Repeat and Predict – Two Keys to Efficient Text Editing

Toshiyuki MASUI
Software Laboratories
SHARP Corporation
2613-1 Ichinomoto-cho
Tenri, Nara 632, Japan
Tel: +81-7436-5-2468
E-mail: masui@shpcsl.sharp.co.jp

Ken NAKAYAMA
Department of Information Science
Faculty of Science
The University of Tokyo
3-8-1 Komaba, Meguro, Tokyo 153, Japan
Tel: +81-3-5478-0520
E-mail: ken@is.s.u-tokyo.ac.jp

ABSTRACT
We propose a simple and powerful predictive interface technique for text editing tasks. With our technique called the *dynamic macro* creation, when a user types a special "repeat" key after doing repetitive operations in a text editor, an editing sequence corresponding to one iteration is detected, defined as a macro, and executed at the same time. Although being simple, a wide range of repetitive tasks can be performed just by typing the repeat key. When we use another special "predict" key for conventional prediction techniques in addition to the repeat key, wider range of prediction schemes can be performed depending on the order of using these two keys.

KEYWORDS: Text Editing, Predictive Interface, Programming By Example, PBE, Programming by Demonstration, PBD, Keyboard Macro, Dynamic Macro Creation

INTRODUCTION
Various techniques for programming by demonstration (PBD) and predictive user interface have been proposed to support easy programming or to reduce the burden of doing similar operations repeatedly[2][6]. Most PBD systems are for graphical user interfaces (GUI,) but PBD techniques for text editors and other keyboard-based systems have also been proposed. For example, Darragh's Reactive Keyboard[3] predicts the user's next keystrokes from the statistic information gathered by the user's previous actions. In Nix's Editing by Example system[7], users can tell the system to infer the editing procedure by showing both the text before modification and the one after modification. The inferred procedure should be of the "gap programming" form, which is a subset of string substitution using regular expressions. Mo's TELS system[5] generalizes users' iterative operations and infers an editing procedure including loops and conditional branches. If the system's guess is wrong, users can incrementally correct it until it does the right thing for them. Since the procedure generated by TELS can include branches and loops, it can perform complex tasks which cannot be done by mere string substitutions.

Although these systems can infer complex editing operations from examples, the procedure to perform the inference is rather complicated and they are not suited for simple repetitive tasks. In this paper, we introduce a simple and powerful prediction technique for text editing tasks called the dynamic macro creation, and show its applications. We also show how this technique can be extended by using conventional simple prediction techniques in combination.

Published in:
Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'94) (April 1994), ACM press, pp. 118–123.

REPEAT PREDICTION
Keyboard Macro
In many text editors, *keyboard macro* is used to substitute a long sequence of operations by another single operation. A keyboard macro is usually defined through the following steps: first, the user tells the editor to start recording a keyboard macro; second, he types the sequence of commands which he wants to define as a new macro; and finally, he tells the editor to stop the recording. For example, if a user of GNU Emacs wants to define a macro to insert a "**%**" at the top of every line, he types "Ctrl-X **(**" to start the recording, types "Ctrl-A **%** Ctrl-N" to insert a "**%**" at the top of the current line and go to the next line, and type "Ctrl-X **)**" to stop the recording. After the recording is finished, he can invoke these operations by typing "Ctrl-X **e**." Although keyboard macro is a general and powerful tool for repetitive editing tasks, it has several disadvantages. First, users have to remember three commands to record and invoke a keyboard macro. Novice users do not tend to remember them just for simplifying repetitive tasks. Second, it is not possible to define the command sequence after they are executed: that is, a user should know that a sequence of commands is used many times, well *before* he actually executes them. In reality, repetitive tasks are often recognized *after* execution. Third, since the procedure of defining a keyboard macro is not simple, it is not useful for short and small repetitive operations.

Dynamic Macro

We propose a simple and powerful method of creating a keyboard macro from repetitive user operations, which we call the *dynamic macro* creation method. Dynamic macro works as follows: All the recent user operations in a text editor is logged as a string, and when a special "repeat" command is issued by typing a special key denoted as [REPEAT], the system looks for repetitive operations from the end of the string. If such operations are found, they are defined as a macro and executed. If [REPEAT] is typed again, the macro is executed again. For example, when a user enters a string "**abcabc**" and types [REPEAT] after that, the system detects the repetition of "**abc**," defines it as a macro, and executes the macro, resulting in another "**abc**." When the user types [REPEAT] again, one more "**abc**" is inserted. Similarly, when a user inserts a "**%**" at the top of two lines by doing the same operations twice and types [REPEAT] after that, the operations are defined as a macro and executed, and as a result, another "**%**" is inserted at the top of the third line.

Dynamic macro does not suffer from the shortcomings of keyboard macro. Users should only remember that typing [REPEAT] makes the system do the repetitive task once more, instead of remembering three different operations of keyboard macro. The macro is defined *after* doing ordinary editing tasks, without telling the editor when to start the recording. More importantly, in spite of its simple-looking appearance, dynamic macro is applicable to great many editing situations, which we will show in later sections by examples.

Details of the Algorithm

The actual process of detecting repetitive operations consists of the following two strategies.

Rule1: If there exist two same consecutive sequences of operations just before typing [REPEAT], define the sequence as a macro. If there exist more than one such sequences, take the longest one. For example, if the user types [REPEAT] after "**abccabcc**," define "**abcc**" as the macro, not "**c**."

Rule2: If there exists no such sequence, look for a pattern XYX just before [REPEAT], where X and Y denote nonempty sequences of operations. If there exist such sequences, define XY as a macro, executing only Y for the first [REPEAT]. If there exist more than one such sequences, take the longest X and take the shortest Y with that X. For example, if the user types [REPEAT] after "**abracadabra**," take "**abra**" as X and "**cad**" as Y, not "**a**" as X and "**br**" as Y.

With Rule2, typing [REPEAT] after "**abcdeab**" makes "**cde**" inserted after it, and one more [REPEAT] makes "**abcde**" inserted. This means that users do not have to carry out the same operations twice before typing [REPEAT], but they have to do one iteration plus only the first part of the second iteration.

Examples

We here show some examples of using dynamic macro implemented on GNU Emacs.

Adding Comment Characters Figure 1 shows how [REPEAT] works for simple tasks like adding comment characters to consecutive lines.

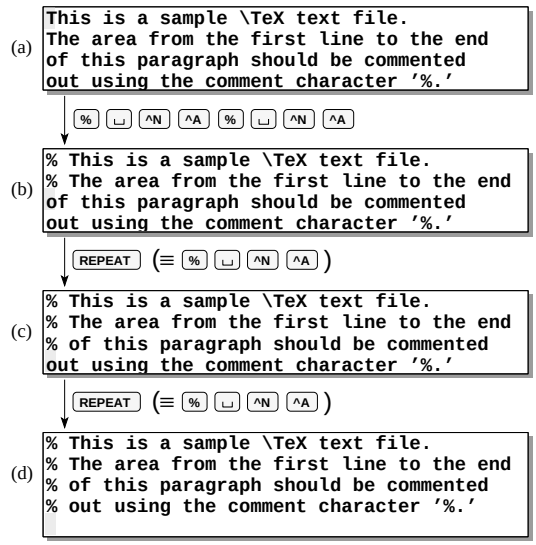

Figure 1: Adding comment characters to each line.

Figure 1(a) shows the original text. When a user types [%] [␣] [^N] [^A] [%] [␣] [^N] [^A], he gets (b) [1] . If he types [REPEAT] here, the system detects the repetition of [%] [␣] [^N] [^A], defines the sequence as a macro (Rule1,) executes the macro, and gets (c). Hitting another [REPEAT] results in (d). To get this result, the user do not have to do exactly the same operations twice before typing [REPEAT]. Figure 2 shows the case when a user types [REPEAT] after typing [%] [␣] [^N] [^A] [%].

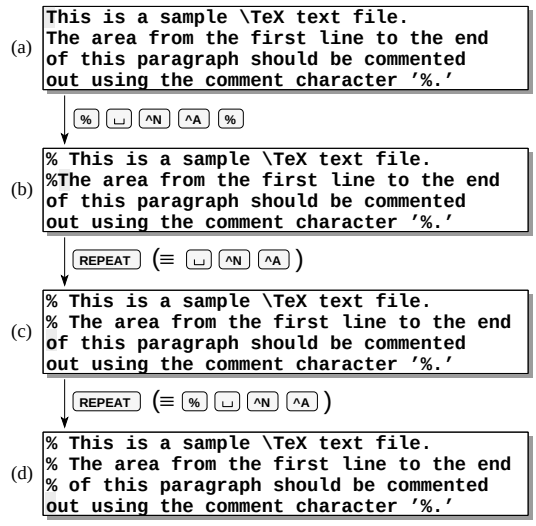

Figure 2: Adding comment characters to each line.

[1] [␣] denotes the space key and [^N] denotes the Ctrl-N key. Ctrl-N moves the cursor to the next line and Ctrl-A moves the cursor to the top of the current line.

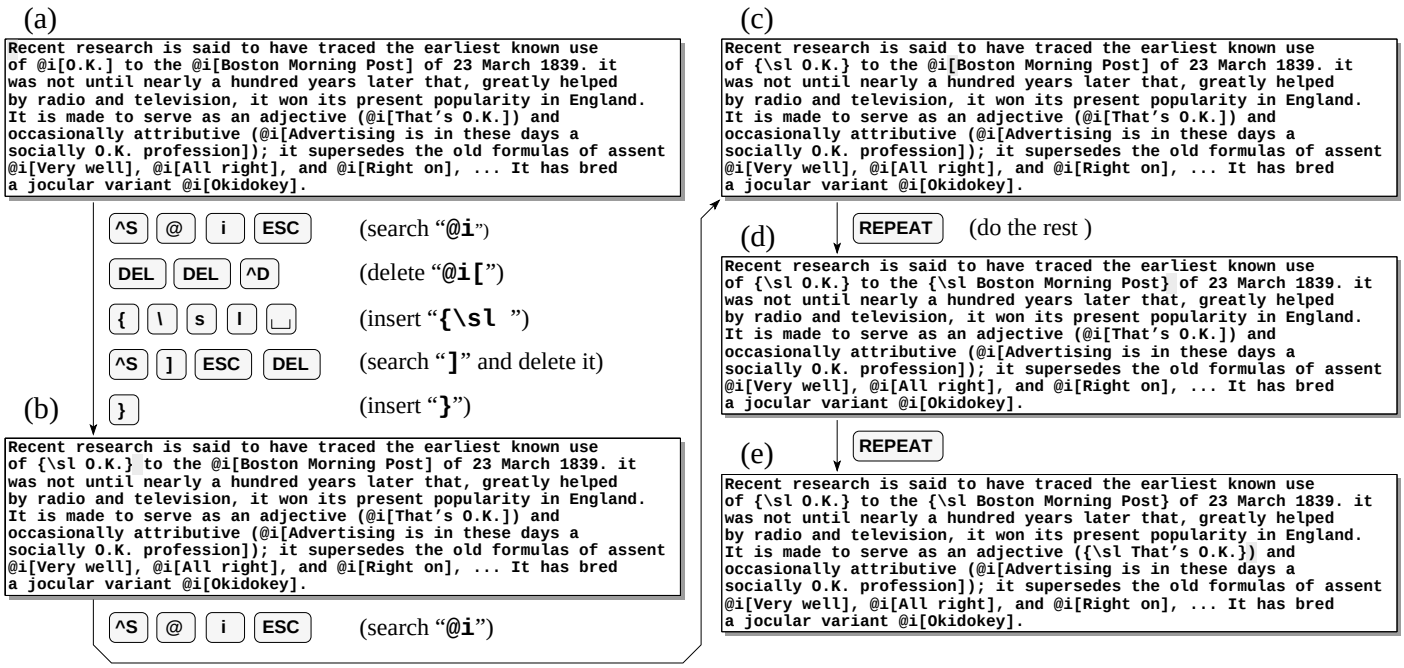

Figure 3: Changing formatting directives. (Example taken from [7].)

In this case, since no sequence of operations is executed twice before REPEAT, the system searches the pattern XYX and gets % for X and ␣ ^N ^A for Y (Rule2.) Then the system executes Y and gets (c). One more REPEAT makes the system execute both X and Y, and results in (d).

In this way, the user gets similar results by typing REPEAT at any moment while he is doing repetitive operations. This means that users do not have to care when to type REPEAT.

Pattern Search and Replace Here we show a more complex example presented in [7]. The objective here is to substitute all the occurrences of "**@i[***text***]**" (Scribe's directive for Italic) in Figure 3(a) by "**{\sl␣***text***}**" (TEX's directive for Italic.) With Nix's system, a user shows the system both the text before modification (e.g. "**@i[O.K.]**") and the one after modification (e.g. "**{\sl␣O.K.}**",) and the system infers the substitution operation. In our system, a user can do the job by typing REPEAT after doing slightly more then one iteration of the substitution operations. (See Figure 3.) Unlike Nix's system, the user do not have to perform any extra operations to the system except typing REPEAT several times after ordinary text substitution procedure.

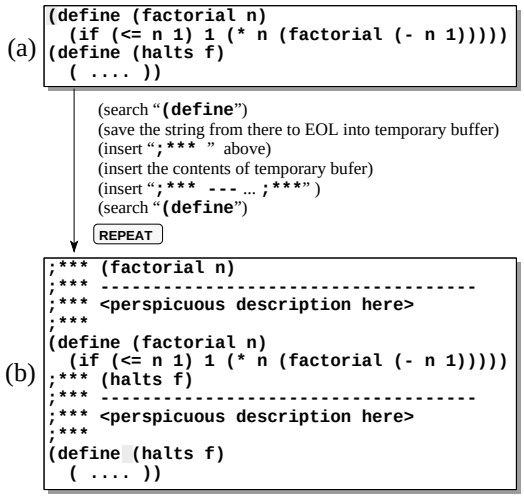

Figure 4: Adding comment lines to Lisp Functions.

Adding Comment Lines to Each Function We show another example that also appeared in [7]. The job here is to add several comment lines above every function definition of Figure 4(a). This is done by long steps of operations, but typing REPEAT after doing the first part of the second iteration results in (b), and more REPEAT will add similar comment lines to the following function definitions.

Advantages of Dynamic Macro

The advantages of dynamic macro is as follows. First, it is *simple to use.* Users only have to remember that they can type REPEAT to make the system do their repetitive chore. They can type REPEAT at any moment during the repetitive operations, and they do not have to tell the system to start or stop recordings. Second, it is *powerful.* As we have shown in previous examples, dynamic macro works well for a variety of simple to complex repetitive tasks where only keyboard macro was applicable. Third, it is *easily implemented.* The system should just keep the log of recent user actions for the

name	system	prediction source
dabbrev	Emacs	strings in the document
completion	Emacs	filenames, etc.
"**.**"	vi	last command
"**!!**"	csh	last command
"**!**(string)"	csh	command history
"ESC **l**"	tcsh	filenames
dynamic macro	(Emacs)	repetitive keystrokes
Reactive Keyboard	shell	keystroke statistics

Figure 5: Comparison of popular predictive interfaces.

implementation of dynamic macro [2] . Forth, it *does not interfere with users* in any sense. Logging user actions is an easy task for most systems and it does not slow down the application. Nothing happens unless users touch [REPEAT]. Finally, it is *general* in that any system with keyboard interface can adopt this technique.

Limitations of Dynamic Macro

Dynamic macro is just a syntactic prediction technique, and cannot generalize any rule from the user input, nor can do any semantic prediction from the context or other information around. Furthermore, although dynamic macro predicts right in most cases, it sometimes guesses differently from the user's expectation. For example, if a user types [REPEAT] after [TAB] [TAB] [a] [b] [c] [RET] [TAB] [TAB], Rule1 applies and another [TAB] is executed, which may be different from the user's expectation of [a] [b] [c] [RET] [3] . If a user types [REPEAT] after "**abracadabra␣ab**," expecting "**racadabra␣**," Rule2 applies and only "**ra␣**" appears, since the pattern XYX matches "**ab**/**ra␣**/**ab**," not "**ab**/**racadabra␣**/**ab**." Since dynamic macro is a prediction technique, it is not possible to make a prediction rule which always fit to every user's expectation. However, these problems can be solved by adding a new key for prediction, denoted as [PREDICT]. We show how this scheme works in the next section.

USING DYNAMIC MACRO WITH CONVENTIONAL PREDICTION TECHNIQUES

In this section, we show how we can use dynamic macro with simple conventional prediction techniques.

Conventional Prediction Techniques

Many kinds of simple prediction techniques are used in popular text editors and other interactive programs. For example, "shell" programs on UNIX have the "history" facility, which allows users to re-execute one of the formerly-issued commands by typing "**!**" and the abbreviation of the command. GNU Emacs provides **dabbrev** function, which expands the substring entered by the user into a full string in the same document which begins with the same substring. Figure 5 shows some of the prediction techniques popular to UNIX users, along with dynamic macro and the Reactive Keyboard[3].

Many of the prediction techniques are functionally equivalent in that they predict the next string from domain-specific dictionaries and other available information. In this sense, using only one "prediction key," or [PREDICT], is usually enough for the prediction, instead of using different keys corresponding to each prediction scheme. Figure 6 shows a sample usage of [PREDICT] using filenames and an English dictionary as the prediction source.

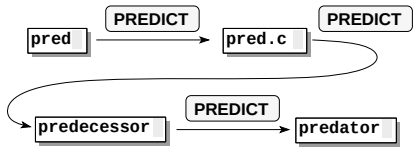

Figure 6: Prediction with filenames and English dictionary.

The semantics of [PREDICT] is as follows: If it is pressed for the first time, predict the next string which is most probable at the context. If [PREDICT] is pressed again, undo the previous prediction and show a new candidate to the user.

Using Prediction Key with Repeat Key

If we use [PREDICT] with [REPEAT] in combination, not only the problem of dynamic macro shown in the last section is solved, but more sophisticated prediction can be performed.

The most serious problem of dynamic macro is that users cannot do anything when the prediction by [REPEAT] was not what they expected. However, using [PREDICT], users can try different candidates whenever the prediction by [REPEAT] was an unexpected one. As we have shown in the previous section, [REPEAT] predicts [TAB] after typing [TAB] [TAB] [a] [b] [c] [RET] [TAB] [TAB], applying Rule1. But typing another [REPEAT] can change the prediction scheme and make Rule2 active, and next candidate, [a] [b] [c] [RET], is predicted instead, undoing the last prediction of [TAB]. If this prediction is the one in the user's mind, the user can then type more [REPEAT] to go on the prediction. (See Figure 7.)

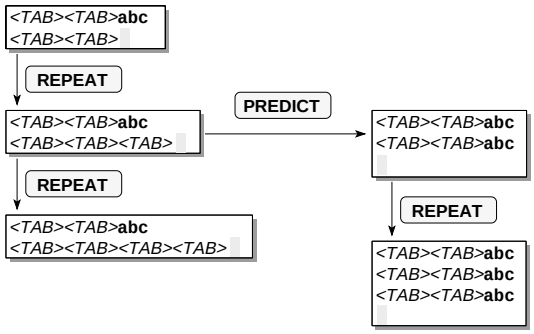

Figure 7: Changing the prediction scheme using [PREDICT].

[2] GNU Emacs is always keeping a list of 100 recent keystrokes and it can be read by **recent-keys** function.

[3] It is not a good idea to make Rule2 have higher precedence than Rule1, though, since [REPEAT] after "**bab** *long-forgotten-sequence* **abab**" would execute the forgotten sequence instead of another "**ab**."

Figure 8 shows the solution to the other example shown in the last section. In this case also, the user can tell the system his expectation by typing PREDICT after REPEAT made a wrong guess.

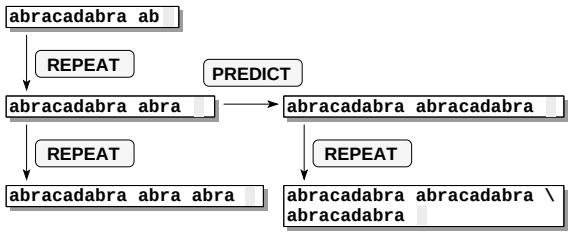

Figure 8: Changing the prediction scheme using PREDICT.

We can go further by extending the semantics of REPEAT as follows: if it is pressed after REPEAT or PREDICT, execute the same prediction scheme again; otherwise, predict the next string using the dynamic macro technique. With this extension, users can first select the prediction strategy using PREDICT, and then apply it repeatedly by REPEAT. Figure 9 shows an example of using this technique.

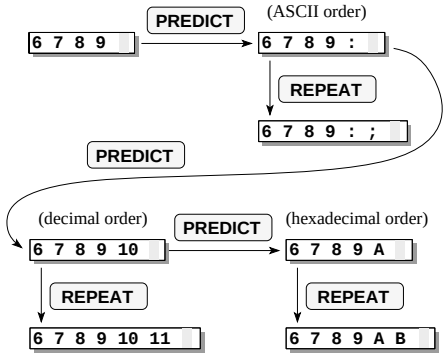

Figure 9: Select and repeat prediction strategies.

Here, the first PREDICT sets the prediction scheme to "ASCII order," and predicts "**:**␣". If it is what the user wants, the user can type REPEAT to continue the prediction under the same prediction scheme and gets "**;**␣". If it is not, the user can type PREDICT again after the first PREDICT, set the prediction scheme to "decimal order," and get the next candidate "**10**␣", and so on.

We can go even further to extend the meanings of REPEAT and PREDICT, and use them as mode-specific prediction keys which correspond to quantitative and qualitative prediction, respectively. Figure 10 shows how PREDICT and REPEAT can be used when writing LaTeX documents with this kind of extension.

First, the system predicts "**{itemize}**..." from the current context ending at "**\begin**," setting the prediction scheme to "LaTeX-itemize." If the user types REPEAT just after that, the system generates "**\item**," from the knowledge that it

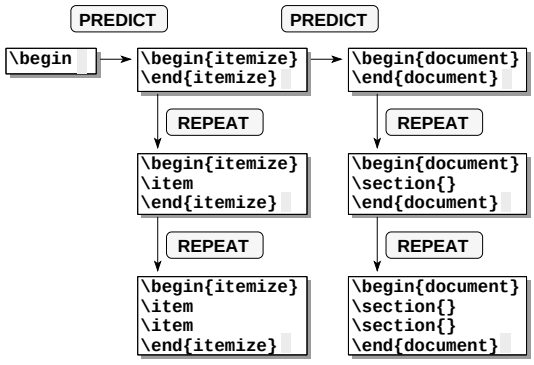

Figure 10: Qualitative and quantitative prediction.

is currently in the LaTeX-itemize mode. If it is not the user's intention, he can type PREDICT again, setting the prediction scheme to "LaTeX-document."

All together, the functions of REPEAT and PREDICT are shown in Figure 11. When REPEAT is pressed, the system goes to state S_{R1} and dynamic macro is executed. If PREDICT is pressed there, the system goes to state S_{P2}, undoes the last prediction, and performs another new prediction. Various other prediction schemes can be tried based on the state transition shown in Figure 11.

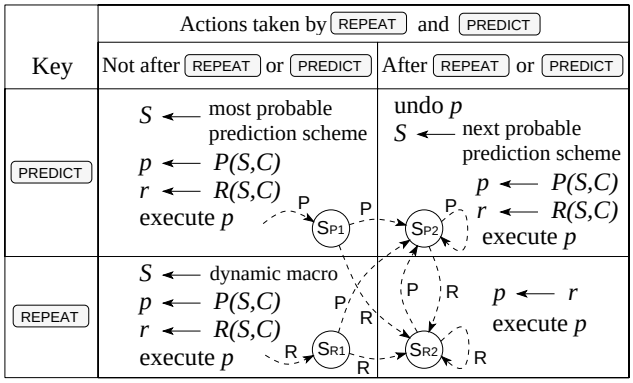

Figure 11: Actions by REPEAT and PREDICT.

Advantages of Using Prediction Key and Repeat Key

Advantages of using PREDICT along with REPEAT is as follows. First, unexpected-prediction problem of dynamic macro can be solved. Second, a variety of prediction techniques can be applied by using only two keys. Third, the meanings associated to the two keys are simple and familiar to current users of text editors. Users of Japanese editors and word processors are especially familiar with this kind of prediction techniques, since Japanese characters are usually "predicted" from ASCII keystrokes using PREDICT-like conversion keys on almost all the Japanese editors and word processors.

DISCUSSIONS

Usability Test

Our system has been used and tested by our colleagues for more than one year, and now it is gradually spreading among the Emacs users community. We have asked some of the users to log their activities related to REPEAT and PREDICT, and got the following results.

- The "hit ratio" of dynamic macro varies from person to person. The highest ratio was 100%, which was achieved by a user who uses REPEAT only occasionaly (five times a month) and very deliberately. The average ratio was around 85%.
- The average length of the predicted keystrokes was about 5, while the most frequent length of the predicted keystrokes was between 3 and 4. This shows that dynamic macro is basically useful for short repetitive keystrokes, while keyboard macro is good for longer sequences.
- Real Emacs experts do not use dynamic macro very often. This is because they already know many special functions corresponding to frequently-used small repetitive tasks like inserting comments, indentation, etc. On the other hand, most non-expert users (including the authors) like the interface and are using it for everyday editing tasks.

Requirements for Predictive Interface

We believe that any predictive interface technique should satisfy the following requirements:

1. Extra operations for prediction should be minimal.
2. Predictions should be correct in most cases.
3. Users who do not use predictive features should not suffer from its existence.
4. Predicted interface should not make users feel uneasy because of its wrongdoings.

We use only two keys, which is minimal. Just as shown in our usability test above, the hit ratio of the prediction is high enough. The system does nothing other than taking a log unless prediction keys are pressed, and it does not interfere with users. With our implementation on GNU Emacs, users can issue an **undo** command at any time to get back to the original state whenever they find something is going wrong because of the unexpected prediction.

System Notification

Some predictive systems notify the user when they can predict the user's next action. KeyWatch[4] generates a beep sound when it finds repetitive user actions. The Reactive Keyboard[3] always shows a candidate of next user action on the command line. Eager[1] displays a special icon when it detects repetitive user operations. In contrast to these systems, our system does not notify anything, because system notification is often a nuisance rather than a help for everyday users.

Macro Definition and Execution

The reason why dynamic macro requires only one key is that macro definition and execution are done at the same time. In many predictive systems, macro definition and execution are separated and users can edit the definition before execution. Although this technique is preferred for more complex systems, we did not take this approach because the prediction by dynamic macro is correct in most cases. One reason for this is that we implemented dynamic macro on GNU Emacs, where different meanings are bound to different keystrokes. In Emacs, the keystrokes for going to the top of a line is different from the keystrokes for going to the left. This condition does not hold for conventional GUI, where systems cannot distinguish absolute mouse motions from relative ones. Sun Microsystem's Textedit system has an "Again" button, using which all the key sequences after the last mouse operation until pressing the button are executed again at the current mouse position. Just like dynamic macro, this can be seen as an implicit definition of keyboard macro. However, it is not as powerful as our system, since users still have to care when and where to start recording the macro, and the functions provided by the editor are much less powerful than those provided by GNU Emacs.

CONCLUSIONS

We proposed a simple and powerful predictive interface technique for editing texts using only two keys, REPEAT and PREDICT. Although being simple, our technique covers a wide area which was formerly covered by keyboard macro and other predictive interface techniques. We like to apply this technique to wider areas other than text editing tasks.

ACKNOWLEDGEMENTS

We thank Makoto Tawada of Nagaoka University of Technology, Nobuhiko Funato, Yuji Ichikawa and Akira Imai of SHARP Corporation for giving us valuable comments to our system.

REFERENCES

[1] Cypher, A. Eager: Programming repetitive tasks by example. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'91)* (April 1991), Addison-Wesley, pp. 33–39. also in [2].

[2] Cypher, A., Ed. *Watch What I Do – Programming by Demonstration*. The MIT Press, Cambridge, MA 02142, 1993.

[3] Darragh, J. J., Witten, I. H., and James, M. L. The Reactive Keyboard: A predictive typing aid. *IEEE Computer 23*, 11 (November 1990), 41–49.

[4] Micro Logic Corp. *KeyWatch*. POB 70, Hackensack, NJ 07602, 1990.

[5] Mo, D. H., and Witten, I. H. Learning text editing tasks from examples: a procedural approach. *Behaviour & Information Technology 11*, 1 (1992), 32–45. also in [2].

[6] Myers, B. A. Demonstrational interfaces: A step beyond direct manipulation. *IEEE Computer 25*, 8 (August 1992), 61–73.

[7] Nix, R. P. Editing by example. *ACM Transactions on Programming Languages and Systems 7*, 4 (October 1985), 600–621.