Learning Users' Habits to Automate Repetitive Tasks

Jean-David Ruvini **Christophe Dony**
LIRMM, University of Montpellier
161, rue Ada - 34392 Montpellier Cedex 5 - FRANCE
{ruvini,dony}@lirmm.fr
Tel: 33 4 67 41 85 86 Fax: 33 4 67 41 85 00

1 Introduction

Entering repetitive sequences of commands (or repetitive tasks) is a well-known characteristic of human-computer interaction. To deal with this problem, early works have associated macro or script languages with interactive environments, for example macros in Excel or Lisp scripts in Emacs. They allow the user to write a program that can be later on invoked to perform a sequence of commands automatically. The limitation of this approach is that, generally, users do not want to or can not spend too much effort on programming: writing a program often takes longer than performing a sequence of commands manually, disrupts the user's work-flow, and requires programming knowledge that many users do not have. Recent advances to overcome these limitations came from different correlated field of research : Programming by Demonstration (PbD), Predictive Interfaces and Learning Interface Agents.

PbD systems [Cypher *et al.*, 1993] let the user demonstrate what the task to automate should do, and create a program from this demonstration. Macro recorders were the first examples of PbD systems, but they were limited because recorded commands are too specific (rote learning, no parameterization) to be reused. Sophisticated PbD systems, such as *Mondrian* [Lieberman, 1993], create programs containing variables, iterative loops or conditional branches from observing user's actions. Although PbD does not require programming knowledge because the user does not have to write code, demonstrating a program takes time and disrupts the user's work-flow.

Predictive Interfaces [Darragh and Witten, 1991] and Learning Interface Agents [Maes, 1994] observe the user while he manipulates the environment, and try to learn, from the correlations between situations the user has encountered and the corresponding commands he has performed, to predict, after each new command what will be the next one. They assist him by afterwards predicting and suggesting to perform automatically some commands. For instance, CAP [Mitchell *et al.*, 1994], an assistant for managing meeting calendars, suggests default values regarding meeting duration, location, time and day-of-week. OpenSesame! [Caglayan *et al.*, 1997] runs in background on Macintosh system 7, and offers to open or close files or applications, to empty trash, or to rebuild desktop on the user's behalf. *WebWatcher* [Armstrong *et al.*, 1995], an assistant for the world wide web, suggests links of interest to the user. Maes's assistants for electronic mail, meeting scheduling and electronic news filtering [Maes, 1994], advise the user for some application specific operations like managing mails, scheduling meetings or selecting articles in news. *ClipBoard* [Motoda, 1997], an interface for Unix, tries to predict the next command the user is going to issue. The main advantages of these systems is that they do not require programming knowledge nor disrupt the user's work-flow because commands are automatically suggested to the user. However, they do not create programs and, thus, only suggest single actions and not sequences of actions. Note furthermore that the set of actions that most of these systems (expect *ClipBoard* and *WebWatcher*), can suggest is small and known in advance.

Eager [Cypher *et al.*, 1993], is one of the most famous attempts to bring together Programming by Demonstration and Predictive Interfaces. *Eager* is an assistant for Macintosh Hypercard. When *Eager* detects two consecutive occurrences of a repetitive task in the sequence of user's actions, it assumes they are the first two iterations of a loop, and proposes to complete the loop. It is a PbD system because it is able to infer loops from observing user's actions and to replay more than one action at once; it is a Predictive Interface because it is able to make suggestions without any user's intervention. It is able to perform loop iterations until "a condition" is satisfied, or following some typical patterns like days of the week or linear sequence of integers. Finally, *Eager* has an important characteristic: it makes a suggestion only after two consecutive occurrences of a repetitive task. As a consequence, it knows exactly when to make a suggestion and which suggestion to make. However, this characteristic is a limitation because in practice such occurrences are frequently not consecutive but interleaved with other actions. *Familiar* (see Paynter's chapter) takes on *Eager* idea and extend it in many ways but does not address this limitation.

The goal of our work has been to design an assis-

tant operating in a context where the number of possible user's actions and possible values for the parameters of these actions are large, where repetitive sequences are not known in advance and not consecutive, and able to replay repetitions composed of several actions, containing loops or conditional branches. None of the above quoted works addresses simultaneously all these issues. In such a context, a key issue is to design an assistant which makes "the right suggestion at the right moment": an assistant who constantly bothers the user with a lot of wrong suggestions is useless because the user would rapidly ignore it. Wolber and Myers's chapter suggests a solution to this problem in the context of PbD system. It proposes to allow the user to demonstrate "When" to make a suggestion as well as "What" to suggest. APE takes another approach. It employs Machine Learning techniques to efficiently and rapidly learn when to make a suggestion, and which sequence of actions to suggest to the user. As a case study, we present the APE (Adaptive Programming Environment) project. APE is a software assistant integrated into the Visualworks Smalltalk programming environment. Like *Eager* and *Familiar*, APE is able to detect loops and to suggest repetitive tasks iteratively. APE is written in Visualworks Smalltalk 3.0 - ObjectShare, Inc, operational and publicly available at `http://www.lirmm.fr/~ruvini/ape`.

In the following we describe APE, we demonstrate what it does and how it can be used. We explain what kind of repetitive tasks it is able to automate and how it automates them. We show what makes learning user's habits difficult and we describe in detail what and how APE learns. We compare experimental results of alternate approaches. We finally summarize lessons learned from this study and give perspectives for future research.

2 Overview of APE

APE is made of three software agents, an Observer, an Apprentice and an Assistant, working simultaneously in the background without any user's intervention. Table 1 defines our terminology and Figure 1 describes the role of each agent.

2.1 The Observer

The Observer traps user's *actions*, reifies them into dedicated Smalltalk objects, instances of classes shown in Figure 2, and stores them in the *trace*. It then sends messages in background to the Apprentice and to the Assistant to notify them that the user has performed a new action.

For example, when the user selects the *doIt* command of a text editor to evaluate an expression, an instance of the class *ActionEditor* is created and references to the involved text editor, the evaluated text and the string "*doIt*" are respectively stored in the *toolID*, *text* and *action* slots. Table 2 shows an example of a part of a trace where each line is a simplified textual representation of an action (for a reason of clarity, we only show the

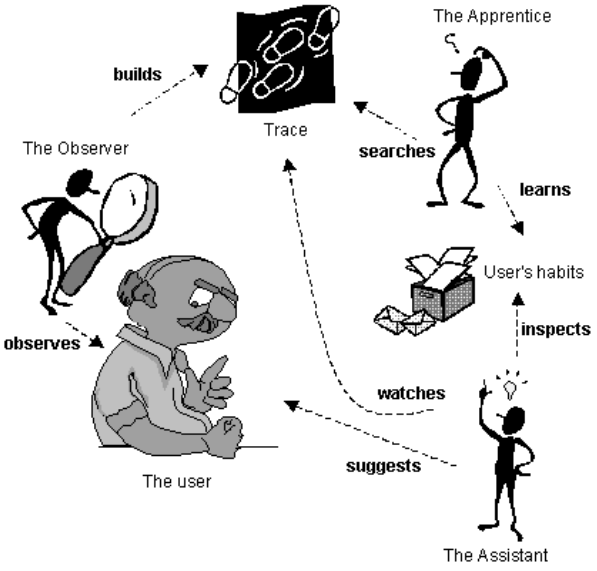


Figure 1: The Observers monitors user's actions and builds the trace, the Apprentice learns user's habits and the Assistant proposes to the user sequences of actions to replay.

Action : high-level intervention of the user on the environment (as opposed to low level interventions such as mouse movements and keystrokes), window manipulation, menu item selection, button pressing, text entering, etc. An action is parameterized by, among other things, the tool (e.g. a Browser, a Debugger, a Text Editor, ...) in which it has been performed.
Trace : history of user's actions.
Task : sequence of actions of the trace.
Repetitive task : task occurring several time in the trace.
Situation : sequence of actions of the trace of a given size n, n being a parameter of the learning algorithms.
Current situation : the last n actions of the trace.
Situation pattern : regular expression matching one or more situations.
Habit : pair "set of situation patterns - repetitive task" such that the situation patterns match the situations in which the user performs the repetitive task.
When-set : set of situation patterns that match the situations in which the user has performed repetitive tasks.
What-set : set of habits.

Table 1: Definition of terms used throughout this chapter.

most informative action parameters). There are different classes to represent the actions held in the different tools of the environment (browser, debugger) because they hold different versions of methods used by the learning algorithm - which does not handle equally all kinds of actions.

In this first implementation, trapping users' actions has been achieved by directly modifying methods (up to 170) of the user interface layer in which user's actions are fired. This is not very satisfactory and should be improved in the future versions. This is a consequence of the lack of a standard mechanisms, such as the "advice/trace" mechanisms of Interlisp [Teitelman, 1978] or such as the Flavors [Moon, 1986] "wrappers" mechanisms, in the Smalltalk environment we have used. Such mechanisms have been developed for Smalltalk, for example in [Böcker and Herczeg, 1990], but no one has been integrated in the Smalltalk environment we have used.

```
Object ()
   Action (type toolName toolID date display)
      ActionApplication (action)
        ActionBrowser (parameter textMode selected)
          ActionDebugger ()
          ActionFileBrowser ()
          ActionParcelBrowser ()
        ActionChangeList (index plug)
        ActionEditor (text index which)
        ActionInspector (parameter on)
        ActionLauncher ()
        ActionParcelList ()
        ActionWindow ()
      ActionError (error object message)
```

Figure 2: Smalltalk hierarchy of action classes.

2.2 The Apprentice

The Apprentice activity is twofold.

1. It detects the user's *repetitive tasks.*
2. It examines the *situations* in which repetitive tasks have been performed and uses two Machine Learning algorithms to learn *situation patterns* and build two sets:
   (a) The When-set of *situation patterns* matching the situations in which the user has performed the detected repetitive tasks.
   (b) The What-set of user's habits i.e. pairs "set of situation patterns - repetitive task" where the set of situation patterns reflects all the situations in which a given repetitive task has been performed.

The Apprentice is able to learn 3 kinds of situation patterns: situation patterns containing wildcards (i.e. a special character, noted ".", that matches any single action or action parameter), unordered situation patterns (the order in which some actions are performed does not matter), or unordered situation patterns containing wildcards. The number of wildcards is not limited.

An occurrence of a situation pattern containing a wildcard is learned when, for example, the user has examined, in a Smalltalk browser, several methods named "=" in the `testing` protocol, for various classes of the `MyGraphics` category (see figure 7). The detected repetitive task is "select protocol(`testing`), select method(=)" and the learned situation pattern is "select category(`MyGraphics`), select class(.)".

2.3 The Assistant

The Assistant observes the user, it uses the When-set to determine **when** to make a *suggestion* to the user and if it has to, it uses the What-set to determine **what** to suggest. More precisely, as shown in Figure 3, after each user's action it inspects the What-set to answer the question: "Is the user going to perform a repetitive task?". If the last user's actions match none of the situation patterns of the When-set, the answer is "no" and the Assistant makes no suggestion. Otherwise, the answer is "yes", and the Assistant inspects the user's habits (What-set) to answer the question: "Which repetitive task is the user going to perform?". It selects all habits[1] with a situation pattern that matches the current situation. Then, it displays in the Assistant window (cf. Figure 4), without interrupting the user's work, the actions composing the repetitive task of the selected habits. The user can ignore this window and these suggestions (non-obtrusive behavior) or mouse-click on one of them. In the latter case, the Assistant successively performs the actions and removes the suggestions from its window.

3 Illustrative Examples

This section provides four examples of what APE is able to learn and to suggest.

Example 1

Repetitive tasks frequently appear while testing applications. Consider a user testing a multi-process simulation of the classical "n-queens" problem, implemented by a main class `Board`. Figure 4 shows two VisualWorks snapshots both including an Assistant window, labeled "Assistant", and the main APE window, labeled "Ape Agents". The `Watch` button shows that the three agents are active. In the top snapshot, labeled "situation before firing a habit", the user has selected, in a simple editor (named Workspace), a Smalltalk expression to create a `board` and to initiate the computation, and is about to select the `inspecIt` item of that editor menu. Because the user is not performing this activity for the first time, a repetitive sequence has been detected and an habit has been learned, a situation pattern of which matches the current situation. The Assistant thus fires the habit i.e. displays in its window a text describing the proposed repetitive sequence of actions (opening four inspectors

[1] Browsing through a huge number of suggestions to find the right one puts a workload on the user. The number of suggestions the Assistant can make is a parameter of APE.

```
ActionEditor(anEditor,'anArray stupidMesage','doIt')
ActionError('doesNotUnderstand','stupidMessage')
ActionDebugger(aDebugger,debug)
ActionWindow(aDebugger,'move')
ActionWindow(aDebugger,'resize')
```

Table 2: A sample of the trace where the user opens, moves and resizes a debugger to correct an error.

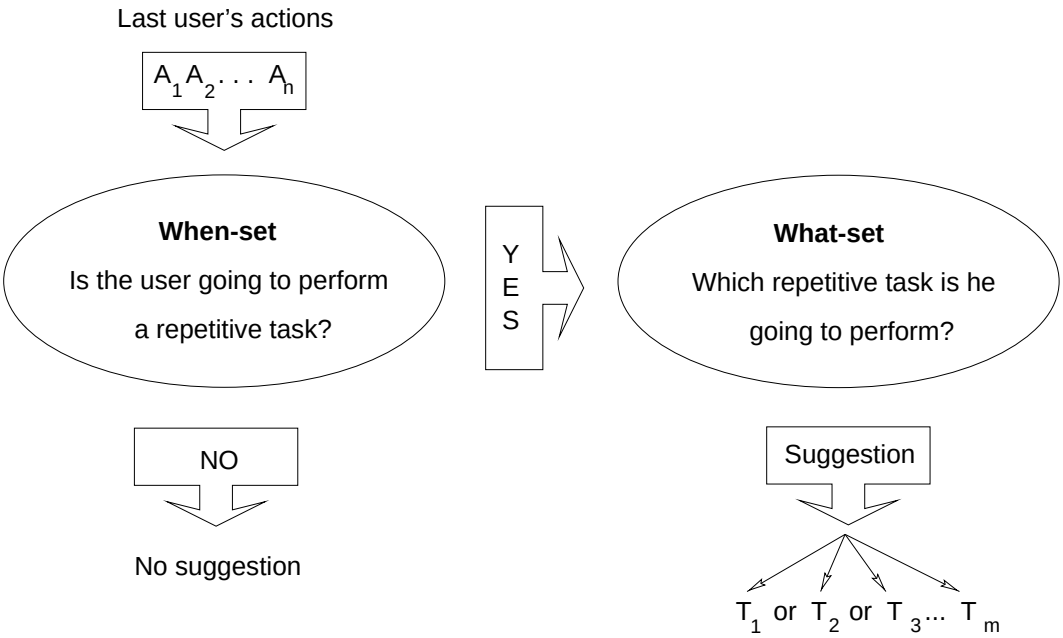
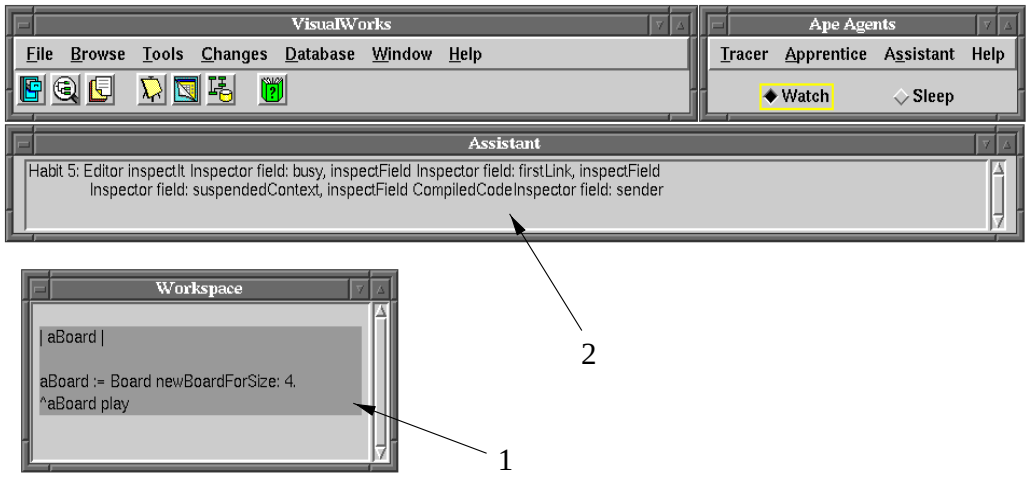


Figure 3: The Assistant inspects the When-set and the What-set to make suggestions.

Situation before firing a habit

The Assistant suggests to open four inspectors...



Situation after a habit has been fired

The user mouse-clicked on the proposition

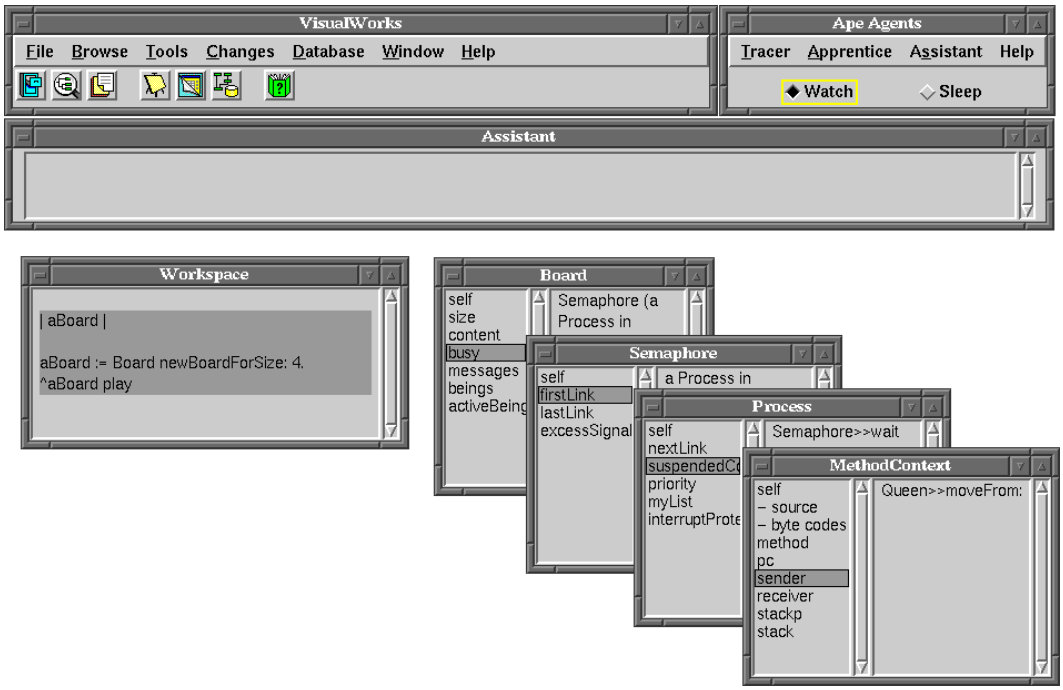


Figure 4: The user has just selected an expression (1) to create and test an instance of class "Board". The Assistant detects a known situation and suggests to execute a registered repetitive sequence of actions, displayed in the Assistant window (2), that evaluates the expressions and open 4 inspectors in cascade, leading to what is shown in the bottom snapshot.

in cascade to show a particular field of a composed object). This repetitive task being exactly what the user intents to do, he mouse-clicks on that text to perform the sequence of actions leading to what is shown in the bottom snapshot labeled "situation after a habit has been fired". In this case, the user has performed seven actions in a secure way with a single mouse-click.

Example 2

Repetitive tasks also frequently appear while debugging applications. Consider the same user now debugging his "n-queens" application. The user has selected a Smalltalk expression (cf. Figure 5, top snapshot, arrow 1) in the "Workspace" window, the evaluation of which (arrow 2) has raised an exception leading to the opening of an "Exception" window (arrow 3). Because, this situation matches a situation pattern learnt by the Apprentice, the Assistant offers to perform the related repetitive task: "open, move, resize a debugger and select stack index 5". This repetitive task is exactly what the user intents to do and he mouse-clicks on the proposition (arrow 4), entailing the creation and correct positioning of a debugger window, as shown in the bottom snapshot.

Example 3

This examples shows that APE is able to automate sequences of actions iteratively (in a loop), even if the iterations are not consecutive. Suppose a user intends to modify the method "area" of all the classes belonging to a category named `MyGraphics`. Before working on a method "area", he wants to save (back up) it. He has first selected and saved the `area` method of the `Circle` class by performing the following actions: select the `MyGraphics` category (cf. Figure 6a, arrow 1), select the `Circle` class of that category (arrow 2), select the `accessing` protocol (arrow 3) and the `area` method of that protocol (arrow 4), select "file out as..." item in the browser menu (arrow 5) to save the method. Later, after having completed various tasks like the modification of that `area` method, the creation of a new `Triangle` class, etc., he has selected and saved the `area` method of the `Diamond` class (cf. Figure 6b). At this point, the Apprentice has detected two non consecutive occurrences of the repetitive task "select the `accessing` protocol, select the `area` method, file out as". The action preceding the first occurrence of this repetitive task is "select the `Circle` class" and the action preceding the second occurrence is "select the `Diamond` class". Because classes `Circle` and `Diamond` belong to the `MyGraphics` category, it infers that the user intends to save the `area` method of all classes of `MyGraphics` category. Hence, it assumes that these two occurrences are two iterations of the following loop: "For all classes of the `MyGraphics` category do select the `accessing` protocol, select the `area` method, file out as" and learns a habit. As a consequence, as soon as the user selects the `MyGraphics` category, and whatever actions he has performed before, the Assistant predicts that he is about to save one more method `area` and offers to complete the loop (cf. Figure 6c). If the user mouse-clicks on the suggestion in the Assistant window, the Assistant saves all methods `area` not yet saved (not shown).

Example 4

This last example shows that APE is able to help the user in the writing of repetitive pieces of code. Suppose a user has written several similar methods named "=", for various classes of `MyGraphics` category, in a browser. He has just selected the `testing` protocol (cf. Figure 7, arrow 1) and is about to write a new method "=". The Assistant offers to insert a text *template* (arrow 2) containing some repetitive code (the asterisks denote non repetitive code). The user has mouse-clicked the suggestion and the template has been inserted (arrow 3 - "situation after a habit has been fired" - bottom snapshot).

4 Detecting repetitive tasks

This section explains what kinds of repetitive tasks the Apprentice is able to detect in the trace and how the Assistant automates them.

4.1 Repetitive sequences of actions

Detection of repetitive sequences of actions is achieved using a classical text searching algorithm [Karp *et al.*, 1972]. "Open, move, resize a debugger and select stack index 5" (cf. Figure 5) is an example of a repetitive sequence of actions.

To automate a repetitive sequence of actions, the Assistant simply replays the actions composing it.

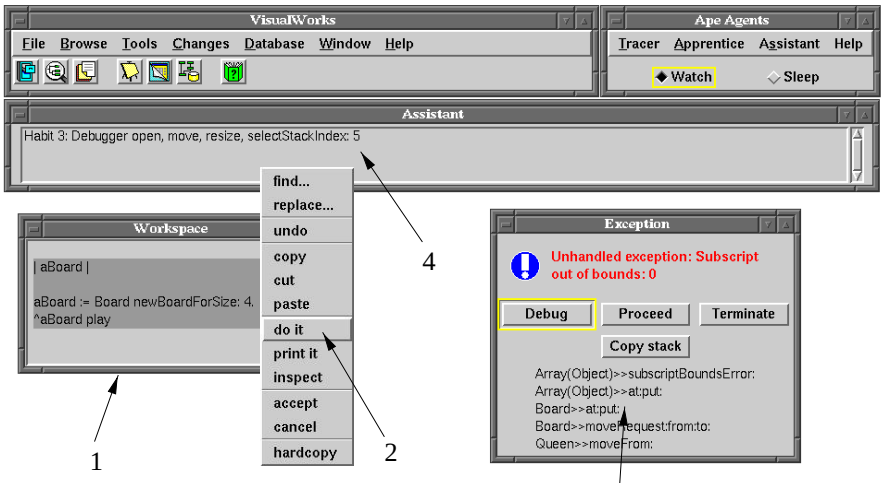
4.2 Loops

Each time the Apprentice detects a repetitive sequence of actions, it supposes that the corresponding sequence of actions could be the "body" of a loop and that each occurrence of the sequence could be an iteration of the loop. It then searches for relations between the actions preceding (or following) each iteration to determine the loop "variable". Examples of such relations are: classes belonging to the same category, methods belonging to the same class, subclasses of the same class, etc. Example 3 in section 3 illustrates this case. The body of the loop is "Select the `accessing` protocol, select the `area` method, file out", the action preceding the first iteration is "Select the `Circle` class" and the action preceding the second iteration is "Select the `Diamond` class". The relation between these two actions is that classes `Circle` and `Diamond` belong to the same category. Thus the Apprentice then infers that the selected class is the loop variable and builds the following repetitive task: "For all classes x of `MyGraphics`, select the x class, select the `accessing` protocol, select the `area` method, file out".

To complete a loop, the Assistant plays the loop body for all the remaining values of the loop variable.

Situation before firing a habit

The Assistant proposes to open a debugger...



Situation after a habit has been fired

The user mouse-clicked on the proposition

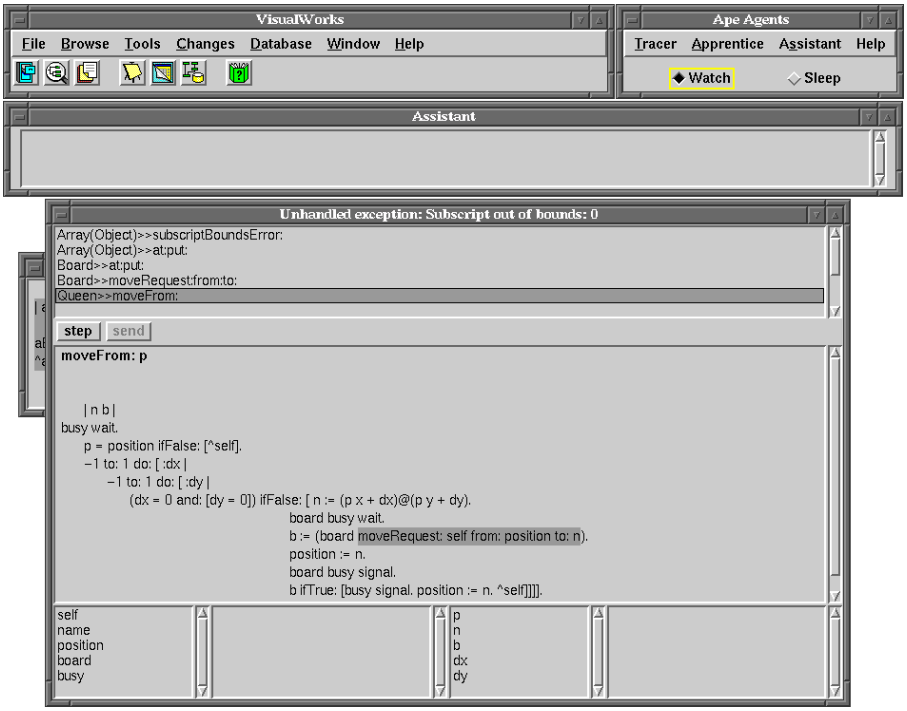
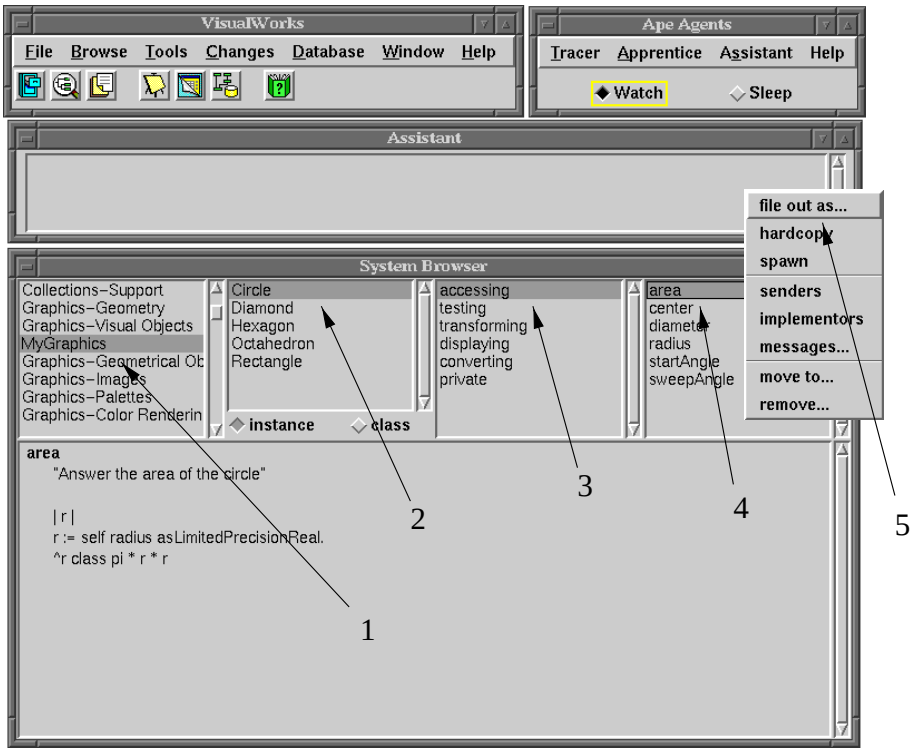
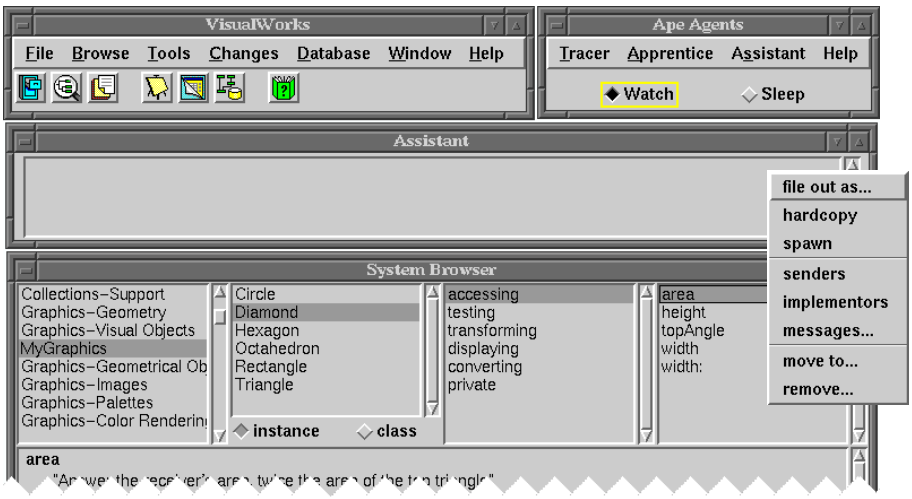


Figure 5: The user has typed (1) and evaluated (2) an expression that raised an exception (3). The Assistant offers (4) to open, move, resize a debugger and to select the fifth item in the debugger stack, as the user typically does (top snapshot). The user has mouse-clicked on the proposition and the Assistant has performed these actions, resulting in a well positioned debugger displaying a user's method (bottom snapshot).



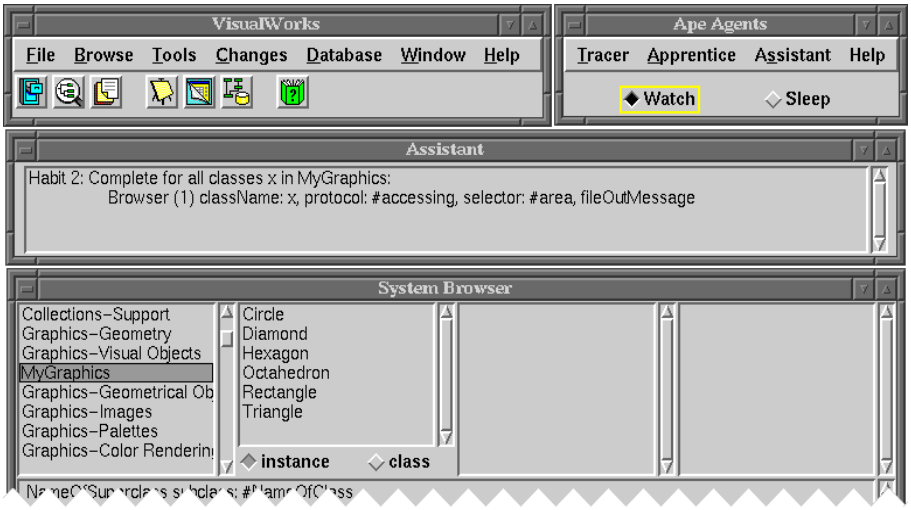
Example 3, first part

Figure 6a : The user saves ("file out as..." command) method `area` of class `Circle`.



Example 3, second part

Figure 6b: After having completed various tasks like the modification of that method `area`, the creation of a new class `Triangle`, etc., he also saves method `area` of class `Diamond`.



Example 3, third part

Figure 6c: The Apprentice has detected a loop in user's actions and has learned a habit. The Assistant offers to complete the loop (to save all method `area`) as soon as it detects that the user is about to save one more method `area`.

4.3 Writing of repetitive pieces of code

To detect these repetitive tasks, the Apprentice compares the methods created by the user, line-by-line, using a simple stringmatch comparison function. When it finds a set of methods that share a certain amount of their respective code in common, it assumes it has found a repetitive portion of code and create a template (cf. Figure 7).

To replay a writing of a repetitive piece of code, it inserts the template in the code window of the browser (again, cf. Figure 7).

4.4 Repetitive corrections of (simple) programming errors

The Apprentice compares the methods the user has modified and the way he did it. When it finds a set of method in which the user has replaced a portion C with another portion of code C', it assumes it has found a repetitive correction and records the replacement.

To replay a repetitive correction of code, it replays the recorded code replacement.

5 Learning User's Habits

5.1 What Makes the Problem Difficult?

We present in this section the requirements that have conducted the choice of the algorithms employed by the Apprentice to learn the situation patterns of the When-set and the What-set. Let us recall that the When-set is a set of situation patterns that match the situations in which the user has performed repetitive tasks and that the What-set is a set of habits i.e. a set of pair "situation patterns - repetitive task". Let AL1 denote the algorithm used to build the When-set and AL2 denote the algorithm used to build the What-set .

Requirement R1 : Low training time

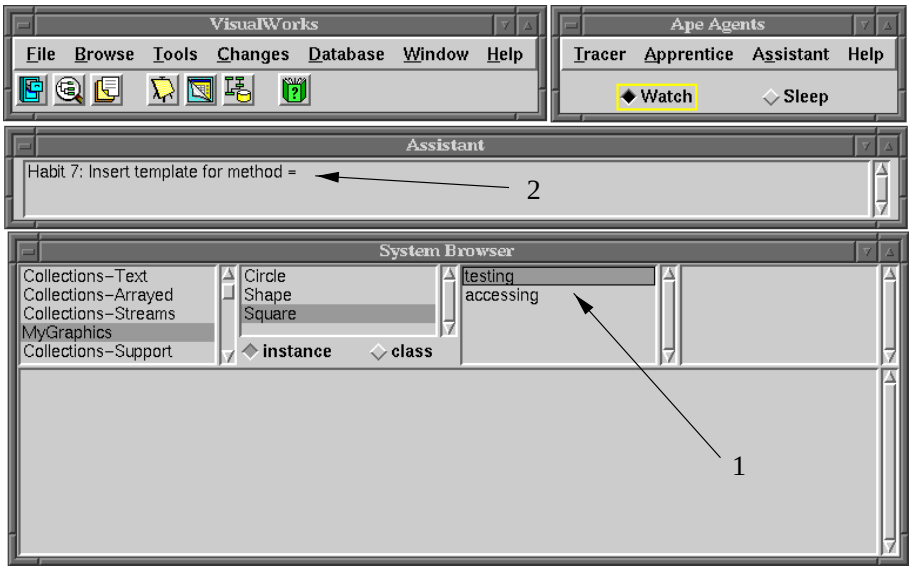
We distinguish "long life" and "short life" repetitive tasks. Short life repetitive tasks are related to specific issues, appear in a small section of the trace and the corresponding situation patterns have to be learned very rapidly from a few situations. Long life repetitive tasks can reflect the general user's habits and can require a very long trace to be detected. Thus, the Apprentice is, on the one hand, able to learn situation patterns very rapidly on small section of the trace in order to capture short life tasks and, on the other hand, can also consider very long traces corresponding to several work sessions. Let us call *training time* the time required by a Machine Learning algorithm to learn situation patterns. AL1 and AL2 must have a low *training time*.

Requirement R2 : Low prediction time

Of course, the Assistant is able to decide when to make a suggestion and which suggestion to make very rapidly. Let us call *prediction time* the time it takes to the Assistant to inspect a set of situation patterns and to determine which ones match the current situation. The prediction time depends on the way AL1 and AL2 represent the learned situation patterns. This prediction time must be very low to allow the Assistant to make suggestions (or to decide to not make a suggestion) after each user's actions.

Situation before firing a habit

The Assistant suggests to insert some repetitive code...



Situation after a habit has been fired

The user mouse-clicked on the proposition

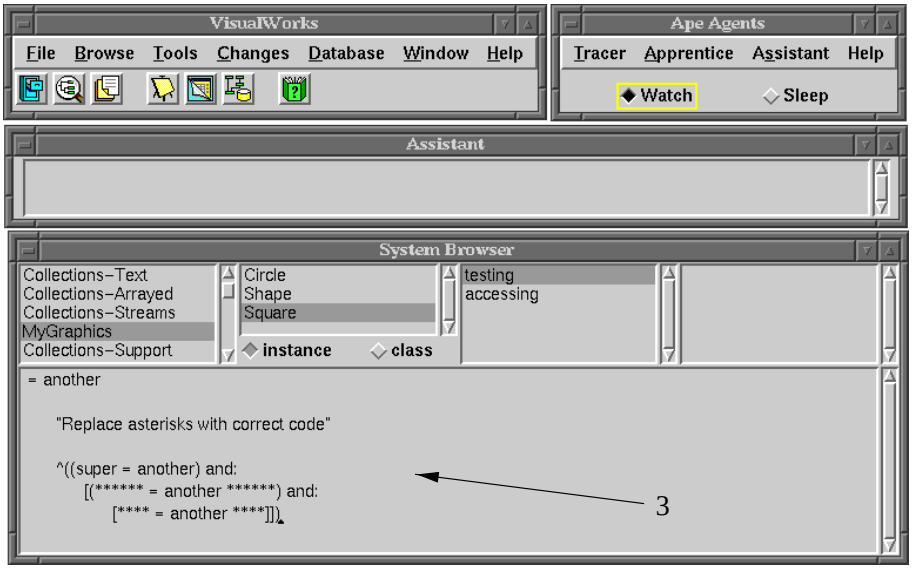


Figure 7: The user has written several similar methods named "`=`", for various classes of `MyGraphics`. He has just selected the `testing` protocol (1) and is about to write a new method "`=`" for class `Square`. The Assistant offers to insert a template (2). The user has mouse-clicked the suggestion and the template has been inserted (3).

Requirement R3 : user intelligible situation patterns

We want the Apprentice to represent situation patterns in a human understandable way. This is not a critical requirement but allows the user to inspect or edit the learned habits. Comprehensible and controllable interfaces give the user the sense of power and control.

Requirement R4 : AL1 - Low error rate

Finally, to be viable our Assistant has to make the "right suggestion at the right moment". This means that it has to correctly determine **when** to make a suggestion. AL1 is said to "make" an error in two cases: (1) when one of the situation patterns it has learned matches the current situation whereas the user is not about to perform a repetitive task, or (2) when none of its situation patterns matches the current situation whereas the user is about to perform a repetitive task. In case (1), the Assistant makes suggestions whereas no suggestions should be done; in case (2), it makes no suggestion whereas suggestions would benefit the user. AL1 must have a low *error rate*.

Requirement R5 : AL2 - Low error rate and low generalization

The Assistant also has to correctly determine **what** to suggest. AL2 is said to "make" an error in two cases: (1) when a situation pattern of an habit matches the current situation whereas the user is not about to perform the repetitive task of that habit, or (2) when a situation pattern of one of the habits matches the current situation whereas AL1 has made an error[2] and the user is not about to perform a repetitive task. In case (1) the Assistant suggests the wrong repetitive task; in case (2) it makes suggestions whereas no suggestion is expected. Case (2) may occur if the situation patterns of the habits are too general. A too general situation pattern will be matched by too many situations and the corresponding task proposed too frequently. AL2 must have a low error rate to make few errors in case (1), but also has to generalize as little as possible to avoid too general situation patterns in case (2).

5.2 Which Algorithms?

Various kinds of algorithms have been proposed in the field of Machine Learning. Concept learning, neural networks and reinforcement learning algorithms do not meet requirement R1, instance-based algorithms do not meet requirement R2, instance-based, statistical, neural networks and reinforcement learning algorithms do not meet requirement R3.

Eligible kinds of algorithms are decision trees algorithms because they miss none of the requirements R1, R2 and R3. Decision trees algorithms, like most of the Machine Learning algorithms, have a low error rate and meet requirement R4. Although they have not been designed to generalize as little as possible, they are better suited for requirement R5 than instance-based or statistical algorithms.

Decision trees learning algorithms have notably been used in CAP [Mitchell *et al.*, 1994]. The state-of-the-art decision trees learning algorithm is C4.5 [Quinlan, 1993][3]. C4.5 has low computing time (incremental versions of C4.5 exist) and is suited to learn the When-set (AL1). However, our tests (cf. Experimental Results) have shown that it learns too general situation patterns and is not suited to learn the What-set (AL2). Hence, APE employs C4.5 to learn the When-set and a new algorithm we have designed to learn the What-set.

5.3 A New Algorithm

Our new algorithm, named IDHYS, is a concept learning algorithm inspired by the Candidate-Elimination [Mitchell, 1978].

Inductive concept learning consists in acquiring the general definition of a concept from training examples of this concept, each labeled as either a member (or *positive example*) or a non-member (or *negative example*) of this concept. Concept learning can be modeled as a problem of searching through a *hypothesis space* (set of possible definitions) to find the hypothesis that best fits the training examples [Mitchell, 1982]. Learning user's habits is a concept learning problem. All the situations preceding a repetitive task T can be seen as positive examples of the concept *"Situations in which the user is going to perform repetitive task T"*. The situations preceding any other task can be considered as negative examples of this concept. The searched definition is a set of situation patterns that match the situations in which the repetitive task T has been performed.

IDHYS searches the hypothesis space of the conjunctions of three situation patterns, one for each kind of situation pattern defined in Section 2: containing wildcards, unordered, or unordered containing wildcards. It learns by building hypotheses that are the most specific generalizations of the positive examples. It processes the positive examples incrementally. It starts with a very specific hypothesis (indeed the first positive example itself) and progressively generalizes this hypothesis with the subsequent positive examples. IDHYS does not build hypotheses for the negative examples which are only used to bound the generalization process. This incremental bottom-up approach makes IDHYS not sensitive to actions with large sets of possible values for their parameters. As a consequence it has a low computing time. Our test (cf. next section) shows it also has a low error rate and does not over-generalize to build the situation patterns.

[2] Whatever algorithm is used to build the When-set, it may sometimes make errors: Machine Learning algorithms with a null error rate have not been discovered yet, and even do not exist for most of the learning tasks.

[3] A commercial version of C4.5, called C5.0, is now available.

Description of an earlier version of IDHYS can be found in [Ruvini and Fagot, 1998], and a more complete in [Ruvini, 2000].

6 Use and Experimental Results

APE, as described in the above section, is implemented and experimentally used by ourselves and by a pool of 50 students enrolled in a Smalltalk course at the master's level. This section first analyses user feedback. It then describes and analyses the technical experimental results.

Concerning user feedback, let us recall that our users are Smalltalk beginners. We do not have yet feedback from experienced programmers. About 70% of our students have considered the Assistant window during 10 minutes approximatively and then have forgotten it. Fortunately, results from the remaining 30% have been very interesting. The main reasons invoked by those who have not used the suggestions are :

- the burden of looking at the Assistant window since nothing, except a modification inside this window, indicates when a suggestion is made,
- the difficulty to read the suggestions presented as a sequence of actions.

This indicates that there is a great deal of work to do in that direction, namely, how to gently alert people and how to provide a better visualization of what the Assistant suggests? Besides, the interesting point is that those who have made the effort to use the tool have rapidly learned how to use it efficiently and have taken advantages of its capabilities. After a while, those users have learned :

- which suggestions are regularly made and which ones interest them,
- when the suggestions are made.

In other words, they have learned to give a look at the Assistant window when they are about to perform a repetitive task and when they do know that the suggestion will be made. The students have been able to anticipate APE's suggestions because APE has a low excess rate and makes few wrong suggestions.

Concerning technical results, APE correctly works and makes the suggestions we expected it to. It also makes many suggestions we did not think of. We report here experiments conducted on 10 traces of 2000 actions long, collected during students' usage of the software.

How well does APE assist its users in practice? One way to answer this question is to train APE on a part of a trace (called the *train trace*) and then to test it on another part (called the *test trace*) to see how often one of its suggestions coincides with user's actions[4]. Figure 8 plots this data. The horizontal axis gives the size of the train traces and the test traces used. APE employs C4.5 to learn the When-set and "IDHYS" to learn the What-set. This is denoted by "C4.5-IDHYS" in Figure 8. However, as a comparison, we also report results when the Apprentice employs C4.5 to build both the When-set and the What-set, this denoted by "C4.5-C4.5", and when it learns the What-set only (and not the When-set) with C4.5 and IDHYS, respectively denoted by "C4.5" and "IDHYS".The percentage of correct suggestions (top graph) is the percentage of repetitive tasks correctly suggested by the Assistant. The percentage of excessive suggestions (bottom graph) is the percentage of actions of the test trace not preceding a repetitive task for which the Assistant has made a suggestion. These percentages have been evaluated for a situation length varying from 1 to 10, and Figure 8 presents average results. This Figure shows that:

- The use of the When-set decreases the amount of excessive suggestions without decreasing the amount of correct suggestions.
- Employing IDHYS to learn the What-set leads the Assistant to make less excessive suggestions and to suggest correctly more repetitive tasks.
- The percentage of excessive suggestions increases with the trace size (see "C4.5-IDHYS").

Both the percentage of correct suggestions and the percentage of excessive suggestions increase with the situation lentgh (not shown here). This shows that there is a tradeoff to find between an assistant which makes few but correct suggestions (and perhaps misses some repetitive tasks) and an assistant which constantly bothers the user with suggestions. Practically, we have chosen a learning frequency of 100 actions and a situation length of 3 actions. In this case, APE correctly suggests 63% of the repetitive tasks and makes an excessive suggestion for only 18% of user's actions. The average size of the repetitive tasks suggested by the Assistant during this experiment was 7 actions (minimum 3, maximum 10).

Another important question is how long it takes APE to learn user's habits. In practice, it takes APE 15 seconds (measured on a PC with a 133 Mh processor) to learn user's habits (i.e. both the When-set and the What-set) from a trace containing 100 actions. A 100 actions trace corresponds to about 6 minutes of user's works. In other words, every 6 minutes of user's work, APE spends 15 seconds to learn new habits. This is quite satisfactory. Note finally that the Assistant makes suggestions (i.e. inspects the When-set and the What-set) in a matter of milliseconds.

7 Conclusion and Prospects

APE is one more step towards assistants which bring together Programming by Demonstration and Predictive Interfaces. It works and operates in a context where the number of possible actions and possible values for the

[4]This is similar to the Machine Learning cross-validation process.

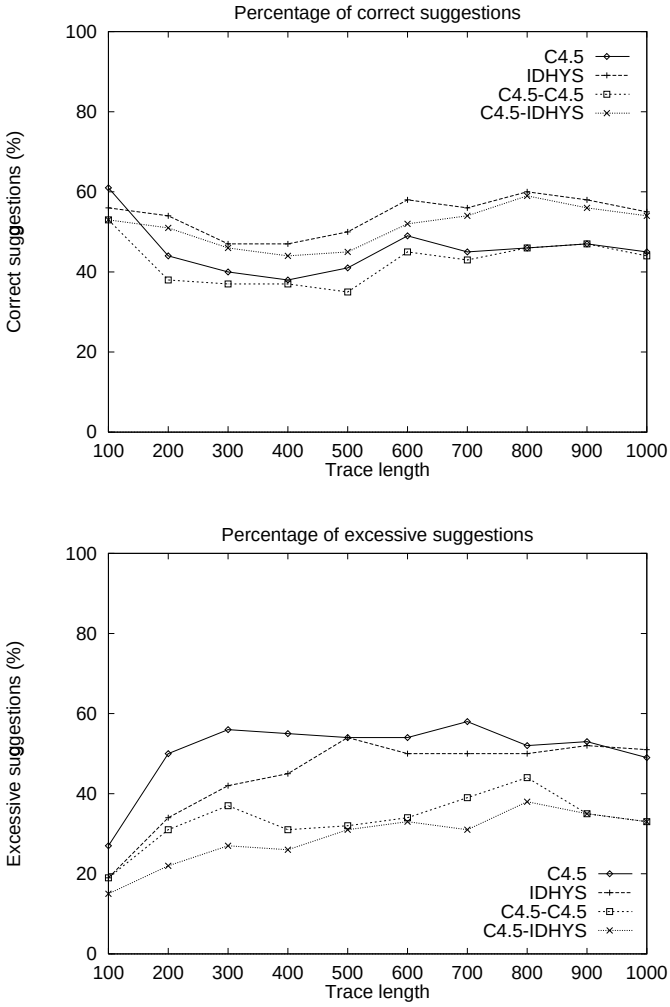


Figure 8: Percentage of repetitive tasks that APE has correctly suggested (top graph) and percentage of actions of the user, not preceding a repetitive task, for which APE has made a suggestion (bottom graph).

parameters of these actions are large, where repetitive sequences are not known in advance and not consecutive. It is able to replay repetitions composed of several actions and containing loops. The lessons learned from this work are:

- Minimizing the amount of wrong suggestions is an important issue. The system has to suggest the "replaying" of the right repetitive task at the right moment.
- Learning when to make a suggestion as well as what to suggest decreases the amount of incorrect suggestions.
- The system must not learn too general habits. We have shown that our new Machine Learning algorithm designed to learn habits reduces the amount of incorrect suggestions without degrading the quality of these suggestions.
- It is possible to learn user's habits to anticipate repetitive tasks. Experimental tests have shown that APE is usable and efficient: it learns user's habits in a matter of seconds and anticipates 63% of the repetitive tasks. It makes irrelevant suggestions for only 18% of the user's actions.

Concerning future work, it is clear that one of the main remaining issue is to present the suggestions made by the Assistant in a more attractive way. Programming by Demonstration studies have addressed the problem of creating a graphical representation of a program or a sequence of actions, and offer a direction for future research.

Although the integration of APE into a programming environment is an originality, it is not restrictive. APE could be integrated in other interactive environments like *Microsoft Windows*, *X window system* or *Apple MacOS*.

References

[Armstrong *et al.*, 1995] R. Armstrong, D. Freitag, T. Joachims, and T. M. Mitchell. Webwatcher: A learning apprentice for the world wide web. In *AAAI Spring Symposium on Information Gathering*, 1995.

[Böcker and Herczeg, 1990] Hans-Dieter Böcker and Jürgen Herczeg. What Tracers Are Made Of. In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 89–99, October 1990.

[Caglayan *et al.*, 1997] A. Caglayan, M. Snorrason, J. Jacoby, J. Mazzu, R. Jones, and K. Kumar. Learn sesame: a learning agent engine. *Applied Artificial Intelligence*, 11:393–412, 1997.

[Cypher *et al.*, 1993] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993. URL: http://www.acypher.com/wwid/.

[Darragh and Witten, 1991] J. J. Darragh and I. H. Witten. Adaptive predictive text generation and the reactive keyboard. *Interacting with Computers*, 3(1):27–50, 1991.

[Karp *et al.*, 1972] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *4th Annual ACM Symposium on Theory of Computing*, pages 125–136, Denver, Colorado, 1–3 May 1972.

[Lieberman, 1993] Henry Lieberman. Mondrian: A teachable graphical editor. In A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors, *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[Maes, 1994] P. Maes. Agents that reduce work and information overload. *Communications of the ACM, Special Issue on Intelligent Agents*, 37(7):31–40, July 1994.

[Mitchell *et al.*, 1994] T. M. Mitchell, R. Caruana, D. Freitag, J. McDermott, and D. Zabowski. Experience with a learning personal assistant. *Communications of the ACM, Special Issue on Intelligent Agents*, 37(7):81–91, July 1994.

[Mitchell, 1978] T. M. Mitchell. Version spaces: An approach to concept learning. Technical Report HPP-79-2, Stanford University, Palo Alto, CA, 1978.

[Mitchell, 1982] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, March 1982.

[Moon, 1986] D. A. Moon. Object-oriented programming with flavors. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–8, New York, NY, November 1986. ACM Press.

[Motoda, 1997] H. Motoda. Machine learning techniques to make computers easier to use. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*. Morgan Kaufmann Publishers, August23–29 1997.

[Quinlan, 1993] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[Ruvini and Fagot, 1998] Jean-David Ruvini and Christophe Fagot. IBHYS: a new approach to learn users habits. In *Proceedings of ICTAI'98*, pages 200–207. IEEE Computer Society Press, 1998.

[Ruvini, 2000] Jean-David Ruvini. *Assistance à l'utilisation d'environnements interactifs: Apprentissage des habitudes de l'utilisateur*. PhD thesis, Université Montpellier II, France, 2000. To appear.

[Teitelman, 1978] W. Teitelman. *Interlisp Reference Manual*. Xerox Palo Alto Research Center, 1978.