RC23768 (W0511-013) November 2, 2005
Computer Science

IBM Research Report

Augmentation-Based Learning: Combining Observations and User Edits for Programming-by-Demonstration

Daniel Oblinger, Vittorio Castelli, Lawrence Bergman
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Augmentation-Based Learning

Combining Observations and User Edits for Programming-by-Demonstration

Daniel Oblinger*
University Of Maryland
A.V. Williams Building
College Park, MD 20742
oblio@pobox.com

Vittorio Castelli IBM T.J.
Watson Res. Ctr
P.O. Box 218
Yorktown Heights, NY 10598
vittorio@us.ibm.com

Lawrence Bergman IBM
T.J. Watson Res. Ctr
19 Skyline Drive
Hawthorne, NY 10532
bergmanl@us.ibm.com

ABSTRACT

In this paper we introduce a new approach to Programming-by-Demonstration in which the user is allowed to explicitly edit the procedure model produced by the learning algorithm while demonstrating the task. We describe a new algorithm, Augmentation-Based Learning, that supports this approach by considering both demonstrations and edits as constraints on the hypothesis space, and resolving conflicts in favor of edits.

1. INTRODUCTION

Programming-by-demonstration (PBD) is a technique for rapidly specifying program operations and logic through the use of demonstration examples. It is generally considered to be a more intuitive and simpler form of specification than traditional programming. The downside of PBD is that the programmer has reduced control over the program content. In traditional programming, the programmer explicitly specifies the objects and operations on them. When using a PBD system, on the other hand, an author typically specifies classes of objects implicitly by providing one or more instances of the class, and infers operations or program control flow from a small set of examples. Typically, learning algorithms are employed to infer user intent. It is impossible for such algorithms to make correct guesses in all cases, particularly with the small set of examples used in PBD.

PBD systems deal with this serious shortcoming in several ways. First, they may allow the user to explicitly specify semantics either during or after the demonstration, for example, by presenting the user with a set of hypotheses from which to select the one most appropriate to each particular situation, as, for example, in Metamouse [4, Chapter 7] or SmartEdit [8].

*This work was done while the author was working at IBM TJ Watson.

Another possibility is manual editing of the program representation, or procedure model, produced by the PBD system. PBD systems typically produce a textual or graphical representation of the program. Some PBD systems, like Chimera [4, Chapter12] allow the author to edit the program representation as a post-processing step. The main appeals of editing over other forms of interaction with a PBD learning algorithm described in Section 4 are that the burden imposed on the user is smaller and that the control of the user over the learning algorithm is more direct, comprehensive, and precise than for all other forms of feedback.

Editing can be used to exert control over program content for a variety of reasons. First, it is common to have spurious actions in the demonstration that need to removed from the model. Second, the demonstration may contain a step that is not sufficiently general and that needs to be replaced. For example, the initial demonstration might have used a shortcut button that is not available to all application users. In this case, replacing the shortcut with a more generic navigation sequence makes the procedure more generally usable. Third, a portion of a procedure may have been demonstrated out-of-sequence, for example, the author may prematurely demonstrate some part of a procedure, and rather than start over, simply demonstrate the missing steps, and then reorder those steps later. Fourth, in some cases, the learning algorithm will produce an overly complex model that the user can readily simplify. Fifth, the author may edit the procedure model to alter its structure, for example, to move an action out of a conditional. Finally, portions of a procedure may become out-of-date due to changes in the application interface. In this case, those portions may need to be replaced by updated sequences. Desirable editing operations include deleting, duplicating, and moving sets of steps; replacing steps with additional demonstrations; and altering the semantics of individual steps (e.g., changing the arguments of a conditional).

The ability to interleave editing operations with procedure demonstrations is a powerful combination, since these two mechanisms provide complementary capabilities. However, interleaving editing with by-demonstration programming is problematic. A serious difficulty arises because an edited procedure representation and the demonstrations recorded before that edit need not be compatible. When additional demonstrations are received after editing, it is not clear how to ensure that the edits are retained, while incorporating new inputs. We note that this problem is very similar to the round-trip problem in model-based program develop-

ment [11]. Without solving the round-trip problem, edits to the output of an automated process will be lost the next time that process is re-run.

Our solution is a novel incremental learning algorithm, Augmentation-Based Learning (ABL), whose hypothesis space is the transitive closure of a set of functions (called *augmentations*) given the existing procedure structure. Thus, when users edit the procedure, they also impose a constraint on the hypothesis space of the learner. Consequently, our solution allows interleaving of both types of input and ensures that manual user edits are always respected by subsequent updates of the procedure caused by new observations..

The rest of the paper is organized as follows: Section 2 contains a formal characterization of ABL. Section 3 explores two typical use cases for interleaved demonstration and editing, and is followed by a discussion and conclusions.

2. AUGMENTATION-BASED LEARNING

2.1 Overview

This section is divided in two main parts. In the first, we formally describe the data obtained from procedure demonstration as particular types of sequences, and then define the notion of *aligning* these sequences with a procedure's structure. In the second part we describe the Augmentation-Based Learning (ABL) algorithm itself.

ABL is unique among incremental learning algorithms in two ways: (1) When presented with a sequence of new training samples, the learning algorithm produces a new procedure structure that is the transitive closure of a set of structure manipulation functions, called augmentations, *given the previous procedure structure.* As a consequence, a user that manually edits a procedure is actually editing the hypothesis space considered by the learning algorithm (this is our solution to the round trip problem). (2) ABL updates procedure structures by selecting from a flexible class of procedure transformation operators called augmentations. These allow for many complex transformation like rolling up a repeating sequence into a loop, or reorganizing linear steps into a flow with embedded conditionals. At the same time augmentations are restricted so that an execution sequence consistent with a procedure will remain consistent with all augmentations of that procedure. This ensures that the learning algorithm's restructurings do not undo paths introduce either by editing or by earlier demonstrations.

2.2 Preliminaries

Procedure model and procedure demonstrations

ABL represents a procedure as a decorated graph $(\mathcal{S}, \mathcal{E}, \mathcal{P})$. A *procedure step* $s \in \mathcal{S}$ is a pair (n, α), where n is a node in the graph and α, the action skeleton, is a generalized, parameterized, and variabilized action that, evaluated in a particular context, yields a completely specified, executable action. Depending on the context, a step produces actions that differ at most by the actual values of parameters. Different procedure steps are in general allowed to yield identical actions. A *directed edge* $e \in \mathcal{E}$ represent sequential ordering of two steps. Each edge in $\mathcal{E}$ has an associated *predicate* $p \in \mathcal{P}$ that is evaluated during playback and denotes whether the edge can be followed. Whenever a node has a single outgoing edge, the associated predicate is always `true`. If there is more than one outgoing edge, one and only one of the associated predicates must evaluate to `true` for any context.

We call *procedure structure* or *uninstantiated procedure* a directed graph $(\mathcal{N}, \mathcal{E})$ where each node $n \in \mathcal{N}$ are associated with a steps, and the edges $\mathcal{E}$ do not have associated predicates. We say that a procedure model is an *instantiated* version of its procedure structure.

Inducing a procedure model involves identifying a procedure structure, constructing a step for each node, and inferring the predicates associated with the edges. Models are induced from collections (called training sets and denoted by $\mathbf{T}$) of procedure demonstrations. A *procedure demonstration*, or *trace* $\mathbf{t}$, is a sequence of *state-action pairs* (SAPs) $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)$ observed while a user performed the task. A trace could be the recording of a complete or of a partial demonstration of the task. A *subtrace* is any contiguous subsequence of a trace. A *state* $\mathbf{x}$ is a representation of the content of the GUI just prior to the execution of the action. The content of the GUI is summarized by extracting from each widgets the values of the attributes that can change as a result of interactions between the user and the application. A state and can also contain information otherwise known to the user, such as the user name. An *action* $\mathbf{y}$ is a representation of a user's interaction with one or more applications, and must be specified to a detail sufficient to support automated execution.

The the state is used during induction to provide the context for the induction and evaluation of edge predicates, and during playback to provide the bulk of the context used to instantiate an executable action $\mathbf{y}$ from an action skeleton α. For example, the action skeleton `Select the first item in the 'Package Explorer' list` is instantiated to `Select project1` when the state information shows that the first item in the list is called 'project1'. Additional information in this context is obtained during run time from previous states and actions. Consider, for example, a procedure to configure in Eclipse a project selected by the user: action skeleta would contain the name of the project as a variable, and during execution actions are instantiated by substituting the variable with the actual name of the project.

Before talking about induction, we need two definitions:

DEFINITION 1. *The* **alignment** *of a subtrace* $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)$ *with a sequence of nodes* $n_1, \ldots, n_n$ *in* $(\mathcal{N}, \mathcal{E})$ *is a one-to-one correspondence that maps* $(\mathbf{x}_i, \mathbf{y}_i)$ *to* n_i*. We say that the subtrace is* aligned *with the path, or, more simply, with the procedure structure.*

Therefore an alignment is just a labeling of the SAPs in a trace that uses steps as labels. Some alignments are meaningful, namely, they make the sequence of SAPs correspond to a sequence of steps that could have generated the observed actions. This idea is captured by the definition of consistency.

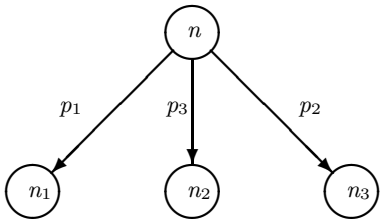
DEFINITION 2. *A subtrace* $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)$ *is* **consistent** *with a model if*

- *the subtrace is aligned to a sequence of nodes* $n_1, \ldots, n_n$*, where* n_i *corresponds to step* s_i*;*
- *Sequentially instantiating the action skeleta* $\alpha_1, \ldots, \alpha_n$ *(where* α_i *is the action skeleton of* s_i*) using the states* $\mathbf{x}_1, \ldots, \mathbf{x}_n$ *yields the sequence of actions* $\mathbf{y}_1, \ldots, \mathbf{y}_n$*;*
- *there is a directed edge from* n_{i-1} *to* n_i *and the associated predicate* p_i *is* `true` *when evaluated using* $\mathbf{x}_i$*;*

- *if the* $(\mathbf{x}_1, \mathbf{y}_1)$ *is the first SAP in a full procedure demonstration,* s_1 *must be an initial step, namely, a step without incoming edges.*

We say that *a model is consistent* with a collection of subtraces if each subtrace is consistent with the model.

We introduce the *generalization problem* as the problem of inducing a procedure model given a procedure structure and an alignment of the training data to that structure. We need a bit of notation to discuss solutions to the generalization problem. For each node $n \in \mathcal{N}$, let $\mathcal{E}(n) \subseteq \mathcal{E}$ be the set of outgoing edges, let $\mathcal{N}(n)$ be the corresponding destination nodes, and let $\mathcal{P}(n) \subseteq \mathcal{P}$ be the corresponding predicates. For example:



In the picture, $\mathcal{N}(n) = \{n_1, n_2, n_3\}$ and $\mathcal{P}(n) = \{p_1, p_2, p_3\}$.

We now propose a sufficient characterization to solutions to the generalization problem.

PROPOSITION 1. **Solution to the generalization problem.** *A procedure model* $(\mathcal{S}, \mathcal{E}, \mathcal{P})$ *is a solution to the generalization problem if it satisfies the following conditions:*

a. *each step in* $\mathcal{S}$ *is induced using only SAPs aligned with the corresponding node;*

b. *consider the collection* $\{(SAP_i^1, SAP_i^2)\} = \mathbf{T}(n)$ *of length-2 subtraces where* SAP_i^1 *aligned with node* n *and* SAP_i^2 *with a node in* $\mathcal{N}(n)$*. The predicates in* $\mathcal{P}(n)$ *are induced using only the states from* $SAP_i^2 \in \mathbf{T}(n)$*.*

c. *the model* $(\mathcal{S}, \mathcal{E}, \mathcal{P})$ *is consistent with all the traces in* $\mathbf{T}$*.*

In simple terms, first induce each step using the aligned SAPs. Then induce the predicates using the SAPs aligned with the destination nodes. If the resulting procedure is consistent with the traces, it is a solution to the problem. Therefore, if a procedure structure and a good alignment is provided, the problem of constructing a procedure model is reduced to that of instantiating steps and inferring predicates. Many existing PBD systems solve only the generalization problem, and the user must provide, either directly (e.g., by labeling action) or indirectly (via feedback mechanisms), the structure and the alignment.

An important contribution of this work is the ability to learn from multiple demonstrations where the alignment between the steps of the demonstrations is not given. When the alignment is not given, the learning algorithm must build a procedure structure using a collection of traces and align these traces to the procedure structure before solving the generalization problem. We call this the *alignment problem.* Clearly, solving the alignment problem alone does not yield a procedure model, and the learner must solve the combined alignment and generalization problem. We now define sufficient properties of a solution to the combined alignment and the generalization problem.

PROPOSITION 2. **Solution to the combined alignment and generalization problem.** *Given a training set* $\mathbf{T}$*, a solution to the alignment and generalization problem consists of a procedure model* $(\mathcal{S}, \mathcal{E}, \mathcal{P})$ *with structure* $(\mathcal{N}, \mathcal{E})$*, and of an alignment of the training set to the structure, such that*

a. *each SAP in* $\mathbf{T}$ *is aligned with one and only one node in* $\mathcal{N}$*; therefore, the alignment is deterministic;*

b. *for each node in* $\mathcal{N}$ *there is at least one aligned SAP in* $\mathbf{T}$*; hence, data is available to instantiate each step;*

c. *for each edge* $e_{i,j}$ *(from* n_i *to* n_j*) there must be at least two SAPs,* sap_t *and* sap_{t+1} *immediately following* SAP_t*, in a trace of* $\mathbf{T}$ *such that* sap_t *is aligned with* n_i *and* sap_{t+1} *is aligned with* n_j*; thus, data is available to infer predicate* $p_{i,j}$*;*

d. *the model* $(\mathcal{S}, \mathcal{E}, \mathcal{P})$ *is consistent with all the traces in* $\mathbf{T}$*.*

As described in Section 4, many existing PBD systems solve constrained versions of the combined alignment-and-generalization problem, by restricting the class of produced procedure structures, for example, by restricting attention to sequence of steps with embedded fixed-length loops, or to procedures consisting of an outer loop with a known or unknown number of steps. The Sheepdog system [6] uses the SimIOHMM learning algorithm [12], which solves the general combined alignment-and-generalization problem.

In the next section we describe an incremental learning algorithms that solves the combined alignment-generalization problem and additionally accommodates the interleaved user-editing operations described in the introduction.

2.3 The ABL Algorithm

Editing Operations

A distinguishing characteristic of ABL is the ability to accept user edits intermixed with actual demonstrations. We formally define an editing operation as follows.

DEFINITION 3. *An* **editing operation** *is a manual transformation of a procedure structure* $(\mathcal{N}, \mathcal{E})$ *into a different procedure structure* $(\mathcal{N}', \mathcal{E}')$ *consisting of adding, copying, or deleting nodes and edges.*

User edits are not constrained to be consistent with pre-existing demonstrations (think of reversing the order of two steps: the resulting model is inconsistent with the data used to create the original model). Therefore, a learning algorithm that derives procedure structure from demonstrated sequences might inadvertently "undo" user edits (in this example, the learning algorithm would try to restore the initial ordering of the steps). In other words, user edits and demonstrations impose constraints on the structure of a procedure model that are potentially in conflict with each other. We resolve conflicts by requiring that user edits take precedence over pre-existing demonstrations.

Before talking about the solution to the alignment and generalization problem with the edits, we need to provide a few definitions. First, by convention we define that SAPs aligned with a node before an editing operation are also aligned with the same node after the editing operation. When a copy operation on a node occurs, SAPs aligned with the original node are also aligned with its copy. Let ϵ_i be the ith user edit, let $\mathbf{T}_i$ be the collection of (complete and partial) traces observed before ϵ_i, and let $\mathbf{T}_i^{\epsilon}$ be the collection

of maximal portions of demonstrations in $\mathbf{T}_i$ that are consistent with the procedure model produced by ϵ_i (namely, if $\tilde{\mathbf{t}} \in \mathbf{T}_i^{\epsilon}$, and $\tilde{\mathbf{t}} \subseteq \mathbf{t}$, any subset of $\mathbf{t}$ strictly containing $\tilde{\mathbf{t}}$ is not consistent with the model).

We now define sufficient properties of a solution to the alignment and generalization problem with edits.

PROPOSITION 3. **Solution to the alignment and generalization problem with edits.** *A procedure model $(\mathcal{S}, \mathcal{E}, \mathcal{P})$ induced using sequences of SAPs intermixed with edits $\epsilon_1, \ldots, \epsilon_h$ is a solution to the alignment and generalization problem with edits if*

a. *is consistent with $\mathbf{T}_h^{\epsilon}$;*

b. *is consistent with the training data observed after ϵ_h.*

By requiring consistency with $\mathbf{T}_h^{\epsilon}$ and not with the entire part of the dataset observed before ϵ_h, we ensure that the learning algorithm does not use the prior demonstration data to "undo" the user edits.

Augmentations

We do not know of any previous algorithm that solves the alignment and generalization problem with edits. We have therefore developed ABL, a real-time incremental learning algorithm that solves this problem by updating a procedure model every time a new SAP is observed using an *augmentation*. An augmentation is a transformation that modifies the structure of a procedure only through adding nodes an edges, but not in any other way. It is this restriction, formalized in the definition below, that ensures that structures produced by the learning algorithm will always retain step transitions previously specified by the user through editing or demonstration. We say that a SAP is associated with the augmentation used by the learning when the SAP is observed.

DEFINITION 4. *An* **augmentation** *associated to a SAP σ is a transformation from a procedure structure $(\mathcal{N}, \mathcal{E})$ to a procedure structure $(\mathcal{N}', \mathcal{E}')$ satisfying $\mathcal{N} \subseteq \mathcal{N}'$ and $\mathcal{E} \subseteq \mathcal{E}'$.*

Hence, the new procedure structure contains all the nodes and edges of the old structure.

We ask that the SAP σ associated with the augmentation be aligned with the new structure; we also require that, if the augmentation yields a new node n, σ be aligned with n.

Since augmentations retain the original procedure structure embedded in the new procedure structure, we ensure that editing operations are not lost.

PROPOSITION 4. *ABL solves the alignment and generalization problem with edits by incrementally updating the procedure structure using only augmentations, aligning new SAPs to existing nodes or to newly created nodes, and solving the generalization problem.*

Consider the model $(\mathcal{S}_h, \mathcal{E}_h, \mathcal{P}_h)$ obtained by applying ϵ_h. This model is consistent with $\mathbf{T}_h^{\epsilon}$ by definition. For each state-action pair σ_i observed after ϵ_h, ABL produces, if possible, a model that contains all the steps in $\mathcal{S}_h$, all the edges in $\mathcal{E}_h$, and those predicates that ensure that the demonstrations in $\mathbf{T}_h^{\epsilon}$ can be aligned with the new model. Additionally, the new model must contain steps, edges, and predicates that ensure the alignment of σ_i.

Procedure Representation

In order to support user edits, the procedure model produced by ABL must yield a representation that is easily understood by a human being. We assume that an advanced user who is willing to manually edit a procedure model is at least somewhat familiar with a programming or scripting language, and therefore we require ABL to produce models that are easily represented as simple programs. For the purpose of this paper, the language for human-readable procedure representations is defined by the following grammar:

```
model := block
block := string blockBody
blockBody := (step | branch | loop | block)+
loop := while predicate block
loop := foreach <item> in collection block
branch := if predicate block
            (elseif predicate block)*
            [else block]
branch := select-one-path block block*
```

The actual forms of the predicates and of the steps are described in the respective induction sections. `<item>` is an identifier, `collection` is a specification of a collection of objects in the GUI, and `select-one-path` is a non-deterministic operator stating that one of the blocks, but the decision of which one to execute is left to the user.

This choice of representation affects the set of allowed augmentations and editing operations. Specifically, we restrict editing to the following set of operations: removing steps, branches, loops, or blocks; cutting-and-pasting steps, branches, loops, or blocks; copying-and-pasting steps, branches, loops, or blocks. We also only allow augmentations belonging to one of the following broad categories: adding a step at the end of a non-completed block; creating a branch step; terminating a branch; adding a path to a branch; creating a loop. It is easy to see that these manipulations transform a string from the described language into another string from the same language.

ABL induction of procedure structure

Consider first the case in which no edits are allowed, and ABL must incrementally solve the combined alignment and generalization problem. It is often possible to find several different augmentations of the current model that produce a new model consistent with both the past observations and the new observation. It is also possible that no such augmentation exists, and therefore the model becomes inconsistent with the data.

ABL maintains a collection $\mathcal{M}$ of procedure models consistent with the observed demonstrations (hence, the training traces are aligned with each model) . When a new SAP σ is observed, ABL updates $\mathcal{M}$, by finding all augmentations that produce new models consistent with the past observations and with σ. More specifically, let $\mathbf{m}$ be a model in $\mathcal{M}$, let n_0 be its (unique) node aligned with SAP observed before σ. Given σ, ABL determines the set $\mathcal{N}(\sigma)$ of steps in $\mathbf{m}$ that are consistent with σ, and adds to $\mathcal{N}(\sigma)$ a new node. Then, for each $n \in \mathcal{N}(\sigma)$ ABL identifies the collection $\mathcal{A}_{\mathbf{m}}(n)$ of augmentations that would create in $\mathbf{m}$ an edge from n_0 to n. Different augmentations have different *costs*: for example, if an edge from n_0 to n already exists in $\mathbf{m}$, no change to the structure of $\mathbf{m}$ are needed, and the corresponding "null" augmentation has small cost. On the other hand, creating a branching edge from $\mathbf{s}$ to a new step has a high cost. Each

augmentation in $\mathcal{A}_{\mathbf{m}}(n)$ yields both a new model and a new alignment of the training set. In this alignment, all previous observations retain their previous alignment with nodes in $\mathbf{m}$, while the new observation is aligned with n. Given the alignment produced by an augmentation $\mathfrak{A}$, ABL tries to produce a new model, by inducing steps and inferring predicates. This operation can fail, in which case $\mathfrak{A}$ is discarded, or succeed, in which case an instantiation cost is computed by adding the costs of instantiating individual steps and predicates (described below). The surviving augmentations applied to the corresponding models yield a new collection of models $\mathcal{M}'(\sigma)$. Each model in $\mathcal{M}'(\sigma)$ has a cumulative cost, computed as the sum of the instantiation cost and of the costs of the augmentations used to produce it. The model with lowest cost is presented to the user. Hence, ABL solves the alignment and generalization problem by incrementally producing a variety of procedure structures and of possible alignments of the data with these structures, and visualizing the structure and alignment with the smallest cost. In practice this means that ABL will display a linear sequence of action steps during a demonstration until the cost of that sequence is greater than the cost of the alternate structure with new branching or looping control flow. Thus during demonstration the system would "rollup" the repetitions of a loop and present them as different iterations of single loop once sufficient evidence was presented.

The induction process can be optimized using a Viterbi-algorithm-like approach. At each point in time, only the model corresponding to the best augmentation to a destination step s is is added to $\mathcal{M}$. The other models are retained but not propagated. If the cost of the current best model becomes higher than that of one of the models retained in the past, backtracking is applied to analyze and propagate this model.

Whenever the user performs an edit ϵ to produce a new model $\hat{\mathbf{m}}$, ABL discards $\mathcal{M}$ and continues the induction with a new collection $\hat{\mathcal{M}}$ containing only $\hat{\mathbf{m}}$.

ABL induction of steps

Inducing a step means constructing the associated parameterized, generalized action α, which can be represented as a quadruple $(\mathfrak{y}, \mathbb{S}, \mathbb{D}, \mathfrak{t})$. Here, $\mathfrak{y}$ is an action type (e.g., "uncheck a check box"), $\mathbb{S}$ is a set of source widgets, $\mathbb{D}$ is a set of destination widgets, and $\mathfrak{t}$ is a text entry. Depending on the type of action, obviously only a subset of the components is meaningful. The induction of a step s in a model $\mathbf{m} \in \mathcal{M}$ is the process of constructing (*generalizing*) the relevant elements of the quadruple using the SAPs aligned with s. Generalization in ABL can be performed with any generalization grammar described in earlier PBD work. In this paper we use an approach consistent with a predefined version space, as described, for example, in [7].

Steps can be *variabilized*, namely, the collection of source widgets, the collection of destination widgets, or the string can be replaced by the learning algorithm with variables that are resolved at playback. A simple example is the induction of a "foreach" loop, which iterates over a collection of items; the "foreach" operator iteratively selects an item from a collection, and the learning algorithm is responsible for identifying the steps that operate on the currently selected item, and replace it with a variable.

Steps can also be manually *parameterized*, namely, the author can manually indicate that the widgets or the text entry must be provided by the user (think of selecting a new password).

The induction algorithm also assigns a cost to the instantiation of a step, which depends on the complexity of the generalization.

ABL induction of predicates

The predicates of the edges leaving a step s are induced using data $\tilde{\mathbf{X}}(s)$ selected as described in Proposition 1. Each state in $\tilde{\mathbf{X}}(s)$ is labeled with the corresponding aligned step. This data is used as input to a decision tree classifier [2], which is then translated into a disjunction of rules, where a rule is a conjunction of terms having the form `attribute.value` $\in$ `valueSet`, where `attribute` is a specification of a property of a widget on the screen, and `valueSet` is a list of strings. Different terms in a rule use different attributes. The reason for the translation is that the disjunction of rules can be presented in a easily readable format to the user, while inspecting a decision tree classifier is typically difficult.

The cost of instantiating a predicate is an increasing function of the complexity of the resulting disjunction of conjunctions: a predicate that always evaluates to `true` has a smaller cost than a predicate associated with a complex expression.

3. EXAMPLES

ABL is the learning algorithm of a system called DocWizards [13], which is used to construct automation and documentation wizards for applications build on top of the Eclipse platform. Figure 1 shows an example of the procedure models that can be induced using ABL, displayed within the DocWizards user interface. Since we do not know of any other algorithm that allows explicit edits of procedures, we are could not perform a comparative study.

Instead, in this section, we present typical examples of the behavior of ABL in specific situations where the user decides to perform manual edits. The two situations selected are characterized by different motivations for the edits. In the first case, we deal with the very common case in which a user makes a mistake during recording, while in the second case the user decides to modify the structure of the procedure model to help the learning algorithm induce a more compact model with fewer demonstrations than with a by-demonstration-only approach.

Recording after deleting steps

In this example, the author records a procedure that adds the "add-javadoc" tabs to all the projects in the workspace. In Eclipse, the projects are represented as tree items in the "Package Explorer" tree. The user makes a mistake by recording the task without ensuring that the initial state of the application is the desired one. Mistakes during recording are very common: in a past user studies [6] all the subjects (and in 30% of the cases, the expert used for control) failed to produce the desired demonstration), and editing yields a pwerful and straightforward mechanism for correcting mistakes without discarding the recorded trace.

In the first demonstration, the author starts recording the procedure, but sees that the "add-javadoc" tag is already associated with the first project properties. The author then removes the tag from the properties, and shows how to add it. The resulting script is:

```
(1) Select toolbar item "Java"
```

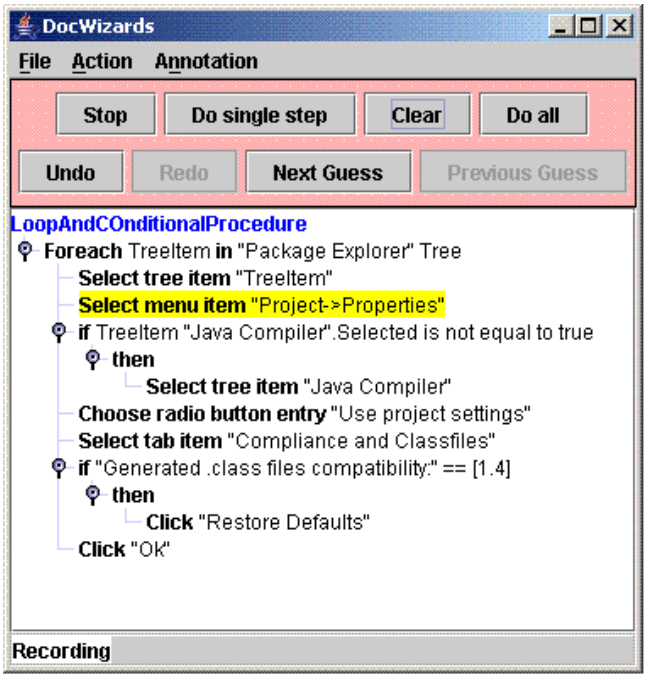


Figure 1: A procedure induced by ABL displayed in the DocWizards' GUI. The procedure contains a loop and two branches.

```
(2) Select tree item "project1"
(3) Select menu item "Project->Properties"
(4) Select tree item "Java Task Tags"
(5) Select table cell "add-javadoc" // spurious
(6) Click "Remove"                  // spurious
(7) Click "New..."
(8) Enter text "add-javadoc" into "Tag:"
(9) Click "OK"
```

where step numbering, in parentheses, and comments, introduced by "//", have been manually added to improve readability. The procedure now contains two spurious steps (5 and 6) that the author does not want as part of the procedure. The user edits out the steps by selecting them in the DocWizards interface and pressing the `Delete` key, to produce

```
(1) Select toolbar item "Java"
(2) Select tree item "project1"
(3) Select menu item "Project->Properties"
(4) Select tree item "Java Task Tags"
(5) Click "New..."
(6) Enter text "add-javadoc" into "Tag:"
(7) Click "OK"
```

The user then continues to demonstrate the task. ABL correctly identifies a loop over the projects, but does not use the demonstrations to reintroduce the removed steps. The resulting script is the following:

```
(1)  Select toolbar item "Java"
(2)  Foreach TreeItem in 'Package Explorer' Tree
(3)    Select tree item "TreeItem"
(4)    Select menu item "Project->Properties"
(5)    Select tree item "Java Task Tags"
(6)    Click "New..."
(7)    Enter text "add-javadoc" into "Tag:"
(8)    Click "OK"
```

This demonstrates the ability of ABL to correctly perform inference after steps have been removed from the model.

Recording after moving steps

The following example discusses a case in which, although ABL produces a procedure structure consistent with the demonstrations, the author decides to manually edit the script both to reduce the number of demonstrations needed to build a complete procedure model, and to help the learning algorithm produce an efficient (i.e., small) model. The part of task used in this example is to ensure that three projects (test1, test2, and test3) are in the workspace before proceeding with further operations.

The first demonstration is performed when all three projects are in the workspace, and the second when only test3 is in the workspace. The resulting script is the following,

```
 (1) if TreeItem "test1" does not exist
 (2)   then
 (3)     Right-click on PWTree // import test1
 (4)     Select popup menu item "Import..."
 (5)     Select "Existing Project into Workspace"
 (6)     Open  "C:\eclipse\workspaces\workspace\test1"
 (7)     Click "Finish"
 (8)     Right-click on PWTree // import test2
 (9)     Select popup menu item "Import..."
(10)     Select "Existing Project into Workspace"
(11)     Open  "C:\eclipse\workspaces\workspace\test2"
(12)     Click "Finish"
(13) Select tree item "test1"
```

where some steps are not shown for sake of brevity. Here, step (13), `Select tree item "test1"`, is the first step of the rest of the task, and is actually followed by other steps aimed at configuring the projects. For sake of brevity, we only show this portion of the task. Note that there are no steps to import test3, because it is already in the workspace in both demonstrations.

The next demonstration is performed when both test2 and test3 are in the workspace, but not test1. The resulting script is the following:

```
 (1) if TreeItem "test1" does not exist
 (2)   then
 (3)     Right-click on PWTree
 (4)     Select popup menu item "Import..."
 (5)     Select "Existing Project into Workspace"
 (6)     Open  "C:\eclipse\workspaces\workspace\test1"
 (7)     Click "Finish"
 (8)     if TreeItem "test2" does not exist
 (9)       then
(10)         Right-click on PWTree
(11)         Select popup menu item "Import..."
(12)         Select "Existing Project into Workspace"
(13)         Open  "C:\eclipse\workspaces\workspace\test2"
(14)         Click "Finish"
(15) Select tree item "test1"
```

This partial model is not incorrect, and with a sufficient number of additional demonstration ABL would correctly infer a model that covers all possible initial states. However, the author decides to manually edit the script to move the second conditional outside the "then" part of the first, and produces the following model

```
 (1) if TreeItem "test1" does not exist
 (2)   then
 (3)     Right-click on PWTree
 (4)     Select popup menu item "Import..."
 (5)     Select "Existing Project into Workspace"
 (6)     Open  "C:\eclipse\workspaces\workspace\test1"
 (7)     Click "Finish"
 (8) if TreeItem "test2" does not exist
 (9)   then
(10)     Right-click on PWTree
```

```
(11)     Select popup menu item "Import..."
(12)     Select "Existing Project into Workspace"
(13)     Open  "C:\eclipse\workspaces\workspace\test2"
(14)     Click "Finish"
(15) Select tree item "test1"
```

The final demonstration is performed when no project is in the workspace. The resulting model is

```
 (1) if TreeItem "test1" does not exist
 (2)   then
 (3)     Right-click on PWTree
 (4)     Select popup menu item "Import..."
 (5)     Select "Existing Project into Workspace"
 (6)     Open  "C:\eclipse\workspaces\workspace\test1"
 (7)     Click "Finish"
 (8) if TreeItem "test2" does not exist
 (9)   then
(10)     Right-click on PWTree
(11)     Select popup menu item "Import..."
(12)     Select "Existing Project into Workspace"
(13)     Open  "C:\eclipse\workspaces\workspace\test2"
(14)     Click "Finish"
(15)     if TreeItem "test3" does not exist
(16)       then
(17)         Right-click on PWTree
(18)         Select popup menu item "Import..."
(19)         Select "Existing Project into Workspace"
(20)         Open  "C:\eclipse\workspaces\workspace\test3"
(21)         Click "Finish"
(22) Select tree item "test1"
```

which is further edited by the user to produce a model with three consecutive if statements.

```
 (1) if TreeItem "test1" does not exist
 (2)   then
 (3)     Right-click on PWTree
 (4)     Select popup menu item "Import..."
 (5)     Select "Existing Project into Workspace"
 (6)     Open  "C:\eclipse\workspaces\workspace\test1"
 (7)     Click "Finish"
 (8) if TreeItem "test2" does not exist
 (9)   then
(10)     Right-click on PWTree
(11)     Select popup menu item "Import..."
(12)     Select "Existing Project into Workspace"
(13)     Open  "C:\eclipse\workspaces\workspace\test2"
(14)     Click "Finish"
(15) if TreeItem "test3" does not exist
(16)   then
(17)     Right-click on PWTree
(18)     Select popup menu item "Import..."
(19)     Select "Existing Project into Workspace"
(20)     Open  "C:\eclipse\workspaces\workspace\test3"
(21)     Click "Finish"
(22) Select tree item "test1"
```

The example shows the power of PBD combined with editing. The final model was produced with four demonstrations and two drag-and-drop operations, namely, a small number of traces and minimal intervention on the part of the user.

4. RELATED WORK

We direct the interested reader for a broad overview of the Programming-By-Demonstration field to the classical references [4, 9]. We are not aware of any learning algorithm that solves the combined alignment and generalization with edits problem. To-date, we are only aware of systems where editing has been done as a post-processing step procedures produced from a single demonstration [4, Chapter12].

In the literature, there are numerous techniques for modeling sequences of dependent input-output pairs that could be used in PBD. In particular, we cite Input-Output Hidden Markov Models (IOHMMs) [1] and Conditional Random Fields [5]. An extension of the IOHMM, called the SimIOHMM [12] solves the alignment problem, and has been successfully used to learn procedure on Microsoft Windows-based machines [6]. These are very powerful statistical methods produce opaque procedure representations that do not lend themselves to human inspection, and, a fortiori, to editing. Even if editing were possible, their learning algorithms are not designed to take into account user edits and would in fact override the edits.

The learning algorithms used in PBD commonly produce deterministic models of the task, and are hence generally less powerful than the cited probabilistic models. Additionally, they often require direct user intervention in the induction process. However, like our approach they often produce a human-readable description of the procedure model.

Typically, the user is allowed to interact with the learning algorithm using other mechanisms. Gamut [10] is an example of PBD system providing a variety of mechanisms for the user to guide the learning algorithm. These mechanisms include “nudges”, “temporal ghosts” (to explicitly refer to past values of properties), “hint highlighting” (to explicitly indicate objects whose properties are relevant to inference), “stop that” (to indicate that the induced behavior is incorrect), “negative examples” to explicitly demonstrate incorrect behavior, “asking questions” (to allow the system to ask elucidating questions to the user when inference fails), “guide objects”, etc. Gamut is able to infer conditionals (based on the properties of objects specified by the user, in contrast with SimIOHMMs and ABL, where the identification of relevant objects and features is performed automatically by the learning algorithm) but it is unclear whether it actually infers loops. The representation of the task constructed by Gamut is opaque and does not allow for explicit user edits.

A variety of systems have been produced that rely on version-space algebra as the main inference algorithm. The paper by Lau, Domingos, and Weld [8] contains recent advances in this area as well discussion of previous work. In this papers, the authors propose an approach to learn a grammar very similar to the one presented in this paper. They show how to learn from demonstrations programs generated by this language under the assumption that the user solves the alignment problem, namely, when each action is manually labeled by the user with a step identifier, which denotes the corresponding step. They also show how to use version-space algebra without manual labeling of the observed actions, but here the structure of the induced program is limited to a loop that contains a variable number of steps. Fixed-length loop induction, where loops are not constrained to contain all the actions in the procedure, was a feature of Eager [3], but this system could not detect complex structure within a loop or nested loops.

5. DISCUSSION AND CONCLUSIONS

In this paper, we have presented an approach to PBD in which the user can interact with the learning algorithm by explicitly editing the procedure model while demonstrating the task. We argue that explicit editing is a powerful mean for controlling the behavior of the learning algorithm.

We have introduced a new algorithm, Augmentation-Based Learning, that supports interspersing demonstrations with

manual edits. At the core of ABL is the approach used to modify the hypothesis space of the learning algorithm when new training samples are provided. ABL relies on a collection of functions, called augmentations, that modify the structure of an existing procedure model when a new training sample is observed, where a training sample is a user action paired with a description of the content of the screen just before the user performed the action. The new hypothesis space of ABL is the transitive closure of the augmentation functions applied to the existing procedure structure. Therefore, when the user manually edits a procedure, ABL can continue the induction process by using the edited procedure as the starting point to expand the hypothesis space, and therefore does not use previously observed data to undo the user edits.

The key advantages of using ABL for PBD are:

- ABL solves the full combined alignment and generalization problem, and therefore:
  - it does not require the user to align observed actions with each other or with existing procedure steps;
  - it does not require intrusive feedback mechanisms, such as specifying the boundaries of loops, marking actions suggested by the learning algorithm as incorrect, or explicitly selecting GUI objects whose properties are important for inference;
  - it is not limited to restricted classes of procedure structures, such as sequences of steps embedded in a top-level loop, or fixed-length loops without conditionals.
- The author may think in terms of concrete action sequences when programming by demonstration, rather than the generalized actions and flows required for traditional programming.
- The author does not need to invent a demonstration sequence in order to force an explicit change in the generated procedure. To enforce a specific change a user need only edit directly the procedure model. This feature can be used to reduce the number of demonstrations that a purely "by-demonstration" approach would require to produce a complete model of the task, as well as to force the learning algorithm to produce a simpler model.
- Obsolete sub-parts of a procedure model can be manually removed and re-demonstrated: this reduces the cost of maintaining procedure models.
- The user need not be concerned with performing exactly all the steps required to demonstrate a task while at the same time ensuring that no spurious action become part of the model. Spurious steps and mistakes can be easily removed using editing operations. In our experience, this has proved to be an extremely useful characteristic.

ABL is the learning algorithm of DocWizards, a PBD system implemented as an Eclipse plugin. ABL operates in real-time, and provides immediate feedback to the user. To date, we have only conducted user studies that focus on the overall user experience, rather than on the specifics of the learning algorithm.

Within DocWizards, we have also provided the ability to inspect the alternative hypotheses that DocWizards maintains, as explained in Section 2.3, and to select one of the alternatives to the model with the smallest score. In practice, however, we have found that this feature is not nearly invoked as frequently as manual editing. However, we found this feature to be very useful for demonstrating how DocWizards manipulates its search space.

In future work we expect to empirically explore this combination of programming techniques to see what second-order benefits or drawbacks exist for this combination of programming techniques. Another direction of investigation is the trade-off between the space of allowable augmentations and the user experience. If only a limited number of augmentations are allowed, ABL is constrained in the kinds of procedure models it can produce, and it is likely not to yield compact representations of the task. This, in turn, reflects on the need to perform more edits, should the user be interested in a simple and efficient representation of the procedure. On the other hand, if the number of augmentations is too large, the search space considered by ABL grows and so does the computational cost.

6. REFERENCES

[1] Y. Bengio and P. Frasconi. Input-Output HMM's for sequence processing. *IEEE Trans. Neural Networks*, 7(5):1231–1249, Sept. 1996.

[2] P. Chou. Optimal partitioning for classification and regression trees. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 13(4):340–354, Apr. 1991.

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

[4] A. Cypher, editor. *Watch what I do: Programming by demonstration*. MIT Press, Cambridge, MA, 1993.

[5] J. Lafferty, A. McCallum, and F. Pereira. Conditional random field: Probabilistic models for segmenting and labeling sequence data. In *Proc. Int. Conf. on Machine Learning*, pages 282–289, 2001.

[6] T. Lau, L. Bergman, V. Castelli, and D. Oblinger. Sheepdog: Learning procedures for technical support. In *Proc. 2004 Int. Conf. on Intelligent User Interfaces*, pages 106–116, 2004.

[7] T. Lau, P. Domingos, and D. Weld. Version space algebra and its application to programming by demonstration. In *Proc. Seventeenth Int. Conf. on Machine Learning*, pages 527–534, June 2000.

[8] T. Lau, P. Domingos, and D. Weld. Learning programs from traces using version space algebra. In *Proc. 2nd Int. Conf. on Knowledge Capture*, 2003.

[9] H. Lieberman, editor. *Your Wish is My Command: Giving Users the Power to Instruct their Software*. Morgan Kaufmann, 2001.

[10] R. G. McDaniel and B. A. Myers. Building applications using only demonstration. In *Proc. 1998 Int. Conf. on Intelligent User Interfaces*, pages 282–289, 1998.

[11] N. Medvivovic, A. Egyed, and D. Rosenblum. Round-trip software engineering using uml: From architecture to design and back,. In *Proc. 2nd Workshop Object-Oriented Reengineering (WOOR 99)*, pages 1–8, Monterey, CA, USA, 1999.

[12] D. Oblinger, V. Castelli, T. Lau, and L. Bergman. Similarity-based alignment and generalization. In *Proc. Sixteenth Europ. Conf. on Machine Learning*, page To appear, October 2005.

[13] M. Prabaker, L. Bergman, and V. Castelli. An evaluation of using programming by demonstration and guided walkthrough techniques for authoring and following documentation. In *submitted to CHI*, 2006.