

Bridging the Translation Gap with ExpandHelp



ABSTRACT

We introduce a flexible command translation framework that can generate a complex command string from vague keywords given by the user. When people use computers to perform tasks, there is often a huge mismatch between the users' intention and required actions. For example, when a Chinese person wants to “房間暗一点” (make the room darker), he may have to find that he should “turn off the ceiling light” for the task, translate it to a required action “toggle the wall switch” or a command like “\$ `iot clight 0`”. There is a huge gap between “房間暗” and “\$ `iot clight 0`”, and people are perpetually suffering from such multi-level “translation gaps”. Translation gaps exist even for experienced computer users, but they are serious for vast amount of people whose mother languages are far from the command syntax required for handling computers. We propose a simple and flexible translation framework ExpandHelp that can be used in various situations where such translation is necessary. We show how our framework can be used in the GitHelp system with which a user can easily generate complex git commands from fragments of users' intentions in various languages.

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

Author Keywords

Help systems; Regular expression, Input method, Generate and filter; ExpandHelp; GitHelp

INTRODUCTION

When people want to perform a task using modern artifacts, people frequently find difficulty handling them properly. Even when manuals and help documents are provided, people don't use them[10] partly because of the vocabulary problem[4]. We should translate our vague intentions in our brain into actions required for the artifacts. When we want to watch a movie, we may pick up a DVD disk, put it into a DVD player, push the play button, and set up the audio and

the monitor. When we feel thirsty, we may go to the kitchen, pick up a glass, and turn the faucet on to fill the glass with water. In either of these cases, there is a tremendous gap between the user's intentions and required actions. All the living creatures are seemingly performing such translations between intentions and actions without difficulty, but in the modern world, people are suffering from serious gaps between what they want to do and what they have to do.

Even when people almost know what they have to do, it is often not easy for them to perform the task without confusion. Even when people know that they can watch a movie on the Internet, finding a movie and playing it is a difficult task for people without experiences. It would be nicer if they could watch a movie just by showing a fragment of the title or attributes of the movie.

We propose a simple, flexible and powerful translation framework ExpandHelp, with which users can generate a complete command string from fragments of users' vague intentions. We use a database consisting of pairs of the description of the task that users want to perform (e.g. 房間暗一点) and the actual command string required by the system (e.g. \$ `iot clight 0`). Descriptions can be written in any natural language using regular expressions and the database are shared on the Wiki pages and can be edited by any user who wants to improve the database.

Contributions of this paper are as follows:

1. Introduction of “translation paradigm” for generating a correct command to an artifact from the user's vague intention.
2. Flexible dynamic help extraction algorithm “generate and filter” using regular expressions.
3. Help data sharing infrastructure for the above strategy.

EXAMPLE: GITHELP

We implemented the GitHelp system with which a user can translate his vague intention in various languages into a complete git command using the ExpandHelp framework.

We assume that the user uses bash for his software development activities. When a user wants to compare the latest README.md file with the same file from 2 versions before, he might try to use git with an option “HEAD^^” for specifying the old version.

However, nothing happens here because HEAD^^ specifies a file included in the older commit, and the command in figure

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

UIST'18, October 14–17, 2018, Berlin, Germany

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 123-4567-24-567/08/06...\$15.00

DOI: http://dx.doi.org/10.475/123_4

```
$ git diff HEAD^^ README.md
$
```

Figure 1. Invoking `git diff`.

1 only compares the README.md with the file included in an old commit, where README.md might not have changed since then.

If GitHelp is installed, the user can type a special shortcut key to invoke GitHelp after typing “git RE 2” to see how he can perform the task.

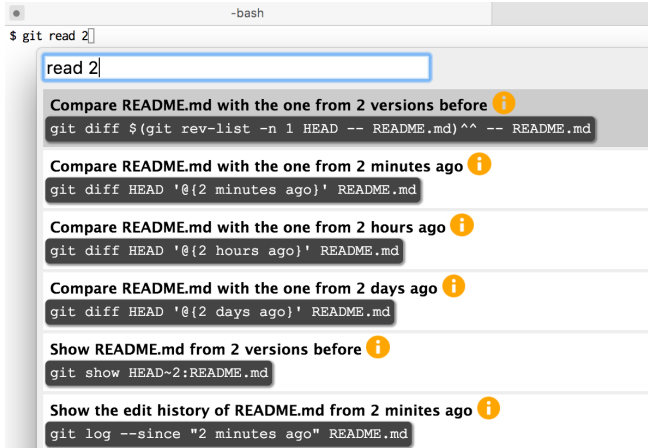



Figure 2. Invoking GitHelp after typing `git read 2`.

Here, various candidate entries with descriptions and command strings are listed, just like text input support systems for non-English languages. One of the entries says “Compare README.md with the one from 2 versions before”, and if the user thinks that’s what he wants to do, he can select the entry by typing arrow keys and type the enter key. Then the user’s input string is replaced by the right command string that satisfies the user’s need.

A description like “2 days ago” is also shown in the candidate list of Figure 2. This means that the user can use the same string “git RE 2” for getting the information of README.md file of the day before yesterday. The generated argument “@{2 days ago}”, may not be familiar to most git users, but users can perform this task only by giving “RE” and “2” without knowing the correct syntax of this option.

When the user types a special shortcut key after typing “git sh reed 3”, entries related to README.md file are listed because “reed” is close to “README.md”.

A Japanese git user can do the same thing just like English-speaking users (Figure 4). He can use the term “表”, etc. to find how he can perform the job.

If the user wants to know more about the command line, he can click the  button and see the Wiki page shown in Figure 5. The user can edit the page if he wants to add information or finds an error. In this page, description part is preceded by a “\$” symbol and the command part is preceded by “%”.

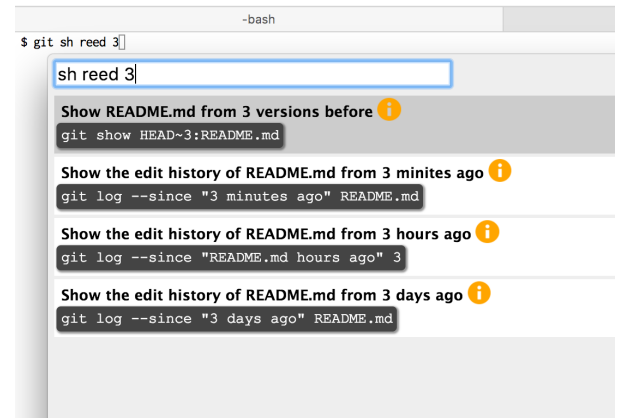


Figure 3. Invoking GitHelp after typing `git sh reed 3`.



Figure 4. Invoking GitHelp in Japanese environment.

IMPLEMENTATION

ExpandHelp

A flexible help data retrieval mechanism “ExpandHelp” is used in GitHelp. Help data used in ExpandHelp are provided as a set of data entries consisting of a regular expression that represents description of the help entry and a corresponding command pattern string. A description string is used to produce natural language texts that are easily understandable by users, while command patterns represents the generated command string corresponding to the description. For example, a help entry for showing the difference between current file and older file can be defined like this:

description:

Compare ({files}) with the one from (1|2|3|4) versions before

command:

```
git diff $(git rev-list -n #{2} HEAD -- #{1})^
-- #{1}
```



Figure 5. A Scrappox page for `git diff`.

Special symbols like `{files}` in the description part is expanded to the list of filenames like `README.md|index.html`. The first regular expression, the description part, is for generating texts like “Compare README.md with the one from 1 versions before”, “Compare index.html with the one from 2 versions before”, etc., and the second part, command part, is used to generate a command to execute a git command. In the first example, “README.md” and “1” match the first and second parentheses of the regular expression, and these values are assigned to `$1` and `$2` for the command part, generating a command “`$ git diff $(git rev-list -n {2} HEAD -- {1})^ -- {1}`”.

The description part can be written in any natural language. For example,

(`{files}`) を (1|2|3|4) 個前のものと比較する

can be used for Japanese users. In this case, “README.md を 1 個前のものと比較する”, “README.md を 2 個前のものと比較する”, etc. are used for the matching.

Generating help menu entries

Finding appropriate entries from the (possibly huge) help document space is performed in two steps. First, a state transition diagram is generated from the regular expressions used in the descriptions. Second, all the description strings are generated by expanding the regular expression into a tree, and filter the generated strings by the pattern specified by the user. Efficient pattern matching is performed every time a new description text is generated, and only the best-matched descriptions are shown to the user as menu entries. We call this the “generate-and-filter” technique, and we will describe the details later.

Phase 1: Generating a state transition diagram from a regular expression

Regular expressions are widely used for finding patterns in text strings in modern programming languages and in the Unix environment. In ExpandHelp, we use regular expressions for generating various patterns of descriptions from a short specification. For example, we use a short regular expression “(remove|delete|erase) (data|file)” for representing expressions like “remove data”, “erase file”, etc.

Converting a regular expression to a state transition machine is a straightforward task. When we have a regular expression “Compare (README.md|index.html|package.json) with the one (1|2|...|10) versions before”, we can convert it into a state transition diagram shown in Figure 6.

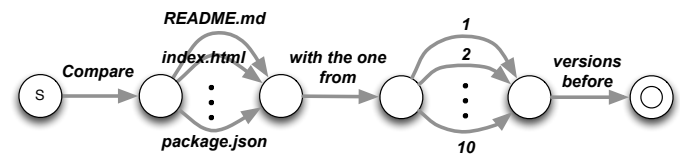


Figure 6. State transition diagram for a `git` command description.

By traversing the nodes and links in this state machine, we can generate the following description texts.

```
Compare README.md with the one 1 versions before
Compare index.html with the one 1 versions before
...
Compare package.json with the one 10 versions before
```

Phase 2: Generating texts from a state transition diagram

Using the state transition diagram shown in Figure 6, ExpandHelp generates all the strings represented in the regular expressions, and filter them by the pattern provided by the user.

We can generate texts from a state transition diagram by traversing nodes one by one. Starting at the start node (S) in Figure 6, we can visit other nodes via edges and generate a tree of generated texts shown in Figure 7. After visiting the initial node (S), the system generates a string “Compare” and proceeds to the next node. In the second generation, the system can add “README.md”, “index.html”, and “package.json” to “Compare”, generating “Compare README.md”, “Compare index.html”, etc. The system can repeat finding edges from previously visited nodes, and eventually generates all the strings described above. Of course, it is impossible to generate all the strings from a regular expression like “(0|1)+” that represents infinite length of strings consisting of 0s and 1s, so the generation should be terminated at a certain generation.

Generate-and-Filter

Generating all the texts in this way before finding matched strings is inefficient, because the amount of generated texts can easily become huge. Instead, the system performs the pattern matching as soon as a text is generated, for saving time and memory.

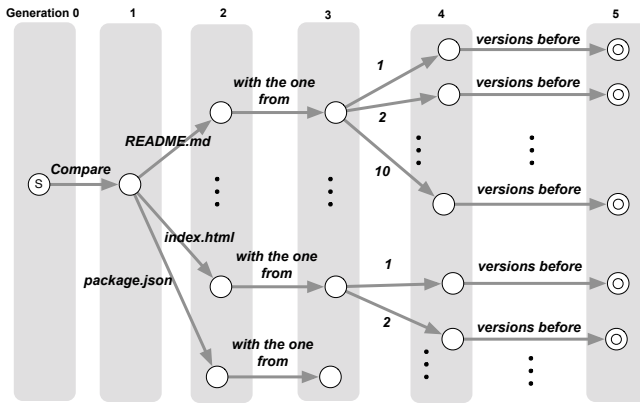


Figure 7. Generating a text tree from a state transition machine.

“Generate-and-test”¹ is a simple and effective strategy used for solving puzzles. For example, the “8-Queen” problem can be solved simply by generating all the possible queen layouts and checking if two or more queens are laid out on the same row, column, or diagonal line. Although this strategy is simple, the cost of generating all the possible solutions is prohibitive, since the solution space of the 8-Queen puzzle is $8^8 = 16,777,216$, which is not a small amount even for today’s computers.

To solve this problem, controlling the generation part from the testing part is effective. In the 8-Queen problem, whenever the testing part finds two queens in the same row or column, it can tell the generation part to give up current layout in the early stage and proceed to the next layout.

Similarly in our case, it is not efficient to perform the matching operation after generating all the texts from the regular expression, and it is better to calculate the matching at the time of generating each text. We call it the “generate-and-filter” strategy, and implemented the algorithm in Expand-Help. Unlike simple puzzle problems where conditions are strict and finding one solution is enough, our goal is to find help description strings that fits the query pattern while allowing errors. For this goal, the system should perform approximate pattern matching without sacrificing processing speed. We could implement flexible generate-and-filter by using a simple and efficient approximate pattern matching algorithm based on the “shifter algorithm”.

Approximate pattern matching by shifter algorithm

The “shifter algorithm”[12] is a simple and efficient text search algorithm that has interesting features not found in more common pattern matching algorithms like KMP[8], Boyer-Moor[1], etc.

When a user wants to find a word “README” in a text, he can use a patter matching state machine like below, where the gray circle denotes that the state is active.

Initially, only the leftmost node is active, but when the state machine receives “R”, both the first and the second node be-

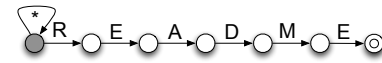


Figure 8. A state machine for “README”.

come active, and the activation state will change to the following pattern.

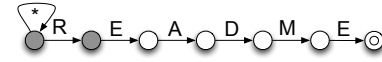


Figure 9. After receiving “R”.

When “README” is given to the state machine, the rightmost node becomes active.

Although this state machine is a nondeterministic finite state automata (NFA), only 7 bits are required to represent the active/inactive states of the nodes, meaning that the whole states can be represented by a single integer value. Also, the state transition can be calculated by a simple combination of logic and bit shift operations.

Approximate pattern matching using shifter algorithm

The state machine shown in Figure 8 can accept only one pattern (“README”), but it can be easily expanded to detect texts that contain a word similar to the pattern (e.g. “REEDEE”).

Adding three more rows of states to Figure 8, we can perform more flexible pattern matcher which can accept strings with 0 to 3 matching errors.

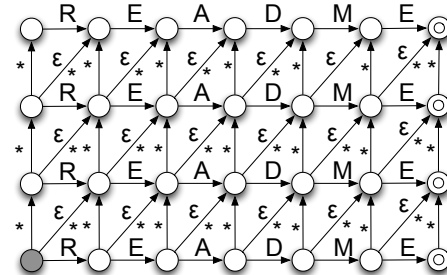


Figure 10. A pattern matcher accepting 0-3 errors.

Figure 10 shows a state machine that can detect a string that matches “README” with 0 to 3 mismatches. Each additional row of nodes represents a matcher with one error, two errors, and three errors, respectively. Vertical and diagonal transition edges are added to allow transitions based on spelling errors.

Initially, only the bottom-left node is active. When a character other than “R” is detected, the transition labeled “*” (wildcard) is activated, and connected nodes become active. At the same time, links labeled as “ε” is also activated without any input character. With these additional links, this expanded machine can detect a text which matches “README” with 0 to 3 errors.

The transition of active nodes while reading “REEDEE” is shown in Figure 11. After reading “REEDEE”, only the upper-right node becomes active, denoting that “REEDEE” matches “README” with 2 matching errors.

¹http://en.wikipedia.org/wiki/Generate_and_test

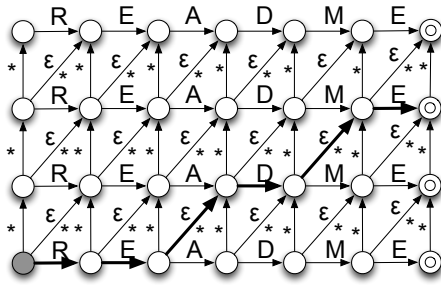


Figure 11. Accepting “REEDEE” using a matcher for “README”.

The state transition shown in Figure 11 looks complicated, but the matching state can be represented by only four integer variables, and the matching algorithm can be implemented efficiently.

Using the matcher for generate-and-filter

We can use this approximate pattern matcher while generating help texts by traversing the state transition machine. Every time a new node is generated by traversing an edge, matching state is calculated and stored in the generated node. The status pattern is calculated from the status pattern stored in the preceding node and the string associated with the edge.

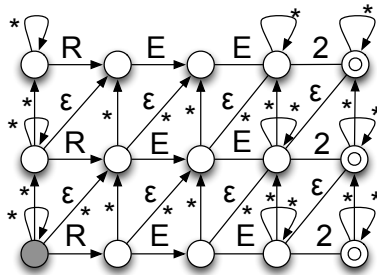


Figure 12. A pattern matcher for “REE 2”.

Figure 12 shows the matcher for “REE 2”², and Figure 13 shows how the matching status is calculated while the description strings are generated from the state transition machine shown in Figure 6. The matching status at each node is calculated from the matching status stored in the parent node and the string associated with the edge. For example, when the system generates the string “Compare README.md with the one from 2 versions before”, the matching status is calculated from the status data at the previous node at “Compare README.md with the one from 2”, and the system can tell that it matches “REE 2” with one matching error, as soon as the string is generated.

The system is keeping the list of generated strings which match the pattern with zero to three errors and it displays the list with minimal number of errors when generating menu entries. Since the pattern matching operation is performed at the time of text generation, the whole generate-and-filter calculation is quick.

²A space (“ ”) in the input is treated as a wildcard character.

Sharing the database on the Web

The data used in GitHelp are represented as “Scrapbox” documents shown in Figure 5. Scrapbox is a general-purpose real-time Wiki system for data sharing. Scrapbox users can edit the contents of Wiki pages directly on Web browsers just like using text editors.

This page shows how to use “git diff”, and the page contains command specifications in ExpandHelp format in addition to the documentation.

Usage examples are often shown on documents and manuals like Unix man pages, but they cannot be used as the database of an online help system. On the other hand, a Scrapbox document contains both the manual document and the help database in the same place, just like writing a source code and the document in the same place under the “Literate Programming”³ style.

Also, since the database is on the Wiki page, any member of the Wiki can correct the document or add additional GitHelp specifications. When a user could not get a good support from GitHelp, he can add ExpandHelp specifications to the Scrapbox document himself so that he and other people can get better support from GitHelp next time.

RELATED WORK

Intelligent help systems

There have long been many researches on intelligent help systems[3], but few number of projects seem to be going on these days, maybe because now we can use the Internet for getting intelligent help from real people, either by searching existing Web pages or by asking in user forums like StackOverflow⁴. Information on the net is rich these days, and we can easily find somebody who can give good answers to our questions. This is a good news, but asking many trivial questions on the net is not considered to be a good manner, and using a powerful and flexible local help system like ExpandHelp is preferable most of the time.

Input Methods

Text input systems for non-English languages are widely used in the world, and they are called as “Input Methods” (IMs). Many research and development on IMs have been going on for years, and people are using them every day for entering texts in their mother language. Figure 14 shows an example of a Japanese IM.

³https://en.wikipedia.org/wiki/Literate_programming

⁴<https://stackoverflow.com/>

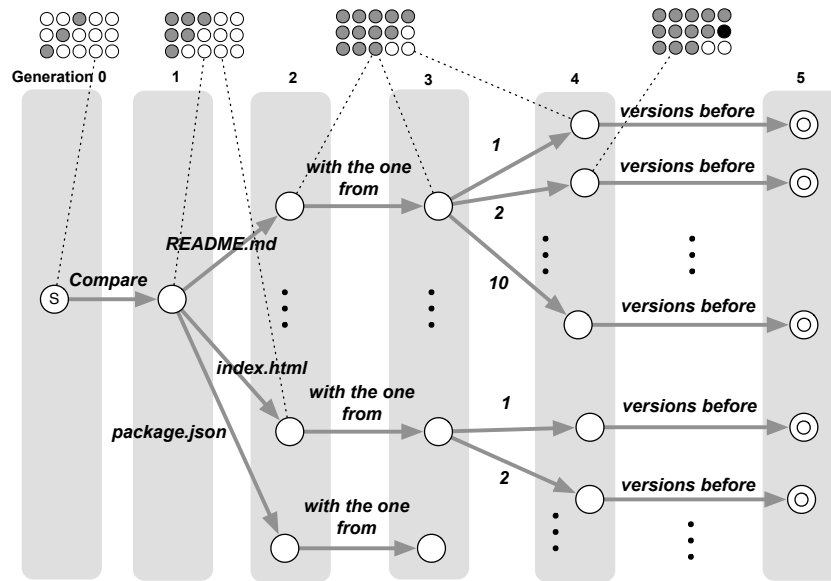


Figure 13. Matching statuses for “REE 2” associated with text generation nodes.



Figure 14. An IM for Japanese text input.

IMs are one form of translation systems that convert one expression (pronunciation, character shape, etc.) into a textual form of a language. In the above example, the pronunciation “kanji” is used for getting “漢字”.

IMs are very popular in the modern computing environment in the world and it is very common for people to use translation systems for using computers. The behavior of GitHelp is similar to conventional IMs, and integration of multiple IM-like systems should be challenged in the future.

Text and code completion

On text editors and Unix shells, “text completion” feature has been available for a long time. When a user types the TAB key after typing “1a” on the Unix shell, “1atex”, “1ast”, and other Unix commands are shown as candidates. When a user types “1s RE” and types the TAB key, the shell checks the directory, find the README.md file, and replace the user’s input with “1s README.md”. This behavior is called “text comple-

tion”, and many variations of this idea have been proposed and adopted in command shells and text editors.

Gitsome⁵ helps git users by dynamically showing possible arguments and related manuals on the Unix shell, just like GitHelp does. The goal of Gitsome is almost the same as GitHelp and we agree with its concept, but Gitsome does not support flexible approximate matching and help data sharing.

Smart IDEs

Integrated Development Environments (IDEs) are widely used for software development, and many smart IDEs for suggesting codes have been proposed recently.

Little’s system[9] generates a complete JavaScript code snippet from keyword fragments give by users. For example, a code snippet “ActiveDocument.PageSetup.LeftMargin = InchesToPoints(2)” can be generated from keywords like “left”, “margin”, “inches”, and “2”. The system generates the code using a template database and heuristics designed for the target system. It is useful for a specific environment, but users of the system cannot modify the algorithm or the database even when no good suggestion was found.

AnyCode[5] is another system that generates a complete code snippet from the user’s input in natural language. AnyCode can handle synonyms database like “make” == “create”, but approximate matching is not supported and users cannot modify the help database.

Active Code Completion[11] is another code completion system for the Eclipse IDE. A database called *palette* are used for code completion and coding support for Java classes. When a user wants to write a code for drawing a rectangle in purple, the user should just tell “purple” to the system, and the system shows the user a color selection window as well as the Java code for drawing a purple rectangle. This is a useful

⁵<https://github.com/donnemartin/gitsome>

feature for the user to write a code including selecting a color, but palettes are difficult to create, and should be provided as IDE plugins.

Han's Abbreviation Completion system[6] can generate a code snippet from abbreviation strings give by the user. For example, a code snippet like “chooser.showOpenDialog()” can be generated from strings like “ch” and “opn”, using the code database and HMM. It can be very useful as a shorthand for handling long names, but users should have a vague knowledge about the correct names (e.g. “chooser”, “dialog”), and large code example corpus should exist before being able to create the database. ExpandHelp database can be created by hand at the time of the system development and no corpus is required.

Using cloud data for development

Many systems support sharing users' development experiences on the web so that they can be used on the developers' IDE.

Using Blueprint[2] as a plugin for Adobe Flex Builder, users can search sample codes from the Web and use it immediately. HelpMeOut[7] tells users how to handle compile errors, based on the people's experiences of handling compile errors. Resources on the net are precious, and such data can be easily incorporated to ExpandHelp databases.

DISCUSSIONS

Applications of ExpandHelp

Since the translation mechanism of ExpandHelp is simple and flexible, it can be used for a wide range of application domains where intelligent help is required. ExpandHelp is actually used as the help page for one of our Web services, hoping that novice users can easily find the feature they need. When a user submits a query that includes vocabularies not in the ExpandHelp document, we are able to add them to the Scrapbox database easily.

Figure 15 and Figure 16 shows how ExpandHelp can be used as a launcher for MacOS. Like GitHelp, users can execute concrete commands only by showing small fragments of their intentions to the system.

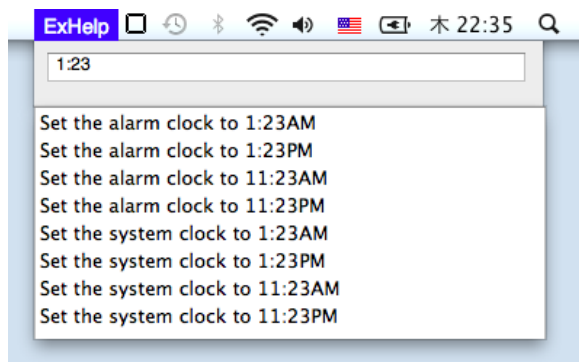


Figure 15. Using ExpandHelp in the menu bar of MacOS.

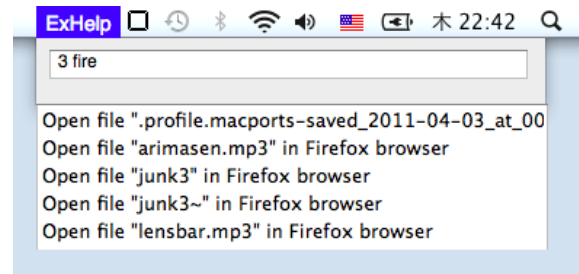


Figure 16. Using ExpandHelp in the menu bar of MacOS.

Order of command and arguments

Using a command shell, users have to enter a command first and enter options and arguments after that. Using a GUI desktop, users can right-click a data icon to show a menu, and select what he wants to do from the menu. Switching between command-oriented usage pattern and data-oriented usage pattern is sometimes confusing. Using ExpandHelp, users can find what they can do either by specifying commands or by specifying data, without worrying about the order.

Usage of regular expressions

An advantage of using a regular expression is that it is fairly easy for users to write description texts. People can easily write a help description like “(Start|Run) Firefox browser” without having a knowledge of automata or formal grammar. We can use arbitrary texts for the database of ExpandHelp. For example, we can collect hundreds of books titled “Windows tips” from the market and convert all the title of the tips into help data of ExpandHelp.

CONCLUSION

We proposed the “translation paradigm” for bridging the gap between users' intention and actual operations. We have introduced a general-purpose flexible framework ExpandHelp that can translate fragments of user's vague intentions into concrete command strings required to control computers and other complex artifacts. We showed the principles and applications of ExpandHelp by using it for GitHelp, a help system for supporting git users based on the shared database that any user can edit to share translation knowledge between users. ExpandHelp framework is useful for any kind of systems that need some kind of user support, and we are planning to use it for various applications and Web services in the future.

REFERENCES

1. Robert S. Boyer and J. Strother Moore. 1977. A fast string searching algorithm. *Commun. ACM* 20 (October 1977), 762–772. Issue 10.
2. Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. 513–522.
3. Sylvain Delisle and Bernard Moulin. 2002. User interfaces and help systems: from helplessness to

intelligent assistance. *Artif. Intell. Rev.* 18 (October 2002), 117–157. Issue 2.

4. G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. 1987. The Vocabulary Problem in Human-system Communication. *Commun. ACM* 30, 11 (Nov. 1987), 964–971.
5. Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java Expressions from Free-form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. 416–432.
6. Sangmok Han, David R. Wallace, and Robert C. Miller. 2009. Code Completion from Abbreviated Input. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. 332–343.
7. Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. 1019–1028.
8. D. E. Knuth, J. H. Morris, and V. R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM Journal of Computing* 6, 2 (June 1977), 323–350.
9. Greg Little and Robert C. Miller. 2006. Translating Keyword Commands into Executable Code. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. 135–144.
10. David G. Novick and Karen Ward. 2006. Why Don'T People Read the Manual?. In *Proceedings of the 24th Annual ACM International Conference on Design of Communication (SIGDOC '06)*. 11–18.
11. Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active Code Completion. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. 859–869.
12. Sun Wu and Udi Manber. 1992. Fast text searching: allowing errors. *Commun. ACM* 35 (October 1992), 83–91. Issue 10.