Recovering from Errors during Programming by Demonstration

Jiun-Hung Chen and Daniel S. Weld
University of Washington
Seattle, WA 98195, USA
{jhchen,weld}@cs.washington.edu

ABSTRACT

Many end-users wish to customize their applications, automating common tasks and routines. Unfortunately, this automation is difficult today — users must choose between brittle macros and complex scripting languages. Programming by demonstration (PBD) offers a middle ground, allowing users to demonstrate a procedure multiple times and generalizing the requisite behavior with machine learning. Unfortunately, many PBD systems are almost as brittle as macro recorders, offering few ways for a user to control the learning process or correct the demonstrations used as training examples. This paper presents CHINLE, a system which automatically constructs PBD systems for applications based on their interface specification. The resulting PBD systems have novel interaction and visualization methods, which allow the user to easily monitor and guide the learning process, facilitating error recovery during training. CHINLE-constructed PBD systems learn procedures with conditionals and perform partial learning if the procedure is too complex to learn completely.

ACM Classification D.2.2 [**Design Tools and Techniques**]: User Interfaces, H1.2. [**Models and principles**]: User/Machine Systems

General Terms Algorithms, Human Factors

INTRODUCTION

The constantly-growing complexity of software applications and greater diversity in the way that people use these applications is increasing the desire of users to customize these applications and their interfaces.

Programming by demonstration (PBD) is one promising customization technique; previous work has resulted in both successful applications, a general machine-learning framework for the problem, and an understanding of the expressiveness / sample-complexity tradeoff [7, 21, 22]. However, several problems remain with most PBD systems:

- **Considerable Domain Engineering:** Substantial domain engineering is required to modify PBD systems to work with a new application. This need to have a human 'in the loop' acts as a barrier to the creation of integrated PBD across the whole suite of a user's applications.

IUI'08, January 13-16, 2008, Maspalomas, Gran Canaria, Spain.

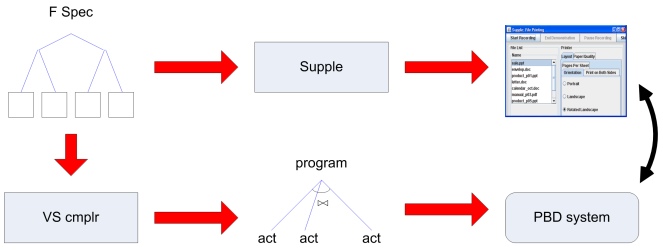

Figure 1. CHINLE **generates a PBD system for an application from the** SUPPLE **functional specification of its interface.**

- **Inscrutability of the Learning Process:** It is often hard for users to understand or influence a PBD system's learning process. What actions are being predicted with what confidence? What hypotheses are being considered? If a hypothesis has been discarded by the system, why did that happen and because of which data?

- **Difficulty Recovering from Training Errors:** A primary motivation for PBD systems is the realization that macro recorders are brittle, but many PBD systems are likewise intolerant of user errors. If a user errs during demonstration, it can be hard to recover from the mistake — without restarting training from scratch.

- **All-or-Nothing Learning:** Sometimes a user's intended procedure is too complex to be automatically learned; in this case, most PBD systems fail to learn anything, rather than learning a partial procedure and presenting it as a wizard. For example, when processing photos, a user might load them into Photoshop, enhance saturation and contrast, resize, save the result as a jpeg, post it into blogging software, and write a caption. Automating this process is impossible today, because no PBD system works across applications. Furthermore, since one typically selects different saturation settings for each picture, PBD systems can't automate that command (or write the caption), but they shouldn't give up. Instead, the PBD system should learn a *partial procedure*, which automates what it can and allows the user to speedily focus on the parts requiring human judgement.

This paper describes CHINLE a novel system which addresses these challenges, automatically constructing PBD systems for an application program from its high-level interface de-

scription (Figure 1). While previous researchers have tackled some of the problems before, CHINLE confronts all problems in concert, introduces new interaction methods, and (we believe) it is the first to use partial learning to create wizards. The resulting PBD systems allow users to quickly detect and correct errors made during training.

The remainder of this paper is organized as follows. First, we explain how CHINLE reuses an application's SUPPLE interface specification to generate the version-space algebraic description necessary for learning. Second, we define a novel probabilistic bias, which guides learning. Third, we introduce the interface, which lets users visualize the state of CHINLE's learning progress, its confidence in hypotheses and in predictions; novel interactors allow the user to quickly recover from errors which might occur during the demonstration process. Some of the recovery techniques lead to interesting learning problems: learning from incomplete data and partial learning from inconsistent traces. Finally, we review related work, state our contributions, and highlight future work.

AUTOMATIC VERSION-SPACE GENERATION

In Lau et al.'s SMARTEDIT system, the human designer manually specified the version-space (VS) algebraic description, which guides learning. In contrast, CHINLE generates the VS automatically from the application's declarative interface specification. We first describe this specification language and then explain how CHINLE automatically transforms these descriptions into version spaces, applying a novel, probabilistic bias.

Specifying Interfaces

We build upon the open-source SUPPLE model-based interface-generation toolkit [11], which uses decision-theoretic optimization to render interfaces in a device-independent manner (Figure 1). For the purposes of this paper, however, the only pertinent aspect of SUPPLE is its functional interface specification (FS) language, described below.

Following earlier work on model-based UIs [31, 13, 27], SUPPLE represents an interface *functionally*, *e.g.*, specifying *what* capabilities the interface should expose, instead of *how* to present those features (SUPPLE's optimization algorithm makes these rendering decisions). Formally, an interface comprises a pair, $\langle \mathcal{G}, \mathcal{C} \rangle$, where $\mathcal{G}$ is a directed acyclic graph of typed *elements* and $\mathcal{C}$ is a set of constraints. Elements correspond to units of information that need to be conveyed via the interface between the user and the controlled appliance or application. Each element is defined in terms of its type. *Primitive types* include integers, floats, strings, Booleans, dates, times, images, *etc.* *Container types*, akin to Pascal records, are used to create groups of simpler elements.

As a running example, consider the functional specification of a Windows-style printer control panel (Figure 2). Interior nodes, such as paper quality, are represented as container types, while leaves are primitives, such as the switch for color *vs.* black and white printing, which is represented as a Boolean. SUPPLE renders this specification as shown in Figure 3.

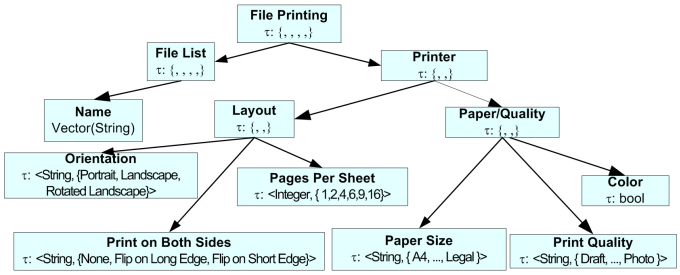

Figure 2. Part of the functional specification (FS) for a printer control panel.

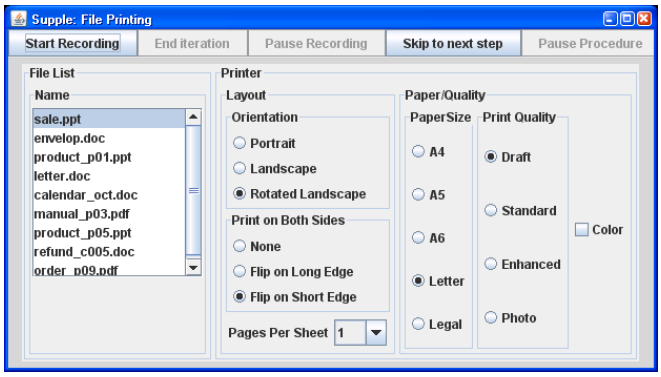

Figure 3. PBD control menu atop SUPPLE's rendering of the printer control panel defined in Figure 2.

Version-Space Algebra

CHINLE compiles functional specifications into a version-space algebraic description for learning. Before describing the compilation process (next subsection), we briefly review VS algebra. A *hypothesis* is a function h that takes as input an element of its domain I_h and produces as output an element of its range O_h. A *hypothesis space* is a set of functions with the same domain and range. The *bias* determines which subset of the universe of possible functions is part of the hypothesis space; a stronger bias corresponds to a smaller hypothesis space. We say that a hypothesis, h, is consistent with a training example (i, o), for $i \in I_h$ and $o \in O_h$, if and only if $h(i) = o$. A version space, $\text{VS}_{H,D}$, consists of only those hypotheses in hypothesis space H that are consistent with the sequence D of examples [26]. When a new example is observed, the version space must be updated to ensure that it remains consistent with the new example. For example, we may define a *ConstInt* version space containing functions of the form $f(x) = C$ for every integer value, C. Given a training example (input: 0, output: 4), the *ConstInt* version space is updated to contain only the function $f(x) = 4$. Further training examples will either be consistent with this version space's single hypothesis, or cause the version space to *collapse*, i.e., contain no hypotheses. This simple version space becomes more powerful when used as a building block in version space *algebra*, which is a method for composing together version spaces to build a complex version space out of simpler parts. The two most important operators are:

- **Union:** combine two or more version spaces to form one space containing the union of the functions in the member spaces, and

- **Join:** combine two or more version spaces to form one space containing the cross product of the functions in the member spaces, subject to a consistency predicate.

Continuing our example, when modeling an n-step program, one uses a join of n actions (Figure 4). Each action can change the value of any parameter in the control panel, and is hence represented as the union of all parameter values. CHINLE considers a wide range of individual hypotheses for each of these parameters. In particular, one possible representation for *String* is a conditional, which is represented as a join of a version space for the condition being tested, and values for the true and false branches [18].

Lau's framework allows the application designer to specify a preference bias by defining a probability distribution over the hypotheses in the hypothesis space. CHINLE uses a particularly useful distribution to weight hypotheses, as we describe in a later section.

A version space is *executed* on an input, i, by applying every hypothesis in the version space to the input and collecting the set of resulting outputs. Formally, after a version space is executed on an input i, a set of outputs $o_1, o_2, ..., o_n$ is generated such that $o_j = h_j(i)$ for some h_j in the version space. Given a probability distribution over the hypotheses, one can then compute the maximum-probability prediction, by summing the probabilities of all hypotheses that agree, $h_j(i) = h_k(i)$, and choosing the o with the greatest probabilistic support.

Lau et al. showed that in many cases, version spaces may be represented efficiently by constraints on the set of consistent hypotheses, such as the boundaries of the set relative to a partial order (one that is convex and definite [12], but not necessarily the generality ordering [19]). The next section explains how CHINLE automatically generates a VS, describes how conditionals are handled, and defines our probabilistic bias.

Recursive-Descent Transformation

Since the user explicitly segments the trace into fixed-length iterations, the number of actions (say n) in the program is known before learning begins.[1] CHINLE creates the version space for the whole program by joining n identical action version spaces.

An individual action version space is computed from the functional specification tree [2] with a simple depth-first traversal as shown in Algorithm 1. The real generalization power comes from the CHINLE's treatment of the leaves in the functional specification, which are each assigned a type-specific version space by the TypedVS routine. For all data types, these base-case version-spaces contain a union of several possible hypotheses, some of which are conditioned on previous actions and others which are unconditional.

Algorithm 1 FS_2VS(**FS-tree** T): **version-space**

```
1: if T is a single leaf then
2:    return TypedVS(T.type)
3: else
4:    for all immediate subtrees S in T do
5:       return ⋃_S FS_2VS(S)
6:    end for
7: end if
```

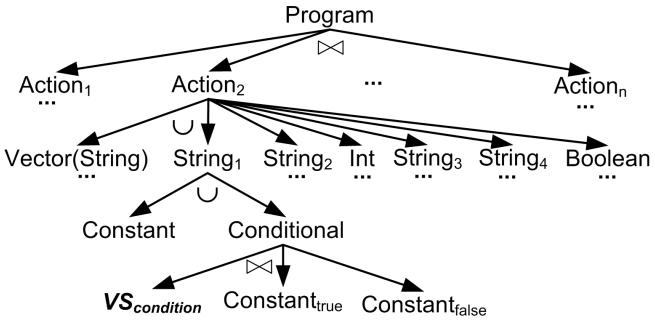

Figure 4. A portion of the automatically generated version space for the printer control panel. For clarity, only one action node is expanded. Note that all string and Boolean nodes would have constant and conditional hypotheses beneath them. The integer and vector nodes would also have delta hypotheses.

For example, consider an integer-valued state variable, such as pages-per-sheet in our printer (Figure 2) or the index into the vector of file names. These variables will be assigned a version space which is the union of the following:

- A **constant value**. This hypothesis would be appropriate if the user selects the same number of pages per sheet each job.

- A **constant delta** — useful if the user linearly increases (or decreases) the value on each iteration. This probably wouldn't make sense for pages per sheet, but would be applicable if the user sequentially prints the files in a directory.

- A **conditional statement** of the form "if *cond* then $value_1$ else $value_2$. We restrict the condition of each rule to a single equality (no logical connectives) comparing the value of some other state variable. A hypothesis of this form would enable the PBD system to learn that if the file type is `.ppt`, then pages-per-sheet should be four, otherwise it should be one.

Other data types (strings, enumerations, etc) afford the same types of hypotheses, with the exception of deltas which require an addition operator and so only apply to numbers (and vector indices).

[1] In fact, the user may add new steps in subsequent iterations and CHINLE will revise the length of learned procedure, but at any time CHINLE knows the current length.

[2] We believe it is straightforward to extend CHINLE to use other specification languages such as Nichols' Pebbles [27], Windows resource files, AppleEvents, or similar semantic descriptions, but we have not yet done this extension.

Figure 4 shows the structure of the version space generated for the printer example. Note that, if fully expanded, the space is quite large. Since many of these hypotheses will be inconsistent with the trace's first iteration, CHINLE ensures efficiency through lazy evaluation; it doesn't actually generate the whole space until it has received the first trace iteration, at which point it can avoid generating many inconsistent elements.

PROBABILISTIC WEIGHTING FOR DISAMBIGUATION

Recall that a version space is simply the set of possible programs (hypotheses) which are consistent with all evidence seen so far. Learning would be simple if the user supplied precisely enough trace data to remove all but a single program from the version space; execution would then be simple. But what should happen if there are multiple consistent programs? In this case, CHINLE iterates through every consistent hypothesis, recording the action which each predicts. If there is disagreement, then CHINLE has the hypotheses vote to determine the actual course of action (following [19]).

But not all hypotheses are equally likely. Occam's Razor suggests that unconditional hypotheses should be preferred over conditional ones. Furthermore, a hypothesis conditioned on the value of the *most recently changed* variable is more likely than a hypothesis which is conditioned on a variable whose value has never been modified.[3] To model these preferences, CHINLE uses a novel probabilistic weighting scheme.

At version-space unions for functional-specification *interior nodes*, a uniform distribution is used. Unions corresponding to *leaf nodes*, i.e. primitive types, give equal probability, P, to each unconditional hypothesis as they do the group of *all* conditional hypotheses. The relative probabilities of conditional hypotheses are normalized based on how recently their conditioning variable has been changed, using an exponential backoff. Suppose the action in question is the $(n+1)^{\text{st}}$ action in the procedure being learned; then up to n variables have been modified previously. If a hypothesis is conditioned on a variable which was changed t steps previously, it is assigned a probability of $\frac{P}{2^t}$. All k hypotheses whose conditioning variables *haven't* changed are assigned the weight $\frac{P}{k2^n}$.

Note that this system of probabilistic weighting means that if the user demonstrates only a single iteration of the procedure, then CHINLE's voting will be dominated by the constant hypotheses, and execution will be identical to that of a standard macro.

VISUALIZING SYSTEM CONFIDENCE

To elaborate our running example, suppose that Sue regularly prints files of different formats sent from her European clients. She typically changes the paper size, paper per sheet and color settings according to the properties of the file to be printed. Because her clients are European, she often uses `A4` paper; however, she prefers `Letter` size for her own documents. She prefers `Pages Per Sheet` to be `1`, except for PowerPoint presentations, where she uses `4`. Unless the file is in color, she prefers the cheaper `Black-And-White` setting. Sue finds it annoying to manually change these printing settings, since she often forgets and wastes paper. So, she wants to automate this repetitive task. Since she needs to print `sale.ppt`, she starts the PBD recorder and changes `Paper Size` from `Letter` to `A4`. Next, she changes `Pages Per Sheet` from `1` to `4`. Finally, she presses `End Demonstration` on the PBD menu to stop recording.

[3]To see this, note that the user has already signified the relevance of any variable she recently changed, whereas there will always be *numerous* irrelevant variables.

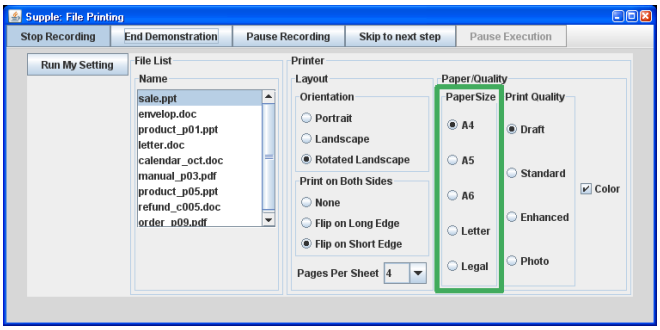

Figure 5. When performing successive demonstrations, CHINLE guides the user to ensure that actions are aligned. Here, the system expects the user to set the `Paper Size` to `A4`. The color of the highlight corresponds to the confidence of the system's prediction; dark green means very confident.

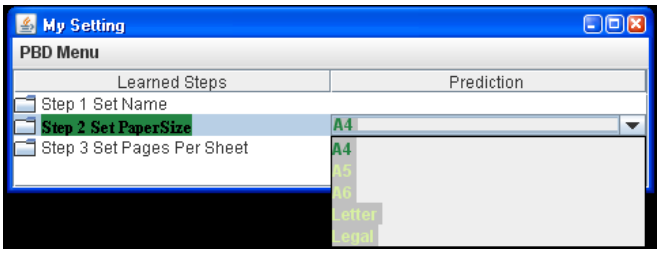

Figure 6. A simple viewer lets the user see a summary of the learned procedure, color coded to denote the system's confidence in each step with darker green indicating higher probability. If the user clicks on the predicted action, alternatives are shown; selecting one adds another training example.

A bit later, Sue needs to print `envelop.doc` so she resumes recording. Since the PBD system knows the rough flow of actions, it guides her to ensure that the actions in the two demonstrations are aligned. It does this by highlighting the expected step[4] (set `Paper Size`) and filling in the anticipated value `A4`; see Figure 5. To indicate the system's confidence in its predictions, CHINLE uses a 6-level sequential color scheme [5], generated by COLORBREWER; its confidence in `A4` is very high (72%), which maps to dark green.

Recovering by Adding a Missing Step

[4]Note that highlighting was similarly used to guide users in PBD systems such as Eager [7, Chapter 9] and DocWizards [2].

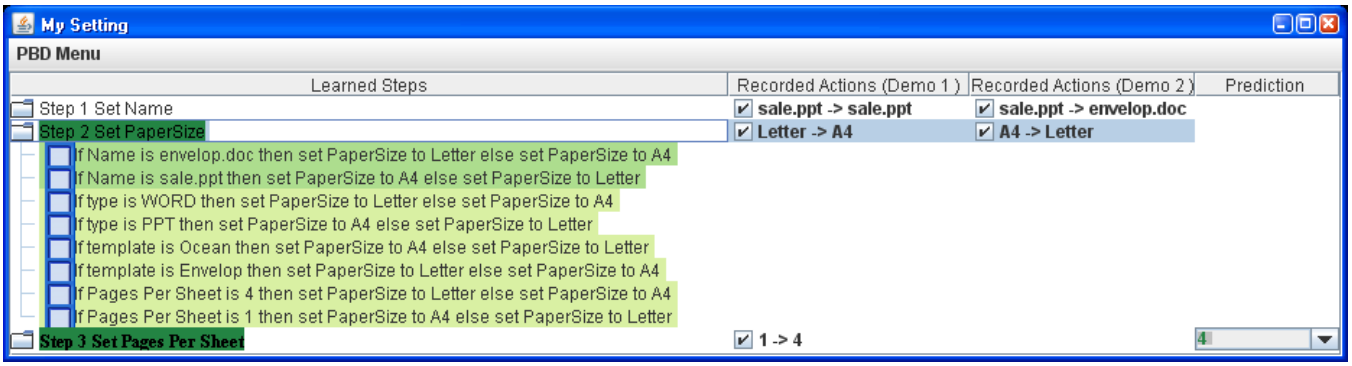

Figure 7. If the user desires, they can inspect the state of the learning process. Additional columns show the training data for each demonstration, and by expanding a step (e.g. Step 2) we see an ordered list of hypotheses, color-coded to mark CHINLE's confidence.

Sometimes a user forgets to execute a step during a demonstration. Many PBD systems fail when actions don't align, requiring the user to reinitiate demonstration. In contrast, CHINLE uses a technique like that of DOCWIZARDS [2] to continue learning.

In our example, Sue realizes that the system is erroneously set to print `sale.ppt` again. To correct this, she selects `envelope.doc` instead. The key point is that while CHINLE tries to align actions with those from a previous demonstration, it is flexible and allows new actions to be added at any point in time — even if this changes the length of the program. In this case, CHINLE adds a matching action to the first demonstration explicitly selecting `sale.ppt`.

At this point Sue inspects the partially-learned procedure in the viewer (Figure 6). CHINLE continues to wait for a `Paper Size` setting, and by clicking on the `A4` prediction, Sue can see the alternative predictions, whose color indicates low probability. In fact, Sue wants to choose `Letter` size, which she can do in several ways: 1) she can select this option in the viewer pulldown (Figure 6), or 2) she can simply click on `Letter` in the application interface (Figure 5). If she had instead wished to keep the `A4` setting, she could have clicked `Skip to Next Step` (top of Figure 5).

Inspecting the Learning Process

Figure 6 provides a high-level summary of the learned procedure, but sometimes a user wants to understand the learning process in more detail. In this case, one may switch to an expanded view as shown in Figure 7. The first column shows the partially-learned procedure using a two-level forest (set of trees) depiction. The leaves of each tree denote hypotheses, sorted and colored by their estimated probability. In the figure, only the hypotheses of Step 2 can be seen — the others are minimized. The root nodes in the forest represent steps in the program. As in the simple view (Figure 6), a step node is colored to indicate confidence in the step's prediction — the intensity of the green is proportional to the probability mass associated with the most likely prediction, summing the weighted votes of all hypotheses. Note that this means that a step may have high confidence even if none of the underlying hypotheses do.

The rightmost column of Figure 7 shows the system's prediction, as before. In this case, CHINLE is confident that Sue will want `4 Pages per Sheet` in Step 3.

The two middle columns of the figure show the actions recorded in the two previous demonstrations. These actions are represented as state transitions, $A \rightarrow B$, where A is the old value of the state variable and B is the new value.[5]

CORRECTING DEMONSTRATION ERRORS

Sometimes a user may make a mistake while demonstrating a procedure, and CHINLE provides methods to recover from such mistakes without starting training again from scratch.

Consider Figure 7 and note that a checkbox precedes each recorded action in the demonstration columns. By unchecking (or rechecking) this checkbox, a user can retract (or reassert) a training example. For example, if when looking at the expanded procedural visualizer, Sue realizes that she has incorrectly selected `Letter` in Step 2 of Demonstration 2, she may simply uncheck that box.

Learning from Incomplete Data

Such a change initiates incremental learning. Unfortunately, the resulting learning problem is complicated, because the state of that variable is now unknown at that time. This is a problem because several types of hypotheses are conditioned on the values of other variables. For example,

- A delta hypothesis predicts that the next value of a variable will be its old value plus some constant c, but this prediction is undefined if the old value is unknown. For example, the system might conjecture that the next file to be printed is the next one listed in a directory, but how can it evaluate the hypothesis if it doesn't know the last file printed?
- Conditional hypotheses (e.g. "If file `Type` is `WORD` then set `Paper Size` to `Letter`") reference the values of

[5]Note that the first demonstration of Step 1 has a null transition `sale.ppt -> sale.ppt`; indeed, Sue skipped this step in the first demonstration. A user can similarly indicate that the current value of a widget is ok in subsequent demonstrations by clicking the `Skip to Next Step` button.

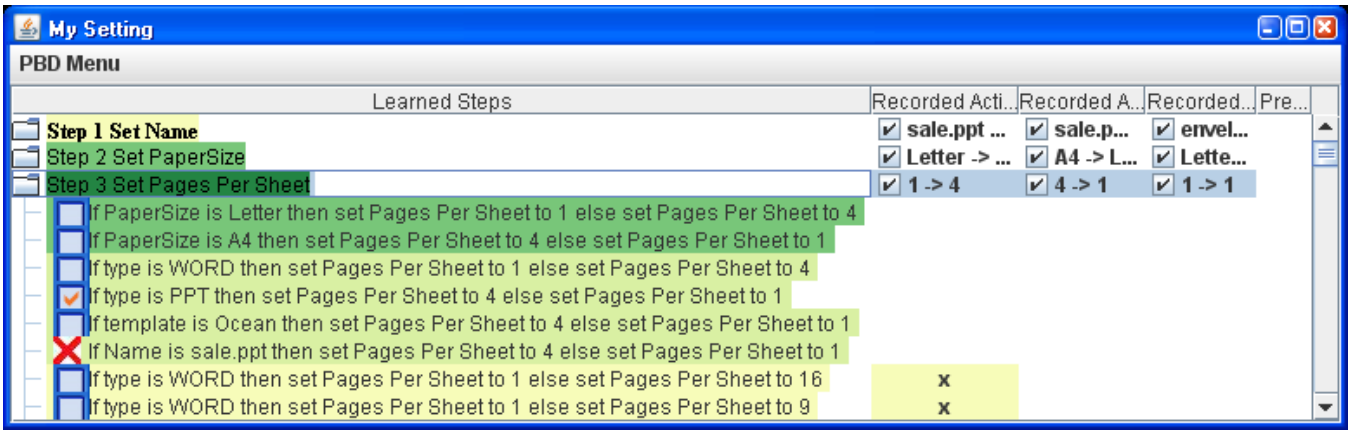

Figure 8. By crossing a hypothesis' box, the user may reject it from consideration. In contrast, a check tells CHINLE that the user has manually selected the hypothesis, overriding all others. Normally, CHINLE doesn't display inconsistent hypotheses, but in this view the user has requested to see them; note that each is associated with yellow shading in one or more "Recorded Action" columns, indicating which previously demonstrated actions conflict with the hypothesis' prediction.

other variables. How can CHINLE evaluate this hypothesis if it doesn't know the file being printed, or its type?

The simplest way of allowing users to retract actions from the training data is to remove all actions after the one which has been retracted. This makes learning easy, because the training data again forms a continuous sequence of fully-specified actions; however, it wastes valuable training data.

Instead, CHINLE treats retracted actions as *missing values* during learning. The key question is how to define the consistency of hypotheses defined in terms of missing values. Borrowing a technique from model-based diagnosis, CHINLE uses *constraint suspension* [8]; all constraints referencing a missing value are automatically considered consistent. For example, a conditional hypothesis is considered consistent with a state transition, $A \rightarrow B$, if the value of the hypothesis antecedent is missing, A is missing, or B is missing.

Manual Selection of Hypotheses

Sophisticated users may wish even more direct control over the learning process. For these users, CHINLE provides a check box to the immediate left of each hypothesis. As can be seen from Figure 8, hypothesis boxes have three possible states:

- **Blank.** This is the default state, indicating that the normally computed probabilistic weighting is in effect.
- **Crossed.** This indicates that the user has manually rejected this hypothesis and it will not vote during prediction or execution. Its color is still automatically updated by CHINLE to show what the probability *would be* if the cross were removed.
- **Checked.** This indicates that the user has manually selected this hypothesis, overriding all other unchecked possibilities. Again, CHINLE still computes the probability and sets the color.

Understanding why Hypotheses are Inconsistent

Figure 8 illustrates additional aspects of CHINLE's visualization and interactions. Normally, CHINLE doesn't show hypotheses which conflict with the training data, but the user may request to see inconsistent hypotheses — which are colored yellow. Each row corresponding to such an inconsistent hypothesis has one or more demonstration columns shaded as well; this indicates which training examples are inconsistent with the demonstrated action. The user may manually retract the conflicting training example if she decides that it was a mistake. However, the user may instead just select an inconsistent hypothesis if she so desires.

PARTIAL LEARNING FROM INCONSISTENT TRACES

Because it uses version-space algebra, CHINLE handles inconsistent training data differently from most machine-learning systems, which strive to be tolerant of noisy data. Systems based on decision trees [29], HMM derivatives [16], support-vector machines, or similar methods continue to predict the highest probability action, even when the user's actions contradict all representable hypotheses.

While noise-tolerant methods are crucial for many learning tasks, we believe PBD is the exception. It is especially important for a PBD system to be predictable, so the user can trust that the system will act correctly when executed. Thus, the learner should be conservative, communicating its confidence at every step and admitting when it is confused. CHINLE's probabilistic color scheme achieves the former and the version-space representation ensures the latter. If the user ever demonstrates a set of actions which, taken together, contradict every hypothesis, then the version space collapses and CHINLE notifies the user, giving her a chance to recover by retracting actions, manually selecting a hypothesis, or accepting partial learning. We have already explained the first two options; we now describe partial learning.

Continuing our example from the situation depicted in Fig-

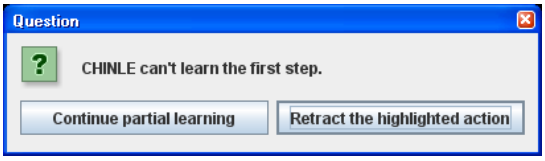

Figure 9. If the version space collapses for a step, the user is given the option of correcting a possible demonstration error, selecting an inconsistent hypothesis, or proceeding with partial learning — leading to a mixed-initiative wizard, which automates part of the procedure.

ure 7, suppose Sue completes the second demonstration by selecting a single page per sheet. Later, when she starts recording a third demonstration, CHINLE confidently predicts that she will print `product_p01.ppt`, because it has matched a delta hypothesis and CHINLE thinks she is iterating through the complete set of files, printing each one. In fact, the system is wrong, and Sue wants to print `iui.doc`. When Sue selects this value, however, the version space for Step 1 collapses — unsurprisingly, none of CHINLE's hypotheses can predict which file Sue wants on paper. CHINLE pops up a warning message (Figure 9) and asks whether the system wants to correct an error or continue with partial learning.

SMARTEDIT [19] also used version-space algebra for PBD, but when confronted with a version-space collapse, SMARTEDIT just reported failure to the user, who could then provide a new trace. In contrast, CHINLE exploits the fact that the algebraic description (illustrated as the tree in Figure 4) factors the version space wherever there is a VS join operator ($\bowtie$). Specifically, if the sub-version-space for any action collapses, the VS for the whole program collapses. But such a case just means that CHINLE can't predict *one* of the actions — it might have a single, clear hypothesis for each of all other steps in the program. In such a case, CHINLE learns a *partial program*. When executed, CHINLE performs as many of the actions as it can, but uses a wizard (or *mixed-initiative* interface) to ask the user to guide the actions which it was unable to learn.

CHINLE supports two types of wizards. Figure 10 illustrates an *in-situ* wizard — the original control-panel interface is used to prompt the user to perform parts of the procedure by herself, while the system automates the rest. The advantage of this approach is its familiarity. Not only does the user get to use a well-known interface to guide the new procedure, but the prompting metaphor is adopted from the guide used during the PBD training phase.

In some cases, however, an in-situ wizard is inappropriate. Suppose, for example, that less screen real-estate were available to render the printer control panel, and as a result it looked as shown in Figure 11. Furthermore, suppose that Sue trained a slightly more complex procedure which used color printing for some documents and not for others. In addition to the version-space collapse for `File Name` CHINLE wasn't able to learn a deterministic rule for `Color`. It might confuse the user to use an in situ wizard to ask for these values, since it would require switching tabs and the resulting

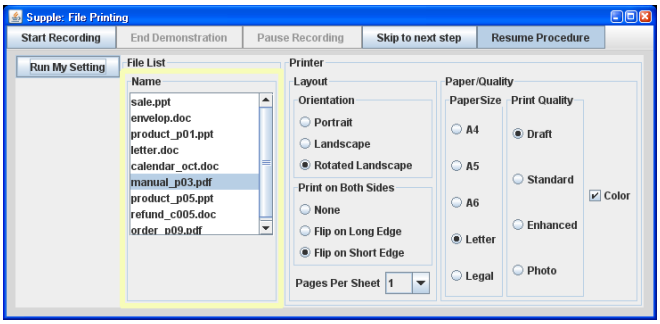

Figure 10. An *in situ* wizard interface which uses the original interface to prompt for necessary values.

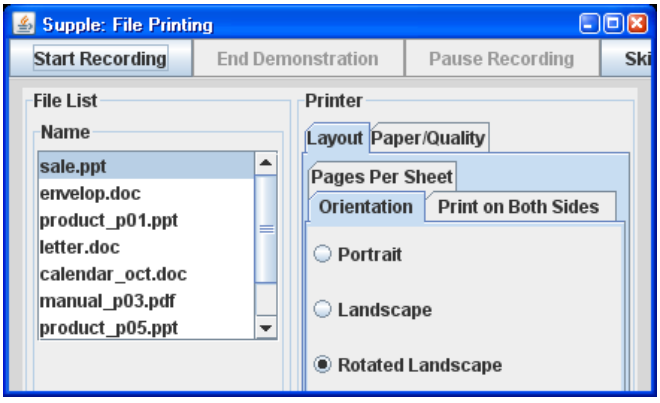

Figure 11. An alternate rendering of the printer control panel, which occupies less screen space. Because functionality is hidden behind tab panes, it might be confusing to use this interface to prompt for unlearnable values when a partially-learned PBD procedure is executed.

activity would deprive the user of a feeling of control.

Instead, CHINLE generates a *traditional* (Microsoft-style) wizard (Figure 12) which prompts for all necessary values and then executes the procedure autonomously.

RELATED WORK

General PBD reviews can be found in [7, 21]. In the following, we discuss related work from three different viewpoints: automatic construction of PBD Systems, use of general machine learning algorithms, and user control of the learning process.

Automatic Construction of PBD Systems

Little work has been done on the automatic generation of PBD systems, but Frank and Foley's early work [10] is an important exception. Unfortunately, the PBD systems it was able to generate had limited generalization capabilities, focusing only on the size and placement of 2D objects.

Piernot and Yvon's AIDE system [7, Chapter 18] provides developers a system substrate for creating application-independent PBD so that developers can add advanced macro capabilities to Smalltalk applications without re-implementing everything from scratch. In particular, AIDE adopts a hierarchical representation of the application history and AIDE

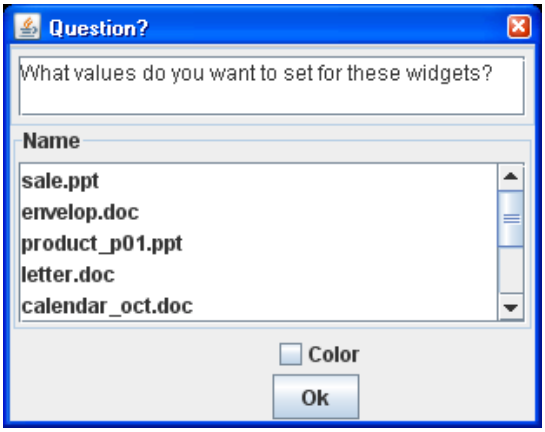

Figure 12. A *traditional* wizard interface which pops up as a separate window to prompt for necessary parameter values. This form of mixed-initiative interface is more appropriate if the main interface (Figure 11) uses tab panes.

generalizes arguments in different events of the same class. AIDE can detect loops but does not learn conditionals or partial procedures. FAMILIAR [21, Chapter 15] is an integrated PBD system for Apple computers, which creates domain independent programming by demonstration with the AppleScript language.

General Machine Learning Approaches to PBD

Many PBD systems are essentially rule-based expert systems, but some researchers have used a general machine-learning framework. Our work is based on Lau et al.'s framework [17, 19] and exploits their compact, factored representation of the version space. In Lau et al.'s SMARTEDIT system, the designer manually specified the VS algebraic description, but CHINLE generates it automatically from the application's functional specification. We also extend their framework by 1) learning partial procedures, 2) learning from incomplete training data, and 3) using a novel probabilistic weighting scheme, and 4) providing a rich visualization scheme for controlling the learning process and recovering from training errors.

FAMILIAR [29] applies decision-tree learning and then combines different predictions with a meta-learner trained on off-line training data provided by an experienced user. In contrast, CHINLE uses probabilistic execution to combine different hypotheses and requires no off-line training data.

Several researchers use fully-probabilistic learning approaches, such as IOHMMS [16], and Relational Markov Models [20], but these approaches are so robust to noise that they may incorrectly generalize a user's actions rather than indicating an inconsistency in demonstration — potentially surprising the user. We believe that the ability to detect a version-space collapse is a crucial benefit of Lau and our approach. In other words, a PBD system should predict a user's intent with extreme confidence — not just 51% probability. CHINLE's color-mapped visualization of probability unobtrusively communicates the system's confidence in its predictions.

One limitation of CHINLE is its requirement that the user manually segment the execution trace into demonstrated segments of fixed length. Other methods, such as IOHMMs [16], Augmentation-Based Learning [28] and Distributed Augmentation-Based Learning [6] relax this assumption, solving the alignment and generalization problem.

To the best of our knowledge, partial learning has not been discussed in the PBD literature, and no one has tackled missing-value problems in the context of PBD. Of course, missing-values have been extensively studied in the broader statistics and machine learning fields. For example, [25] is a classic reference on incomplete data in statistics, and [30] explains how C4.5 deals with missing attribute values.

In addition to machine learning, researchers have suggested using common-sense knowledge bases to power PBD generalization. For example, Creo [9] is a PBD web browser, which allows users to create a general-purpose procedure from a single example; Creo's generalization ability comes from ConceptNet and TAP.

User-Control of the Learning Process

In order to control automatic generalization of procedures, users must understand the programmatic representations. Ko and Myers's WHYLINE [15] answers users' "Why?" and "Why not?" questions; they show that this capability greatly helps users, reducing debugging time. A recent study on supporting end-user debugging [14] shows that the greatest need for explanation falls in the Oracle/Specification category: figuring out whether a value was right or wrong and how to fix values. The DOCWIZARDS [2] system addresses the understandability of learned procedures, allowing human annotation and integrating with documentation. CHINLE's ability to determine which demonstrated actions contradict a hypothesis might enable a similar facility.

Many systems ask users to help disambiguate the generalization process by selecting one of several consistent hypotheses [7, 19]; however, we aren't aware of previous PBD systems which allow users to select *inconsistent* hypotheses as a form of error recovery. Nor are we aware of a prior PBD system which uses sequential color-coding [5] to indicate confidence in its predictions.

Witten and Mo's TELS system [7, Chapter 8] supported mixed-initiative PBD procedures. If TELS makes a mistake, the user is invited to enter a debugging phase which reverts to learning mode. Active-learning approaches, such as that of Wolfman *et al.* [32], use the system's understanding of its uncertainty to proactively determine which question might best be asked of the user in order to speed learning. We wish to incorporate these ideas into CHINLE. On the other hand, partial procedure learning can be helpful for mixed-initiative PBD systems because the time of version-space collapse is likely a good opportunity to ask the user for help.

Augmentation-Based Learning [28] and its extension, Distributed Augmentation-Based Learning (DABL) [6], support learning from multiple traces, demonstrated by disparate users.

Furthermore, they allow users to directly edit the learned procedure and can learn from both traces and users' edits. CHINLE also allows the user to directly modify a procedure, but through a different interaction method — selection or rejection of a specific step hypothesis. DABL supports a greater range of programmatic control, which is appropriate for the more sophisticated intended users of ECLIPSE; however, our approach allows CHINLE to continue to update the likelihood of all hypotheses, even after the user has intervened.

PLOW [1] allows a different type of user input, using a spoken narrative to choose between possible generalizations of a demonstrated action. Some end-user programming work takes natural-language instruction even further. Tailor [4, 3] allows the user to modify a procedure with ordinary English commands. Sloppy programming [24, 23] lets users automate Internet browsing actions, again with English instructions.

CONCLUSIONS

This paper describes CHINLE, which (like FAMILIAR [29]) automatically generates PBD systems for applications, given only their interface specification. CHINLE is based on the version-space algebra approach of Lau et al. [19], because it is *intolerant to noise*; the ability to detect a version-space collapse is a crucial benefit, reducing the chance that the system will learn an unintended procedure. CHINLE makes two main contributions:

- **Novel Visualization and Interaction Methods.** PBD systems have long sought to communicate to the user the state of learning and the nature of the generalized procedure. CHINLE introduces new interactors for this task ranging from a simple viewer (Figure 6) to an expanded visualization (Figure 8). The simple view summarizes the learned procedure, predicts the next action, and communicates CHINLE's confidence using a sequential color map for probability. The expanded view extends the color map to indicate confidence in CHINLE's prediction and its underlying hypotheses; the expanded view also shows which hypotheses have been discarded as a result of which demonstrated actions.

- **New Ways to Recover from Errors during Demonstration.** CHINLE provides direct-manipulation metaphors for several error-correction techniques: 1) if the user executed the wrong action during demonstration, she can purge it from the training data with a simple checkbox (Figure 7); note that implementing this feature requires learning from incomplete date, which is complicated by cross-action dependencies in the hypothesis space, 2) the user can select or reject specific hypotheses — regardless of their consistency with training data, 3) when all hypotheses are inconsistent with an action's training data, CHINLE performs partial learning to create a wizard, which can be rendered *in situ* or *traditionally* (Figures 10 and 12). Each of these interactors is integrated with CHINLE's sequential color model.

In addition, we present a novel scheme for probabilistic weighting of hypothesized actions which allows iteration over sets and conditional branches, but ensures that a straight-line "macro" is learned if the user demonstrates just a single example.

Future Work

The biggest task for future work is a user study to evaluate the usability of our novel visualizations and interactors. Quite simply, do users find the system useful and do they have confidence in the learned procedures? We also anticipate more focused studies addressing questions such as 1) which wizard style (in situ or traditional) is preferred by users and when? 2) which is easier for users to understand, CHINLE's representation of conditional actions, that used by block structured languages, or sequential decision stumps? 3) what kinds of errors do users make when demonstrating macros and procedures? 4) will users actually understand and use CHINLE's error-correction capabilities? 5) do users understand and find useful CHINLE's visualizations of the learning process?

Since CHINLE is built atop SUPPLE and uses the latter's functional specification as the input for version-space construction, one must question the scope of this representation. To date SUPPLE has been used to build one to two dozen application interfaces — the most complex being a fully-functional email client. While these data-rich applications demonstrate SUPPLE's generality, we wish to continue to extend the complexity of the PBD systems generated by CHINLE.

On a more technical front, we want to remove CHINLE' current limitations such as manual segmentation, fixed-length loops, and the inability to use logical connectives inside conditionals. In addition, we note that currently CHINLE allows symmetric hypotheses in its version space (e.g. "If A then X else Y" and 'If not A then Y else X". It's well known that novice users struggle with if/then/else constructs, and unnecessary duplication is likely to exacerbate the problem. In the future, we will revise the version-space generation algorithm to eliminate symmetries.

Acknowledgements

We thank Eytan Adar, Richard C. Davis, Mira Dontcheva, Krzysztof Gajos, Raphael Hoffmann, Michael Toomim, Fei Wu, Jia-Chi Wu, Miao Xu and anonymous reviewers for their valuable feedbacks and comments. This research was supported by NSF grant IIS-0307906, ONR grant N00014-06-1-0147, SRI CALO grant 03-000225 and the WRF / TJ Cable Professorship.

REFERENCES

1. J. Allen, N. Chambers, G. Ferguson, L. Galescu, H. Jung, M. Swift, and W. Taysom. Plow: A collaborative task learning agent. In *Proc. of AAAI'07*.

2. L. Bergman, V. Castelli, T. Lau, and D. Oblinger. Docwizards: a system for authoring follow-me documentation wizards. In *Proc. of UIST '05*.

3. J. Blythe. An analysis of procedure learning by instruction. In *Proc. of AAAI-05*

4. J. Blythe. Task learning by instruction in tailor. In *Proc. of IUI '05*

5. C. A. Brewer. Color use guidelines for mapping and visualization. In A. MacEachren and D. Taylor, editors, *Visualization in Modern Cartography*, pages 123–147. Elsevier Science, 1994.

6. V. Castelli and L. Bergman. Distributed augmentation-based learning: a learning algorithm for distributed collaborative programming-by-demonstration. In *Proc. of IUI'07*

7. A. Cypher. *Watch what I do: Programming by demonstration*. MIT Press, Cambridge, MA, 1993.

8. R. Davis. Diagnostic reasoning based on structure and behavior. *Aritificial Intelligence*, 24, 1984.

9. A. Faaborg and H. Lieberman. A goal-oriented web browser. In *Proc. of CHI '06*.

10. M. R. Frank and J. D. Foley. A pure reasoning engine for programming by demonstration. In *Proc. of UIST '94*.

11. K. Gajos and D. S. Weld. Supple: automatically generating user interfaces. In *Proc. of IUI'04*.

12. H. Hirsh. Theoretical underpinnings of version spaces. In *Proc. of IJCAI'91*.

13. W. C. Kim and J. D. Foley. Providing high-level control and expert assistance in the user interface presentation design. In *Proc. of CHI'93*.

14. C. Kissinger, M. Burnett, S. Stumpf, N. Subrahmaniyan, L. Beckwith, S. Yang, and M. B. Rosson. Supporting end-user debugging: what do users want to know? In *Proc. of AVI '06*.

15. A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proc. of CHI '04*.

16. T. Lau, L. Bergman, V. Castelli, and D. Oblinger. Sheepdog: learning procedures for technical support. In *Proc. of IUI '04*

17. T. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *Proc. of ICML'00*.

18. T. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *Proc. of K-CAP '03*.

19. T. Lau, S. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1–2):111–156, October-November 2003.

20. L. Liao, D. Fox, and K. H. Location-based activity recognition using relational markov networks. In *Procs. of IJCAI'05*.

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

22. H. Lieberman. *End User Development*. Springer, 2006.

23. G. Little, T. Lau, A. Cypher, J. Lin, E. Haber, and E. Kandogan. Koala: Capture, share, automate, personalize business processes on the web. In *Proc. of CHI '07*.

24. G. Little and R. C. Miller. Translating keyword commands into executable code. In *Proc. of UIST '06*.

25. R. J. A. Little and D. B. Rubin. *Statistical analysis with missing data*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

26. T. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

27. J. Nichols, B. A. Myers, M. Higgins, J. Hughes, T. K. Harris, R. Rosenfeld, and M. Pignol. Generating remote control interfaces for complex appliances. In *Proc. of UIST'02*.

28. D. Oblinger, V. Castelli, and L. Bergman. Augmentation-based learning: combining observations and user edits for programming-by-demonstration. In *Proc. of IUI '06*.

29. I. H. Paynter, G. W.and Witten. Applying machine learning to programming by demonstration. *Journal of Experimental and Theoretical Artificial Intelligence*, 16(3):161–188, 2004.

30. J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

31. C. Wiecha, W. Bennett, S. Boies, J. Gould, and S. Greene. ITS: a tool for rapidly developing interactive applications. *ACM Transactions on Information Systems (TOIS)*, 8(3):204–236, 1990.

32. S. A. Wolfman, T. Lau, P. Domingos, and D. S. Weld. Mixed initiative interfaces for learning tasks: Smartedit talks back. In *Proc. of IUI'01*.