**RC24055 (W0609-064) September 19, 2006**
**Computer Science**

# IBM Research Report

## Distributed Augmentation-Based Learning: A Learning Algorithm for Distributed Collaborative Programming-by-Demonstration

**Vittorio Castelli*, Lawrence Bergman****
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218*
P.O. Box 704**
Yorktown Heights, NY 10598

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

# Distributed Augmentation-Based Learning

## A Learning Algorithm for Distributed Collaborative Programming-by-Demonstration

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

*Lawrence Bergman*
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
bergmanl@us.ibm.cm

**ABSTRACT**
The learning algorithms used in Programming-by-Demonstration (PBD) are either on-line and incremental or off-line and batch. Neither category is entirely suitable for capturing know-how from demonstrations in a distributed, collaborative environment, where multiple expert can independently provide examples to improve the model.

In this paper we describe Distributed Augmentation-Based Learning (DABL), the first real-time PBD learning algorithm suited for distributed know-how acquisition. DABL is an incremental learning algorithm that uses a version-control-like paradigm to combine independently constructed procedure models. An expert can check out a procedure model from a repository and modify it by means of new demonstrations or by manually editing it. The expert then reconciles the changes with those concurrently made by other experts and checked into the repository.

DABL automatically merges the two procedures, learns new decision points based on reconcilable differences, and identifies conflicts where there are multiple valid ways of combining the changes or where the combination produces an invalid model, that is, one that does not lie in the search space of the learning algorithm.

**ACM Classification:** I.2.6[Artificial Intelligence]: Learning - Knowledge Acquisition.

**General terms:** Algorithms

**Keywords:** Example-and demonstration-based interfaces, Programming-By-Demonstration, Artificial Intelligence

### Introduction

Programming-by-demonstration (PBD) [8, 4] has been widely proposed as an attractive approach to improving personal productivity. A user can automate a task by demonstrating it to a PBD system, which *generalizes* the task into an executable *procedure model*. The early work in PBD learning algorithms focused on a single demonstration of a task—often the performance of one or more iterations of a repetitive sequence of operations.

*Collaborative PBD* has been more recently proposed as a way of developing richer procedure models that can address a wide range of use cases [3]. These uses range from authoring intelligent documentation [2], to creating automation tools that combine the knowledge of a group of experts (e.g., technical support professionals [6], system administrators [7], or developers), to building guided application walkthroughs. Collaborative PDB involves combining multiple demonstrations to capture the structure of a complex task, for example, to create decision points that depend on differences in the task execution environment.

A collaborative approach to PBD offers at least two advantages over alternative approaches to knowledge capture and dissemination: through the combination of examples produced by multiple experts, PBD directly captures their collective knowledge; and since procedure models can be incrementally modified by adding new examples, PBD can simplify the program life cycle: the structure of a captured task can be incrementally learned, refined, and updated.

We note two prior approaches to collaborative learning algorithms for PBD. The first we call *parallel collaborative PBD*. This approach involves combining multiple independently collected recordings of a task (we will use the term *traces* to refer to these individual recordings) to induce a complex procedure model. SimIOHMM [10] is an example of such a learning algorithm. SimIOHMM extends the Input-Output Hidden Markov Model [1] by biasing the learner with a similarity function defined on the *states* (i.e., representations of the GUI content prior to each action). A distinctive characteristic of SimIOHMM is its ability to automatically identify equivalent steps in different traces. This step *alignment* is used to construct action descriptions and to explain the differences between traces in terms of probabilistic decision points. We say that SimIOHMM solves the *alignment-and-generalization problem*, since action generalization and trace alignment are performed simultaneously. SimIOHMM suffers from two limitations. First, it produces an opaque model

that is very difficult to present to the end-user in a readable form. Second, it is a batch algorithm—the user does not see the real-time effects of her actions on the model.

We call the second approach *incremental collaborative PBD*. This involves using a previously created model to guide the user during subsequent recordings; each recording is used to incrementally add information to an evolving procedure model. An example, Augmentation-based learning (ABL) [9], is an incremental learning algorithm that overcomes the two main limitations of SimIOHMM. By restricting the search space of the learner to the set of programs produced by a deterministic grammar, ABL incrementally induces procedure models in real time that are easily converted into a human-readable, script-like form. ABL is also the first PBD learning algorithm that allows the user to directly edit the procedure model. In ABL, editing is treated as a source of bias to the learning algorithm and, as a consequence, demonstrations and editing operations can be seamlessly interweaved.

In this paper, we propose a third class of collaborative PBD learning algorithms. We call these *distributed collaborative PBD*. Distributed collaborative algorithms combine the benefits of both prior approaches—providing support both for learning incrementally from multiple examples and for merging possibly conflicting recordings or procedure models generated in parallel by different experts.

We will describe a particular instance of this class, Distributed Augmentation-Based Learning (DABL), an extension to ABL that builds procedure models from user demonstrations, from manual edits, and by combining procedure models separately modified by different authors. We describe DABL using a scenario centered around a procedure repository that supports version control. Experts can check out procedure models, independently modify them, and check in the improved models. DABL's version control algorithm is responsible for merging independently updated versions of the model, for identifying conflicts, and for providing conflict-resolution facilities.

To support distributed procedural know-how acquisition, the version control system must satisfy two novel requirements.

First, the combination of independently modified versions of a procedure model must be a *learning* operation. This requirement states that, besides ensuring that the merged procedure is syntactically correct, the merge process must also enhance the model to account for differences between the versions being merged, for example, by introducing new decision points.

Second, the merged procedure model must be in a format compatible with the learning algorithm. This requirement states that, besides ensuring that the structure of the merged procedure model belongs to the search space of the learning algorithm, the data structures used for learning must be updated to be consistent with both versions of the procedure.

The main contributions of this paper are the following:

1. The idea of distributed collaborative PBD, which combines incremental procedure updates with batch learning.
2. The first PBD learning algorithm for distributed know-how capture. DABL attains this goal by: i) learning both incrementally in real time and in an off-line mode, and ii) solving the full alignment-and-generalization problem while supporting manual edits.
3. The concept and implementation of an intuitive, version-control-like front-end for a distributed PBD learning algorithm.
4. The concept of version reconciliation as a new learning paradigm—the learning algorithm can infer novel structure of the task not present in either of the procedure models being reconciled; and an algorithm that embodies this concept.

The rest of the paper is organized as follows. We first describe the Augmentation-Based Learning algorithm extended by DABL. We then describe the components of DABL: identifying conflicts in separately modified versions of the same procedure model, merging portions of the procedures where no conflict occur, and supporting conflict resolution. We show examples of the behavior of a DABL implementation integrated with the DocWizards system [2]. We conclude the paper with a summary and a discussion of future work.

## AUGMENTATION-BASED LEARNING

Augmentation-Based Learning (ABL) [9] is an incremental PBD learning algorithm that builds procedure models in real-time from observations of the interactions between a user and an application. Each observation is summarized by a *state-action pair* (SAP). A SAP consists of a representation of the content of the UI prior to the action (the *state*) and a description of the user action at an appropriate level of abstraction (the *action*). In the ABL implementation used in the DocWizards system [2], the state is constructed by walking the widget tree of the target application UI and extracting descriptive attributes from each widget. The abstraction level of the action is the one typically used in how-to instructions. For example, a typical action description has the form: `Uncheck "Use default compiler options"`. ABL produces a program specified by a grammar selected to address the needs of the intended consumers. The learning algorithm can be easily adapted to any grammar, and is not limited to those used in the DocWizards system or in the current paper. For sake of simplicity, in the current paper we use the simplified grammar described in Figure 1. The ABL implementation used in DocWizards is based on a more complex grammar that composes subtask groups of steps, loops, and branches.

The atomic learning step of ABL consists of extending a possibly empty procedure model with a new SAP as input, and can be described as follows. ABL maintains a "current action step" cursor $\mathfrak{C}$; this cursor is initialized to the beginning of the procedure or points to an action step specified by the user. An action step is a step in the procedure that represents the generalization of an action; other types of steps are control steps, such as decision points and loop steps, and structural steps, such as subtask groups. When it receives a new

**Procedure Representation Grammar**

```
<model>     := <step>+
<step>      := <actionStep> | <branch>
<branch>    := Decision Point <pathExpr>+
<pathExpr>  := if <predicate> then <step>+
```

Figure 1: The procedure representation grammar used in this paper. A predicate <predicate> is a boolean expression on values of UI widget attributes. An action step <actionStep> is a human-readable description of a specific action.

SAP as input, ABL finds the collection $C$ of action steps in the model that are consistent with this input. An action step is consistent with a SAP if the step is a generalization (abstraction) of the action in the SAP. For each consistent step, say $s$, ABL checks whether the model can be modified to contain a direct path from $\mathfrak{C}$ to $s$. By direct path we mean a path without intervening action steps.

ABL is only allowed to modify an existing model via transformations called *augmentations*. An augmentation is a transformation that preserves all existing steps and control structures and all existing direct paths between action steps. An augmentation can therefore only add new action steps and new direct paths between action steps (by inserting control steps , such as branches or loops). An augmentation is *valid* if it produces a new model consistent with the grammar. Note that, if the grammar allowed the "go to" statement, a valid augmentation would always exist from any action step to any other action steps. In general, however, it is not always possible to find valid augmentations from $\mathfrak{C}$ to a step in $C$. If no valid augmentation exists from $\mathfrak{C}$ to any step in $C$, ABL modifies the model by adding to it a new action step $\hat{s}$ and inserting a decision point right after $\mathfrak{C}$ that introduces a direct path from $\mathfrak{C}$ to $\hat{s}$. If ABL finds a valid augmentation to a step in $C$, ABL produces a new augmented model. If more than one model is produced, the simplest one is selected for display. As new SAPs are observed, ABL augments these models in parallel. Several heuristics are used to prune the collection of parallel models. Finally, ABL generalizes the action steps and automatically infers the decision point predicates and loop conditions that might be affected by the new observation. After augmenting a model, ABL advances the cursor $\mathfrak{C}$ to $s$.

Note that, during the described learning step, ABL automatically aligns the SAPs to the procedure model. The alignment information is maintained in an appropriate alignment data structure, consisting of: a map where the keys are the action steps and the values are the list of SAPs aligned with the step used as key; a map from SAPs to the aligned procedure step; and a collection of correspondences between SAPs and steps maintained dynamically while learning from demonstrations (this collection is indexed using the models augmented in parallel as keys). The third data structure is temporary, and its content is used to selectively update the first two at the end of a demonstration. The first two data structures are persistent and are stored with the procedure model.

Note also that ABL generalizes the procedure model using the aligned data. Therefore, unlike most learning algorithms used in PBD, ABL solves the alignment-and-generalization problem for general grammars.

ABL allows the user to provide partial demonstrations starting from a user-specified place in the procedure model, and to insert new action steps in the procedure by demonstrating the corresponding actions. The latter function will be used in the examples section.

Finally, ABL allows the user to edit the procedure model and supports free interweaving of demonstrations and manual edits. ABL treats editing as a source of constraints for the induction process. In particular, ABL ensures that data observed before the edit is not used to "undo" the effects of editing. Additionally, ABL incorporates algorithms for carefully disregarding data that is inconsistent with the manual edits, as discussed in [9]; this is particularly relevant while inducing control structures. Previous PBD systems, such as Chimera [4, Chapter12], support editing of the program representation as a post-processing step, not as part of the induction process: users cannot edit in the midst of a demonstration and the learning algorithm cannot refine a manually edited model.

There are several reasons to support manual edits of procedure models. First, manual edits can substantially reduce the number of demonstrations required to produce a desired procedure model. An example is given in [9]. Second, editing is a powerful mechanism for maintaining procedure models: (1) as the life cycle of the procedure unfolds, parts of the model can become obsolete due to changes in the application GUI or in the process being captured; (2) the order in which subtasks are executed could change to reflect changes in the underlying process; (3) mistakes can be introduced in the model via erroneous task demonstration. In all three cases, direct editing is the most natural way of correcting these problems.

The DocWizards implementation of ABL supports a rich set of editing operations. In the interest of clarity and simplicity, in this paper we will only consider two editing operations: deleting steps and moving steps (cut-and-paste). These operations are by far the most common, and are of fundamental importance when multiple authors contribute to a procedure model over an extended period of time. DABL uses ABL as the core learning algorithm; it extends ABL by adding the ability to learn by combining independently modified procedure models.

## THE DABL ALGORITHM

### Motivating Scenario

Alice and Bob independently check out a procedure model $m^O$ from the repository. Alice enhances the model with new recordings and manually edits it; she then commits the result to the repository. The repository now contains a new model $m^R$. Bob also enhances the model with new recordings and edits, and produces a local version of the procedure, $m^L$. Before being allowed to commit his changes, Bob now

**INITIALIZE**
Initialize the conflict set $\mathbb{C}$ to empty

*Preprocess the repository version*
Identify $\mathbf{T}^R$, the unique traces of $m^R$
Determine $\mathfrak{D}^R$, the deleted steps in $m^R$
Determine $\mathfrak{M}^R$, the moved steps in $m^R$
Determine $\mathfrak{N}^R$, the new steps in $m^R$

*Preprocess the local version*
Identify $\mathbf{T}^L$, the unique traces of $m^L$
Determine $\mathfrak{D}^L$, the deleted steps in $m^L$
Determine $\mathfrak{M}^L$, the moved steps in $m^L$
Determine $\mathfrak{N}^L$, the new steps in $m^L$

**DETERMINE CONFLICTS**
Add to $\mathbb{C}$ the steps in $\mathfrak{D}^R \bigcap \mathfrak{M}^L$
Add to $\mathbb{C}$ the steps in $\mathfrak{M}^R \bigcap \mathfrak{D}^L$
Add to $\mathbb{C}$ the steps in $\mathfrak{M}^R \bigcap \mathfrak{M}^L$
  with different destinations
  in $m^L$ and $m^R$
Add to $\mathbb{C}$ the steps in $\mathfrak{N}^L$ having context
  that intersects $\mathfrak{M}^R \bigcup \mathfrak{D}^R$
Add to $\mathbb{C}$ the steps in $\mathfrak{N}^R$ having context
  that intersects $\mathfrak{M}^L \bigcup \mathfrak{D}^L$
Add to $\mathbb{C}$ the decision points in $\mathfrak{N}^R$
  and $\mathfrak{N}^L$ that cannot be reconciled

**MERGE PROCEDURES**
In the order in which they appear in $\mathbf{T}^L$,
  Incorporate steps from $\mathfrak{N}^L$ into $m^R$
  Align the unique traces of $m^L$
    with the updated $m^R$
  Apply edits from $\mathbf{T}^L$ to $m^R$
  Manually reconcile conflicts in $\mathbb{C}$

Figure 2: The procedure reconciliation algorithm.

needs to reconcile $m^L$ and $m^R$. The DABL algorithm extends ABL by providing support for reconciliation. When asked to combine $m^R$ and $m^L$, DABL identifies and appropriately merges differences that can be reconciled, and also detects possible conflicts. Each conflict is presented to Bob, who has the option to discard his local changes, to override the changes in the repository procedure, or, if appropriate, to specify a merge of the two versions. Once all conflicts are resolved, Bob commits the procedure model to the repository.

Bob is presented with a familiar user interface, functionally equivalent to the front-end to CVS [11] provided by the Eclipse platform [5]. The similarity with CVS, however, ends here. The merge process in DABL is not based simply on a lexical comparison between the procedures, but is actually a new form of learning. To support the scenario described: (1) DABL must ensure that automatic merges and conflict resolutions produce a model consistent with the grammar used for learning from demonstrations. (2) DABL must also use the differences between the models being reconciled to learn structural information that is not available from either model alone. (3) DABL must ensure that the new data produced by both Alice and Bob is properly aligned

| $\mathbf{m^R}$ | $\mathbf{m^L}$ | | **Outcome** |
|---|---|---|---|
| $s$ is unchanged | $s$ is unchanged | | $s$ is unchanged |
| $s$ is unchanged | $s$ is moved | $(*)$ | $s$ is moved |
| $s$ is unchanged | $s$ is deleted | $(*)$ | $s$ is deleted |
| $s$ is moved | $s$ is deleted | $(*)$ | conflict |
| $s$ is moved | $s$ is moved | | possible conflict |
| $s$ is deleted | $s$ is deleted | | $s$ is deleted |
| $s$ is a new step | context unchanged | $(*)$ | $s$ is added |
| $b$ is a new branch | context is moved | $(*)$ | conflict |
| $b$ is a new branch | context is deleted | $(*)$ | conflict |
| $b$ is a new branch | new $b^{'}$ overlaps $b$ | | possible conflict |

Table 1: Differences between two independently modified procedure models, $m^R$ and $m^L$ and corresponding implications for the reconciliation algorithm. The six cases that are symmetric to those marked with a star are omitted.

with the merged procedure model. This last requirement, together with requirement (1), must be satisfied to ensure that DABL can correctly augment the merged procedure model when new demonstrations are provided.

In the rest of this section we describe how DABL identifies the differences between $m^R$ and $m^L$, how it detects conflicts, how differences that do not result in conflicts are merged, and how the user reconciles conflicts.

### DABL Overview

The repository model $m^R$ and the local model $m^L$ share the collection of traces $\mathbf{T}^O$ used to produce the common ancestor $m^O$. Additionally, the repository version contains the collection of traces $\mathbf{T}^R$ recorded and checked in by others since the current user checked out $m^O$. Similarly, the local version has a collection of traces $\mathbf{T}^L$ recorded by the user since the check-out of $m^O$. Together, $m^R$, $\mathbf{T}^R$ and $\mathbf{T}^L$ contain all the information needed to describe the differences between $m^R$ and $m^L$: in ABL and DABL, the traces contain both demonstrations and editing operations.

Figure 2 describes the DABL reconciliation algorithm, which is divided into three parts: initialization, conflict determination, and merge. The initialization consists of extracting the information needed to determine the conflicts from $\mathbf{T}^R$ and $\mathbf{T}^L$, and to perform the merge. Conflict determination consists of identifying changes that cannot be merged automatically. The procedure merge phase consists of merging automatically the portions of the procedure that do not contain conflicts, reconciling conflicts under the direction of the user, and updating the data structures used for subsequent learning.

### Detecting differences and conflicts

The grammar and the editing operations described earlier constrain the differences between $m^O$ and one of the derived models, say $m^L$, to fall into one of three categories:

1. $m^L$ contains a new action step $s$ not in $m^O$: this action step must correspond to a recorded action in $\mathbf{T}^L$. Additionally, $m^L$ can contain a new decision point or a new branch of an existing decision point that contains $s$.

| $\mathbf{m^R}$ | $\mathbf{m^L}$ |
|---|---|
| `step 1` | `step 1` |
| `if condition1` | `step 2` |
| `  step 2` | `if condition2` |
| `  step 3` | `  step 3` |
| `else` | `  step 4` |
| `  newStep 1` | `else` |
| `end` | `  newStep 2` |
| `step 4` | `end` |

Figure 3: Example of interlocking decision points in independently modified procedures that cannot be reconciled automatically to produce a program consistent with the grammar of Figure 1.

2. $m^L$ and $m^O$ both contain an action step $s$, and $s$ has been manually moved in $m^L$.
3. $m^O$ contains an action step $s$ that has been deleted in $m^L$.

These departures of $m^R$ and $m^L$ from $m^O$ might give rise to conflicts. More specifically, a change in $m^L$ and a change in $m^R$ can result in either of two types of conflicts:

*Type 1*. It is not possible to create a new model consistent with the grammar that contains both changes;

*Type 2*. The changes could be reconciled in at least two different ways.

Table 1 summarizes the 16 cases that can be encountered when analyzing a specific step during procedure reconciliation (note that symmetric situations are implicit in the figure): eight of these can be reconciled automatically, six always result in a conflict, two can give rise to a conflict under specific circumstances, described in detail below.

When a step is affected by a change in one model but not in the other, the merged model contains the change. A conflict is always declared when a step is moved in one of the models and deleted in the other. When a step is moved in both $m^R$ and $m^L$, a conflict is declared only if the step is moved to two different places within the procedure. The conflicts described so far are statically identified by analyzing $\mathbf{T}^L$ and $\mathbf{T}^R$ as a preprocessing step.

Conflicts can also be introduced when creating new branches. The following example illustrates this case: a user enhances $m^O$ by demonstrating a new step, `newStep 1`. The learning algorithm constructs a decision point as a result. The user checks in the changes. Another user independently modifies $m^O$ by demonstrating `newStep 2`. The learning algorithm constructs another decision point. The resulting models are shown in Figure 3. In this case, the learning algorithm cannot reconcile the interlocking branches, and consequently declares a conflict. The example illustrates the following general principle: when new steps are introduced in one of the modified models and the neighborhood of these steps is perturbed in the other model, a conflict is declared. Perturbation can be caused by manual editing or by new demonstrations. While the conflicts discussed so far are of type 1, violations of this general principle can produce conflicts of type 2. For example, if a decision point is introduced in one of the models, and the steps surrounding the decision point are moved in the other, then the models can be reconciled in four possible ways: the decision point is left in place and the surrounding steps are moved; the decision point is moved together with the surrounding steps; the decision point is left in place and the move is discarded; the decision point is discarded and the steps are moved. In the first two cases, both changes can be applied to the merged procedure without producing an invalid procedure, but DABL cannot determine which of the two alternatives is the desired one.

#### Merging the procedures

In general, a PBD procedure model consists of two main components: the program presented to the user and executed during playback, and the alignment of the traces to the program used by the learning algorithm. DABL simultaneously reconciles both components by updating the repository procedure $m^R$ with information from the local copy. DABL guarantees that the alignment of the traces $\mathbf{T}^O$ used to construct the original model is preserved in both $m^R$ and $m^L$. Therefore, the alignment of $m^R$ only needs to be updated with information from the traces $\mathbf{T}^L$ unique to $m^L$.

To reconcile the programs, DABL identifies $\mathfrak{N}^L$, the set of steps that are unique to $m^L$. In the DocWizards implementation, this is accomplished by inspecting the step labels; when a procedure model is checked out from a repository, the local copy is given a unique identifier used to prefix the label of each new step. Steps in $\mathfrak{N}^L$ that have not been identified during preprocessing as being part of a conflict are grouped into contiguous, non-adjacent groups. Each such group $\mathbb{G}^L$ is analyzed as follows:

- DABL determines whether $m^L$ contains changes equivalent to those in $m^R$. To accomplish this, the DocWizards's implementation of DABL identifies the subtraces $\mathbf{T}^{\mathbb{G}^L}$ of $\mathbf{T}^L$ aligned with $\mathbb{G}^L$ using the existing alignment data structure. DABL then iterates on each element $\mathbf{t}^L$ of $\mathbf{T}^{\mathbb{G}^L}$; it invokes the learning algorithm, initializes the current step cursor to the last action step that occurs before $\mathbb{G}^L$ and that exists in both $m^L$ and $m^R$, and provides the SAPs of the subtrace $\mathbf{t}^L$ as input to the learning algorithm. If the learning algorithm can align all the subtraces with $m^R$ without introducing new steps, then the reconciliation algorithm declares that the changes are equivalent.
- If the changes are not equivalent, DABL identifies the location in $m^L$ where the new steps should be added. If the steps preceding or following that location have been moved in $m^R$, or if the location cannot be determined because of a delete operation in $m^R$, DABL declares a conflict (further details are given below). Otherwise, DABL invokes the learning algorithm with $\mathbf{T}^{\mathbb{G}^L}$ as input and tries to augment $m^R$. If there is no valid augmentation, DABL declares a conflict. An example that leads to such a failure is shown in Figure 3. If reconciliation succeeds, no conflict is declared, and DABL analyzes the next group of new steps.

Similarly, DABL identifies $\mathfrak{N}^R$, the collection of steps that are present in the repository model and not in the local model. Steps in $\mathfrak{N}^R$ that have not been identified during preprocessing as being part of a conflict are grouped into contiguous,

non-adjacent groups. Each such group $\mathbb{G}^R$ is analyzed as follows:

- The traces in $\mathbf{T}^L$ are scanned to find subtraces of length two with successive SAPs $\sigma_1$ and $\sigma_2$, such that $\sigma_1$ is aligned with the action step immediately preceding $\mathbb{G}^R$, and $\sigma_2$ is aligned with the action step immediately following $\mathbb{G}^R$. Call $\mathbf{T}^{\mathbb{G}^R}$ the set of such subtraces.
- If the scan is successful, DABL uses $\mathbf{T}^{\mathbb{G}^R}$ to infer a decision point that explains the conditions under which the steps in $\mathbb{G}^R$ should be executed or ignored.

During this part of the reconciliation process, DABL extracts information on the structure of the task that is not available from either model being analyzed alone. Thus, procedure reconciliation is a new *learning mode* that sets DABL apart from all other PBD learning algorithms. Example 1 in the examples section shows this process in action.

When all new steps in $m^L$ have been analyzed, DABL is left with the tasks of reconciling the conflicts and updating the alignment. These are described in the following paragraphs.

DocWizards provides a user interface integrated with DABL to help the user reconcile conflicts. The reconciliation UI is depicted later in Figure 11. Each conflict is depicted as a tree with two children. The first child is labeled with the operation on the local copy of the procedure model that causes the conflict, and contains a subtree consisting of those steps and the surrounding context. The second child is the corresponding labeled subtree for the repository copy. In the figures of this paper, the context consists of the preceding step and the following step (if they exist); deleted steps are shown without a context.

The user controls the reconciliation process by means of three buttons, labeled "Use Local", "Use Repository", and "Use Both", respectively. The "Use Local" button instructs DABL to override the changes in the repository with those in the local procedure. Conversely, the "Use Repository" button instructs DABL to discard the local changes and resolve the conflict in favor of the version in the repository. The "Use Both" button is always inactive for conflicts of type 1. It is active for conflicts of type 2 (where there are multiple ways of reconciling the changes) when applying both changes leads to a valid procedure model. An example of this situation is the case of a move operation and a conflicting delete operation in which a proper subset $\tilde{\mathbb{G}}$ of the moved steps $\mathbb{G}$ are deleted: here, the "Use Both" reconciliation operation deletes the steps in $\tilde{\mathbb{G}}$ and moves the undeleted steps in $\mathbb{G}$ to the location specified in the move operation. A second example consists of a conflict between a move and the introduction of a new branch overlapping the moved steps. Here, "Use Both" results in moving the new branch together with the other steps.

Once all conflicts are resolved, the user can complete the check-in, which consists of updating the alignment data structures (as described below), saving the reconciled procedure model into the repository, and replacing the local version with a copy of the reconciled procedure model. The user can also abort the reconciliation process, in which case both local and repository version are unchanged.

Updating the alignment consists of the following operations:

1. The traces $\mathbf{T}^L$ are added to the repository model.
2. The alignment data structure for each new step that is created during the merge process is updated automatically by the learning algorithm.
3. The alignment data structure for each step of the common ancestor model $m^O$ is updated by adding to it the SAPs in $\mathbf{T}^L$ that are aligned with that step.
4. The alignment data structure for each step involved in a conflict are updated as in 3, with the exception of the alignments of steps that appear only in $m^L$ and that are discarded during the merge.

When all the reconciliation steps are completed, the learning algorithm is invoked again to generalize the action steps using the combined data, and to update the predicates of the decision points. This is a second main difference between the reconciliation process of traditional version control systems and that of DABL. In particular, DABL can combine data from the two procedure models and produce predicates that could not be inferred from either local or repository model alone. Note that through the reconciliation process, DABL can refine predicates even without producing structural modifications to the procedure model.

#### Remarks on the merge algorithm

Before presenting and discussing examples that describe the behavior of DABL, we comment further on specific aspects of the reconciliation process.

To focus the discussion of this paper, we have chosen a restricted grammar and a selected subset of editing operations that illustrate all the main features of DABL and highlight the main difficulties of the reconciliation process. No additional conceptual difficulties arise in implementing DABL for the full grammar and full set of editing operations used in the DocWizards system.

The reconciliation process treats the repository version and the local version differently. The clearest example is the management of discarded steps. Steps that are present in the local version of the procedure model and that are discarded by the reconciliation process are entirely forgotten. In contrast, DABL maintains all relevant information on steps belonging to the repository model that are discarded during reconciliation. The reason is that rejected changes in the repository procedure might later be restored by other users as part of subsequent reconciliations.

Finally, DABL's support for edits eliminates the need for a complex and cumbersome reconciliation process. In particular, the current implementation of DABL provides the user with three alternative choices for reconciliation, even when there are numerous other possibilities. The three options offered to the user are the most intuitive ones. In those rare

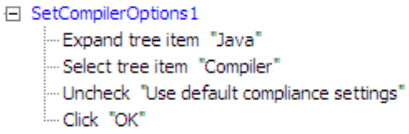


Figure 4: Ex. 1. Original repository procedure.

occasions in which another alternative would be more desirable, the user can accept the model closest to the desired result and then manually edit it.

### Implementation

DABL is implemented in Java 1.4, using SWT for the UI. DABL has been incorporated into DocWizards, a PBD system for capturing tasks on Eclipse-based applications. DABL procedures and traces are stored in XML. The procedure and trace files are compressed together in a ZIP file that simplifies their transmission and distribution. An HTML version of the procedure is also available for quick inspection.

## EXAMPLES

A set of related examples will show different aspects of the merge algorithm, the effects of user-in-the-loop reconciliation, and the ability of the algorithm to continue learning after repository merge.

The examples are based on a simple set of configuration operations on the Eclipse IDE. The common scenario involves setting compiler options as part of a project build. In order to ensure a correct build, two prerequisites must be ensured. First the compiler compliance level must be set appropriately; if the wrong level is used for generated ".class" file or source compatibility, code may not compile properly. Similarly, the appropriate Java JRE version must be used for correct compilation. In addition, build path information may need to be adjusted. We further motivate the example by noting that options within Eclipse can be set for an entire workspace, or separately for individual projects within the workspace.

In each of the examples, we look at parallel check-outs. In other words, we assume that two separate users each check-out the same code, with both check-outs prior to any code updates. The first user modifies the code and checks in. This is followed by the second user performing independent modifications and checking in, necessitating reconciliation between his modifications and those of the first user.

### Example 1

The first example illustrates how DABL treats the reconciliation step as a learning process. The initial procedure snippet unchecks the "Use default compliance setting" checkbox. Figure 4 shows this initial procedure model. The first user performs the task and later realizes that the configuration step is not complete; the source compatibility level must be set to 1.4. The user inserts the relevant step into the procedure model. The result is shown in Figure 5, where the newly inserted step is the second from the bottom. The second user independently performs the task. Being a more advanced user, she realizes that one should ensure that the Compiler compliance level be set to 1.4 if it is currently 1.3

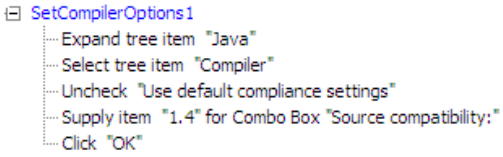


Figure 5: Ex. 1. Procedure modified by first user by inserting a new statement.

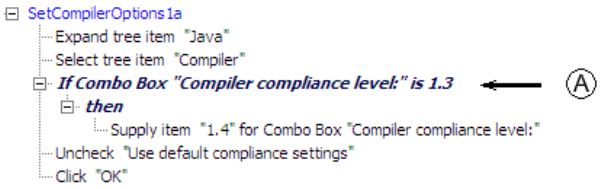


Figure 6: Ex. 1. Procedure modified by first user by inserting a conditional.

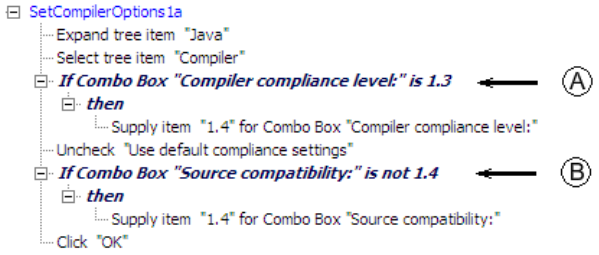


Figure 7: Ex. 1. Merged procedure.

before unchecking the 'Use default compliance setting' check box. The user turns on recording before performing the alternative, and DABL correctly learns the condition under which the compliance level should be reset. This user makes no change to the 'Source compatibility' because it is already set to 1.4 it her environment

The resulting procedure model is shown in Figure 6. Upon check-in, DABL incorporates the decision point, A, shown in Figure 6 into the reconciled model of Figure 7. So far, DABL behaves (superficially!) like CVS. However, this similarity ceases to exist when DABL attempts to explain the other difference between the repository procedure checked in by the first user and the one produced by the second user. This difference is the presence of the additional step Supply item '1.4' for Combo Box 'Source compatibility' only in the first check-in. This discrepancy can be explained in terms of differences between the environments: DABL infers a new decision point, B, that describes the conditions under which the 'Source compatibility' should be changed to 1.4 in the final version of the procedure model (Figure 7). When combining procedure models, DABL can learn new information on the structure of the task: this is a unique and distinctive characteristic of DABL that sets it apart from traditional version control systems and from the other learning algorithms used in PBD.

### Example 2

The second example shows detecting conflicts between a generated conditional and a code move, with user intervention to resolve the conflict. The original procedure, shown

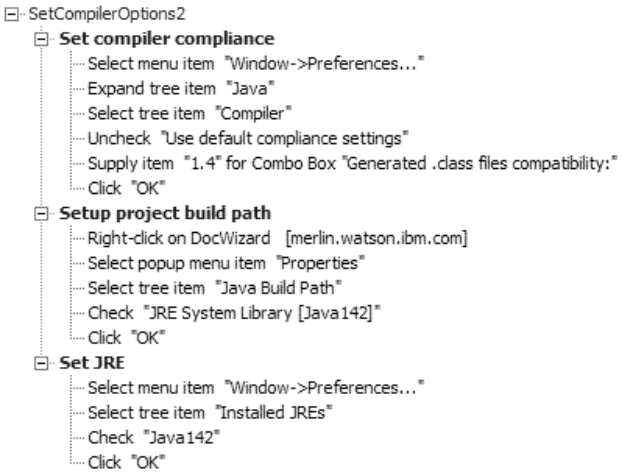


Figure 8: Ex. 2. Original repository procedure.

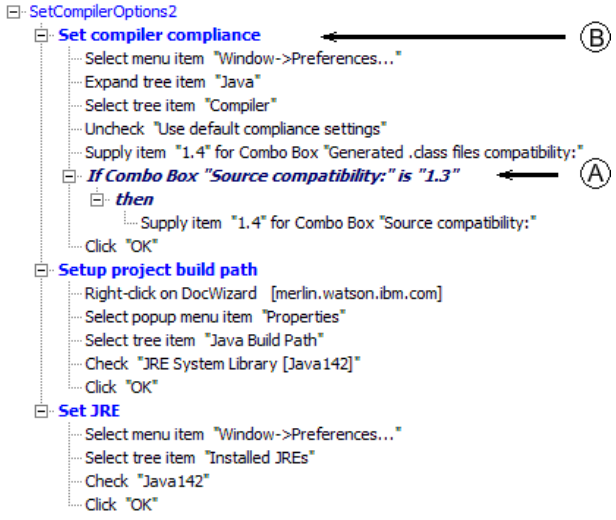


Figure 9: Ex. 2. The result of the changes introduced by the first user. Note the new decision point, A, and the unchanged position of the `Set Compiler compliance` subtask, B.

in Figure 8, consists of three subtasks - setting the generated .class file compatibility compliance for the workspace, setting up the build path for the DocWizards project, and setting the JRE for the workspace. Note that this procedure has been manually organized so that the recorded actions for each subtask are contained within a subtask group. The first user adds the same conditional described in example one to the first subtask, then checks the procedure (shown in Figure 9) into the repository. The second user independently moves the compiler compliance subtask at the end of the procedure. The resulting procedure, shown in Figure 10, is inconsistent with the repository version, since we now have independent edits that modified a subtask and also moved it.

Should the merge *both* move and edit the subtask, or should only one of those operations be included? We involve the user in making the decision through the reconciliation interface (Figure 11). A single conflict has been detected, and is presented in two parts: at the top DABL highlights

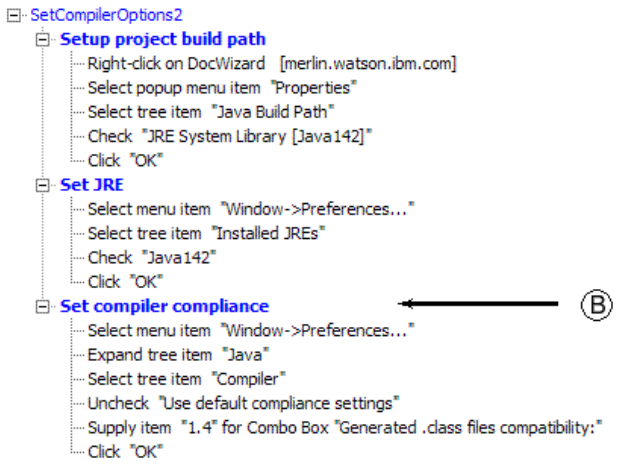


Figure 10: Ex. 2. The result of the changes introduced by the second user. Note the changed position of the `Set Compiler compliance` subtask, B.

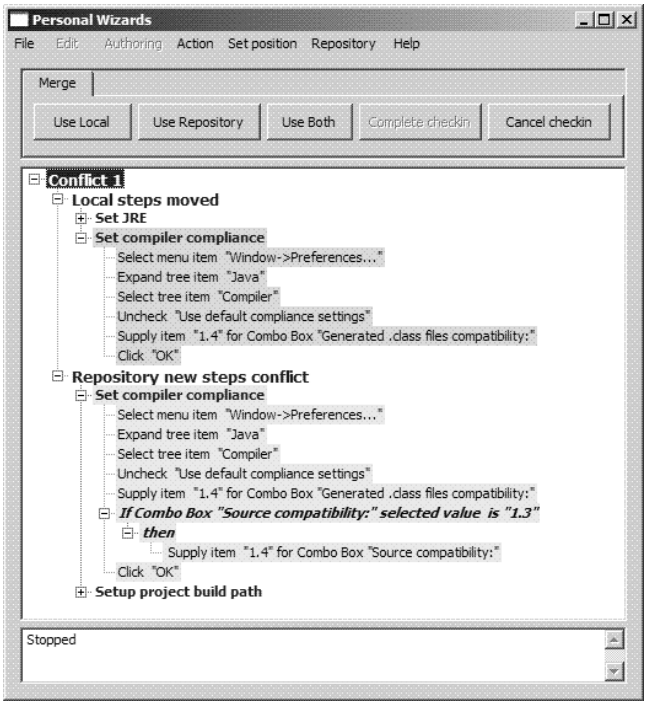


Figure 11: Ex. 2. The reconciliation UI with the conflict resulting from the changes shown in Figures 9 and 10. Note the three buttons labeled “Use Local”, “Use Repository”, and “Use Both” above the panel that shows the conflict. In the main panel at the top, DABL shows the procedure steps in the local version causing the conflict within their context and, at the bottom, the procedure steps in the repository version causing the conflict within their context.

the statements that were moved in the current local copy, and shows the surrounding context; below these DABL highlights the corresponding unmoved statements with the conditional from the repository copy (as checked in by the first user), and shows the surrounding context.

The user operates on this representation by selecting a partic-

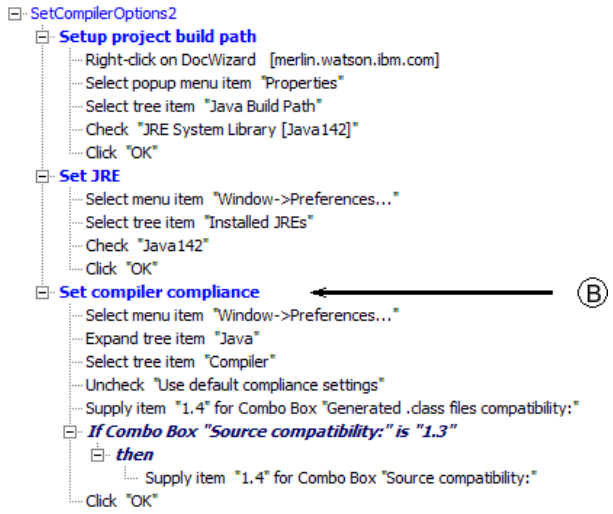


Figure 12: Ex. 2. The result of merging the conflict shown in Figure 12 when the user selects the "Use Both" option. Note that `Set Compiler compliance` subtask is in the same position as in Figure 10 (B), and that the decision point is incorporated into this subtask, as in Figure 9 (A).

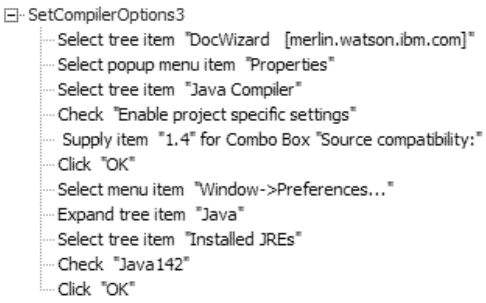


Figure 13: Ex. 3. Original repository procedure.

ular conflict, and then specifying what to do with it. In this example, the second user selects "conflict 1" (in this case, the only conflict), and then clicks the "Use Both" button to specify that DABL is to do a smart merge of her changes with those in the repository. The resulting procedure, checked into the repository, is shown in Figure 12.

### Example 3

The third example shows another merge conflict, in this case, a conflict between a code move and a code delete. The original procedure, shown in Figure 13, sets the compiler source compatibility for the DocWizards project, and then sets the JRE for the workspace. The first user moves the code that sets the compatibility below the code that sets the JRE to ensure that the compatibility level is based on the JRE selected, then checks in the edited procedure (Figure 14). The second user realizes that the procedure will be more generic if, instead of setting the compiler options for a particular project, they are set for the entire workspace. She deletes the section of the procedure that sets the compiler options for the project, then inserts statements that set the compiler options for the workspace (Figure 15). When the second user checks the edited procedure into the repository, the merge algorithm

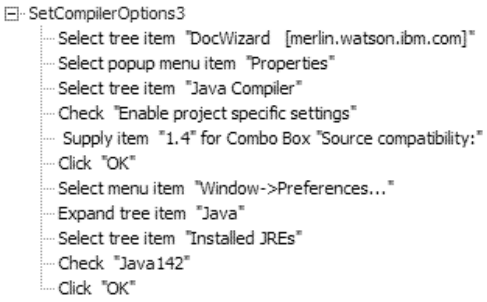


Figure 14: Ex. 3. Procedure edited by the first user.

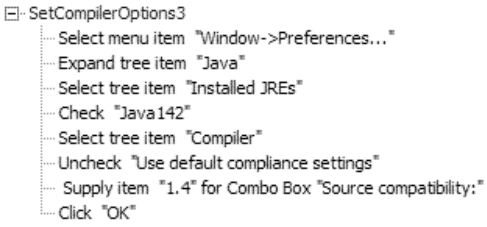


Figure 15: Ex. 3. Procedure edited by the second user.

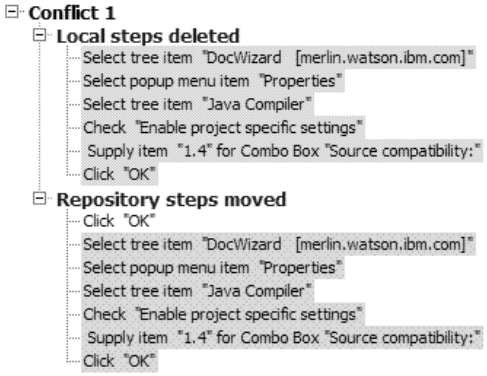


Figure 16: Ex. 3. Unreconciled statements.

is able to successfully merge in the additional statements that she has added. However the statements deleted by her are the same statements that the first user moved, so the merge algorithm is unable to reconcile the edits. The unreconciled statements are presented as shown in Figure 16. The user specifies that the local changes are to be retained, which causes the move in the repository to be discarded, and the result of the merge is the same procedure shown in Figure 15.

### Example 4

The last example demonstrated the ability of DABL to perform additional learning on a previously updated repository entry. The final check-in from example three is the starting point. A third user checks out the merged code, then adds an additional demonstration, resulting in the same conditional described in example one. The updated code is shown in Figure 17. Note that DABL is able to learn from additional demonstrations in exactly the same way as with "original" unmerged code.

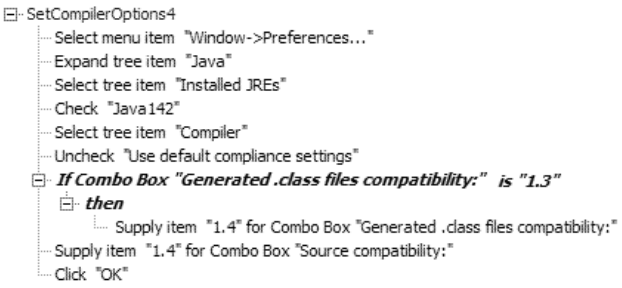


Figure 17: Ex. 4. Result of enhancing a merged procedure with demonstrations.

## CONCLUSIONS

We have presented DABL, the first incremental learning algorithm for distributed collaborative PBD.

DABL can sequentially learn the structure of a task and generalize predicates and action steps from multiple recordings of user actions. Additionally, DABL allows the user to directly edit the procedure, and seamlessly integrates the editing operations with the learning process.

These characteristics of DABL are not sufficient for creating procedure models by combining models produced separately by multiple experts. DABL supports distributed collaborative PBD by providing a version-control-like environment for merging separately enhanced procedure models. DABL supports the familiar and intuitive CVS paradigm: an expert checks out a procedure model from the repository, enhances it, and reconciles it with the repository version before checking in the changes.

What sets DABL aside from existing version control systems is the fact that reconciliation is a learning operation—while merging the repository procedure model and the user's version, DABL can infer structural information and logical predicates that are not obtainable from either model.

The merge operation consists of identifying conflicts between versions, automatically merging changes that do not result in conflicts while learning new structural and logical information, and asking the user to manually reconcile conflicts by accepting or discarding changes. This reconciliation paradigm lends itself to the construction of intuitive user interfaces, such as the one incorporated in the DocWizards system and used to generate the examples shown in this paper.

By supporting incremental, real-time learning from multiple demonstrations and manual edits, and by offering a novel learning mode integrated in the procedure reconciliation process, DABL is the first learning algorithm with the features required to support distributed collaborative PBD.

Future work includes usability studies. We are particularly interested in determining how to best deal with Type 2 conflicts with additional valid reconciliation alternatives beyond the three currently supported by the interface. Specifically, it is unclear whether providing all the alternatives would be more effective or more confusing than limiting the choice to the currently supported three alternatives and requiring the user to edit the reconciliation result. We also intend to study the benefits of previewing the results of the different alternative conflict resolutions.

### Acknowledgments

We would like to thank Apratim Purakayastha, Paul G. Crumley, Chandrasekhar Narayanaswami, and Siddhartha Chatterjee for their support of our work.

## REFERENCES

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

2. L. Bergman, V. Castelli, T. Lau, and D. Oblinger. DocWizards: a system for authoring follow-me documentation wizards. In *Proc. 18th annual ACM Symp. on User Interface Software and Technology, UIST2005*, pages 191–200, 2005.

3. L.D. Bergman, T. Lau, V. Castelli, and D. Oblinger. Personal Wizards: collaborative end-user programming. In *Proc. of CHI2003 Workshop on Perspectives in End User Development*, Fort Lauderdale, FL, Apr. 5–10 2003.

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

5. Eclipse platform technical overview . http://www.eclipse.org/articles/index.php?filter= whitepaper, July 2001.

6. T. Lau, L.D. 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, D. Oblinger, L.D. Bergman, V. Castelli, and C. Anderson. Learning procedures for autonomic computing. In *Proc. of Workshop on AI and Autonomic Computing: Developing a Research Agenda for Self-Managing Computer Systems, IJCAI 2003*, Acapulco, Mexico, Aug. 9–15 2003.

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

9. D. Oblinger, V. Castelli, and L.D. Bergman. Augmentation-based learning, combining observations and user edits for programming-by-demonstration. In *Proc. 2006 Int. Conf. on Intelligent User Interfaces*, pages 202–209, February 2006.

10. D. Oblinger, V. Castelli, T. Lau, and L.D. Bergman. Similarity-based alignment and generalization. In *Proc. Sixteenth Europ. Conf. on Machine Learning*, pages 657–664, Berlin, October 2005. Springer.

11. Per Cederqvist, et al. Version management with cvs (for cvs 1.11.6), http://www.cvshome.org/docs/manual/, 1993.