*Appears in the Papers from the AAAI 2009 Spring Symposium on Agents that Learn from Human Teachers*

# Iteration Learning by Demonstration

**Steven Eker**
Computer Science Laboratory, SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
eker@csl.sri.com

**Thomas J. Lee** and **Melinda Gervasio**
Artificial Intelligence Center, SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
{tomlee,gervasio}@ai.sri.com

**Abstract**

Programming by demonstration (PBD) utilizes a human teacher to train a computer system to perform tasks within that system. This technique effectively improves productivity for many tasks, in many domains. Many tasks are repetitive in nature; these can be learned by PBD by recognizing the repetitions and generalizing these to iterative programs. We present a domain-independent approach to iteration learning by demonstration based on a dataflow model of user actions. We discuss alternatives to our approach and their tradeoffs. In doing so, we identify criteria useful for characterizing iteration learning by demonstration.

## Introduction

Programming by demonstration (PBD) (Cypher 1993; Lieberman 2001) is a simple yet powerful paradigm for automating repetitive tasks. PBD lets human users easily teach a computer system general procedures by simply demonstrating an example of the procedure to be learned. From the user's perspective, performing the actions he would execute anyway makes PBD a natural and appealing approach.

Programming by demonstration is useful for automating tedious, repetitive tasks. Iteration — repeating a sequence of actions over some collection of items — is one of the most common forms of repetition. Indeed, learning repetitive tasks is often cast as that of learning a (single) looping program. In our work on LAPDOG (Gervasio, Lee, and Eker 2008), we address the more general problem of learning programs containing one or more loops. Also, while other approaches (e.g., (Halbert 1993), (Allen et al. 2007)) have relied on user annotations to define the loop boundaries and the collection to be iterated over, LAPDOG employs a pure PBD approach for loop recognition.

LAPDOG has been deployed in several applications: CALO (Cognitive Agent that Learns and Organizes) (CALO 2008), a personal assistant for the electronic desktop; CPOF (Command Post of the Future) (CPOF 2008), a collaboration and visualization tool used by the US military; and WebTAS (WebTAS 2008), a data fusion, visualization, and pattern analysis toolset for large and disparate datasets. While LAPDOG's approach to iteration learning is powerful and, in some ways, is less intrusive than approaches that require user annotations, it has its limitations. Based on our experience with these deployments, we devised alternative methods to loop learning from demonstration.

In this paper, we present LAPDOG's pure PBD approach to iteration learning as well as various alternatives and expound on their individual strengths and limitations. We begin with an overview of the LAPDOG approach to learning from demonstration. We then present LAPDOG's loop recognition and generalization algorithm in detail. We identify useful criteria for characterizing iteration learning and compare our approach to the alternative approaches to iteration learning. Finally, we discuss related and future work and conclude with some recommendations for iteration learning in terms of the application and its users.

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

## Learning from Demonstration

LAPDOG (Learning Assistant Procedures from Demonstration, Observation, and Generalization) is a domain-independent system for programming by demonstration. LAPDOG learns *dataflow* procedures, where outputs of actions generally serve as inputs to succeeding actions. LAPDOG takes as input one or more demonstrations provided by the application, each consisting of a sequence of actions that represents the user's execution of a particular task within that application. LAPDOG outputs a parameterized program that reproduces and generalizes its input. It learns all programs that are consistent with its input demonstration(s), performing a final reification step to produce a single executable program. Execution is performed either directly by the application (e.g., if it supports a scripting language) or through a *task manager* that passes primitive actions to the application for execution, executing nonprimitive actions (such as loops) itself.

An action consists of a unique name and a set of input and output arguments. A simple type system, defined by the application in terms of a few base types defined by LAPDOG, is used to assign a data type to each argument. Notationally, inputs and outputs are prefixed with + and - respectively. For example,

$$Convert(+infile, +format, -outfile)$$

might represent the user converting a document to a particular format, and

$$GetCreationDate(+file, -date)$$

might query the creation date of a file. The set of actions defined by an application is its *action model*.

A demonstration is a sequence of actions representing user activity. In general, a demonstration expresses a dataflow, with the outputs of an action serving as inputs to succeeding actions. This is termed a *support* of the input argument by the output argument.

All arguments in a demonstration are literals (constants). Literals may be primitive or structured. A structured literal may be a list or a set of literals all of the same type, or a *tuple* which maps a set of string keys to literals of disparate types. Structures may be nested arbitrarily deeply. In the learned procedure, an argument may be an *expression* that is either a literal, a variable, or a function invocation that in turn has input arguments.

The learned procedure, like an action, has inputs and outputs which specify its *parameter signature*, which specifies its dataflow. Inputs provide values which may support arguments within the procedure. Outputs provide results of the procedure to the invoker, which is either the user or another procedure.

The learned procedure consists of its signature and a sequence of *statements* which are generalized actions or loops. A loop replaces a repeated sequence of actions with a loop body that generalizes the sequences. Loop formulation is a form of *structure generalization*.

If a literal occurs multiple times in a demonstration, some or all occurrences of it in the learned procedure may be replaced by a single variable. By *variablizing* the demonstration, LAPDOG generalizes the demonstration, rather than simply learning a verbatim repetition of it. Variablization is a form of *parameter generalization*.

### Algorithm Overview

LAPDOG takes as input a demonstration, and outputs a generalization of it in the form of an executable procedure. First, dataflow completion is performed on the demonstration. Parameter and structure generalization are then performed, producing a hypothesis space of alternative procedures consistent with the demonstration. Finally, the hypothesis space is reified, producing a single procedure.

**Dataflow completion** In order for the learned procedure to be executable, each input argument of each action must be supported. LAPDOG performs dataflow completion to assure this condition is met. Because demonstrations may include unobservable actions (e.g., mental operations by the user) (McDaniel 2001), LAPDOG performs depth-bounded forward search over a library of *information-producing actions* to find a sequence that will generate required inputs from known outputs. Examples of information-producing actions are knowledge base queries, information extractors, and string manipulation operations. Information-producing actions have no side effects and may thus be executed to verify the expected results without changing the state of the world. If the search fails, the unsupported value may be either left as a literal or made an input parameter to the learned procedure; the choice is left to the user.

**Generalization** In general, dataflow completion produces a set of viable completions. The generalization step considers each in turn, halting when a valid generalization is discovered. Currently, dataflow completion outputs only the shortest completions, but alternative control strategies could consider all completions or order them according to some other cost metric.

Sometimes alternative parameter generalizations are valid in a situation. For example, if an argument has two different supports, either could be used to support it. Similarly, there may be multiple valid structure generalizations.

Generalization produces an intermediate representation that retains alternative generalizations. Statement arguments are represented as sets of expressions, rather than single expressions. To accommodate alternative structure and dataflow generalizations, the procedure body is represented as a set of statement sequences rather than a single sequence.

Supporting and supported arguments are replaced with a common variable, enforcing codesignation in the learned procedure. An argument that is a function of other arguments (e.g., the first element of a list) is replaced with the corresponding function invocation. An argument with multiple supports is replaced with a set of expressions, one per support.

Repeated patterns of actions are tested to determine if they present a valid iteration over one or more supported list or set arguments; if so, structure generalization is performed, forming a loop statement over the list(s) or set(s) with repeated actions unified into a loop body. This is discussed in detail in the next section.

All parameter and structure generalizations consistent with a given demonstration are retained, forming the space of all valid procedures. Additional demonstrations can then be used to further refine the hypothesis space.

**Reification** Reification selects from the hypotheses computed by generalization to form an executable procedure. First, an element from the set of structurally different procedures is selected. Then each argument in this procedure with a nonsingleton set of alternatives is replaced with a single alternative. The resulting procedure is output.

Reification may exploit application-independent and application-dependent heuristics for program preference, and may involve the user. For example, an application could express a preference for shorter programs, or that certain literals be retained as such rather than variablized.

## Iteration Learning in LAPDOG

We are concerned with loops that iterate over the contents of one or more containers and that might create new containers of the same type. For the purposes of this paper, we will only consider looping over lists although the algorithm can be extended to support other container data types, and looping over sets is also supported in the current implementation. We will illustrate the problem and notation with an example. Consider the sequence of actions:

$A(-[a\ b\ c], -k)$
$C(+a)$
$B(+k, +a, -s)$
$C(+s)$
$B(+k, +b, -t)$
$C(+t)$
$B(+k, +c, -u)$
$C(+u)$
$D(+[s\ t\ u])$

Here the uppercase letters denote actions, the lowercase letters letters denote primitive data values and lists are enclosed in square brackets. We wish to find programs consistent with the demonstration that may contain zero or more non-nested loops. For example:

$A(-L, -X)$
$C(+head(L))$
**for** $Y$ **in** $L$ **building** $H$ **do**
  $B(+X, +Y, -Z)$
  $C(+Z)$
  $H$ **accumulate** $Z$
**od**
$D(+H)$

Here capitalized arguments denote variables. In a program, output positions of actions may only contain variables — no pattern matching is allowed. Input positions in general may contain expressions formed from variables, constants, and operators. For the purposes of this paper, we will only consider variables and the operators $head$ and $tail$, which extract the first and last elements respectively from a list. *For-loops* iterate over one or more lists (which must have the same length if multiple lists are involved) and generate zero or more lists where each generated list accumulates one element at the end of the loop body. The accumulated elements must be given by a variable occurring as an output in the loop body. Generated lists are not available for use until after the end of the loop.

A program $P$ is *consistent* with a demonstration $D$ if $D$ can be produced by executing $P$ under the assumption that each action, when executed in the same context (i.e., with the same previous actions having taken place) and with the same inputs as an occurrence in $D$ will generate the same outputs as seen in that occurrence.

In general, there may be multiple programs consistent with a given demonstration, and we are interested in generating them all. In this way we can apply some post learning selection mechanism, for example by pruning programs that are inconsistent with a second demonstration, to yield the final learned program.

Often when multiple programs are learnable from a given demonstration, some programs will have the same structure but differ in the support for a given value (i.e. they will have a different variable or expression as an input to some action). For example consider the demonstration:

$A(-[a\ b\ c], -k)$
$B(+k, -a)$
$C(+a, +a)$
$C(+a, +b)$
$C(+a, +c)$

Here there is ambiguity about where the $a$ that is passed as the first argument to each of the $C$ actions comes from. In order to represent this situation compactly, we will instead work with *generalized programs* where each input argument to an action, and each container to be looped over is actually a nonempty set of alternatives. For example, the two alternative programs that could be learned from the above example can be represented by the single generalized program:

$A(-L, -X)$
$B(+\{X\}, -Y)$
**for** $Z$ **in** $\{L\}$ **do**
  $C(+\{Y, head(L)\}, +\{Z\})$
**od**

Here all the input arguments are sets and the first argument of the $C$ action is the non-singleton set $\{Y, head(L)\}$. This is because the value $a$, which occurs as the first argument to each of the demonstration instances of $C$, could have come from the list that was the output of the $A$ action (and which was replaced by the variable $L$), or the from the sole output of the $B$ action (which was replaced by the variable $Y$).

The overall structure of the algorithm is that of a search, where there may be alternative choices to be considered.

1. There may be alternatives for supports; these are captured by sets of alternatives for input arguments or containers to be looped over.
2. There may be choices about where and whether to start a loop and how long the loop body should be; these are structural choices and are captured by different generalized programs.

Only the second case requires actual branching. Each branch of the search returns a set of generalized program fragments consistent with a suffix portion of the demonstration sequence. This set may be empty in the case that the branch of the search failed.

The key data structure in this search is a multimap $M$ that maps data values (primitive and containers) to supports. A support for a data value $v$ is an entity that can potentially provide $v$; for example, a fresh variable that has replaced an instance of $v$ on the current branch of the search, or a container data value that contains $v$. Because $M$ will need to be updated on one branch in a search and then restored to its previous state before exploring a parallel branch, it is crucial to efficiency that this multimap is stored as a *fully persistent* data structure where any previous state is available and can be used as a basis for future updates.

The algorithm can be considered as a pair of mutually recursive procedures, LinearGenerator and LoopGenerator, that search for generalizations of the demonstration. LinearGenerator takes an index $i$ into the demonstration and the multimap $M$ of known supports and generates a set of generalized programs that is consistent with the demonstration suffix starting from $i$, and using the supports in $M$. It generates linear code by variablizing demonstration actions, detecting potential loops as well. LoopGenerator takes an index $i$ into the demonstration and the multimap $M$ of known supports together with a loop body length $b$ and a number of iterations $n$ and generates a set of generalized programs that

are consistent with the demonstration suffix starting from $i$, and using the supports in $M$, where the next $n \times b$ actions are explained by a loop over one or more lists of length $n$ that has a loop body of length $b$.

The algorithm starts with a call to LinearGenerator with index 0 and an empty multimap of supports.

## Variablization

In creating a program action from a demonstration action $\alpha$, each output value $u$ is replaced by a fresh variable, which is then added to the multimap $M$ as a support for $u$ in later actions. If $u$ happens to be a list, then $u$ itself is recorded as a support for both its first and last elements.

Each input value $v$ is replaced by a set of *support expressions* which is computed recursively. The set of support expressions for $v$ is the union of the sets of support expressions for each support $s$ that $v$ has in $M$.

In the base case, $s$ is a variable and the resulting set of support expressions is the singleton set containing just $s$ itself. If $s$ is a container $c$ that contains $v$, we recursively compute the set $S$ of support expressions for $c$ and for each $e \in S$, we generate a new expression by using $f(e)$ where $f$ is the appropriate accessor function for obtaining $v$ from $c$.

When an input value $v$ to a demonstration action $\alpha$ has no supports, the attempt to variablize $\alpha$ fails.

## LinearGenerator

LinearGenerator starts with an empty set $G$ of generalized program fragments and considers demonstration actions in order, starting at index $i$. When an action $\alpha$ is considered, LinearGenerator attempts to variablize it as explained above. If variablization fails then all branches of the search that required linear code from index i up to and including the failed index are dead and LinearGenerator returns the set $G$ of generalized program fragments accumulated so far.

In variablizing an action $\alpha$, every time we form a support expression $head(e)$ for input value $v$, there is the possibility that instead of accessing $v$ using the head operation, $\alpha$ could be the first action that accesses the loop variable in a loop that iterates over the list given by support expression $e$. In this case we say $\alpha$ is the *anchor* of the loop. Let $l$ be a list that supported $v$. To account for the possible iteration over $l$ we insert the length of $l$ into a set $I$ of iteration counts for loops anchored by $\alpha$. We do not record $l$ itself as it is possible that $\alpha$ could be part of a loop that is iterating over multiple lists (of the same length). It is also possible that the loop body actually started before $\alpha$.

To detect potential loops anchored at $\alpha$ we consider each iteration count $n \in I$ in turn, and try alternative loop body lengths, $b$, and loop body start indices, $j$, looking for a pair $(j, b)$ where the sequence of action names in the demonstration fragment $[j, j + b - 1]$ that corresponds to the first iteration is equal to sequence of action names in the demonstration fragments $[j + k.b, j + k.b + b - 1]$ for each $k \in \{1, \ldots, n-1\}$ which correspond to the subsequent $n-1$ iterations. Such a pair $(j, b)$ describes a putative list loop of $n$ iterations.

For each such pair $(j, b)$ we call LoopGenerator on $(j, M_j, b, n)$ where $M_j$ is the state of the support multiset $M$ just before the variablization of the demonstration action at index $j$. This previous state of $M$ is available because $M$ is implemented by a fully persistent data structure. LoopGenerator will return the sets of generalized program fragments starting with a list loop of body length $b$ with $n$ iterations that can be constructed, continuing from index $j$ in the demonstration, using supports in $M_j$. Since the same combination of arguments to LoopGenerator can potentially be found from multiple anchors, a set of already tried argument combinations is maintained to avoid redundant work.

The generalized program fragments returned by LoopGenerator are each grafted on to the current generalized program fragment for actions up to but not including index $j$ and the resulting generalized program fragments are added to the set $G$.

After all the putative loops anchored at $\alpha$ have been explored, LinearGenerator moves on to the next action. After the last demonstration action had been processed, the set $G$ of generalized program fragments accumulated so far is returned.

## LoopGenerator

LoopGenerator tries to generate a loop with a body of length $b$ that iterates over one or more lists of length $n$, starting with the action at index $i$ and using the supports from multiset $M$. This occurs in two phases.

In the first phase, the putative loop body is constructed by variablizing the first $b$ actions starting at index $i$. Finding supports for inputs and variablizing outputs and adding supports to $M$ works much like it does in LinearGenerator. Because supports will be local to this iteration of the loop, a reference to the original state of $M$ is kept for use after the end of the loop. Each input value $v$ that is supported as the first element of a list $l$ of length $n$ causes $l$ to be recorded as a potential list to be looped over and a loop variable is created for it. This loop variable then becomes an alternative support for $v$. For example, suppose we have the demonstration:

$$A(-[a\ b\ c], -[a\ b\ d])$$
$$B(+a, +a, -s)$$
$$B(+b, +a, -t)$$
$$B(+c, +a, -u)$$

Here the first action will have already been variablized, say to:

$$A(-K, -L)$$

in LinearGenerator and the first $B$ action will trigger a call to LoopGenerator to look for a loop with 3 iterations and body length of 1. LoopGenerator will variablize the first occurrence of input value $a$ to $\{head(J), head(K)\}$. This use of the first elements of lists variablized by $J$ and $K$ means they are both potential lists to be looped over, and so fresh variables $X$ and $Y$ are created to iterate over $J$ and $K$ respectively. These variables then become supports for $a$. A fresh variable $Z$ is created to variablize the output argument of $B$ so the after the first phase, the variablization of the first $B$ action looks like:

$$B(\{head(J), head(K), X, Y\},$$
$$\{head(J), head(K), X, Y\}, Z)$$

In the second phase, the putative loop body is executed for the remaining $n-1$ iterations, with inconsistent support expressions for inputs being pruned.

Variables are bound to values at the start of each iteration. Each loop variable receives its value from the list that it is iterating over. Each action in the loop body is then compared to the corresponding action for the current iteration in the demonstration. Alternative support expressions for input values in the loop body whose evaluations disagree with the actual values found in the demonstration are pruned away. If an input loses all of its support expressions, failure occurs and an empty set of generalized program fragments is returned. Output values from the demonstration are used to update the value bindings for this iteration. In our example, the sets of support expressions for the inputs arguments of $B$ are pruned, leaving:

$$B(\{X\}, \{head(J), head(K)\}, Z)$$

Also each output occurring in the loop body is potentially a list that is generated by the loop and is accumulated in a fresh variable for later use.

If the loop body survives the second phase, the loop variables are checked, and those that no longer appear in a support expression for some action input are themselves pruned. If all the loop variables are pruned we again have failure. Otherwise we form a loop statement with the surviving loop variables iterating over their lists, using the pruned body. Thus, in our example, we end up with a for-loop:

**for** $X$ **in** $\{J\}$ **building** $L$ **do**
  $B(\{X\}, \{head(J), head(K)\}, Z)$
  $L$ **accumulate** $Z$
**od**

Finally LoopGenerator calls LinearGenerator, passing it the pre-loop state of the support multiset $M$, to generate a set of generalized program fragments for the rest of the demonstration. Each of these generalized program fragments are grafted, in turn, onto the end of the newly constructed loop to obtain the set of generalized program fragments that LoopGenerator will return.

### Extensions

The basic algorithm can be extended to handle other container types and other operators. The current implementation supports sets and tuples, as well as operators to access the elements of tuples, and to construct fixed length lists from their elements. Also supported are input arguments to an action that are designated as *ungeneralizable* by the action model. An ungeneralizable argument is passed though unchanged, remaining a constant in the resulting program.

Loops that iterate over sets present an additional problem in that the order in which the elements of a set are accessed is undefined. This limits set-based for-loops to iterating over a single set since otherwise we would have an undefined correlation between multiple loop variables. Also we cannot accumulate lists in a set-based for-loop since otherwise we would be adding order information that we do not have. We can however accumulate sets.

Detection of a putative set-based for-loop is triggered by the use of any member $e$ of a set $s$. The code for checking the correspondence of action names between the first and subsequent iterations also collects a list of the input arguments that mirror the argument $e$ in the remaining iterations. This list serves as early check on the feasibility of a set-based for-loop (i.e. each element of $s \setminus \{e\}$ must occur exactly once in this list) and defines an order on the elements of $s \setminus \{e\}$ to be used in the execution based pruning phase after the loop body as been constructed.

## Alternative Approaches

Based on informal observations in deployed applications, iteration learning in LAPDOG has proven useful and efficient. However, it suffers from the following restrictions to its applicability:

- A container $C$ containing all elements iterated over to be defined and explicit in the demonstration is required.
- $|C|$ bodies must be demonstrated.
- If $C$ is ordered, prospective loop bodies are required to be demonstrated in that order.

We discuss these limitations and various techniques for addressing them. In doing so, we provide a framework for characterizing iteration learning in PBD systems.

**Demonstration versus specification** LAPDOG uses a pure PBD approach to iteration learning, inferring loops strictly from the demonstration, with no guidance and no loop specification by the user, and no user actions other than the natural workflow. This has advantages and disadvantages. Our experience indicates that PBD is intuitive to a wide range of users, and leads to early acceptance of machine learning and creative exploration of learning opportunities. One reason for this is that the user trains the learner by following their normal workflow, which is natural and unobtrusive. Also, demonstration is often more appealing to nonprogrammers and less technically-oriented users. Regardless of the type of user, it can be disruptive to force the user to switch to 'programmer mentality' while performing their tasks.

On the other hand, pure PBD can be unwieldly to use, requiring long or multiple demonstrations to make intent clear; specification can be concise and less error-prone than demonstration. Sometimes the intention of the user is known only to the user; this 'hidden state' can be difficult to expose (McDaniel 2001), leading to incorrect generalizations or the inability to generalize.

The continuum of pure demonstration to pure specification provides a useful criterion for characterizing various facets of iteration learning and their efficacy for a particular application and user base. Very loosely speaking, nonprogrammers may be more comfortable towards the pure demonstration end of the spectrum. Experienced programmers may be more comfortable with specification, due to its similarity with conventional programming.

**Container identification** Container identification addresses the question 'What is to be iterated over?'. Pure PBD requires that $C$ be explicit in the demonstration; depending on the application, the user might have to contrive

this. For example, a user might demonstrate a set of operations on all files within a directory display. If this display is open at the start of the demonstration, there is no action to support the directory. The contrivance might be to reopen the directory, making it explicit in the demonstration.

Rather than require a demonstrated $C$, a learner might infer $C$ from the state of the application. For example, if there is only one file with a particular name within a file system, and that file is mentioned in a prospective loop body, the learner could infer that $C$ is the directory it resides in. If the file name occurs in multiple places, other file names within the demonstration may serve to disambiguate $C$. Finally, if this is insufficient, providing multiple examples may do so.

The problem of container identification can be obviated by relaxing the requirement that the iteration be over the contents of a container. Instead, loops are detected based on finding a pattern of repeated actions, with little or no regard for the data values of those actions. This permits the discovery of other kinds of loops, such as counting loops and those that terminate based on an arbitrary condition.

Though powerful and general, we feel that such an approach has significant technical challenges. If any pattern of repeated actions is generalized to a loop, there is significant likelihood of false positives for loop detection and therefore incorrect generalizations. Furthermore, since the collection of elements is enumerated in the demonstration but not otherwise identified, it remains an issue as to how these elements are to be determined when the learned program is executed. One possibility is to require the user to enumerate them. Another is to attempt to infer at learning time a relationship among the elements that is expressible as information-producing actions. For example, given the availability of the information-producing action $\mathit{ChildrenOf}(+\mathit{Parent}, -\mathit{Children})$ the demonstration

$$\begin{array}{l} A(-charles) \\ B(+william) \\ B(+harry) \end{array}$$

would generalize to

$$\begin{array}{l} A(-\mathit{Parent}) \\ \mathit{ChildrenOf}(+\mathit{Parent}, -\mathit{Children}) \\ \textbf{for } E \textbf{ in } \mathit{Children} \textbf{ do} \\ \quad B(+E) \\ \textbf{od} \end{array}$$

though $[william, harry]$ is explicit neither in the demonstration nor the application environment. We plan to investigate this approach in future work.

Finally, the application could simply ask the user to identify $C$. This could be initiated by the learner when an action pattern indicates a potential loop. It could also be part of a more specification-driven approach to iteration learning, in which the user indicates the intention of generating a loop, with the attendant tradeoffs of specification versus demonstration.

**Order-constrained iteration** If the iteration container $C$ is ordered, the repeated actions forming the loop body must be executed in this order, or no loop will be discovered by LAPDOG. For tasks that are order-constrained, it is important for the learned program to respect ordering. Unfortunately, producing a demonstration in the correct order can be inconvenient for the user.

The application can address this by using unordered (set) containers wherever possible; however, this poses further difficulties if some actions require an ordered container while others do not. An unordered container may not support an ordered container, even if they contain the same elements. Hence they will not be codesignated with the same variable. This can necessitate the user specifying the contents of both the ordered and unordered containers when the learned task is executed. LAPDOG can address these difficulties to some extent by exploiting information-producing actions of the form $\mathit{Sort}(+C, -C')$, each of which orders a set or reorders a list $C$ in a useful way, for orders that are known *a priori*.

**Partial iteration demonstration** In LAPDOG, all steps in the iteration must be explicit in the demonstration; this is burdensome and error-prone if $|C|$ is large. Indeed, any task that is difficult to demonstrate correctly is problematic for PBD systems. Ideally, the system itself can help reduce demonstration errors.

A technique employed by users of (CPOF 2008) is to first populate a container from which to learn the iteration with artificial or partial data (manipulation of data containers is particularly natural in this application). The demonstration iterates over this container, rather than the (presumably larger) container that would be used in the actual task. This container typically contains two elements, but may contain any number greater than one, and may be a subset of $C$. Making $|C|$ small effectively relieves the user of demonstrating many repetitions of the loop body. However, it introduces new problems by causing a detour from the desired workflow of the user. It takes the user out of pure PBD, requiring that the artificial container be managed at learning time, and requiring care at execution time so that the desired container is provided. Also, if an iteration is only partially demonstrated, the state of the application will not, in general, be in the same state as if the iteration were fully demonstrated. This may pose a problem for any learning that happens subsequent to the iteration, as well as for the user whose later work may depend upon a correct execution state.

A proactive learner that monitors the user continuously might offer to complete the iteration for the user, as in Eager (Cypher 1991). When the (partial) iteration is detected, the application asks the user if it may execute the loop body for the remaining elements of $C$. Similarly, a passive learner could be directed to complete the iteration by the user after a small number of demonstrated iterations.

**Separate demonstration of loop body** For large $|C|$ or long loop bodies, requiring even two demonstrations of the loop body might be onerous for the user. LAPDOG supports iteration learning from a single demonstration of the loop body as follows. First, an auxiliary program comprising the loop body alone is demonstrated and learned. Then, it is demonstrated once for each element of $C$.

This presumes that the learner and the application support program composition by allowing a previously learned program to be executed in a subsequent demonstration and invoked from the learned program. To date, all LAPDOG applications do so. It also presumes to some extent that executing learned programs is reasonably simple; otherwise this technique may not be effective in saving time and reducing the cognitive load of the user.

While this technique does rely on demonstration rather than specification, the requirement that an auxiliary subprogram be learned can be disruptive to the user's workflow, and violates the naturalness of PBD. On the other hand, users comfortable with programming are likely to be comfortable with this technique, as it is a form of modular programming and task decomposition.

The user must still demonstrate $|C|$ loop bodies. For large $|C|$ it may be useful to combine this technique with partial iteration to minimize training time spent by the user.

A variant of this technique may be used to completely obviate any demonstration of repetition. The demonstration of the iteration is replaced with direct specification by the user of the container to be iterated over.

**Iteration without learning** Finally, we point out that in some applications it may be acceptable to defer iteration decisions to task execution time. The idea is that any learned task that accepts some parameter $E$ may be invoked iteratively over a container designated by the user, drawing $E$ from that container.

The user need not anticipate the need for iteration. This is particularly appealing for tasks which, in exceptional cases only, need to operate on lists or sets rather than single instances. It is also advantageous in applications with highly modular tasks which are freely composed by the user during workflows.

The chief disadvantage is that the knowledge that a particular iteration is expected by a task is not captured when that task is learned. This pushes the burden onto the task manager — it must adapt task execution at runtime, turning noniterative tasks into iterations and distinguish valid iterations from runtime errors. Specifically, when a task is invoked with an argument that is a container with elements of type $T$, and the corresponding parameter in the learned program is of type $T$, the task manager wraps a loop around the invocation of the task which iterates over the container and collects outputs into appropriate containers. If the parameter type is inconsistent with the element type, an error is issued.

## Future Work

We plan to implement a new loop-finding algorithm which is based upon finding repeated patterns of actions without additional constraints on their arguments being consistent with demonstrated container values. This will enable learning iteration of forms other than iteration over a container, such as those that are terminated by an arbitrary condition.

In the various deployed LAPDOG applications, we plan to implement extensions to current loop-finding techniques that involve the user in ways other than providing demonstrations. Within these applications, we will conduct user studies to determine the combination of techniques that is most effective for a range of iteration problems and user skills and styles.

## Related Work

Loop learning has always been a primary focus of PBD. Approaches have varied in the amount of user involvement, the number of examples required for learning, the background knowledge used, and the representation of the learned procedure. For example, SmallStar (Halbert 1993) supports loop learning through direct manipulation, letting users teach loops by recording one iteration and then editing in control structures. PLOW (Allen et al. 2007), a PBD system for information extraction from Web services, also learns loops from single iterations but does so by utilizing the structure of the Web page and annotations derived from user utterances during demonstration. By verbalizing his action as he highlights the collection to be looped over and the items to be extracted from the first element, the user provides PLOW with enough information to generalize the actions to the rest of the collection. In contrast, LAPDOG learns loops purely by demonstration, requiring no additional input from the user regarding the loop body or the collection to be looped over.

Many PBD systems learn explicit loops. For example, Eager (Cypher 1991) tried to detect iterative patterns as the user interacted with a HyperCard application and after seeing two or three iterations, it generalized the actions into a loop and offered to complete the execution of the loop for the user. Familiar (Paynter and Witten 2001) continually observed the users actions to find repeated patterns and started generalizing after two iterations, similarly offering to complete the remaining iterations for the user. It was one of the earliest systems to be able to handle noise — specifically in the form of extraneous actions at the start of some iterations. SmartEdit (Lau et al. 2001), a PBD system embedded with a text application, took a similar proactive approach to loop learning, backed by a different machine learning methodology. It also supported learning nested loops by providing users with UI tools for indicating the start and end of a nested loop within a demonstration. These systems are particularly useful for providing in-situ automation help. In contrast, LAPDOG is designed for learning long-lived procedures and, as such, focuses on generalizing the entire demonstration for future executions rather than predicting loops for the purpose of assisting with the current execution.

Other approaches have learned loops within alternative program representations. For example, Metamouse (Maulsby and Witten 1993) learned production rules that effectively simulated a variety of control structures, with loops essentially represented as recursive production rules. Sheepdog (Lau et al. 2004) focused on the structural generalization problem and learned (unparameterized) procedures including conditional branches and loops through the application of alignment techniques over a set of demonstration traces. Meanwhile, WIT (Yaman and Oates 2007) treated the generalization task as one of model merging and relied on grammatical inference techniques to identify and merge the observed individual iterations into a loop. In contrast, LAPDOG learns standard procedures, which lends it-

self more naturally to display meant for human consumption.

## Conclusions

We present a systematic method of iteration learning based on the LAPDOG dataflow model of actions and a strict approach to PBD requiring no user involvement other than producing the demonstration. Our method provides the following capabilities:

- learns iteration over both lists and sets;
- produces all valid generalizations of the demonstration, including both loop and non-loop variants for iterative demonstrations;
- supports single example and multiexample learning;
- permits arbitrarily structured data values, and learns corresponding compositions of data accessors;
- identifies loops that are intermixed with noniterative action sequences without the need for user involvement;
- learns multiple sequential (though not nested) loops;
- learns parallel iteration over multiple lists (but not sets);
- recognizes iterative computations of values within a loop that are required later in the demonstration (perhaps by a subsequent iteration), and generates a generalization that computes them.

We identify and discuss various tradeoffs in terms of LAPDOG's approach. In doing so we provide a framework for characterizing iteration learning in PBD systems.

LAPDOG covers a fairly narrow region in the space of possible iteration learners. While it has proven to be effective in deployed applications, an application-independent learner should provide multiple techniques to the application in order to support a range of users in automating a variety of repetitive tasks.

## Acknowledgments

We thank Carl Angiolillo, Matt Gaston, and Karen Myers for their contribution to many of the alternative approaches to iteration learning presented here.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-07-D-0185/0004. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), or the Air Force Research Laboratory (AFRL).

## References

Allen, J.; Chambers, C.; Ferguson, G.; Galescu, L.; Jung, H.; Swift, M.; and Taysom, W. 2007. Plow: A collaborative task learning agent. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence*. AAAI Press.

CALO. 2008. CALO - A Cognitive Agent that Learns and Organizes. http://www.caloproject.sri.com.

CPOF. 2008. Command Post of the Future (CPOF). http://www.darpa.gov/sto/strategic/cpof.html.

Cypher, A. 1991. Eager: Programming repetitive tasks by demonstration. In *Watch What I Do: Programming by Demonstration*. MIT Press. 33–39.

Cypher, A., ed. 1993. *Watch What I Do: Programming by Demonstration*. MIT Press.

Gervasio, M.; Lee, T.; and Eker, S. 2008. Learning email procedures for the desktop. In *Proceedings of the AAAI Workshop on Enhanced Messaging*. AAAI Press.

Halbert, D. C. 1993. Smallstar: programming by demonstration in the desktop metaphor. In *Watch What I Do: Programming by Demonstration*. MIT Press. 103–123.

Lau, T.; Wolfram, S.; Domingos, P.; and Weld, D. 2001. Learning repetitive text-editing procedures with smartedit. In *Watch What I Do: Programming by Demonstration*. Morgan Kaufmann. 209–226.

Lau, T.; Bergman, L.; Castelli, V.; and Oblinger, D. 2004. Sheepdog: learning procedures for technical support. In *Proceedings of the 2004 International Conference on Intelligent User Interfaces*. ACM.

Lieberman, H., ed. 2001. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann.

Maulsby, D., and Witten, I. 1993. Metamouse: an instructible agent for programming by demonstration. In *Watch What I Do: Programming by Demonstration*. MIT Press. 155–181.

McDaniel, R. 2001. Demonstrating the hidden features that make an application work. In *Watch What I Do: Programming by Demonstration*. Morgan Kaufmann. 163–174.

Paynter, G., and Witten, I. 2001. Domain-independent programming by demonstration in existing applications. In *Watch What I Do: Programming by Demonstration*. Morgan Kaufmann. 297–320.

WebTAS. 2008. WebTAS. http://www.webtas.com.

Yaman, F., and Oates, T. 2007. Workflow inference: What to do with single example and no semantics. In *Proceedings of the AAAI Workshop on Acquiring Planning Knowledge via Demonstration*. AAAI Press.