Visual AWK: A Model for Text Processing by Demonstration

Jürgen Landauer
Lehrstuhl für Programmiersprachen
Universität Erlangen-Nürnberg
Martensstr. 3,
91058 Erlangen, Germany

Masahito Hirakawa
Information Systems Laboratory
Hiroshima University
1-4-1 Kagamiyama,
Higashi-Hiroshima 724, Japan

{landauer, hirakawa}@isl.hiroshima-u.ac.jp

Abstract

Programming by Demonstration systems often have problems with control structure inference and user-intended generalization. We propose a new solution for these weaknesses based on concepts of AWK and present a prototype system for text processing. It utilizes 'vertical demonstration', extensive visual feedback, and program visualization via spreadsheets to achieve improved usability and expressive power.

Keywords

Programming by Demonstration, generalization, control structure, pattern-action paradigm, program visualization, spreadsheet metaphor.

1 Introduction

In text editing users are often confronted with reformatting tasks which involve large portions of texts, sometimes consisting of hundreds of lines. For example, let us assume we want to create mailing labels (figure 2) out of a given address list (figure 1). The task seems to be easy to automate since all paragraphs are similarly structured, containing a name, an address, and a phone number each.

However, both the built-in find & replace function and the macro recorder of the editor prove to be not flexible enough to handle the task, because their facilities for specifying search patterns and for dealing with special cases and exceptions are limited.

On the other hand, most current end-users estimate solving such tasks with one of today's programming languages as too difficult for them.

Programming by Demonstration (PBD) is a promising remedy here [3][6] since, by contrast, it promises nearly unlimited programming power though ease of learning and usage.

Therefore, a variety of PBD systems were proposed for this application domain [5] - [7] in the past. But PBD is not yet very widespread in commercial text editors because of some serious weaknesses.

In this paper we examine these weaknesses and present a new approach for the solution of the deficiencies of PBD. We introduce Visual AWK, a prototype text processing system developed at the Information Systems Lab of Hiroshima University based on the programming language AWK [1] which incorporates the new design approach. Extensive visual feedback and program visualization via spreadsheets improve both usability and expressive power.

Visual AWK is aimed at users without previous knowledge in programming, but with experience in text editor use.

The application domain are *semi-structured* texts. That is, texts that consist of equally structured entities, for instance lines or paragraphs, but may contain a few syntactically classifiable sets of exceptions with a different structure.

2 Design Principles

PBD systems "watch" the user performing a task on one or more concrete examples and infer an abstract *generalization*, i.e. an algorithm that can perform the given task not only on the presented examples but on a more "general" class of data with equal properties.

In this paper we focus on two of the problems involved in finding this generalization, which we estimate as essential for further progress in PBD (cf. D. Maulsby in [10]):

- The generalizations are often not what users intended, and
- the identification of control structure is often insufficient.

1049-2615/95 $4.00 © 1995 IEEE

```
John Bix, 2416 22 St., N.W., Calgary, T2M 3Y7. 284-4983
Tom Bryce, Suite 1, 2429 Banff Blvd., N.W., Calgary, T2L 1J4. 289-5678
Brent Little, 2429 Cherokee Dr., N.W., Calgary, T2M 2J6. 234-0001
Mike Hermann, 3604 Centre Street, N.W., Calgary, T2M 3X7. 229-4567
Helen Binnie, 2416 22 St., Vancouver, E2D R4T. (405)220-6578
Mark Willianms, 456 45Ave., S.E., London, F6E Y3R. (678)234-9876
Gorden Scott, Apt. 201, 3023 Blakiston Dr., N.W., Calgary, T2L 1L7. 289-8880
```

Figure 1: An address list (data taken from Witten and Mo [14])

2.1 User-intended generalization

Incompleteness of the example set: PBD systems usually request the user to start demonstrating a few or even only one example.

Further examples are used to *incrementally* compute converging constraints about the data. For instance, "v1.ps" and "v2.ps" allow for inferring "v#.ps", an additional example "wow.ps" suggests "*.ps" [10].

However, this approach relies on the user's ability to provide a complete set of examples. That is, a set with at least one representative of each case for which the system is expected to create correct output. The example set given above is hence incomplete since it also allows generalizations such as "##.ps OR ###.ps" and even "*.*" and numerous others with different levels of generalization.

In order to avoid possibly incorrect programs, we expect the user to provide *all examples before demonstration starts*. This is possible in our application domain since this complete set of examples is the entire text given as input.

Furthermore, in Visual AWK a generalization derived from *a demonstration with one example is immediately applied to all other examples*, i.e the entire text (users can restrict the scope of an operation with patterns as described below). So users can verify the correctness of the derived generalization and adjust the demonstration if necessary after each demonstration step.

So by contrast with other PBD systems which process data *horizontally* in the sense that they completely edit an example before incrementally adding new examples, Visual AWK can be said to process data *vertically*:

Intermediate demonstration steps are immediately applied to the entire example set so that users can verify the result before continuing demonstration.

Incompleteness of data descriptions of single examples: To infer a general rule from a set of demonstrations the PBD system must find out what the user intended when selecting and editing a piece of data. What is the similarity between the examples, how can they be undistinguishably *classified*? (Cf. "data description" in [4].)

Of course, it is reasonable to assume that the user knows the reason for a selection and is hence able to provide the data description. But this information is not available to the system if the user only motions, for instance, with a mouse click at a specific word in the given text without any explanation.

Many PBD systems try to recover the user's intent for selecting data by using domain-specific knowledge. If a spreadsheet user often rearranges ciphers in a specific way and if the built-in knowledge base has a description of the *syntactic class* "date", it can infer that the date format should be changed.

However, often more than one syntactic pattern matches the set of examples. An ambiguity must be dissolved: maybe all examples happen to be in bold characters, but they also have the common prefix "Friday".

Whereas this approach works fairly well in a limited application domain, it fails for general-purpose systems like the one proposed here because only few heuristics are available to reduce ambiguities.

Therefore, we use another approach in Visual AWK. Since it is only the users who have full knowledge about the intent of their demonstrations, we rather let them "explain" why they selected a piece of data. They augment the given text with grammar patterns and associated actions for processing it.

Can end-users describe grammars? Most end-users are not aware of the structure of their texts, especially not in computer (syntactic) terms, even if they wrote them. They rather have a hidden knowledge about common concepts such as "family name" and about concepts of their special field, for instance "bibliography entry in IEEE format".

In our case, however, users are only interested in a few syntactic entities at a time. The other data can just be skipped. Moreover, since the scope of the program to be generated is restricted to the given text, coincidental similarities of the data can be utilized. If all phone numbers in an address list happen to be local, the search pattern needs not include special cases such as international numbers. Hence,

less complex grammars are likely to be sufficient. Similar to Visual AWK's model AWK [1] we utilize regular grammars.

```
John Bix,
2416 22 St., N.W.,
Calgary,
T2M 3Y7

Tom Bryce,
Suite 1,
2429 Banff Blvd., N.W.,
Calgary,
T2L 1J4

Brent Little,
2429 Cherokee Dr., N.W.,
Calgary,
T2M 2J6

Mike Hermann,
3604 Centre Street, N.W.,
Calgary,
T2M 3X7

Helen Binnie,
2416 22 St.,
Vancouver,
E2D R4T

Mark Willianms,
456 45Ave., S.E.,
London,
F6E Y3R

Gorden Scott,
Apt. 201,
3023 Blakiston Dr., N.W.,
Calgary,
T2L 1L7
```

Figure 2: Output: mailing labels

Users specify grammar rules via *trial & error demonstrations*. They formulate patterns and text processing actions out of some primitives provided by the system and try them on an example. The system infers a generalization and immediately applies it to *all* other data which fulfills the predicate. Users then verify the result and adjust the predicate and/or action as needed.

This approach would not be possible without the ability of human beings to scan textual data *visually* with considerable speed and accuracy: Just try to *scan* this text for all instances of the word "visual" and you will find them quickly without having to *read* it, despite the fact that the lines of this text are far away from being similarly structured.

2.2 Control structure

Loop detection: Existing PBD systems were designed mainly to detect loops by identifying "similar" action sequences in an event history or program trace, heavily relying on the user's ability to demonstrate several loop steps in the same order so that similar patterns can be found.

But we believe that in the era of modeless computing users should no longer be forced to demonstrate in a specific order, especially if loop bodies to be repeated exceed a certain length.

On the other hand, some systems infer only sequences and let the users add control structure statements to a visual or textual program representation ([7], see also [8]). In our estimation this is too difficult for the intended users of Visual AWK.

Instead of explicit control structures, we present an editable control structure "form" to the user, which has shown to be sufficient for a large class of problems. It is the pattern-action paradigm of AWK: A single loop over all input lines (or "records") encloses a set of conditions ("patterns") and associated code ("actions") which is executed if the condition evaluates to true (figure 3). Visual AWK users can add new pattern-action pairs or edit them without knowing about the underlying program structure.

```
FORALL (input records) DO
  IF pattern1 THEN { action1 }
  IF pattern2 THEN { action2 }
  ...
  IF patternn THEN { actionn }
END DO
```

Figure 3: The AWK control structure

It was probably A. Aho et al. who made this observation first: "The implicit input loop and the pattern-action paradigm simplify and often entirely eliminate control flow" ([1], p. 185).

Conditionals: Early PBD systems inferred the reason for the user performing an unpredicted action (i.e. a special case) from the selected example itself, or its surrounding context, e.g. the previous or following word, style attributes etc.

But in general, there is no reasonable upper limit for the amount of context which needs to be taken into consideration. It could be one word, one line, or the entire text.

Apart from the fact that this involves an enormous computational effort and increases the possibility of ambiguities when inferring-data descriptions, it also reduces the expressive power of generated programs. Consider the usual conditional statement form:

IF (predicate) THEN { action }

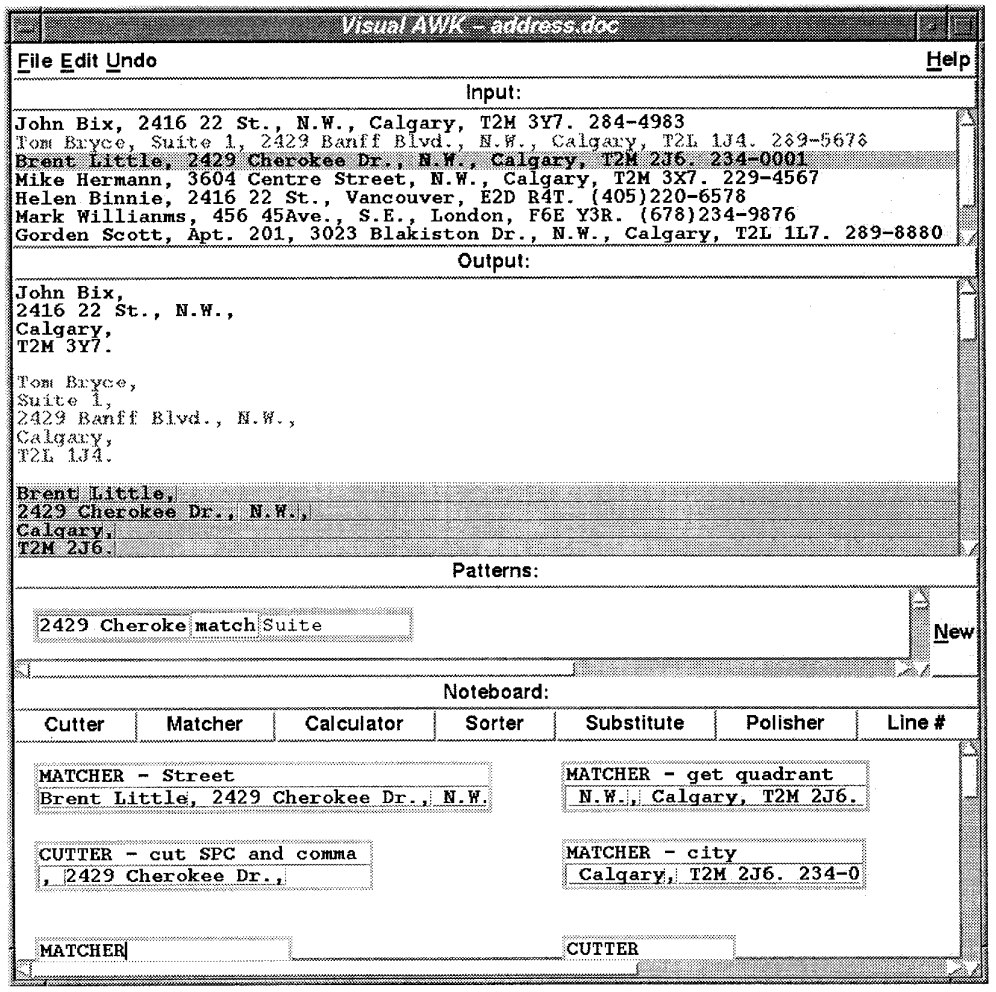

Figure 4: The Visual AWK main interface: third input line is selected for demonstration.

Here, the predicate and the action need notbe related at all. They can operate on different parts of the input text.

We make this difference explicit by completely separating the demonstration of predicate and action. Furthermore, predicates can consist of compound functions of arbitrary complexity.

3 Visual AWK System Overview

3.1 Actions

Visual AWK users need not start from scratch when defining structure rules. The system has two built-in concepts inherited from its model AWK because of their broad usefulness: A text is assumed to consist of lines (or paragraphs) with a newline character at its end, and lines consist of words ("fields") separated by whitespace characters.

So selecting an argument for an action could be done like this (figure 4): Users first select an arbitrary line for demonstration in the topmost text window by clicking on it. The system reacts by changing the background color of this line.

Then, for example, they select the third word on this line with a double-click. The system infers "word #3" and informs the user about this generalization by immediately highlighting the third word in all other lines, too.

Users can now drag the selection from the current example line into the corresponding (i.e. same color) line in the output window below the input window and drop it at the desired position. For example, if the example string is inserted after the (previously added) 6th item, corresponding strings are inserted at this position in each line by the system.

By typing text in the example output line, strings which are constant for each line can also be inserted.

The widget titled "Noteboard" is used for other, more complex ways of selecting data than by word position.

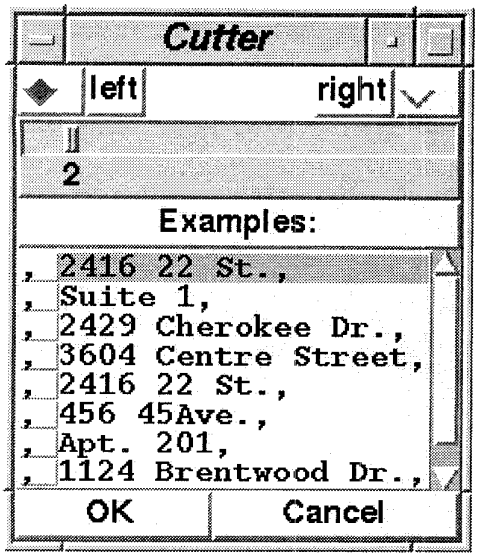

Figure 5: The Cutter: cut at second character from left.

For example, to select data by character position a selection is defined by pushing the "Cutter" button. A new rectangular icon appears on the Noteboard, initially empty. Users can now specify an argument by selecting another piece of data, e.g., word #3 in the input window (as described above), drag it from there and drop it onto the icon. The other parameter of the function, i.e., the character position is specified by double-clicking on the icon. A popup window appears (figure 5), and users can either edit the parameter by dragging the mouse over the example data displayed or by using the slider. Here all other instances of examples (i.e. all other third words) and the effect of the "cut" operation on them are also shown.

Other direct manipulation interfaces exist for other ways of selecting data by regular grammar expressions. For instance, the "Matcher" (figure 6) selects by string content. Regular expressions can be typed in or constructed using predefined subexpressions such as "numeric", "word", or "whitespace".

If these popups are closed by pushing OK, the corresonding Noteboard icon is updated and displays the calculated result for the current input line. For example, if the user has demonstrated a Cutter operation at character position 2 (counted from left), the icon displays the resulting parts in different colors.

If another input line is selected for demonstration, the Noteboard is updated accordingly. That is, its functions are evaluated with that line as input and the icons are updated accordingly.

The resulting string value can be further processed in two ways: users can either drag it directly to the output window and insert it there, or they can drop it on other newly created icons, thus making the output of one function the input of another one. By repeatedly connecting icons in that way, *functions* of arbitrary complexity can be *compounded.*

Therefore, the Noteboard can be said to visualize the generalized program and thus enable editing. Note the similarity to *spreadsheets*: Cells (icons) primarily display evaluation results and secondarily allow for access of their underlying functions. In Visual AWK, they can be accessed via double clicking their icon. The associated subwindow is popped up and users can edit their function specification.

The spreadsheet metaphor is utilized because a majority of users can nowadays be expected to be familiar with spreadsheets. Moreover, note that users appreciate that computations are *visible* and *local* to cells [5]. However, rather than representing functions textually, the direct manipulation interfaces mentioned above are used since end users are unlikely to be familiar with string formulas.

The Button panel contains not only selection but also composition functions. For example, the Calculator represents a simple formula editor for numerical calculations (not yet implemented in the prototype). "Substitute" allows for string substitutions.

3.2 Patterns

Users recognize 'special cases' while scanning the generated output after an edit operation when they encounter an output different from what they expected.

They first might try another way of demonstration, probably by using an instance of the special case as the current example line. But after some experimenting (which can easily be undone by selecting 'Undo' from the main menu) they realize that editing the special case towards a 'better' result makes all the other lines 'worse'.

The next step is finding an answer for the question "Why is this case special?", i.e. finding a characteristic predicate for all lines with this special case.

Again, this is done by trial & error. Users select the "New" button in the "Patterns" subwindow and choose a Boolean operation out of a menu. Presently arithmetic comparison (<, ==, >) and string comparison (similar to the "Matcher") are implemented. Then users provide parameters for the operations, again via drag & drop from the input window or from the Noteboard, or by typing in constant text.

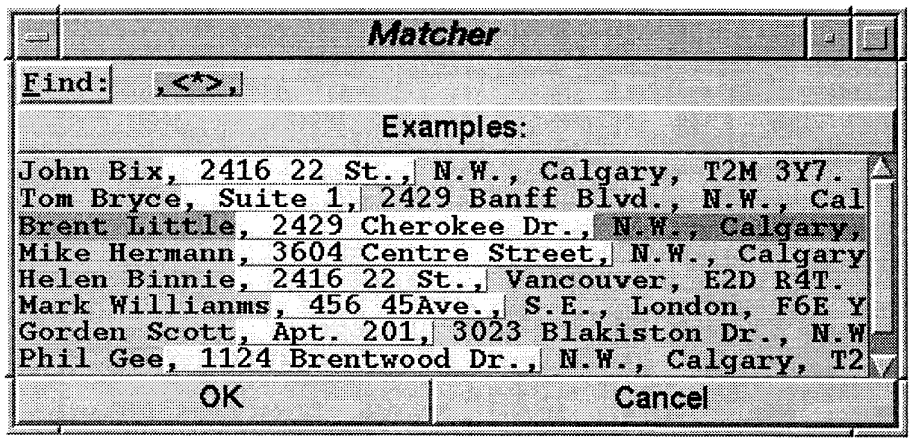

Figure 6: The Matcher: user selected the third line for demonstration; matching strings highlighted; pattern is: "arbitrary string between two commas"

As soon as the parameters are specified completely the system applies the patterns to all lines and visualizes the result: Non-matching input lines and their corresponding output lines are grayed out to focus attention on matching lines. If the users are content with the result, i.e. the predicate selects all lines with the desired property, they can edit the output. All editing actions in the output window are now only applied to lines matching the pattern currently visible in the pattern window.

When encountering new special cases additional patterns are specified. In the end the whole input file is partitioned into categories described by patterns.

If re-editing of a previously edited class of data is needed, users can access the line edit facility either by selecting the pattern or (more likely since it is easier for them) just select an instance of the category in the input window. The pattern associated with this line becomes the current pattern and again all non-matching lines are grayed out.

It is possible that several patterns match a line. To dissolve this ambiguity, the most recently created pattern has precedence.

3.3 Feedback

Since data is always copied when applying drag & drop operations, several identical instances of one string are likely to be visible at several places on the interface. Feedback about the source of information is crucial. Therefore, when moving the mouse cursor above a displayed string, the corresponding source string of the drag & drop operation is flashed. By iteratively using this mechanism users can backtrack to where the data originated.

In addition, icons can be labeled similar to comments in conventional programming languages.

The interrelationship between the current input line and its corresponding output line(s) is visualized by changing their background color.

4. An Example Session

For further illustration we now present an example session, adopted from [14]: The address list from figure 1 is to be reformatted so that "the name, street address, city, and postal code are on separate lines and the phone number does not appear" (p. 184).

Visual AWK users might start editing with, for instance, the first line as example. They select the first word (the person's first name) and drag it to the output window. Scanning the generated output shows that this trial was correct. So they do the same with the second word (the family name), and this also comes out to be correct.

However, after inserting a newline by pushing return and copying words no. 3, 4, and 5 they detect a mistake. This is because the street address sometimes consists of either two or three words.

In this situation users have basically two options: they can either define a new special case or try one or more new ways of selection.

Basically the latter is preferable since each new conditional branch needs to be demonstrated at least partly again.

Let us assume the users were told so and hence try another way of selection. After some experimenting they might find one of the proper solutions: The Matcher is the ideal tool for selecting strings between delimiters, in our case it is the commas (see figure 6). The street name is not yet extracted completely because

the matching string starts with a comma and a space character. They can be removed with the Cutter (figure 5), cutting after the second character (counted from left). The resulting tail part of the Cutter is quickly dragged into the output window and dropped there.

When selecting the next item on the example line, maybe again by selecting the string between commas, an irregularity becomes obvious. The output of line two does not show "N.W." but the street name instead. This is because line two contains a suite number as an additional data field. But so far this did not attract attention since the output seemed to be reasonable.

A good idea here might be to define a new pattern. There are various possibilities, but the fact that the constant string "Suite" appears in the previously created street description can be used to distinguish such lines from others.

Hence, the user can create a new string "match" pattern (figure 4) and copy the previously extracted street description pattern via drag & drop to the left side of the match operator. The right side is the constant string "Suite".

The pattern is now evaluated, and all non-matching input lines and their corresponding output lines are grayed out to visualize this partition of the data.

By iteratively selecting new pieces of data as described and adding it to the output, the desired list (figure 2) is created.

Other special cases such as an apartment number in line 7 can be treated in the same way as the "Suite".

5 End-User Testing

Eight individuals (students from Hiroshima university's faculties of education and literature) were asked to perform editing tasks similar to the example described above. All of them had no programming experience, but they were frequent text editor users. Some of them had used spreadsheets in the past. These tests were rather qualitative than quantitative since the testers do not represent an average user population. For example, they were rather homogeneous concerning their age.

Apart from the most difficult task, all users could solve the given problems. Difficulties arose with the visual representation of regular expressions in the Matcher which is very rudimentary. Obviously more research must be conducted to obtain user-friendly ways for visualizing regular expressions (for a promising approach see [2]).

Another observation made is that users tended to underestimate the importance of the verification of the generated output after each demonstration step. If they did not detect errors immediately they became confused later when errors were propagated through subsequent demonstrations.

6 Implementational Notes

Immediate feedback is expensive in terms of computational power and so systems like the one presented here have only become possible since recently.

Fortunately the users by themselves cannot scan the whole data at once, only the part which is currently visible on the screen. So only this much smaller portion needs to be updated.

By separating output display and program interpretation into two processes users can work parallel to the interpreter. This is especially helpful for users with some experience with the system's behavior because they need not verify the results after each step.

The prototype was written in Tcl/Tk [6] because it combines both rapid user interface creation and string processing facilities. Tcl/Tk is interpreted, and so the Visual AWK prototype requires a fast workstation in order to maintain a reasonable response time. However, an optimized and compiled full-scale implementation will without a doubt achieve satisfying response times comparable to the ones of corresponding AWK programs, even on a PC.

7 Comparison with Related Work

Visual AWK is not the first system which utilizes a pattern-action scheme: The Triggers system [12], for example, has condition-action rules which are executed in an implicit while loop. The Find & Do facilities of systems such as Moctec [8] implement a similar control structure.

What makes Visual AWK unique are the vertical demonstrations, the immediate application of generalizations to the entire data, resulting in a better control of the user over the system.

Concerning the application domain, TELS [14] and Turvy [9] are probably the most similar systems to Visual AWK.

But neither provides means to specify conditions which depend on data other than the data to be affected. However, Turvy has introduced the concept of a "focus of attention" to which users can point in order to guide the pattern inference. But still it always takes the primarily

selected data into consideration, thus reducing the possible expressive power of programs.

Spreadsheets have been proposed generally for end-user programming systems before [11]. Visual AWK utilizes it as the first system for PBD program visualization.

9 Conclusion

The approach presented in this paper makes the following contributions to PBD research:

(1) Being a general-purpose system, Visual AWK does not rely on knowledge-based inference. Instead, it introduces the concept of *trial & error demonstration*. It is the user who "explores" and thus specifies patterns and actions, resulting in a better control of the system through the user and hence in a higher reliability.

(2) It lifts the limitations of previous PBD systems by allowing separately demonstrated conditions and actions for conditionals and by introducing a scheme for function composition.

This can only be achieved through the visualness of the approach, explained below:

(1) Trial & error demonstration entirely depends on the ability of humans to scan large portions of similarly structured data visually.

(2) Spreadsheet-like visual program representation enables end-users to edit programs after erroneous demonstrations and thus allows for more complex programs.

Full-scale systems modeled after the presented prototype can be included in existing text editors as an additional feature. Users can select parts of their texts, process it with Visual AWK in a new window and put it back or save it as a new file.

Note, however, the generality of the proposed design principles: For example, similar to AWK, Visual AWK can also serve as a model for a demonstration-based visual database query language. A text with similarly structured entities can be viewed as a relational database. Paragraphs become records, and "semi-structured" means that records can have fields which possibly differ in number and type.

The preconditions that the entire input data (examples) must be available before beginning and that demonstrated programs are only valid for this data seem to limit the scope of Visual AWK to single-use programs. However, these conditions can be weakened to *the input data must be structurally equivalent to the data which was used for creating the program.* End-users have no means to discover this, but by simply experimenting they should always be able to process their monthly report with the same program, or maybe adjust it to slightly changed needs. This is possible since the program itself is represented by the internal state of the interface which can be saved and later restored.

Acknowledgments

The authors wish to thank Professor T. Ichikawa from Hiroshima University and G. Viehstaedt from Erlangen University, for useful comments. The first author was supported by a grant from German Academic Exchange Service (DAAD).

References

[1] A. Aho, B. Kernighan, P. Weinberger, "The AWK programming language", Reading (MA), 1988 (reprint).

[2] M. A. Bell, D. Jackson, "String Pattern Matching in a Visual Programming Language", Techn. Report, Univ. of Liverpool, UK, 1994.

[3] A. Cypher (ed.), "Watch What I Do: Programming by Demonstration", MIT Press, 1993.

[4] D. C. Halbert, "SmallStar: Programming by Demonstration in the Desktop Metaphor", in [3], pp. 102-123.

[5] D. G. Hendry, T. R. G. Green, "Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model", Int. J. Human-Computer Studies (1994) 40, pp. 1033-1056.

[6] M. Hirakawa, T. Ichikawa, "Visual Language Studies - A Perspective", in: Software - Concepts and Tools (1994) 15, Springer-Verlag, pp 61-77.

[7] H. Lieberman, "Tinker: A Programming by Demonstration System for Beginning Programmers", in [3], pp. 48-64.

[8] D. Maulsby, "Prototyping an Instructible Interface: Moctec", Proc. SIGCHI '92, 1992.

[9] D. Maulsby, "The Turvy Experience: Simulating an Instructible Interface", in [3], pp. 239-269.

[10] B. A. Myers, "Demonstrational Interfaces: Coming Soon ?", Proc. SIGCHI '91, New Orleans (LA), 1991.

[11] B. A. Nardi, J. R. Miller, "The spreadsheet interface: A basis for end user programming." INTERACT '90, pp. 977-983. Amsterdam, 1990.

[12] R. Potter, "Triggers: Guiding Automation with Pixels to Achieve Data Access", in [3], pp. 360-380.

[13] J. Ousterhout: "Tcl and the Tk Toolkit", Addison-Wesley, Reading (MA), 1993.

[14] I. H. Witten, D. Mo, "Learning Text Editing Tasks from Examples", in [3], pp. 182-203.