example. A propositional rule learner can learn disjunctive concepts and might eliminate some of TGEN_{VS}'s weaknesses. Since machine learning algorithms are often sensitive to the available attributes, it is important to experiment with alternate ontologies and representations. A propositional learner might be able to handle actions as training examples if given a vocabulary of program templates (this approach has been successfully applied to wrapper induction [7]).

Another open question is how our system should deal with noisy data. Both of our implementations would be confused if the user inserted an irrelevant action or switched the order of two actions in subsequent iterations — even if the ordering is unimportant. Planning-style reasoning with precondition-effect models [26] might alleviate this type of noise. The segmentation problem can be viewed as a different type of noisy data. Throughout this paper we have talked about generalization of pristine traces from which nonrepetitive actions have been pruned, but a PBD system should be capable of identifying the correct subsequence to generalize. One promising approach is to use data mining to learn only nuggets of high predictive value, rather than attempting to classify the entire training set.

Finally, we plan to increase the breadth of our work by incorporating it into a significant application such as Microsoft Office, addressing interaction management, and performing user studies to quantify potential benefit.

CONCLUSION

Previous approaches to Programming by Demonstration have used brittle, domain-dependent heuristics in order to minimize the number of user actions which have to be observed before the system can generate a useful generalization. In contrast, we implement PBD trace generalization as an inductive learning problem, raising the possibility of a robust, domain-independent method for applying PBD to any application. We have developed two prototypes, TGEN_{VS} and $\text{TGEN}_{\text{FOIL}}$, and tested them in an email context motivated by Eager [3]. Both systems learn quickly; TGEN_{VS} has generally lower sample complexity due to a strong conjunctive bias, but $\text{TGEN}_{\text{FOIL}}$ learns variable length conditional loops which frustrate TGEN_{VS}. $\text{TGEN}_{\text{FOIL}}$ learns nontrivial programs with as few as four iterations. We have also measured the effect of the ontology on sample complexity.

REFERENCES

[1] M. Bauer. Acquisition of Abstract Plan Descriptions for Plan Recognition. In *Proc. 15th Nat. Conf. AI*, 1998. To appear.

[2] William W. Cohen. Grammatically biased learning: learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.

[3] Allen Cypher. Eager: Programming repetitive tasks by demonstration. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 205–217. MIT Press, Cambridge, MA, 1993.

[4] Allen Cypher, editor. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993.

[5] H. Kautz. *A Formal Theory Of Plan Recognition.* PhD thesis, University of Rochester, 1987.

[6] David Kurlander. Chimera: Example-Based Graphical Editing. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 270–290. MIT Press, Cambridge, MA, 1993.

[7] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *Proc. 15th Int. Joint Conf. AI*, 1997.

[8] Neal Lesh and Oren Etzioni. A sound and fast goal recognizer. In *Proc. 14th Int. Joint Conf. AI*, pages 1704–1710, 1995.

[9] Neal Lesh and Oren Etzioni. Scaling up goal recognition. In *Proc. 5th Int. Conf. Principles of Knowledge Representation and Reasoning*, pages 178–189, 1996.

[10] Henry Lieberman. Tinker: A Programming by Demonstration System for Beginning Programmers. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 49–64. MIT Press, Cambridge, MA, 1993.

[11] Pattie Maes and Robyn Kozierok. Learning interface agents. In *Proceedings of AAAI-93*, pages 459–465, 1993.

[12] David Maulsby and Ian H. Witten. Cima: An Interactive Concept Learning System for End-User Applications. *Applied Artificial Intelligence*, 11:653–671, 1997.

[13] T. Mitchell. Generalization as search. *J. Artificial Intelligence*, 18:203–226, 1982.

[14] T. Mitchell. *Machine Learning.* McGraw Hill, 1997.

[15] Brad A. Myers. Peridot: Creating User Interfaces by Demonstration. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 125–153. MIT Press, Cambridge, MA, 1993.

[16] C.G. Nevill-Manning and I.H. Witten. Detecting sequential structure. In *Proc. Workshop on Programming by Demonstration, ML'95, Tahoe City*, July 1995.

[17] C.G. Nevill-Manning and I.H. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[18] Gordon W. Paynter. Generalising Programming by Demonstration. In *Proceedings Sixth Australian Conference on Computer-Human Interaction*, pages 344–345, Nov 1996.

[19] M. Pazzani and D. Kibler. The utility of prior knowledge in inductive learning. *Machine Learning*, 9:54–97, 1992.

[20] M. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94, 1997.

[21] M. Pollack. *Inferring domain plans in question-answering.* PhD thesis, University of Pennslyvania, 1986.

[22] J.R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5:239–266, 1997.

[23] R. Reiter. On closed world databases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978.

[24] J. Schlimmer and L. Hermens. Software agents: Completing patterns and constructing user interfaces. *J. Artificial Intelligence Research*, pages 61–89, 1993.

[25] R. Weida and D. Litman. Terminological Reasoning with Constraint Networks and an Application to Plan Recognition. In *Proc. 3rd Int. Conf. Principles of Knowledge Representation and Reasoning*, October 1992.

[26] D. Weld. An introduction to least-commitment planning. *AI Magazine*, pages 27–61, Winter 1994. Available at `ftp://ftp.cs.washington.edu/pub/ai/`.

[27] I.H. Witten, C.G. Nevill-Manning, and D.L. Maulsby. Interacting with learning agents: implications for ml from hci. In *Workshop on Machine Learning meets Human-Computer Interaction, ML'96*, pages 51–58, July 1996.