Locating Need-to-Translate Constant Strings 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

Modern software applications require internationalization to be distributed to different regions of the world. In various situations, many software applications are not internationalized at early stages of development. 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 into 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 necessary task that developers must complete for internationalization. In this paper, we present an approach to automatically locating need-to-translate constant strings. Our approach first collects a list of API methods related to the Graphical User Interface (GUI), and then searches for need-to-translate strings from the invocations of these API methods based on string-taint analysis. We evaluated our approach on four real-world open source applications: RText, Risk, ArtOfIllusion, and Megamek. The results show that our approach effectively locates most of the need-to-translate constant strings in all the four applications.

1 Introduction

1.1 The Problem

Modern software applications often need to be distributed to different regions of the world. To be better used by users in a certain region, a software application should have a local version for local users. Typically, a local version's user-visible texts should be in the local language, and its numbers, times and dates should also be in the local formats. In general, techniques for obtaining and managing these local versions are usually referred to as software internationalization.

Developers of some software applications consider internationalization in the beginning of development. That is to say, 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 need to apply software internationalization on existing code with the following reasons. First, many popular software applications are originated from open source prototypes or research prototypes, whose developers do not expect their users to have requirements specific to particular regions in the beginning. 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 [5, 13]. The need-to-translate elements include constant strings, time/date objects, number-format objects, culture-related objects, etc. In particular, locating need-to-translate constant strings is often the most tedious task. The reason is that, a software application typically contains a large number of constant strings, many but not all of which need translation.

1.2 Existing Support

There exist tools (e.g., GNU gettext[1], Java internationalization API[2]) to help developers externalize need-to-translate constant strings after the developers locate them. Other tools such as KBabel[3] help developers edit and manage resource files (called PO files in KBabel) containing

*Corresponding author

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

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

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

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

externalized constant strings. Some development environments (e.g., Eclipse) provide help 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 in Section 5 show that in 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, translation of some constant strings may introduce bugs. For example, if the name of a field in an SQL query for a database is translated into another language, the software application may suffer from runtime failures when retrieving data from the database.

1.3 Our Approach

In this paper, we present an automatic approach to locating need-to-translate constant strings in source code based on string-taint analysis. The basic idea of our approach is to locate invocations of API methods that may output strings to the application Graphical User Interface (GUI), and trace from the output strings to the constant strings that need translation. In particular, our tracing approach includes four techniques.

- The first technique adapts string-taint analysis [14] to trace from the output strings to their sources. Sources that are constant strings should be translated.
- The second technique deals with an application involving network communication where a source of an output string may be a string variable whose value is transmitted across the network. There we further analyze string transmission across the network to locate hard-coded constant strings in one side of the network but may appear on the GUI of the other side of the network.
- The third technique deals with complications where the translation of some to-be-translated constant strings also impacts strings that are compared with these to-be-translated constant strings. There we analyze string comparisons in the code to locate constant strings that string translation impacts.
- Since not all constant strings that are viewable on the GUI need translation, the last technique filters out strings that are not likely to be translated.

This paper makes the following main contributions:

- An approach to automatically locating need-to-translate constant strings in source code based on tracing from invocations of API methods that output strings to the GUI.
- Adaptation of string-taint analysis and three other practical techniques to further cope with issues of string transmission, string comparison, and filtering.
- An empirical study of applying our approach on four real-world open source applications to demonstrate the effectiveness of our approach. The empirical results show that our approach not only locates most of the strings that the developers externalized, but also finds some strings that the developers missed. We reported in a bug report 17 missed strings that are still missing in the latest version of the Megamek application[4]. All of the 17 strings were confirmed and later translated by Megamek developers.

We organize the rest of this paper as follows. Section 2 presents an example of locating need-to-translate constant strings. Section 3 presents our approach in detail. Section 4 presents the implementation of our approach. Section 5 reports an empirical study of our approach. Section 6 further discusses related issues. Section 7 discusses related work and Section 8 concludes with future work.

2 Example

We next present an example to illustrate a situation that a developer may face when manually locating need-to-translate constant strings in source code. The example comes from Risk[5] (Version 1.0.7.5), a real-world open source project used in our empirical study. Consider the following code portion in Risk:

```
1 public class Risk{
2   private RiskController gui;
3   private String message;
4   private RiskGame game;
5   public void GameParser(String mem){
6      message=mem;
7      StringTokenizer StringT = new StringTokenizer(message," ");
8      String addr = StringT.GetNext();
       ...
9      if(addr.equals("CARD")){
10        if(StringT.hasMoreTokens()){
11           String name = StringT.GetNext();
12           String cardName;
             ...
13           if(name.equals("wildcard"))
14              cardName = name;
15           else cardName = card.getName() + " " + name;
16           gui.sendMessage("You got a new card:\""
17               + cardName + "\"", false , false);
          }
       ...} }
18    public void DoEndGo(String mem){
     ...
19    GameParser("CARD "+game.getDeservedCard());
     ...}
  }
20 public class RiskGame{
21   public String getDesrvedCard(){
22      Card c = cards.elementAt(r.nextInt(cards.size()));
23      if(c.getCountry() == null)
24           return "wildcard";
25      else
26           return c.getCountry.getName();
     ...}}
```

In the preceding code portion, Lines 16-17 include an invocation of `RiskController.sendMessage(...)`, and the expression `"You got a new card:\"" + cardName +"\""` corresponds to parameter `output` in `RiskController.sendMessage(...)`, which sends the value of `output` to the GUI. Now the developer knows that

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

[5] On Sept. 2, 2008, we found that the name of Risk was changed to Sametime Risk recently.

"You got a new card:" needs translation. Furthermore, the value of variable cardName also appears on the GUI. So the developer needs to further trace to the sources of cardName. Line 14 indicates that name is a source of the value of cardName. Furthermore, the value of name comes from a token of StringTokenizer StringT as shown in Line 11. In Lines 6-7, the content of StringT comes from parameter mem of Risk. GameParser(String), and the tokenizer splits mem into two parts. The first part is used for the branch condition in Line 9, while the second part is passed to variable name and output to the GUI. Only the second part needs translation.

Then the developer finds an invocation of Risk.GameParser(String) in Line 19, which passes the actual argument "CARD "+game.getDeservedCard() to the method. Furthermore, the developer needs to look into the implementation of RiskGame.getDeservedCard() and finds that it returns two possible values: "wildcard" and c.getCountry.getName(). A possible value of the latter is actually a country name from a data file, and the related code is not shown for simplicity. Thus, two possible values of the actual argument in Line 19 and the parameter in Line 5 can be "CARD wildcard" and "CARD XXXX", where "XXXX" is a country name from the data file.

From the preceding analysis, the developer can know that the first part of StringTokenizer StringT is "CARD" and the second part is either "wildcard" or "XXXX". Therefore, the constant string "CARD" in Line 19 is used for only the branch condition and does not require translation, while the constant string "wildcard" in Line 24 is output and requires translation. Furthermore, the developer can know that "CARD" in Line 9 does not require translation, because "CARD" in Line 9 is compared to only the first part of StringT. However, "wildcard" in Line 13 requires translation, because "wildcard" in Line 13 is compared to the second part of StringT and the second part requires translation because it is passed to the GUI.

From this example, we can see that a developer needs to perform a tedious and error-prone analysis to determine which string requires translation and which string does not, and the developer needs to be experienced enough to do so. In particular, it is also necessary to analyze contents of string variables and comparisons of strings. Such analysis helps determine that constant strings "wildcard" in Lines 13 and 24 require translation while constant strings "CARD" in Lines 9 and 19 do not.

3 Approach

Our approach consists of three main steps. The first step is to maintain a list of API methods that output strings to the GUI. This step is a preparation step before we begin to locate need-to-translate strings in software applications that have not been internationalized. We refer to these API methods as the output API methods. The second step is to search in the application for invocations of the output API methods and determine the actual arguments that are output to the GUI. We refer to these actual arguments as the initial output strings. The third step is to trace from the initial output strings to other places that may contain need-to-translate constant strings. This step includes four practical techniques: string-taint analysis, string-transmission analysis, string-comparison analysis, and filtering.

3.1 Collecting Output API Methods

To represent an API method in the list of output API methods, we use the signature of a method with full package and class names, because method names may be overloaded and classes in different packages may have the same name. Furthermore, for each output API method, we also specify which parameters of the method are output to the GUI and denote them as output parameters.

Note that, in API libraries of a programming language such as Java, there are typically a small number of classes (or modules) containing output API methods. For example, in API libraries for Java, packages java.awt.* and javax.swing.* are the main sources of output API methods for general Java programs, and package org.eclipse.swt.* is the main source of output API methods for Java programs running on Eclipse. That is to say, output API methods often involve only a small subset of all the API methods, and thus can be collected manually with reasonable effort.

3.2 Locating Initial Output Strings

For each output API method, we search for all possible invocations of the method in the software application and record locations of the invocations. Due to polymorphism, such an invocation may not appear as an invocation of an output API method syntactically. We consider all the invocations that may be bound to an output API method. Note that searching for possible invocations of a given method under polymorphism is a mature technique and has been implemented in IDEs such as Eclipse.

After we locate all the possible invocations of output API methods, we trace to the actual arguments corresponding to the output parameters of the output API methods. These actual arguments are the initial output strings.

3.3 String-Taint Analysis

From each initial output string, we perform an adapted string-taint analysis to locate the possible sources of the initial output string in the code. Among the located sources, hard-coded constant strings and strings transmitted across the network are of particular interest to us.

String-taint analysis is a technique recently proposed by Wassermann and Su [14] based on string analysis [1, 11] whose purpose is to predict the possible values of a certain string variable in the code. Wassermann and Su adapted string analysis to further analyze whether some substrings in the string variable might come from insecure sources. With source code, a string variable, and insecure locations as input, string-taint analysis predicts the given string variable's possible values and determines whether the possible values might contain insecure substrings (i.e., those from insecure sources).

The general idea of string analysis is as follows. First, the program is changed to the static single assignment (SSA) [3] form. Second, string assignments and concatenations that the string variable under analysis depends on are abstracted as an extended context free grammar (CFG) with string operations (i.e., library methods managing strings such as `String.subString(int,int)` in Java) on the right-hand side. Then these string operations are simulated with finite-state transducers (FSTs) [11]. Finally, the language of the extended CFG includes all the possible values of the string variable under analysis. String-taint analysis adapts string analysis by adding annotations to the substrings that are involved in string analysis and propagating these annotations during the process of string analysis. Thus, if strings from insecure sources are properly annotated, examining for their annotations can help determine whether the possible values of the string variable 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 annotated them as "&FileInput" and "transmitted", etc.

To illustrate this process, we next describe how our approach analyzes the code in Section 2. First, we compile and transformed the code to the SSA form as below.

```
if(c.getCountry==null){
    return1 = "wildcard";
}else{
    return2 = &FileInput;
}
return3 = φ(return1, return2);
parseCard = "CARD"+return3;
message = φ(parseCard,{other actual arguments});
StringT = new StringTokenizer(message, " ");
addr = StringT.getNext();
if(addr.equals("CARD")){
    name = StringT.getNext();
    output = You got a new card: +name;
}
```

Then, we transform the SSA form to the extended CFG as below. In the transformation, we add the exact location of each constant string as the annotation to the constant string. For example, for string `wildcard`, we add "RiskGame.java:6767" (`"wildcard"` starts from the 6767th character in RiskGame.java) as its annotation, we do not show annotations in the following grammar for briefness.

```
return1 → wildcard
return2 → &FileInput
return3 → return1|return2
parseCard → CARD return3
message → parseCard|...
StringT → message
addr → nextToken(stringT, " ")
StringT1 → reduceToken(StringT, " ")
name → nextToken(stringT1, " ")
StringT2 → reduceToken(StringT1," ")
output → You got a new card: name
```

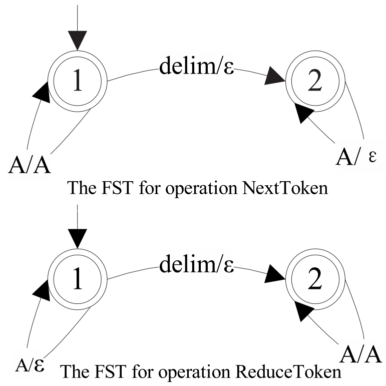
1
delim/ε
2
A/A
A/ ε
The FST for operation NextToken
1
delim/ε
2
A/ε
A/A
The FST for operation ReduceToken

Figure 1: FSTs for `StringTokenizer.getNext()` (where A denotes Σ*/delim)

In the SSA code, there are two types of string operations of `StringTokenizer`: the constructor `StringTokenizer()` and `getNext()`. `StringTokenizer` is a string-manipulating class in Java and is initialized with a string and a delimiter (denoted as delim), and the string is divided into segments with the delim as the separator. Then we can obtain the segments using the method `getNext()`. In the grammar, the constructor `StringTokenizer()` is treated as an ordinary string assignment, and `getNext()` is replaced by two continuous operations: `nextToken()` and `reduceToken()`. In the two operations, `nextToken()` returns the value of the first token while `reduceToken()` returns the remaining string after cutting the first token off the head. The two operations are simulated by two finite state transducers (FSTs)[6] shown in Figure 1. We deduce from `output` in the last line of the extended CFG and obtain possible results[7]: `You got a new card: wildcard` and `You got a new card:&FileInput`. From the annotations of the two constant strings `"You got a new card:"` and `"wildcard"`, we obtain the locations of these two constant strings and marked them as need-to-translate.

3.4 String-Transmission Analysis

Using string-taint analysis in Section 3.3, we are able to trace to string variables whose values are transmitted across the network. We next present our technique to further trace the transmitted strings. A straightforward idea for tracing a transmitted string on one side of an application over the network is to locate the corresponding string variable on the

[6] In Figure 1, the character before "/" is the input to the FST and the character after "/" is the output from the FST.

[7] We do not deduce strings with infinite length, so when our deduction meets the same non-terminal for the second time, we ignore it.

other side of the application, and use string-taint analysis to trace the corresponding string variable on the other side.

However, string variables holding transmitted strings are typically also used to hold strings that do not appear on the GUI. Let us consider a piece of code that implements data transmission between a client and a server. The transmitted data are encapsulated in a class defined as below.

```
1 class Packet {
2    int command;
3    String data;
4    public Packet(int command, String data)
5       {this.command=command; this.data=data;}
6    public int getCommand()
7       {return command;}
8    public String getData()
9       {return data;}}
```

On the server side, the following code portion is used to send two different objects of `Packet` to the client side.

```
10 Packet packet = new packet(Packet.ENDOFGAME,
11     "Automatic Shuts Down")
       ...
12 Packet packet = new packet(Packet.CHAT,
13     "Game saved to"+sFilename);
14 ObjectOutputStream out = new
15     ObjectOutputStream(socket.getOutputStream());
16     out.writeObject(packet);
```

On the client side, the following code portion is used to receive objects of `Packet` transmitted from the server side.

```
17 ObjectInputStream in =
18     new ObjectInputStream(socket.getInputStream());
19 Packet packet = (Packet)in.readObject();
20 switch(packet.getCommand()){
21    case Packet.CLOSECONNECTION:
22            disconnected(); break;
         ...
23    case Packet.CHAT:
24            Output(packet.getData()); break;
         ...
25    case Packet.ENDOFGAME:
26            saveEntityStatus(packet.getData());break;}
```

From the preceding code portions, we know that the client side may receive different objects of `Packet`. However, only when the value of `command` in `Packet` is `Packet.CHAT`, the value of `data` in `Packet` is output to the GUI on the client side. In the preceding code portions, `"Game saved to"` (Line 13), which is sent with `Packet.CHAT`, is passed to the GUI and thus needs translation while `"Automatic Shuts Down"` (Line 11), which is sent with `Packet.ENDOFGAME`, does not need translation. Thus, if we continue to trace `data` in `Packet` on the server side using string-taint analysis, we may trace to some constant strings that are assigned to `data` in `Packet` when the value of `command` in `Packet` is not `Packet.CHAT`. The reason is that string-taint analysis does not analyze different values of `command` in `Packet`.

In fact, the preceding way of data transmission represents a typical mechanism used in object-oriented software for data transmission. First, data for transmission is implemented as objects for transmission. Second, in the class definition of objects for transmission, there is a member variable (i.e., `command` in the preceding code) serving as the label variable of the data for transmission. In addition, there is another one or more member variables (i.e., `data` in the preceding code) holding the data for transmission. If there are strings for transmission, one or more such member variables are defined as strings. Third, after receiving a transmitted object, the receiver needs to check the value of the label variable before using the data, as the receiver needs to interpret the meaning of the data according to the value of the label variable.

To make more precise analysis of transmitted strings, we adopt the following strategy. First, in the class that defines objects for transmission, we determine the member variable serving as the label variable through analyzing the receiver's code. The distinct characteristic of such a member variable is that after receiving an object, the receiver should check this member variable before using the data in the received object. Second, if the received object contains a member variable as a string, for each different value of the label variable in the object, we view the string as a different source of a transmitted string. For example, the two occurrences of `packet.getData()` in Lines 24 and 26 are viewed as different sources. Third, after we trace to a string in a received object in the receiver's code, we further analyze the instantiation of the object for transmission in the sender's code. If the label value of the instantiated object matches the label value of the transmitted object in the receiver's code, we further trace the sources of the corresponding string in the sender's code using our adapted string-taint analysis. If the label value does not match, no string-taint analysis is performed on the sender's code.

3.5 String-Comparison Analysis

In Sections 3.3 and 3.4, our aim is to trace constant strings that may be viewable on the GUI. However, not only strings viewable on the GUI require translation in software internationalization. In the example presented in Section 2, `"wildcard"` in Line 24, which is a source of `name`, needs translation. Since the constant string `"wildcard"` in Line 13 is compared to `name`, `"wildcard"` also needs translation. Therefore, after we locate constant strings viewable on the GUI, we need to further locate the strings that are compared with these viewable strings.

To address this issue, we first locate all the comparisons between strings in the source code. In particular, we locate comparisons between strings through identifying invocations of string-comparison methods provided by the supporting libraries (e.g., `String.endWith()` in Java, `strcmp()` in C). Then for each side of each comparison, we perform our adapted string-taint analysis to locate all the constant strings that are the sources of the side. If any constant string located as a source for one side is in the set of viewable strings located with the techniques in Sections 3.3 and 3.4, we include all the constant strings located as sources for the other side as need-to-translate strings. We iteratively perform the preceding string-comparison analysis until we cannot locate any more need-to-translate strings.

3.6 Filtering

As a practical matter, not all the strings located with the techniques described in Sections 3.3, 3.4, and 3.5 require translation. Some strings should remain the same in most or even all local languages (e.g., strings composed of arabic numerals), while some other strings may be intentionally reserved as untranslated (e.g., trademarks). Therefore, as the final technique of our approach, we further filter out some located constant strings that may not need translation. Currently, we use two simple heuristics. First, we filter out any constant string that does not include any letter character. Second, we filter out any constant string that is equal (ignoring the case) to the name of the project. For example, we filter out the constant string `"\""` in Line 17 in the code portion in Section 2 according to the first heuristic.

4 Implementation

We implemented an Eclipse plug-in called TranStrL (need-to-Translate String Locator) for our approach. In TranStrL, we chose Java as the target language because Java is a widely used programming language in open source applications. For a Java application, TranStrL presents all the located need-to-translate constant strings, the source files that contain these strings, and the exact locations of these strings in the source files. We first used the front-end of the JSA tool (a Java string analyzer) [1] to obtain the extended context free grammar (CFG) with string operations; then we built FSTs [11] to simulate string operations in Java; and finally we added annotations before deducing with the extended CFG. In TranStrL, we collected output API methods from two packages: `java.awt.*` and `javax.swing.*`. So TranStrL currently supports Java applications using only these two packages to implement their GUI.

5 Empirical Study

5.1 Study Setup

In our empirical study, we used four real-world open source applications as subjects: RText, Risk, ArtOfIllusion (AOI) and Megamek. All the four applications are accessible from the web site of sourceforge[8]. RText is a programmer-oriented text editor started since Nov. 2003. Risk is a board game started since May 2004. AOI is a graph editor started since Nov. 2000. Megamek is a real-time strategy game started since Feb. 2002. We chose these four applications with two main reasons. First, all the four applications are among the highest ranked programs that meet the requirement of our study (i.e., having versions before and after internationalization, and having GUIs built on `java.awt.*` and `javax.swing.*`). Second, the four applications represent software in different categories and their GUI structures are different. Both RText and AOI have typical component-based GUIs (i.e., GUIs built with buttons, dialogs, menus, etc.). As AOI is a graph editor, it includes more operations on canvas and graphs. By contrast, Risk and Megamek are two different games with more stylized and complex GUIs. We chose two games as subjects because the GUIs of games are typically more complex than other types of applications and it would be interesting to see how our approach performs on this type of applications.

The developers of all these four subjects did not consider internationalization at the beginning, and they used many hard-coded constant strings in English in early versions of theses subjects. In Summer 2004, the developers of RText internationalized RText and updated the application from Version 0.8.6.9 to Version 0.8.7.0. During this time of internationalization, they internationalized only the core package of the application (i.e., package `org.fife.rtext`). The primary aim of the internationalization was to create a version for Spanish users. In Winter 2004, the developers of Risk internationalized Risk and updated the application from Version 1.0.7.5 to Version 1.0.7.6. In Summer 2002, AOI was internationalized and updated from Version 1.1 to Version 1.2. In Spring 2005, the developers of Megamek began to internationalize Megamek and finished the first internationalized version (i.e., Version 0.29.73).

For all the four subjects, the developers externalized some hard-coded constant strings to resource files and translated the externalized constant strings to the target languages during internationalization.

To evaluate how useful our approach is for real-world internationalization tasks, for each subject, we applied TranStrL to the version before internationalization and compared the results achieved by TranStrL with the actual changes for internationalization made by the developers. Before we report the empirical results, we present the statistics of the subjects in Table 1. For each subject, Columns 1-6 show the name and version number of the application, the starting month of the application, the number of developers involved in the development of the application[9], the number of lines of code (LOC) of the application, the number of files of the application, and the number of constant strings of the application, respectively[10].

Column 7 shows the number of the need-to-translate constant strings, which serve as the golden solution in our empirical study. We obtained our golden solution as follows. First, we deemed constant strings in the version before internationalization as need-to-translate constant strings, if the developers externalized them in the

[8] `www.sourceforge.net`, accessed on June 20, 2008

[9] Sourceforge counts all the persons who contributed to a project as developers, so it is no wonder that there are 16 developers for RText.

[10] The statistics for Ver 0.8.6.9 of RText are only for package `org.fife.rtext`, as the developers internationalized only this package.

subsequent internationalized version. Second, since our approach did find a number of need-to-translate constant strings that were not externalized in the subsequent internationalized version for each subject, we also deemed as need-to-translate constant strings the constant strings that were located by TranStrL and manually verified by us to need translation.

In particular, when TranStrL located a constant string not externalized in the subsequent internationalized version, we further checked versions later than the subsequent internationalized version. If the constant string was externalized in a later version, we also deemed it as need-to-translate. If not, we used some manually generated input data to execute the subsequent internationalized version. If the string was viewable on the GUI and not understandable to a user not familiar with English, we deemed it as need-to-translate; otherwise, we deemed it as not needing translation. In principle, we adopted a conservative policy to avoid misclassifying strings that do not need translation as need-to-translate. That is to say, we tried to avoid biasing our evaluation favorably to our approach.

5.2 Empirical Results

5.2.1 Overall Results and Analysis

Results. We present the results of applying TranStrL to the four subjects in Table 2. In this table, we refer to strings that need translation but are not located by our approach as false negatives, and strings that are located by our approach but actually do not need translation as false positives. From the table, we have the following observations.

First, our approach (using all the tracing techniques) is able to locate most of the need-to-translate strings. In RText, our approach locates all the need-to-translate strings, while in Risk, AOI, and Megamek, our approach locates 491 of 509, 1215 of 1221, and 1724 of 1734 need-to-translate strings, respectively. That is to say, the false negatives of our approach for all the four subjects are quite small.

Second, for each subject, our approach does find a few false positives. In RText, Risk, AOI, and Megamek, the numbers of strings that are located by our approach but do not need translation are 37, 7, 65, and 41, respectively. Compared to the numbers of need-to-translate strings in the four subjects, the numbers of false positives are also quite small.

Third, for each subject, our approach is able to locate some constant strings that the developers did not externalize in the subsequent internationalized version but were verified by us as need-to-translate. The developers might have either missed them or did not externalize them at that time due to time or workload limit. In both cases, locating such strings should be helpful for the developers to produce a version with better quality of internationalization earlier.

In total, our approach locates 1670 such strings in the four subjects. Among the 1670 strings, 1422 (87 in RText, 10 in Risk, 746 in AOI, and 579 in Megamek) were externalized and translated in a later version and 248 still remained hard-coded in all the later versions or were removed due to modifications other than internationalization. We next present two examples of the two preceding situations.

The first example is from RText. In the subsequent internationalized version (i.e., 0.8.7.0), the text editor shows the position of the cursor at the lower right corner of the panel in the form of `"Line xx, Col.  xx"`. However, constant strings `"Line"` and `"Col."` are not externalized. The developers of RText externalized and translated the two strings 11 months later in Version 0.9.1.0.

The second example is from Megamek as shown in the following piece of code.

```
public MechView(Entity entity) {
    ...
    StringBuffer sBasic;
    sBasic.append( Messages.getString("MechView.Movement") )
    ...
    sBasic.append(entity.getMovementTypeAsString())}
public String getMovementTypeAsString(){
    switch (getMovementType()) {
        ...
        case Entity.MovementType.TRACKED:
            return "Tracked";
        case Entity.MovementType.WHEELED:
            return "Wheeled";
        ...}}
```

Variable `sBasic` in the method `Mechview()` (in `megamek.client.Mechview.java`) is finally passed to the GUI as the description of weapons in the game. Therefore, the developers externalized the first part of `sBasic` as `Messages.getString("MechView.Movement")`, and added an item `"Mechview.Movement"` in the resource file (i.e., `"Movement:"` for English and `"Bewegung:"` for German). But even in the latest version they did not externalize the second part, which is a return value from method `getMovementTypeAsString()`. Therefore, a strange string with its first part translated to German but second part remaining in English appears on the GUI of the German version of the game. We have reported all 17 untranslated need-to-translate strings located by TranStrL to the Megamek developers as bug report #2085049 and all these 17 strings were confirmed and fixed by the developers. We next further discuss the reasons for the false negatives (i.e., need-to-translate strings not located) and the false positives (i.e., located need-to-translate strings that actually do not need translation).

Analysis of false negatives. Generally, the false negatives fall into three categories. The first category is constant strings that are compared to string variables whose values come from viewable items in widgets (such as an AWT Table item or an AWT List item) on the GUI. If the developers translate constant strings whose value appears in a widget on the GUI, the translation may impact string variables that are compared to these translated strings. This category includes all the 6 false negatives from AOI, all the 10 false

Table 1: Basic information of the subjects

Application /Version	Starting Month	#Developers	#LOC	#Files	#Constant Strings	#Need-to-Trans(Not externalized in the subsequent version)
Rtext 0.8.6.9(Core Package)	11/2003	16	17k	55	1252	408(121)
Risk 1.0.7.5	05/2004	4	19k	38	1510	509(55)
AOI 1.1	11/2000	2	71k	258	2889	1221(816)
Megamek 0.29.72	02/2002	33	110k	338	10464	1734(678)

Table 2: Results of applying our tool on the four applications

Application	Need-to-Trans(Not externalized in the subsequent ver)	Located	False Neg (FN)	False Pos (FP)
RText	408(121)	445	0	37
Risk	509(55)	498	18	7
AOI	1221(816)	1280	6	65
Megamek	1734(678)	1765	10	41

negatives from Megamek, and 3 of the 18 false negatives from Risk. In principle, string-comparison analysis should be able to locate strings in this category. The reason that our tool failed to do so in our empirical study is as follows. This category involves some string assignments or even string comparisons implemented in library code. Our tool cannot trace into library code whose source code is not available, but if we can extend string-taint analysis and string-comparison analysis to library code, we can address this category of false negatives.

The second category is constant strings related to the names of language-related file folders (e.g., maps and cards). 10 of the 18 false negatives in Risk belong to this category. Let us take map folders as an example. Since Risk is a game application, various maps are used. As maps may contain texts specific to particular languages, versions for different languages may require different sets of maps. To internationalize maps, the developers used different folders to store maps for different languages. Thus, when switching from one language to another, the names of map folders should also be switched.

The third category is debugging messages viewable on the console but not output through output API methods. In Risk, 5 of the 18 false negatives belong to this category. Note that developers may choose to or not to internationalize debugging messages. For RText, our approach located 2 debugging messages (which are output through API methods), but the developers did not externalize them and we counted them as false positives. However, developers of Risk externalized 5 debugging messages, which we counted as 5 false negatives.

Analysis of false positives. Generally, the false positives fall into four categories. The first category of false positives consists of strings that are viewable on the GUI but may be intentionally left as not translated. Such strings include version information, copyright information, acronyms, etc. Since we used a conservative policy when verifying strings that are located by our approach but not externalized by the developers, we counted these strings as false positives. In total, 18 of 37 false positives in RText, 3 of the 7 false positives in Risk, 4 of 65 false positives in AOI, and 6 of 41 false positives in Megamek belong to this category.

The second category of false positives consists of strings that are viewable on the GUI but cannot be translated. For example, file-extension or directory names (such as “*.txt” or “C:/abc”) appear in dialogs related to file selection, but these names should be the same for different languages. Another example is the names of fonts (e.g., Times New Roman). These names may also appear on the GUI, but should remain the same for different languages. Furthermore, string-comparison analysis introduces more false positives if strings are compared with false positives in this category. In total, 14 of 37 false positives in RText, 4 of 7 false positives in Risk, 61 of 65 false positives in AOI, and 35 of 41 false positives in Megamek belong to this category.

The third category includes 3 of 37 false positives in RText. These strings are HTML tags. They are passed to some texts in the HTML format and these texts are then passed to a window that displays HTML files. That is to say, the texts are for display on the GUI, but translating these HTML tags may result in improper display.

The fourth category is those used for debugging. This category includes 2 false positives in RText. That is to say, these 2 strings can appear in windows for displaying debugging information. As the developers may not be familiar with multiple languages, translating these strings may impact the debugging process negatively.

Summary. For all the four subjects, our approach is able to locate most of the need-to-translate strings while producing only small numbers of false negatives and false positives. Among the false negatives, the first category may result in run time errors but can be addressed by extending the analysis to analyze library code. The second category can be easily detected by analyzing the file system. The third category is relatively trivial for users to detect. Among the false positives, the first category actually can be removed by translators who know about the customs of local users. The second and the third categories of false positives may result

Table 3: Turning on and off string-transmission analysis

Application	Need-to-trans	Located	FN	FP
Megamek	1734	1765	10	41
Megamek(NT)	1734	1188	585	39
Megamek(ALL)	1734	1777	10	53

in run time errors, but can be detected by heuristic-based checkers. The fourth category is also trivial for users to detect. Furthermore, for each subject, our approach is able to locate some need-to-translate strings that the developers did not locate when internationalizing the subject.

The preceding results show that our approach is useful in at least the following two scenarios. First, developers can use our approach to generate candidates for translation, since our approach achieves acceptable results for developers to start internationalization. Second, since our approach can find some strings that developers cannot easily find by themselves, they can use our approach to check internationalized versions and find missed need-to-translate strings.

5.2.2 Effects of Different Techniques

In our approach, the basic tracing technique is string-taint analysis, and we also develop three other techniques (i.e., string-transmission analysis, string-comparison analysis, and filtering) to cope with practical complications. To evaluate the effects of the three techniques in our approach, we performed a series of evaluations. The baseline was to use all of the three techniques with string-taint analysis, and we turned off each technique at a time to see how the specific technique affects the results.

Effects of string-transmission analysis. We show the results of turning on and off string-transmission analysis in Table 3. Since only Megamek transmits strings across the network, turning on or off string-transmission analysis affects the result of only this subject. We considered two ways of turning off string-transmission analysis. In the first way, we did not analyze string variables whose values are transmitted across the network. In the second way, we used string-taint analysis to analyze all string variables whose values are transmitted across the network without considering the label variable in transmitted objects.

In Table 3, the line marked with "(NT)" presents the results of our approach with turning off string-transmission analysis in the first way, while the line marked with "(ALL)" presents the results of our approach with turning off string-transmission analysis in the second way.

From the table, we observe that, compared to the first way of turning off string-transmission analysis, string-transmission analysis helps find 575 more need-to-translate strings (or reduce 575 false negatives) in Megamek, introducing 2 false positives (falling into the second category of false positives discussed in Section 5.2.1). Compared to the second way of turning off string-transmission analysis, string-transmission analysis helps reduce 12 false positives.

Table 4: Turning on and off string-comparison analysis

Application	Need-to-trans	Located	FN	FP
RText	408	445	0	37
RText(NC)	408	445	0	37
Risk	509	498	18	7
Risk(NC)	509	474	42	7
AOI	1221	1280	6	65
AOI(NC)	1221	1280	6	65
Megamek	1734	1765	10	41
Megamek(NC)	1734	1730	36	32

Table 5: Turning off the string filter

Application	Need-to-trans	Located	FN	FP
RText	408	445	0	37
RText(NF)	408	581	0	173
Risk	509	498	18	7
Risk(NF)	509	532	18	41
AOI	1221	1280	6	65
AOI(NF)	1221	1487	6	272
Megamek	1734	1765	10	41
Megamek(NF)	1734	2080	10	356

Effects of string-comparison analysis. We show the results of turning on and off string-comparison analysis in Table 4, in which the lines marked with "(NC)" present the results of turning off string-comparison analysis.

First, string-comparison analysis is helpful to find more need-to-translate strings (or reduce false negatives) in two of four subjects (i.e., 24 in Risk and 26 in Megamek). Second, string-comparison analysis brings in 9 false positives in Megamek. Specifically, these 9 false positives belong to the second category of false positives. That is to say, string-comparison analysis locates these 9 strings because they are compared directly or indirectly to some strings viewable on the GUI but cannot be translated.

Effects of filtering. We show the results of turning on and off filtering in Table 5, in which the lines marked with "(NF)" present the results of turning off filtering.

From Table 5, we observe that, in each subject, filtering can effectively reduce the number of false positives. Furthermore, filtering does not cause any false negatives in all the four subjects. The reason is that we use conservative heuristics in filtering. Actually, if we use some aggressive heuristics, we may further reduce the number of false positives, but the number of false negatives may increase. In fact, we tried some aggressive heuristics as well, but in general the aggressive heuristics did not significantly outperform our simple conservative heuristics.

5.3 Threats to Validity

The main threats to internal validity lie in the way we verify constant strings not externalized in the subsequent internationalized version to be need-to-translate strings for each subject. First, it may be error-prone to verify constant strings as need-to-translate in versions later than the subsequent internationalized version, because the later versions involve various modifications for other purposes. Second, manually verifying constant strings not externalized in any later version as need-to-translate may be prone to accidental mistakes or personal perspectives to the notion of being "need-to-translate". To reduce these threats, for each subject, we examined all these strings in all later versions carefully, executed the internationalized subject to see whether these strings appear on the GUI and decided whether they are not understandable to a user not familiar with English using a conservative policy. In fact, some of the false positives are related to this policy. The second threat to internal validity is that we did not consider the strings that were missed by both the developers and our approach. To reduce this threat, we chose popular software applications to carry out our experiments, so that the quality of manual string externalization should be high. The third threat to internal validity is that we collected output API methods manually and the collected list may not be complete. Although an incomplete list is not in favor of our results, it may affect the false positives and false negatives in our evaluation.

The main threats to external validity are as follows. First, the results of our study may be specific to the applications used in the evaluation. To reduce this threat, we chose applications from various domains and their GUI structures are different from one another. Second, the four subjects used in our empirical study are all open source applications in Java, and all of them are of moderate sizes. Therefore, the findings of our empirical study may be specific to open source applications in Java with moderate sizes, and may not be generalized to other applications. Third, we evaluated the effects of string-transmission analysis only on Megamek, as among the four subjects only Megamek transmits strings across the network. Therefore, the findings on string-transmission analysis in our empirical study may not be generalized to other applications. To further reduce these threats, we plan to apply our approach to more applications, especially those for commercial use, with larger code bases, or having strings transmitted across the network.

6 Discussion

The basis of our approach is string-taint analysis, which was developed based on data-flow analysis. Compared to traditional data-flow analysis, which should also be applicable to trace possible sources of the initial output strings, the main strength of string-taint analysis lies in that it can further analyze contents of strings through formulating string assignments and concatenations as an extended CFG and string operations as FSTs. As a result, string-taint analysis can help reduce some false positives, such as the string `"CARD"` in Line 19 of the example in Section 2.

The main weakness of string-taint analysis is that string-taint analysis deems all the sources of each initial output string as need-to-translate. However, when whether a particular value of an initial output string is output to the GUI depends on values of variables of other types, string-taint analysis may induce inaccuracy, which then may result in some false positives. One possible way to reduce this kind of false positives is to use dynamic analysis [2]. The main disadvantage of using dynamic analysis for locating need-to-translate constant strings is that dynamic analysis requires a set of test data to cover possible usages of the software application under analysis. Furthermore, according to our experience, developers typically do not use other variables to determine whether a particular value of an initial output string is output to the GUI. Thus, the weakness of string-taint analysis may not result in many false positives in practice. One exception that we know is transmitted strings. There a label variable is used to determine which values of a transmitted string are output to the GUI and which are not. To deal with this situation, we developed a technique for transmitted strings.

Our current string-transmission analysis is able to deal with the situation of string transmission via objects through sockets. However, there are still other ways to transmit strings across the network. One popular way to transmit strings is to use a remote function call such as RPC and RMI. Our approach can address this situation with minor adaptation by matching object names rather than socket numbers. Other transmission strategies such as SOAP and EventBus require more specific techniques beyond our technique, and we plan to address them in future work.

7 Related Work

To our knowledge, our work is the first reported effort directly focusing on automatically locating need-to-translate constant strings. There have been a couple of published books on how to internationalize a software application [5, 13]. The books provide some guidelines on how to find out need-to-translate constant strings and externalize them. Some researchers analyzed the process of internationalization and presented issues to be considered during the process, including locating need-to-translate strings [9, 4]. However, none of them provides any automatic approach to locating need-to-translate strings.

String analysis and string-taint analysis are recent advances in static data-flow analysis [10]. Christensen et al. [1] first suggested string analysis, which is an approach for obtaining possible values of a string variable. Gould et al. [6] used string analysis to check the correctness of dynamically generated query strings. Halfond and Orso [8] used string analysis to detect and neutralize SQL injection attacks. Minamide [11] suggested to simulate string operations in an extended CFG with FSTs, and implemented a string analyzer on PHP code to check contents of dynamically generated web pages. Recently, Wassermann and Su developed string-taint analysis [14] based on Minamide's work and further applied the technique on detecting cross-site scripting [15]. In our approach, we adapted string-taint analysis for a new problem (i.e., locating need-to-translate constant strings), and developed techniques to cope with practical complications in the problem.

Our approach can also be viewed as determining a subset of constant strings that are related to the GUI. From this perspective, our approach is also related to research on abstract type determination, which tries to decide the semantic role of a variable in the code. O'Callahan and Jackson [12] proposed a technique based on static data-flow analysis to decide the abstract type of a variable. Guo et al. [7] further improved the approach to the same problem using dynamic data-flow analysis. Our approach differs from these approaches in two main aspects. First, our approach targets at a new problem not addressed by these approaches. Second, our approach is based on various techniques for analyzing strings in source code while these approaches do not focus on strings.

8 Conclusion and Future Work

In this paper, we present a novel approach to automatically locating need-to-translate constant strings. Our approach is based on string-taint analysis, and proposes three practical techniques to cope with the complications in the targeted problem. Furthermore, we implemented our approach as an Eclipse plug-in. We evaluated our approach on four real-world open-source applications: RText, Risk, ArtOfIllusion (AOI), and Megamek. The empirical results show that our approach is able to locate most of the constant strings externalized by the developers, with small numbers of false positives and false negatives.

In future work, we plan to extend our approach to address the following research issues. First, we plan to extend our tool for analyzing Java library code, because the current inability to trace into Java library code causes some false negatives. Second, we plan to extend our approach to support other ways of string transmission across the network. Third, we plan to further automate the collection of the output API methods in our approach. In particular, we plan to mine the list of output API methods from existing internationalized software applications, in which we can trace forwardly from the externalized strings to the methods that eventually send the strings to the GUI. Fourth, there are factors other than text translation that affect the quality of software internationalization (e.g., date/time, number formats, different colors for emphasis in different cultures), we plan to further address such problems. Finally, we plan to extend our approach to locate need-to-translate strings in Web applications, in which texts viewable on Web pages are typically concatenated with non-viewable tags.

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] J. A. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proc. ISSTA*, pages 196–206, 2007.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Prog. Lang. and Sys.*, 13(4):451–490, October 1991.

[4] V. Dagiene and R. Laucius. Internationalization of open source software: framework and some issues. In *Intl. Conf. on Information Technology: Research and Education*, pages 204–207, 2004.

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

[6] C. Gould, Z. Su, and P. T. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. ICSE*, pages 645–654, 2004.

[7] P. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *Proc. ISSTA*, pages 255–265, 2006.

[8] W. G. J. Halfond and A. Orso. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proc. ASE*, pages 174–183, 2005.

[9] J. Hogan, C. Ho-Stuart, and B. Pham. Current issues in software internationalisation. In *Proc. Australian Computer Science Conf.*, 2003.

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

[11] Y. Minamide. Static approximation of dynamically generated web pages. In *Proc. WWW*, pages 432–441, 2005.

[12] R. O'Callahan and D. Jackson. Lackwit: a program understanding tool based on type inference. In *Proc. ICSE*, pages 338–348, 1997.

[13] E. Uren, R. Howard, and T. Perinotti. *Software Internationalization and Localization: An Introduction*. Van Nostrand Reinhold, 1993.

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

[15] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proc. ICSE*, pages 171–180, 2008.