TranStrL: An Automatic Need-to-Translate String Locator for Software Internationalization

Xiaoyin Wang[1,2], Lu Zhang[1,2]*, Tao Xie[3]*, Hong Mei[1,2], Jiasu Sun[1,2]
[1]Institute of Software, School of Electronics Engineering and Computer Science
[2]Key laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
Peking University, Beijing, 100871, China
{wangxy06, zhanglu, meih, sjs}@sei.pku.edu.cn
[3]Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
xie@csc.ncsu.edu

Abstract

Software internationalization is often necessary when distributing software applications to different regions around the world. In many cases, developers often do not internationalize a software application at the beginning of the development stage. To internationalize such an existing application, developers need to externalize some hard-coded constant strings to resource files, so that translators can easily translate the application to be in a local language without modifying its source code. Since not all the constant strings require externalization, locating those need-to-translate constant strings is a basic task that the developers must conduct. In this paper, we present TranStrL, an Eclipse plug-in tool that automatically locates need-to-translate constant strings in Java code. Our tool maintains a pre-collected list of API methods related to the Graphical User Interface (GUI), and then searches for need-to-translate strings in the source code starting from the invocations of these API methods using string-taint analysis.

1 Motivation

Modern software applications often target at users from different regions around the world. To better serve users in a certain region, a software application must be adapted to its local version to meet the users' requirement. This adaptation often includes translation of the user-visible text, conversion of number and date formats, etc. In general, techniques for obtaining and managing these local versions are usually referred to as software internationalization.

Some software applications adopt the technique of internationalization at the beginning of the development stage. Developers of these applications try to avoid hard-coding elements that need to be changed from one local version to another. However, in many cases, developers adopt software internationalization only during or after the development of the first non-international version with two main reasons. First, many popular software applications originate from open source prototypes or research prototypes, whose developers do not expect their users to have requirements specific to particular regions in the beginning of the tool development. Second, developers of an international software application may reuse some non-international software components. Thus, the developers may have to internationalize these reused components. In all these cases, developers need to internationalize existing code, which typically contains many hard-coded elements specific to one local version.

When internationalizing existing code, developers usually need to locate those hard-coded elements that need translation [2, 4]. The need-to-translate elements include constant strings, time/date objects, number-format objects, etc. In particular, locating need-to-translate constant strings is often the most tedious task. The reason is that, unlike other need-to-translate elements, a software application typically contains a large number of constant strings, many but not all of which need translation. Some need-to-translate constant strings are easy to locate, such as the string `"Name"` in `new Button("Name")`, but many others may be very difficult to locate, such as those strings that are output to the Graphics User Interface (GUI) through several assignments, parameter transmissions and string operations [5]. The existence of a large number of need-to-translate strings and the difficulty of locating some of them make the task of locating need-to-translate strings tedious and error-prone. Therefore, an automatic need-to-translate string locator would provide great help to the developers.

*Corresponding author

In this paper, we present TranStrL[1], an Eclipse plug-in tool that locates need-to-translate constant strings in Java code. The basic idea behind our tool is to locate invocations of API methods that output strings to the GUI, and trace from the output strings to the constant strings that need translation [5].

2 Related Work

To the best of our knowledge, our tool is the first one focusing on automatically locating need-to-translate constant strings in source code. There are some published books [2, 4] on how to manually internationalize a software application. However, none of them provides any automatic approach or tool to locating need-to-translate strings.

There exist tools (e.g., GNU gettext[2] and Java internationalization API[3]) to help developers externalize need-to-translate constant strings after the developers locate them. Other tools such as KBabel[4] help developers edit and manage resource files (called PO files in KBabel) containing externalized constant strings. Other commercial tools such as Redpin[5] and Passolo[6] provide an integrated localization environment that includes file visualizers, translation editors, term translators, and translation consistency checkers.

Some development environments (e.g., Eclipse String Externalizing and Susilizer[7]) provide features to locate and externalize all constant strings in the code of an application. However, not all of the constant strings need translation. Our empirical results [5] show that in many real-world software applications, less than half of the constant strings need translation. Thus, it may be a waste of time for translators to translate all the constant strings. To be even worse, some constant strings should not be translated; otherwise, bugs could be introduced to the application. For example, if the name of a field from a database table is translated to be in another language, the application may suffer from runtime failures when retrieving data from the database.

3 TranStrL Architecture

As shown in Figure 1, TranStrL consists of six components: an API invocation search engine, an adapted string-taint analyzer, a string-comparison analyzer, a string-transmission analyzer, a filter, and the tool GUI. TranStrL takes the source code of a Java application as input and produces a list of need-to-translate strings. Additionally, TranStrL requires a pre-collected Output API Method list as input.

The main process of TranStrL includes five steps: (1) TranStrL uses the API-invocation search engine to search for invocations of the methods in the Output API Method List. (2) TranStrL takes the actual arguments (in these invocations) that are output to the application GUI as the initial Output Strings and passes them to the string-taint analyzer. (3) The string-taint analyzer traces from the initial Output Strings to their data origins and obtains a list of basic need-to-translate strings. (4) TranStrL passes the obtained list to the string-transmission analyzer to further trace to the strings that are passed to the application GUI through network communications, and to the string-comparison analyzer to further trace to the strings that are compared with the already-known need-to-translate strings. The fourth step is performed iteratively until no more need-to-translate strings are added to the string list. For optimization, only the need-to-translate strings newly located by the current iteration are used as the input to the next iteration. (5) TranStrL filters the constant strings in the string variable list acquired in Step 4 according to some heuristics and passes the filtered list to the GUI of TranStrL.

The Output API Method List is a list of method signatures, in which each method can output at least one of its parameters to the application GUI. We denote the parameters that can be output to the application GUI as Output Parameters so that we can trace from them in the tool's process. We manually collected the Output API Method List from packages `"java.awt.*"` and `"javax.swing.*"`. Thus, TranStrL currently supports only applications with their GUI written using these libraries. We next present the six components of the tool in detail.

3.1 API-Invocation Search Engine

TranStrL searches for the invocations of the Output API Methods with Eclipse's Java Search Engine, which is a powerful tool to search for the declarations and references to different syntactic elements in Java. The engine can precisely locate possible invocations of a given Java method in the presence of polymorphism. Then TranStrL uses the actual arguments (in the invocations) that correspond to Output Parameters as the initial output strings that are passed to the String-Taint Analyzer.

3.2 String-Taint Analyzer

From each initial output string, TranStrL performs an adapted string-taint analysis to locate the possible sources of the initial output string in the code.

String analysis and string-taint analysis are recent advances in static data-flow analysis [3]. Christensen et al. [1] first suggested string analysis, which is an approach

[1] pronounced as [ˈtrenstrəl]

[2] `http://www.gnu.org/software/gettext/manual/gettext.html`

[3] `http://java.sun.com/docs/books/tutorial/i18n/index.html`

[4] `http://kbabel.kde.org/`

[5] `http://www.redpin.eu/`

[6] `http://www.passolo.com/`

[7] `http://www.susilizer.com`

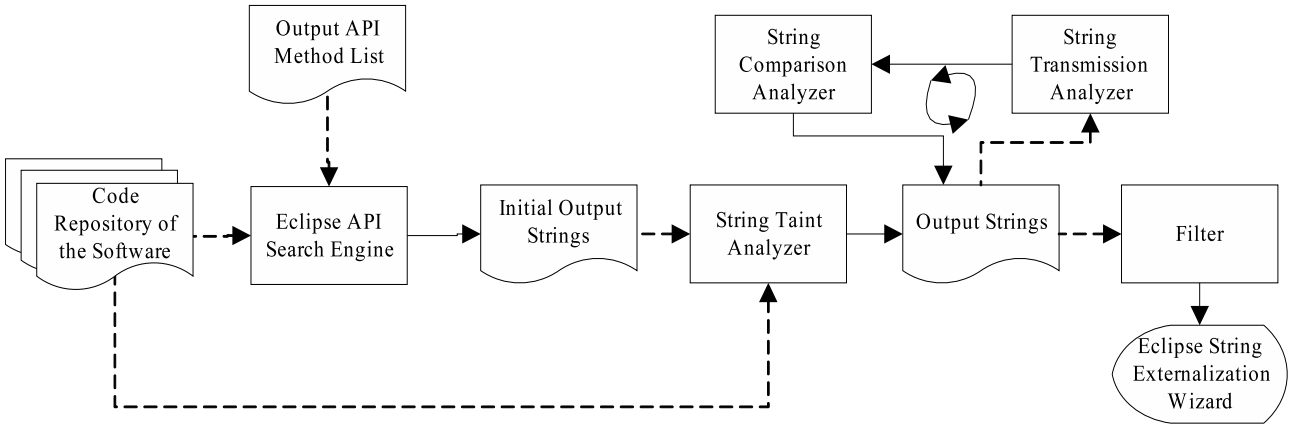

Figure 1: Overview of TranStrL

for obtaining possible values of a string variable. Recently, Wassermann and Su [6] developed string-taint analysis based on string analysis. String-taint analysis further analyzes whether some substrings in the string variable might come from insecure sources. The input of string-taint analysis is the source code and a string variable in the code. String-taint analysis predicts the possible values of the given string variable and determines whether the possible values might contain insecure substrings.

To apply string-taint analysis for our problem, we need to do some adaptation. As we are interested in hard-coded constant strings, we use the locations of these strings as their annotations. For strings from other sources such as files and network, we further annotate them as "&FileInput" and "transmitted", etc. TranStrL adopts the adapted string-taint analysis, with which TranStrL can locate a list of basic need-to-translate strings.

3.3 String-Transmission Analyzer

Using string-taint analysis, TranStrL can trace to string variables whose values are transmitted from the network. For these variables, we further apply a string-transmission analyzer in TranStrL. By matching the socket number and the flag variables transmitted along with the data in a network packet, the string-transmission analyzer locates the strings whose values are passed to the strings in the previously obtained need-to-translate string list. Currently, TranStrL can cope with only network transmissions by passing objects through sockets.

3.4 String-Comparison Analyzer

After string-taint analysis and string-transmission analysis, TranStrL can locate the constant strings viewable on the application GUI. However, we need to further locate the strings that are compared with these viewable strings because otherwise the logic of the program may go wrong. Thus, we apply our string-comparison analyzer for TranStrL to address this issue. The analyzer first locates all the comparisons between strings in the source code. Then, for each side of a comparison, the analyzer performs string-taint analysis to locate all the constant strings that are the sources of the side. After this step, the analyzer checks if any located constant string as a source for one side is in the list of previously located viewable strings. If so, the analyzer adds all the strings located as sources for the other side as need-to-translate strings. TranStrL uses the string-comparison analyzer iteratively until TranStrL cannot locate any more need-to-translate strings.

3.5 String-Candidate Filter

As a practical matter, not all the located strings require translation. Some strings should be the same in all local languages (e.g., strings composed of arabic numerals), while some other strings may be intentionally untranslated (e.g., trademarks). Therefore, we further apply a string-candidate filter to filter out some located constant strings that may not need translation. By default, TranStrL filters out two kinds of strings: a constant string includes no letter and a constant string that is equal (ignoring the case) to the name of the project. TranStrL also uses a property file, in which the tool users can provide the types of strings that the users want to filter out in the form of regular expressions.

3.6 Tool GUI

We reuse the Eclipse String Externalize Wizard as the GUI of TranStrL. The reason is that the String Externalize Wizard provides a context view for the developers to check the context of a certain constant string and edit it. Furthermore, it provides a default strategy to externalize a list of strings automatically.

The Eclipse String Externalize Wizard gets as its input a list of all the constant strings (each string is identified by its value and its offset in the file enclosing it) in the code. In TranStrL, we reuse the wizard by popping up an identical wizard and setting its input as the need-to-translate string list produced by TranStrL.

4 Tool Usage

4.1 Example Usage Scenarios

TranStrL can be especially useful to developers in two main scenarios:

- When a software application that has not been internationalized is required to be translated to other languages, the developers can run the tool to get a list of need-to-translate constant strings. The developers may choose to inspect the list of strings manually one by one. In this case, TranStrL can reduce by about 70% of the strings that the developers need to inspect, according to our recent empirical results [5]. That is to say, TranStrL usually retrieves only 30% of all the constant strings in the code as need-to-translate. The developers can also choose to externalize them all using the Eclipse string externalize strategy. This step may bring in a small number of incorrectly-located and/or unlocated need-to-translate strings, and further testing and debugging may be required, but the developers can effectively reduce the effort on manually checking each located string.
- When the software application has been internationalized but the developers are not certain about the quality of the internationalization, they can run TranStrL to find out a list of suspicious missed need-to-translate strings, and then check the list to confirm whether each string needs translation. In fact, based on our recent research [5], we reported 17 such missed strings in the latest version of Megamek[8] (the most downloaded real-time strategy game in Sourceforge) to the developers, and the developers confirmed and translated all of the 17 strings.

4.2 Starting TranStrL

TranStrL is easy to start. A user just needs to open the project in Eclipse, right click on the project icon to get a context menu, and press the button of "internationalize..." in the menu. Then, a list of need-to-translate strings of the project is output to a result file.

TranStrL can be started without any configurations. However, if a user already knows that some of the methods in the client code can output some of their parameters to the application GUI before starting TranStrL, the user may add these methods to the Output API Method List. Adding these methods to the list may reduce some inaccuracy. Furthermore, as mentioned previously, a user can also customize the filtering strategies in TranStrL to improve the effectiveness of the filter.

4.3 Using the Results

After locating need-to-translate strings with TranStrL, the user can press the button "TranStrL Externalize..." to open the tool GUI to externalize these strings. Since TranStrL reuses the tool GUI and the string externalizing strategies of "Externalize Strings" in Eclipse, the user can easily externalize and edit the located strings either in the way that the developer wants or using a default externalizing strategy provided by Eclipse. When used on internationalized applications, TranStrL automatically filters out the strings that have already been externalized, so that the developers can focus on only the potentially missed need-to-translate strings.

[8] `http://sourceforge.net/projects/megamek/`

5 Conclusion

In this paper, we present TranStrL, an Eclipse plug-in to automatically locate need-to-translate constant strings. TranStrL is mainly based on the Eclipse Java Search engine and an adapted string-taint analyzer, and supplemented with three other components to further enhance the accuracy. TranStrL reuses the GUI of Eclipse String Externalization as the user interface.

In future work, we plan to extend the Output API Method List to other popular GUI libraries to support more software applications. Furthermore, we plan to extend the String Transmission Analyzer to cope with other network transmission strategies such as SOAP or event Bus.

Acknowledgment

The authors from Peking University are sponsored by the National Basic Research Program of China (973) No. 2009CB320703, the High-Tech Research and Development Program of China (863) No. 2007AA010301 and No. 2006AA01Z156, the Science Fund for Creative Research Groups of China No. 60821003, and the National Science Foundation of China No. 90718016. Tao Xie's work is supported in part by NSF grants CNS-0720641, CCF-0725190, and Army Research Office grant W911NF-08-1-0443.

References

[1] A. Christensen, A. Mller, and M. Schwartzbach. Precise analysis of string expressions. In *Proc. SAS*, pages 1–18, 2003.

[2] B. Esselink. *A Practical Guide to Software Localization: For Translators, Engineers and Project Managers*. John Benjamins Publishing Co, 2000.

[3] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, January 1976.

[4] E. Uren, R. Howard, and T. Perinotti. *Software Internationalization and Localization: An Introduction*. 1993.

[5] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-translate constant strings for software internationalization. In *Proc. ICSE*, 2009.

[6] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. PLDI*, pages 32–41, 2007.