# Locating Need-to-Externalize Constant Strings for Software Internationalization with Generalized String-Taint Analysis

Xiaoyin Wang, Lu Zhang, Tao Xie, *Senior Member*, *IEEE*,
Hong Mei, *Senior Member*, *IEEE*, and Jiasu Sun

**Abstract**—Nowadays, a software product usually faces a global market. To meet the requirements of different local users, the software product must be internationalized. In an internationalized software product, user-visible hard-coded constant strings are externalized to resource files so that local versions can be generated by translating the resource files. In many cases, a software product is not internationalized at the beginning of the software development process. To internationalize an existing product, the developers must locate the user-visible constant strings that should be externalized. This locating process is tedious and error-prone due to 1) the large number of both user-visible and non-user-visible constant strings and 2) the complex data flows from constant strings to the Graphical User Interface (GUI). In this paper, we propose an automatic approach to locating need-to-externalize constant strings in the source code of a software product. Given a list of precollected API methods that output values of their string argument variables to the GUI and the source code of the software product under analysis, our approach traces from the invocation sites (within the source code) of these methods back to the need-to-externalize constant strings using generalized string-taint analysis. In our empirical evaluation, we used our approach to locate need-to-externalize constant strings in the uninternationalized versions of seven real-world open source software products. The results of our evaluation demonstrate that our approach is able to effectively locate need-to-externalize constant strings in uninternationalized software products. Furthermore, to help developers understand why a constant string requires translation and properly translate the need-to-externalize strings, we provide visual representation of the string dependencies related to the need-to-externalize strings.

**Index Terms**—Software internationalization, need-to-externalize constant strings, string-taint analysis

## 1 Introduction

In this era of globalization, a software product is often distributed to different regions of the world. To be better used by users in a certain region, the software product should have a corresponding local version for local users in the region. Typically, in a local version of a software product, user-visible texts should be in the local language, and other locale-related elements such as measures and dates should also be in the local formats. For example, for the English version, all user-visible texts should be in English, the length measure should be in miles, feet, inches, etc., and the dates should be in the format of MM/DD/YYYY or DD/MM/YYYY. To generate and manage all the local versions, a typical process is to internationalize the software product. During software internationalization, all the locale-related elements in the source code are externalized to resource files. After a software product is internationalized, developers can generate a different local version via automatically synthesizing the corresponding set of local resource files together with the internationalized version.

In some software products, developers consider internationalization in the beginning of the development process. That is to say, developers need to avoid hard-coding locale-related elements (e.g., constant strings) from the beginning of the development process. However, developers sometimes need to apply internationalization on existing code in two major situations. First, many popular software products originate from open source prototypes or research prototypes. When these prototypes were developed, developers usually did not expect the improved versions of their prototypes to be distributed to the global market so that they might write hard-coded constant strings in the code for coding efficiency. When developers improved these prototypes and planned to distribute improved versions to the global market, they then needed to perform internationalization on the existing code. Second, due to the adoption of software reuse, developers of an internationalized software product may reuse some uninternationalized software components. In such a situation, they may have to internationalize the existing code of these reused components. Furthermore, in an internationalized software product, there may also exist some mis-externalized locale-related elements for the following two reasons. First, since the GUI and the program structure can be very complex, the developers may forget to externalize

- *X. Wang, L. Zhang, H. Mei, and J. Sun are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, and the Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, P.R. China. E-mail: {wangxy06, zhanglu, meih, sjs}@sei.pku.edu.cn.*
- *T. Xie is with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695. E-mail: xie@csc.ncsu.edu.*

*Manuscript received 17 Aug. 2011; revised 26 Apr. 2012; accepted 18 May 2012; published online 5 June 2012.*
*Recommended for acceptance by F. Tip.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-08-0243.*
*Digital Object Identifier no. 10.1109/TSE.2012.40.*

some need-to-externalize elements or make wrong decisions on whether an element needs to be externalized. Second, for relatively large software products, developers writing the code of the GUI and developers writing the code of the back-end logic may be different. In such a case, imperfect interaction between developers may also result in misexternalized elements. For example, the back-end developers believe that a certain field of a certain Java class will not be output to the GUI, so they do not externalize all the constant strings that flow to the field. However, the GUI developers may output this field later for some reasons and forget to inform the back-end developers.

Therefore, developers need to locate those need-to-externalize elements when they want to internationalize an existing software product or to check for the misexternalized elements in an internationalized software product. The need-to-externalize elements typically include constant strings, time/data objects, number objects, measures, and culture-related objects (e.g., color), etc. [5], [19]. Among all these need-to-externalize elements, need-to-externalize constant strings are the most tedious and error-prone to locate. This is because both the number of constant strings and the number of need-to-externalize constant strings are often very large, but not all constant strings need to be externalized.

To help developers locate need-to-externalize constant strings, many Integrated Development Environments (IDEs) (e.g., Eclipse) provide support to locate all the constant strings in source code. However, according to the results of our empirical study (see Table 2 in Section 4), only 11-42 percent of the constant strings actually need to be externalized. Therefore, if all the constant strings are externalized, translators would put much more efforts into the localization process for each new region. Furthermore, some constant strings must not be externalized and translated. For example, the constant strings that are used as keys for a database must not be externalized and translated; otherwise, an internal error will occur. Therefore, existing support to locating need-to-externalize constant strings is insufficient and there is a strong need for more tool support that can differentiate need-to-externalize constant strings from other constant strings.

In this paper, we propose an automatic approach to locating need-to-externalize constant strings. Our approach is mainly based on generalized string-taint analysis and the basic idea is as below. Before we process any software product, we collect a list of GUI-related API methods that output values of their string argument variables to the GUI. With the list of collected API methods, for a given software product, we identify all the invocation sites of these API methods, and trace back from the output string variables related to these method invocations to find constant strings that flow to the GUI-related API methods in the source code based on four techniques. We deem the found constant strings as need-to-externalize constant strings and report them to the developers.

The four techniques in our tracing approach are as below.

- The first technique generalizes string-taint analysis [25] to trace from the output string variables to their data sources. We report the data sources that are constant strings as need-to-externalize constant strings. We choose to base our approach on string-taint analysis instead of basic data-flow analysis because string-taint analysis can further analyze contents of strings through formulating string assignments and concatenations. Actually, due to the variety and complexity of string operations, it is likely that constant strings participating in some string operations may not have their values flow to the final result of the string operations. If the final result of the string operations further flows to the GUI, without string-taint analysis, such constant strings will be mistakenly determined as need-to-externalize constant strings (see the example in Section 2).
- The second technique handles those software products that include network communication features. In such software products, constant strings may be transmitted from one side of the network to the other side. So we further develop string-transmission analysis to analyze the transmission of values of string variables across the network to locate those hard-coded constant strings that are in the source code of one side of the network but may appear on the GUI of the other side of the network.
- The third technique handles the comparisons between string variables. Some need-to-externalize constant strings located via the preceding two techniques are compared with other string constants or variables. If these compared string constants or variables are not externalized in the internationalized software product, the result of the comparison may be wrong and some internal errors may occur. So we further develop string-comparison analysis to locate the data source of the string variables that are compared with need-to-externalize constant strings.
- The fourth technique handles trivial constant strings that do not require translation. Some constant strings (e.g., strings that contain only arabic numbers) do not require translation even if they may appear on the GUI and visible to the users. So we further develop a filter to remove trivial constant strings.

In summary, this paper makes the following main contributions:

- We propose an automatic approach to locating need-to-externalize constant strings in source code based on collecting GUI related API methods and tracing from their output string variables. Specifically, our approach is based on generalized string-taint analysis, and involves three practical techniques that handle string transmission, string comparison, and trivial constant strings.
- We conducted an empirical evaluation on seven real-world open source software products that demonstrates the effectiveness of our approach on locating need-to-externalize constant strings in uninternationalized software products. 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.[1] All 17 strings were confirmed and later translated by Megamek developers.

- We designed and implemented visualization of the string dependencies related to the need-to-externalize constant strings so that the developers can better understand how these constant strings go to the GUI and how to externalize and translate them.

This paper is an extended and revised version of our previous conference paper [22]. The main differences between this paper and our previous conference paper are as follows: First, we generalize string-taint analysis to formulate generalized string-taint analysis. Although generalized string-taint analysis is technically similar to string-taint analysis, it extends string-taint analysis by allowing various types of taints. Thus, the generalized string-taint analysis provides a more general application scenario of string-taint analysis, and enables us to present our approach in a more clear and precise way. Second, on top of the empirical study on software products of editors and games that we reported in our previous paper, we further applied our approach on two other categories of software products: content-presentation applications and GUI-related libraries. Specifically, we applied our approach on two content-presentation applications (i.e., TV-Browser and StoryBook) and one GUI-related library (i.e., JFreeChart). Our new empirical evidence confirms most conclusions in our previous paper and reveals some new categories of false negatives. Third, we designed and implemented GUI tool support to visualize the string dependencies related to a need-to-externalize constant string to further help developers to externalize and translate these strings. Furthermore, we use a string example graph to simplify the string dependencies to help developers explore them. Fourth, we present the time and memory usage of applying our approach on the studied subjects to evaluate the performance of our approach.

The rest of this paper is organized as follows: We present an example of locating need-to-externalize constant strings in Section 2. We present our approach in detail in Section 3. We report the empirical evaluation of our approach on locating need-to-externalize constant strings in Section 4. We describe our visualization support in Section 5. In Section 6, we further discuss related issues. In Section 7, we discuss related work, and in Section 8, we conclude with future work.

## 2 Example

We next present an example to illustrate the situation that a developer may face when manually locating need-to-externalize constant strings in source code. The example comes from Risk (Version 1.0.7.5), a real-world open source project used in our empirical evaluation. 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.nextToken();
       ...
9      if(addr.equals("CARD")){
10        if(StringT.hasMoreTokens()){
11          String name = StringT.nextToken();
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.getDeserved
      Card());
      ...
     }
   }
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 `"You got a new card:"` needs externalization. In addition, 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 `StringTokenizerStringT` as shown in line 11. In lines 6-7, the value 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

1. http://sourceforge.net/projects/megamek/.

passed to variable `name` and output to the GUI. Only the second part needs externalization.

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 here 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 need externalization, while the constant string `"wildcard"` in line 24 is passed to the GUI and needs externalization. Furthermore, the developer can know that `"CARD"` in line 9 does not need externalization, because `"CARD"` in line 9 is compared to only the first part of `StringT`. However, `"wildcard"` in line 13 needs externalization because `"wildcard"` in line 13 is compared to the second part of `StringT` and the second part needs externalization 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 needs externalization and which string does not, and the developer needs to be experienced enough to do so. It should be noted that analyzing contents of string variables and comparisons of strings may also be necessary. Such analysis helps determine that constant strings `"wildcard"` in lines 13 and 24 need externalization while constant strings `"CARD"` in lines 9 and 19 do not. In contrast, basic data-flow analysis cannot detect the value change of `StringT` at line 8, so it cannot decide that `StringT` at line 11 does not contain `"CARD"` as a part of its value. Therefore, basic data-flow analysis would erroneously determine all the four constant strings as need-to-externalize. To avoid such imprecision, we choose to base our approach on string-taint analysis, which is able to analyze the contents of string variables. Furthermore, we need some new techniques to handle various complications such as string comparisons.

## 3 Approach

There are three main steps in our approach. The first step of our approach is to collect a list of API methods that output strings to the GUI. This step is a preparation step before we begin to locate need-to-externalize strings in source code. These API methods are referred to as *output API methods* in the rest of this paper. The second step of our approach is to search in the source code for invocations of the *output API methods* and identify the actual arguments that are output to the GUI. These actual arguments are referred to as *initial output strings* in the rest of the paper. The third step of our approach is to trace from each *initial output string* to places that may contain need-to-externalize constant strings.

In particular, the third step includes four techniques: generalized string-taint analysis (which is adapted from string-taint analysis [25]), string-transmission analysis, string-comparison analysis, and filtering of trivial constant strings.

### 3.1 Collecting Output API Methods

We observe that, in the API libraries of most programming languages, there are typically only a small number of GUI-related packages and modules containing *output API methods*. For example, in the standard API libraries for the Java language, 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. Therefore, when manually collecting the *output API methods*, we need to consider only a small number of packages/modules. That is to say, the collecting effort should be limited.

In our collected list of *output API methods*, we use the signature of the method with its class name and full package path to represent an API method. The reason is that method names may be overloaded and classes in different packages may have the same name. Furthermore, for each *output API method*, we also specify the method parameters that are output to the GUI and these parameters are referred to as *visible parameters* in the rest of the paper.

### 3.2 Locating Initial Output Strings

To locate *initial output strings*, we search for all possible invocations of each *output API method* in the source code 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. Therefore, we can directly use a method-invocation search engine in IDEs in our approach.

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

### 3.3 Generalized String-Taint Analysis

To locate the possible data sources of each *initial output string*, we generalize string-taint analysis and apply the generalized string-taint analysis for data-source tracing.

#### 3.3.1 Generalizing String-Taint Analysis

String-taint analysis [25], proposed by Wassermann and Su, is a recent improvement of string analysis [1], [15] 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 data sources (e.g., user inputs). 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 substrings from insecure data sources).

String analysis contains the following three main steps: 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. The arguments of the string operations are nonterminals of the extended CFG. Third, these string operations are simulated with Finite-State Transducers (FSTs) [15], and the extended CFG is converted to a normal CFG through FST-CFG intersection. The language of the normal CFG generated in the third step includes all the possible values of the string variable under analysis.

String-taint analysis adapts string analysis by adding a Boolean annotation (i.e., the taint) to each string data source (corresponding to each terminal in the extended CFG) that is involved in string analysis. If the string data source is secure (e.g., supporting files, constant strings), the value of the annotation is false (denoting security). Otherwise, the value of the annotation is true (denoting insecurity). For a string variable whose value is the operation/concatenation result of some string data sources, the value of the annotation is temporarily false to represent not found to be insecure. Then these annotations are propagated through the extended CFG along with the string analysis process. Specifically, when performing FST-CFG intersection, the annotation values of the data sources in the extended CFG are assigned to the corresponding data sources in the generated normal CFG. Then, in the normal CFG, annotation values are propagated from the right-hand side to the left-hand side for each production. The propagation is iteratively executed until the annotation values of all nonterminals become stable. Thus, we can determine whether a string variable contains insecure substrings by examining whether its corresponding nonterminal in the CFG is annotated with true.

For the ease of applying string-taint analysis to our problem, we generalize string-taint analysis to allow propagations of more complex annotations. Compared to string-taint analysis, our generalized string-taint analysis allows user-defined annotations and user-defined propagation operations on the annotations from the right-hand-side to the left-hand-side of a production in the CFG. Besides the Boolean type, user-defined annotations in our generalized string-taint analysis can also be of the integer type, the string type, the set of a basic type, or even user-defined complex structural types. Furthermore, we explicitly define a propagation operation, which is a function specifying how the annotation of the left-hand-side nonterminal of a production can be derived from the annotations of all the nonterminals/terminals in the production. Note that such a propagation operation may take the current annotation of the left-hand-side nonterminal as one of its inputs. Specifically, during the FST-CFG intersection phase, our approach leverages exactly the same technique as string-taint analysis to assign the annotation values of the data sources in the extended CFG to the corresponding data sources in the generated normal CFG, except that the assigned annotation values can be of user-defined annotation type. Then, in the normal CFG (with no string operations), we calculate the annotations of the left-hand-side nonterminal in each production according to the user-defined propagation operations. Similarly, this calculation process is iteratively performed until the annotation values on all the nonterminals become stable. It should be noted that, like many other iterative processes in program analysis, to ensure the termination of this iteration, all the possible values of the user-defined annotation should form a lattice with finite height, and the result of a user-defined propagation operation should be larger than or equal to its inputs in the lattice (see lattice-based program analysis [16]).

Existing string-taint analysis is actually a special case of our generalized string-taint analysis using the following definition of annotations and propagation operations: First, the annotation of any terminal or nonterminal in the CFG is of the Boolean type. Second, the propagation operation is as follows: If any terminal or nonterminal at the right-hand side of a production is annotated as true, the nonterminal at the left-hand side should also be annotated as true.

### 3.3.2 Applying Generalized String-Taint Analysis

As we are interested in locating hard-coded constant strings, we apply our generalized string-taint analysis using the following definition of annotations and propagation operations. First, each terminal or nonterminal is annotated with a set of locations, and each location specifies a unique location in the source code, including the enclosing file path, the offset, the string length, and a flag to mark the category of the data source (i.e., "constant" for constant strings, "transmit" for function invocations reading from the network, "fileInput" for function invocations reading from a file, and "databaseInput" for function invocations reading from the database). Second, we use the following propagation operation: The resulting annotation of the left-hand side nonterminal is the union of the existing annotation of the left-hand side and the existing annotation of each right-hand side terminal or nonterminal. Note that the main difference between the generalized string-taint analysis used in our approach and the original string-taint analysis lies in that the propagation of annotations in our approach is able to distinguish all possible data sources of each terminal.

To illustrate the process of applying our generalized string-taint analysis, we next describe how our technique analyzes the code in Section 2. First, we transform the code to the SSA form as below. In the SSA form presented below, for simplicity we use "&FileInput" to represent the method call `c.getCountry().getName()`, whose return value comes from an input from supporting files through several assignments.

```
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.nextToken();
```

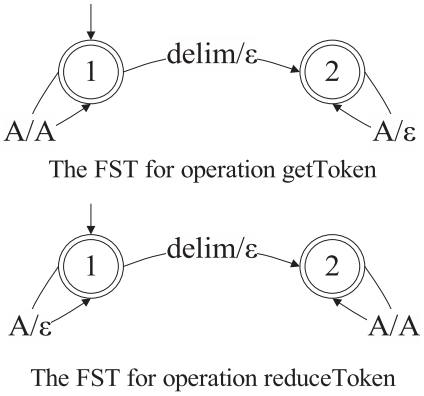



Fig. 1. FSTs for `StringTokenizer.nextToken()` (where A denotes any character except delim.

```
if(addr.equals("CARD")){
  name = StringT.nextToken();
  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 to the annotation of the terminal corresponding 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. For each nonterminal, we initially set its annotation as the empty set. We do not show empty sets as annotations in the following grammar for simplicity of presentation.

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

In the SSA code, there are two types of string operations for `StringTokenizer`: the constructor `StringTokenizer()` and `nextToken()`. `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 `nextToken()`. In the grammar, the constructor `StringTokenizer()` is treated as an ordinary string assignment, and `nextToken()` is replaced by two continuous operations: `getToken()` and `reduceToken()`. In the two operations, `getToken()` 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 FSTs[2] shown in Fig. 1.

2. In Fig. 1, the character before "/" is the input to the FST and the character after "/" is the output from the FST.

Then, we propagate the annotations from the terminals to the nonterminals according to our predefined propagation rule for the task. Specifically, the propagation process is presented in Algorithm 1. The inputs of the process include $C[N, T, S, P]$, which denotes the CFG with operations extracted from the code. $N$, $T$, $S$, and $P$ denote the nonterminal set, the terminal set, the start nonterminal, and the production set, respectively. The other input of the process is $Map$, which denotes a mapping between each terminal and its location in the code. The output of the process is $Res$, which denotes the set of code locations as the final annotation of $C.S$ (i.e., the start variable of $C$). In the process, Lines 1-6 depict the initialization of the annotations. $C.T$ denotes the terminal set of $C$, and $x.annot$ denotes the annotation of $x$ ($x$ can be a terminal or a nonterminal). The $get$ method of $Map$ returns the corresponding code location of a given terminal. Therefore, in the initialization phase, the annotations of each terminal are initialized as a singular set whose element is the corresponding code location of the terminal. The annotations of each nonterminal is initialized as an empty set. Line 7 in the process uses the string-operation-resolving technique in string-taint analysis to resolve all the string operations in $C$ and generate $C'$. The string-operation-resolving technique in string-taint analysis guarantees that the generated $C'$ is equivalent to $C$. Lines 8-14 depict the iterative propagation of annotations. $C'.N'$ and $C'.P'$ denote the nonterminal set and the production set of $C'$, respectively. Moreover, $p'.left$ and $p'.right$ denote the left-hand side nonterminal and the right-hand side of production $p'$, respectively. Thus, as shown in the algorithm, in one iteration, for each of all the productions, we merge the annotations of all the (non)terminal of the production to the annotation of its left-hand side nonterminal. The iteration ends when the annotations of all nonterminals become stable and do not change any more.

**Algorithm 1.** The process of propagating annotations
**Input:** $C[N, T, S, P]$: The CFG with operations extracted from the code. $Map$: The map between any terminal $t$ and its code location $loc$.
**Output:** $Res$: The set of code locations as the annotation of $C.S$

```
1: for each terminal t in C.T do
2:     t.annot = {Map.get(t)};
3: end for
4: for each nonterminal n in C.N do
5:     n.annot = {};
6: end for
7: Resolve operations in C[N, T, S, P]) and generate
   C'[N', T', S', P'];
8: repeat
9:     for each p' in C'.P' do
10:        for each (non)terminal v in p'.right do
11:            p'.left.annot = p'.left.annot ∪ v.annot;
12:        end for
13:    end for
14: until n'.annot does not change for all n' in C'.N'
15: res = C'.S'.annot;
16: return res;
```

In our example, for the first production in the CFG, the annotation of return1 after propagation is the union of the annotation of return1 before propagation (i.e., initially the empty set) and the annotation of "wildcard" (i.e., {RiskGame.java:6767}). For the last production in the CFG, the annotation of output after propagation is the union of the annotation of "You got a new card:" and the annotation of name. We do the propagation until all the annotations do not change any more. Then, we check the annotation of output, and find the locations of two constant strings "You got a new card:" and "wildcard" in its annotation. So, we mark these two constant strings as need-to-externalize.

### 3.4 String-Transmission Analysis

Using generalized 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 other side of the application, and use generalized 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 the Packet class to the client side.

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

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

```
15 ObjectInputStream in =
16   new ObjectInputStream(socket.getInput
  Stream());
17 Packet packet = (Packet)in.readObject();
18 switch(packet.getCommand()){
19   case Packet.CLOSECONNECTION:
20     disconnected(); break;
    ...
21 case Packet.CHAT:
22   Output(packet.getData()); break;
    ...
23 case Packet.ENDOFGAME:
24   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 is the value of data in Packet output to the GUI on the client side. In the preceding code portions, "Game saved to" (line 11), which is sent with Packet.CHAT, is passed to the GUI and thus needs externalization, while "Automatic Shuts Down" (line 10), which is sent with Packet.ENDOFGAME, does not need externalization. Thus, if we continue to trace data in Packet on the server side using generalized 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 member* of the data for transmission. In addition, there is another member variable (i.e., data in the preceding code) holding the data for transmission, which we refer to as *data member*. 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 member before using the data member, as the receiver needs to interpret the meaning of the data according to the value of the label member.

To make more precise analysis of transmitted strings, we propose string transmission analysis, which is presented as Algorithm 2. The inputs of Algorithm 2 are $C$, the code base, and $DS_o$, the data origins of Initial Output Strings acquired with generalized string-taint analysis. The output is $DS_n$, the data origins of Initial Output Strings at the other side of the network.

**Algorithm 2.** Procedure of string transmission analysis
**Input:** $C$: Project Code Base. $DS_o$: Data Origins of Initial Output Strings acquired with generalized string-taint analysis
**Output:** $DS_n$: Data Origins of Initial Output Strings at the other side of network

1: Locate all received variables $Set_r$;
2: $Ds_n = \{\}$;
3: **for** each $r$ in $Set_r$ **do**
4:  **if** $r$ relates to $DS_o$ **then**
5:  Get the data flows $dfs$ from $r$ to the GUI;

```
 6:        Generate Enabling Constraint cons from dfs;
 7:        Locate all sent variables Set_s;
 8:        for each s in Set_s do
 9:          if s is a String then
10:            Locate data origins ds_v of s;
11:            DS_n = DS_n ∪ ds_v;
12:          else if s is an Object with actual non-final
             members then
13:            Locate data origins ds_v of the data member
               of s;
14:            DS_n = DS_n ∪ ds_v;
15:          else
16:            Locate initializations inis of s;
17:            for each ini in inis does not violate cons do
18:              Locate data origins ds_v of the data
                 member initialization in ini;
19:              DS_n = DS_n ∪ ds_v;
20:            end for
21:          end if
22:        end for
23:      end if
24:  end for
25:  return DS_n;
```

In line 1 of Algorithm 2, we determine the objects that are transmitted through the network. To achieve this purpose, we locate in the source code all the API method invocations for socket output, and acquire all the variables that receive the return value of these invocations (e.g., the `packet` in line 19 of the above code). We mark such variables as *received variables*. Line 2 of Algorithm 2 just initializes the output set as an empty set.

For each received variable $r$ that is related to the known data origins of Initial Output Strings,[3] lines 5-22 locate all the data origins from the other side of the network that flows to Initial Output Strings via $r$.

Specifically, the locating process includes three phases.

The first phase is to build the enabling constraint of $r$ that is or whose member is a data origin of Initial Output Strings. This phase is presented in Lines 5-6 of Algorithm 2. In this phase, we first extract the data flow path from the received variable or its member variables to the Initial Output String. The member variables involved in this process are recorded as *data members*. Then we gather all the branch conditions that enable the data flow and contain the transmission variable or its member variables (e.g., the condition `packet.command==Packet.CHAT` combined by Lines 20 and 23). Finally, we generate an *enabling constraint* by combining the gathered branch conditions. We record the member variables involved in the constraint as *label members*.

The second phase (line 7 of Algorithm 2) is to locate all the API method invocations for socket input, and acquire all the variables (we ignore variables which are of primitive types) that are sent into the sockets (e.g., `packet` in line 16), which we refer to as *sent variables*.

The third phase (lines 8-22 of Algorithm 2) is to decide which sent variables will flow to the GUI through the network, and locate the data origins of these sent variables. Specifically, if a sent variable $s$ is of the string type, we simply apply generalized string-taint analysis on it (lines 10-11 of Algorithm 2). If $s$ is not of the string type, we check whether all its label members are declared as final variables or are actually final variables (i.e., no assignments to them except at the initialization). If they are, we check the initializations $inis$ of the *sent variable* to decide whether to apply generalized string-taint analysis on its data member based on the value of its label member and the enabling constraint (lines 16-20 of Algorithm 2). Otherwise, we simply apply generalized string-taint analysis on its data members (lines 13-14 of Algorithm 2).

It should be noted that we make conservative decisions in the above steps. For the string transmissions, we consider all the socket inputs/outputs and all possible data flows from the received variable to Initial Output Strings, while, for the enabling constraint, we add only confirmed branch conditions (i.e., inner-procedure guard conditions), and we decide the data member of a sent variable as not flowing to the GUI, only if its corresponding label member is final (so that it will not be changed) and its assigned value at the initialization is a constant that will violate the enabling constraint for sure.

### 3.5 String-Comparison Analysis

In Sections 3.3 and 3.4, our aim is to trace constant strings that may be visible on the GUI. However, more constant strings than those visible strings on the GUI need externalization for software internationalization. In the example presented in Section 2, `"wildcard"` in Line 24, which is a source of `name`, needs externalization. Since the constant string `"wildcard"` in line 13 is compared to `name`, this `"wildcard"` also needs externalization. Therefore, after we locate constant strings visible on the GUI, we need to further locate the strings that are compared with these visible 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 and `strcmp()` in C). Then for each side of each comparison, we perform generalized 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 visible 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-externalize constant strings. We iteratively add need-to-externalize constant strings until we cannot locate any more need-to-externalize constant 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 numbers), 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. Second,

3. Note that here "related to a data origin" means that a received variable (if it is of the string type) or its member variables (if it is an object with member variables) is a known data origin of Initial Output Strings.

TABLE 1
Basic Information of the Subjects

| Application | Description | Starting Month | #Developers |
|---|---|---|---|
| ArtofIllusion | 3D editor | 11/2000 | 2 |
| JFreeChart | chart drawing library | 11/2000 | 9 |
| Megamek | real-time strategy game | 02/2002 | 33 |
| Risk | board game | 05/2004 | 4 |
| Rtext | text editor | 11/2003 | 16 |
| StoryBook | writing support software | 01/2008 | 1 |
| TV-Browser | electronic TV guide | 04/2003 | 20 |

we filter out any constant string that is equal (in a case insensitive way) 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 Empirical Evaluation

### 4.1 Tool Implementation

To perform our empirical evaluation, we implemented an Eclipse plug-in called TranStrL for our approach [23].[4] We chose Java as the target language because Java is a widely used programming language among open source applications. For TranStrL, we collected *output API methods* from two packages: `java.awt.*` and `javax.swing.*`. So TranStrL currently supports analysis of Java applications that use only these two packages to implement their GUI.

### 4.2 Evaluation Setup

In our empirical evaluation, we used seven real-world open source applications as subjects: ArtOfIllusion, JFreeChart, Megamek, Risk, RText, StoryBook, and TV-Browser. All seven applications are accessible from the website of sourceforge,[5] and the versions used in our empirical evaluation are packaged and downloadable from our project website.[6] Among the seven applications, ArtOfIllusion is a 3D editor for designers to build and edit various 3D models and animations, JFreeChart is a library for drawing various charts from datasets, Megamek is a large real-time strategy game, Risk is a turn-based board game, RText is a programmer-oriented text editor, StoryBook is a writing support software that helps authors organize scenes, chronological lines, characters and so on, and TV-Browser is an electronic TV guide that provides information about future TV programs for a large number of TV channels. Table 1 depicts the basic information of these applications. In Table 1, the first column presents the name of the application, the second column presents the short description of the application, the third column presents the month in which the application is registered to sourceforge, and the fourth column presents the number of developers involved in the development of the application. Since sourceforge counts all the people who contributed to a project during its life cycle as developers, the presented number of developers may be larger than the number of developers who are active simultaneously at a certain time of the project's life cycle.

We chose these seven applications for two main reasons. First, the seven applications are among the most downloaded programs that meet the requirement of our evaluation (i.e., having versions before and after internationalization, and having GUIs built on `java.awt.*` and `javax.swing.*`) in their own domains. Second, the seven applications represent software in different categories and their GUI structures are different. ArtOfIllusion and RText are both editors and they have typical component-based GUIs (i.e., GUIs built with menus, buttons, and edit panels). As ArtOfIllusion is a 3D editor, it includes more operations on canvas and graphs. JFreeChart is a library with no standalone GUI, but it may generate GUI components in the GUIs of the applications that invoke JFreeChart APIs to draw charts. So we chose JFreeChart to explore the results of our approach on such GUI-related libraries. Megamek and Risk are two different kinds of games with more stylized components and complex GUI structures. StoryBook and TV-Browser are two content-presentation applications with rich GUI components to show complex structures of elements with structured text descriptions and pictures. We chose two games and two content-presentation applications because the GUIs of these applications are typically more complex than other types of applications and it would be interesting to see how our approach performs on different applications of these types.

The developers of all seven of these applications did not consider internationalization at the beginning, and they used many hard-coded constant strings in their native languages (i.e., English and German in the seven applications used in our empirical evaluation) in early versions of theses applications. For all seven applications, the developers began to internationalize them some time later, and during the internationalization, 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 the usefulness of our approach for real-world internationalization tasks, for all seven applications we applied our approach to their versions as they were right before the internationalization and compared the results achieved by our approach with the actual changes for the internationalization made by the developers in their versions right after the internationalization. For StoryBook and TV-Browser, where no source code package of early released versions is available, we applied our approach on the SVN versions submitted before internationalization and checked the change with the SVN versions submitted after the internationalization,[7] while, for other applications, we used the source code package of the released versions right before and after the internationalization.

In Table 2, we present the information of the software versions from right before internationalization and to which we applied our approach. For each software version, Columns 1-5 show the application name and the version number, the date when the version is released, the number of the lines of code (LOC) in the software version, the number of files in the software version, and the number of constant strings in the software version, respectively.[8] Note

4. TranStrL can be downloaded from http://sourceforge.net/projects/transtrl/files/TranStrL/.

5. http://sourceforge.net/, accessed on 20 June 2011.

6. http://sourceforge.net/projects/transtrl/files/Evaluation/.

7. Internationalization may take a number of code submissions to finish. We determined that the internationalization ended after a code submission if none of the 30 subsequent code submissions are related to the internationalization.

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

TABLE 2
Basic Information of the Software Versions to Which We Apply Our Approach

| Application /Version | Version Generation Month | LOC | Files | Constant Strings | Need-to-Externalize (Not Externalized in the Internationalized Version) |
|---|---|---|---|---|---|
| ArtOfIllusion 1.1 | 06/2002 | 71k | 258 | 2889 | 1221(816) |
| JFreeChart 0.9.11 | 08/2003 | 95k | 339 | 1212 | 130(7) |
| Megamek 0.29.72 | 02/2002 | 110k | 338 | 10464 | 1734(678) |
| Risk 1.0.7.5 | 05/2004 | 19k | 38 | 1510 | 509(55) |
| Rtext 0.8.6.9(Core Package) | 05/2004 | 17k | 55 | 1252 | 408(121) |
| StoryBook svn-30 | 01/2008 | 11k | 73 | 1185 | 202(18) |
| TV-Browser svn-27 | 04/2003 | 11k | 67 | 623 | 187(1) |

TABLE 3
Overall Results of Applying Our Tool on the Seven Applications

| Application | Need-to-Externalize (Not Externalized in the Internationalized Version) | Located | False Negatives | False Positives |
|---|---|---|---|---|
| ArtOfIllusion | 1221(816) | 1280 | 6 | 65 |
| JFreeChart | 130(7) | 130 | 1 | 1 |
| Megamek | 1734(678) | 1765 | 10 | 41 |
| Risk | 509(55) | 498 | 18 | 7 |
| RText | 408(121) | 445 | 0 | 37 |
| StoryBook | 202(18) | 223 | 1 | 22 |
| TV-Browser | 187(1) | 216 | 10 | 39 |

that for StoryBook and TV-Browser, the month of their internationalization is the same as their starting month (when they are registered to sourceforge). However, their sizes show that they were being developed for some time before their registration. So our approach is still applicable to them. Column 6 shows the number of the need-to-externalize constant strings, which serve as the golden solution in our empirical evaluation. We obtained our golden solution as follows: First, we deemed constant strings in the version before internationalization as need-to-externalize constant strings if the developers externalized them in the subsequent internationalized version. Second, since our approach did find a number of need-to-externalize constant strings that were not externalized in the subsequent internationalized version for each subject, we also deemed as need-to-externalize constant strings the constant strings that were located by TranStrL and manually verified by us to be need-to-externalize.

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-externalize. If not, we carefully analyzed the source code to determine whether the constant string is visible the GUI and whether the untranslated version of this constant string hinders users' comprehension of using the application. In principle, we adopted a conservative policy to avoid misclassifying constant strings that do not need translation as need-to-externalize. That is to say, we tried to avoid biasing our evaluation favorably to our approach.

## 4.3 Empirical Results

In our empirical evaluation, we are interested in answering the following four research questions:

- How effective is our approach when applied on real-world applications for locating need-to-externalize constant strings?
- What are the reasons for the false positives and false negatives?
- What is the performance of our approach?
- What is the impact of the different techniques on the result of our approach?

To answer these four research questions, we collect the results of applying our approach on the seven real-world open source applications and analyze the results. Specifically, we present and analyze our evaluation results to answer the first and second research questions in Section 4.3.1. We present and analyze our evaluation results to answer the third research question in Section 4.3.2, and we present and analyze our evaluation results to answer the fourth research question in Section 4.3.3.

### 4.3.1 Overall Results and Analysis

**Overall results.** We present the overall results of applying TranStrL to the seven subjects in Table 3. In this table, we refer to constant strings that need translation but are not located by TranStrL as false negatives, and constant strings that are located by TranStrL 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-externalize strings. In RText, our approach locates all the need-to-externalize strings, while in ArtOfIllusion, JFreeChart, Megamek, Risk, StoryBook, and TV-Browser, our approach locates 1,215 of 1,221, 129 of 130, 1,724 of 1,734, 491 of 509, 201 of 202, and 177 of 187 need-to-externalize constant strings, respectively.

Second, for each subject, our approach does produce a few false positives. In ArtOfIllusion, JFreeChart, Megamek, Risk, StoryBook, and TV-Browser, the numbers of strings that are located by our approach but do not need translation is 65, 1, 41, 7, 37, 22, and 39, respectively. Compared to the numbers of need-to-externalize strings in the seven subjects, the numbers of false positives are quite small.

TABLE 4
The Distribution of False Negatives

| Application | All | First category (library related) | Second category (folder path) | Third category (other tasks) | Fourth category (debugging messages) | Fifth category (project name) |
|---|---|---|---|---|---|---|
| ArtOfIllusion | 6 | 6 | 0 | 0 | 0 | 0 |
| JFreeChart | 1 | 1 | 0 | 0 | 0 | 0 |
| Megamek | 10 | 10 | 0 | 0 | 0 | 0 |
| Risk | 18 | 3 | 10 | 0 | 5 | 0 |
| RText | 0 | 0 | 0 | 0 | 0 | 0 |
| StoryBook | 1 | 1 | 0 | 0 | 0 | 0 |
| TV-Browser | 10 | 3 | 0 | 6 | 0 | 1 |

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 which were verified by us as need-to-externalize. 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 1,696 such strings in the seven subjects. We used version tracking and code difference tools to trace the changes on these strings in the versions later than the internationalization, and we find that, among the 1,696 strings, 1,429 (746 in ArtOfIllusion, 579 in Megamek, 10 in Risk, 87 in RText, 6 in StoryBook, and 1 in TV-Browser) were externalized and translated in a later version and 267 strings 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 shown in the following piece of code:

```
public MechView(Entity entity) {
    ...
    StringBuffer sBasic;
    sBasic.append( Messages.getString
    ("MechView.Movement") )
    ...
    sBasic.append(entity.getMovementTypeAs
    String())}
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 before our study, 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 the second part remaining in English appears on the GUI of the German version of the game. We reported all 17 unexternalized need-to-externalize strings located by TranStrL to the Megamek developers as bug report #2085049 and all 17 of these strings were confirmed and fixed by the developers. We next further discuss the reasons for the false negatives (i.e., need-to-externalize constant strings not located by TranStrL) and the false positives (i.e., located need-to-externalize strings that actually do not need translation).

**Analysis of false negatives.** Generally, the false negatives fall into five categories, and Table 4 presents the distribution of the false negatives in different subjects.

The first category includes constant strings that are output to GUI through API invocations in the library code or that are compared with visible strings in the library code. This category includes all 6 false negatives from ArtOfIllusion, the 1 false negative from JFreechart, all 10 false negatives from Megamek, 3 of the 18 false negatives from Risk, the 1 false negative from StoryBook, and 3 of the 10 false negatives from TV-Browser. In principle, generalized string-taint analysis and string-comparison analysis should be able to locate strings in this category. The reason that our tool failed to do so in our empirical evaluation is as follows: This category involves some string assignments, Output API invocations, and string comparisons implemented in library code. Our tool cannot trace into libraries whose source code is unavailable, but after extending our implementation to analyze library code, we should be able to address this category of false negatives.

The second category includes constant strings related to the names of language-related file folders (e.g., maps and cards). Specifically, 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

TABLE 5
The Distribution of False Positives

| Application | All | First category (intentionally left) | Second category (untranslatable strings) | Third category (HTML tags) | Fourth category (debugging messages) |
|---|---|---|---|---|---|
| ArtOfIllusion | 65 | 4 | 61 | 0 | 0 |
| JFreeChart | 1 | 1 | 0 | 0 | 0 |
| Megamek | 41 | 6 | 35 | 0 | 0 |
| Risk | 7 | 3 | 4 | 0 | 0 |
| RText | 37 | 18 | 14 | 3 | 2 |
| StoryBook | 22 | 20 | 0 | 2 | 0 |
| TV-Browser | 39 | 4 | 35 | 0 | 0 |

languages. Thus, when switching from one language to another, the names of map folders should also be switched.

The third category includes constant strings that did not go to the GUI in the version before the internationalization, but are sent to the GUI by the developers with some code changes on the version after the internationalization. In TV-Browser, 6 of the 10 false negatives belong to this category. The reason for this category of false negatives is that the developers perform some other development tasks along with the internationalization. Since it is impossible to predict what other tasks the developers will perform along with the internationalization, our approach cannot detect this category of false negatives. However, we believe that the developers should know that they are performing these tasks along with internationalization, so they should keep these affected strings in mind and so that they may not miss these strings during the internationalization. Furthermore, if the developers apply our approach on their application again after they finish the tasks, they can still find these constant strings.

The fourth category includes debugging messages visible 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, the developers of Risk externalized 5 debugging messages, which we counted as 5 false negatives.

The fifth category includes only 1 false negative from TV-Browser, being the application name "TV Browser," which appears as the label of the main menu of the application. Since we remove application names in our filtering process, we miss this constant string. However, since most developers choose not to translate application names, our filtering is still reasonable. Actually, the constant string "TV Browser" that appears as the title of the main window of the same application remains unexternalized.

**Analysis of false positives.** Generally, the false positives fall into four categories, and Table 5 presents the distribution of the false negatives in different subjects.

The first category of false positives consists of strings that are visible 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 located by our approach but not externalized by the developers, we counted these strings as false positives. In total, 4 of 65 false positives in ArtOfIllusion, the 1 false positive in JFreeChart, 6 of 41 false positives in Megamek, 3 of the 7 false positives in Risk, 18 of 37 false positives in RText, 20 of 22 false positives in StoryBook, and 4 of 39 false positives in TV-Browser belong to this category.

The second category of false positives consists of strings that are visible 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. Other examples include the names of fonts (e.g., Times New Roman) and the TV channel names in TV-Browser. 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, 61 of 65 false positives in ArtOfIllusion, 35 of 41 false positives in Megamek, 4 of 7 false positives in Risk, 14 of 37 false positives in RText, and 35 of 39 false positives in TV-Browser belong to this category.

The third category includes 3 of 37 false positives in RText and both of the 2 false positives in StoryBook. 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 seven subjects, our approach is able to locate most of the need-to-externalize strings while producing only a small number of false negatives and a small number of false positives. Among the false negatives, the first category may result in untranslated strings on the GUI but can be addressed by extending our tool implementation to analyze libraries. The second category can be easily detected by analyzing the file system. The third category can be addressed by applying our approach to the application again after the outputting of constant strings to the GUI is implemented. The fourth and fifth categories are 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

TABLE 6
Execution Time and Memory Consumption of Our Approach

| Application | Execution Time (s) | | | Maximum Memory Consumption (MB) |
|---|---|---|---|---|
| | Whole Process | Taint | Comparison | |
| ArtOfIllusion | 138.57 | 98.71 | 37.79 | 306 |
| JFreeChart | 99.28 | 63.43 | 34.81 | 116 |
| Megamek | 740.86 | 303.79 | 433.79 | 664 |
| Risk | 355.68 | 104.69 | 250.39 | 155 |
| RText | 208.82 | 84.13 | 124.11 | 223 |
| StoryBook | 75.22 | 58.76 | 16.13 | 117 |
| TV-Browser | 75.79 | 56.61 | 16.46 | 155 |

positives are project-specific and can be easily detected by users' inspection or user-defined heuristics. The fourth category is trivial for users to detect. Furthermore, for each subject, our approach is able to locate some need-to-externalize strings that the developers did not locate when internationalizing the subject.

The preceding results demonstrate that our approach is useful in at least the following two scenarios: First, developers can use our approach to generate candidates for externalization 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-externalize constant strings.

#### 4.3.2 Execution Time and Memory Consumption

To study the performance of our approach, we record execution time and memory consumption of our approach when applied to the seven subjects in our empirical evaluation, and present the data in Table 6. In Table 6, the first column presents the name of the subject. The second, third, and fourth columns present the execution time of the whole approach, the execution time of the generalized string-taint analysis (including the string transmission analysis), and the execution time of the string comparison analysis, respectively. The fifth column presents the maximal memory usage during the execution. Note that we did not record the execution time of the string transmission analysis solely. This is because string transmission analysis is not an independent step, but is invoked whenever generalized string-taint analysis locates strings read from the network. Furthermore, after string transmission analysis locates the string data source on the server side, generalized string-taint analysis is invoked again to trace string data flows on the server side. Therefore, it is not easy to exactly record execution time of merely string transmission analysis. It should also be noted that there are the class loading process and the filtering process, so the execution time of the whole approach is slightly larger than the sum of the execution time of the generalized string-taint analysis and the string comparison analysis.

From Table 6, we have the following two observations: First, the time and memory usage of our approach is reasonable on all the subjects in our empirical evaluation. For the largest and most complex project (i.e., Megamek), the execution time of our approach is less than 13 minutes, and the memory consumption is less than 700 MB. Second, the execution time and memory consumption may not positively correlate with the size of the application. For example, JFreeChart is much larger than RText, but our approach costs less time and memory on JFreeChart. The reason is that our approach focuses on only the string variables, constants, and operations related to the GUI. Therefore, the main factor that affects the performance of our analyses is not the size of the whole application that we apply our approach on, but the number of GUI-related string variables, constants, and operations. Furthermore, the most time-consuming part in the generalized string-taint analysis is the handling of string operations (i.e., using FSTs to approximate string operations in the CFG and propagate taints through FSTs). So, for an application whose GUI-related string data flows involve very complex string operations, our approach will spend more time on it. As an instance, Risk takes more time than ArtOfIllusion although it has fewer need-to-translate constant strings.

#### 4.3.3 Impacts of Different Techniques

In our approach, the basic tracing technique is generalized 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 impacts of the three techniques in our approach, we performed a series of experiments. The baseline was to use all of the three techniques with generalized string-taint analysis, and we turned off each technique at a time to observe how each specific technique impacts the results.

**Impacts of string-transmission analysis.** We show the results of turning string-transmission analysis on and off in Table 7. Since only Megamek transmits strings across the network, turning string-transmission analysis on or off impacts 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 generalized string-taint analysis to analyze all string variables whose values are transmitted across the network without considering the label variable in transmitted objects.

TABLE 7
Turning String-Transmission Analysis On and Off

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

TABLE 8
Turning String-Comparison Analysis On and Off

| Application | Need-to-Externalize | Located | FN | FP |
|---|---|---|---|---|
| Risk | 509 | 498 | 18 | 7 |
| Risk(NC) | 509 | 474 | 42 | 7 |
| Megamek | 1734 | 1765 | 10 | 41 |
| Megamek(NC) | 1734 | 1730 | 36 | 32 |

In Table 7, the line marked with "(NT)" presents the results of our approach with turning string-transmission analysis off in the first way, while the line marked with "(ALL)" presents the results of our approach with turning string-transmission analysis off 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-externalize strings (i.e., reduce 575 false negatives) in Megamek, introducing only 2 false positives (falling into the second category of false positives discussed in Section 4.3.1). Compared to the second way of turning off string-transmission analysis, string-transmission analysis helps reduce 12 false positives.

**Impacts of string-comparison analysis.** We show the results of turning string-comparison analysis on and off in Table 8, in which the lines marked with "(NC)" present the results of turning string-comparison analysis off. Since only Megamek and Risk contains string comparisons that involve need-to-externalize constant strings, turning string-transmission analysis on or off impacts the result of only these two subjects.

First, string-comparison analysis is helpful to find more need-to-externalize strings (i.e., reduce false negatives) in the two 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 visible on the GUI but cannot be translated.

**Impacts of filtering.** We show the results of turning filtering on and off in Table 9, in which the lines marked with "(NF)" present the results of turning filtering on and off.

From Table 9, we observe that, for most of the subjects, filtering can effectively reduce the numbers of false positives. Furthermore, since we use conservative heuristics in filtering, there is only one false negative caused by filtering among all seven subjects. The reason for the false negative has been discussed in Section 4.3.1. Actually, if we use some aggressive heuristics, we may further reduce the number of false positives, but the number of false negatives may also increase.

## 4.4 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-externalize strings for each subject. First, it may be error-prone to verify constant strings as need-to-externalize in versions later than the subsequent internationalized version because the later versions involve various modifications for other purposes.

TABLE 9
Turning the Filter On and Off

| Application | Need-to-Externalize | Located | FN | FP |
|---|---|---|---|---|
| ArtOfIllusion | 1221 | 1280 | 6 | 65 |
| ArtOfIllusion(NF) | 1221 | 1487 | 6 | 272 |
| JFreeChart | 130 | 130 | 1 | 1 |
| JFreeChart(NF) | 130 | 130 | 1 | 1 |
| Megamek | 1734 | 1765 | 10 | 41 |
| Megamek(NF) | 1734 | 2080 | 10 | 356 |
| Risk | 509 | 498 | 18 | 7 |
| Risk(NF) | 509 | 532 | 18 | 41 |
| RText | 408 | 445 | 0 | 37 |
| RText(NF) | 408 | 581 | 0 | 173 |
| StoryBook | 202 | 223 | 1 | 22 |
| StoryBook(NF) | 202 | 262 | 1 | 61 |
| TV-Browser | 187 | 216 | 10 | 39 |
| TV-Browser(NF) | 187 | 223 | 9 | 45 |

Second, manually verifying constant strings not externalized in any later version as need-to-externalize may be prone to accidental mistakes or personal perspectives to the notion of being "need-to-externalize." 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 due to the conservative nature of 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 evaluation so that the quality of manual string externalization would 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 numbers of false positives and false negatives in our evaluation.

The main threats to external validity are as follows: First, the results of our evaluation 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 seven subjects used in our empirical evaluation are all open source applications written in Java, and all of them are of moderate sizes. Therefore, the findings of our empirical evaluation may be specific to open source applications in Java with moderate sizes, and may not be generalized to other applications. Third, we evaluated the impacts of string-transmission analysis and string-comparison analysis only on a few subjects among the seven subjects. Therefore, the findings on string-transmission analysis and string-comparison analysis in our empirical evaluation may not be generalized to other applications. To further reduce these threats, we may need to apply our approach to more applications, especially those for commercial use, with larger code bases, having strings transmitted across the network, and/or having string comparisons involving need-to-externalize constant strings.

## 5 Extension for Visualization

When applying our approach to software internationalization, software developers may encounter the following two issues: First, as our approach does concede some false positives, developers may not be confident enough on whether a particular constant string needs externalization. They may need more information about how the constant string goes to the GUI to make sure that the path is feasible at runtime. Second, when translating a particular externalized constant string, developers may need to know the string's concatenation pattern with other strings. For example, in the following code portion, the pattern of the output to the GUI should be "You get [number] [jobs done]" in which "[number]" is for any number, and "[jobs done]" is for some done jobs such as "car(s) fixed." Without the context, it is difficult to correctly translate the phrase "You get" because the word "get" has broad meanings. In the code portion, the word "get" is a sign for the perfect tense, while in the sentence "You get 3 cars," the word "get" means "acquire." In many languages such as Chinese, a single word cannot represent these different meanings, so developers may need to check concatenation patterns when translating located need-to-externalize constant strings.

```
JTextArea ta = new JTextArea();
int number = getNumber();
String out = "You get " + number + getTask();
ta.append(out);
...
private String getTask()
...
Switch(this.currentTask)
...
case Task.CarFix: return "car(s) fixed.";
...
```

To help developers tackle these two issues, we implemented a tool named TransVis as an extension of our TranStrL tool to visualize the context of a given constant strings located by TranStrL.[9] In particular, TransVis provides two kinds of visualization support.

Given a constant string (denoted as *Str*) located by TranStrL, the first kind of visualization support in TransVis is to visualize each String Flow Graph [1] that *Str* may be involved in. The nodes in a String Flow Graph represent all the string elements that may flows to a certain hot spot. Note that a hot spot is an *initial output string* in our problem. Here, a string element can be a string constant, a string variable, or a string expression with a string operation (such as concatenation). The edges in a String Flow Graph represent all the flows among these string elements. Given a node (denoted as $A$) in a String Flow Graph, there are two ways of strings from other nodes (e.g., $B$ and $C$) flowing into $A$. In the first way, $A$ can be assigned with the value of either $B$ or $C$. We denote this way as a solid triangle in the String Flow Graph. In the second way, $A$ is assigned with the result of a string operation (e.g., concatenation) over $B$ and $C$. We denote this way as a solid rhombus in the String Flow Graph. Note that, for any node in the String Flow Graph, there is only one way for the strings from other nodes to flow into the node. Supposing *Str* may go to only one *initial output string* (denoted as *Output*), TransVis is able to calculate and visualize the String Flow Graph (denoted as $G$) of *Output*. From $G$, developers may know visually how *Str* goes to *Output* and how *Str* is concatenated with other strings.

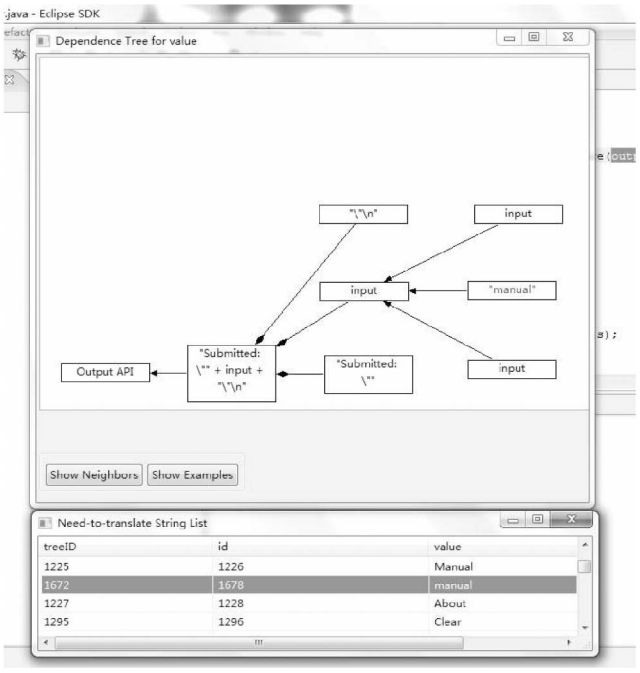



Fig. 2. A string flow graph for the need-to-externalize constant string "manual."

Fig. 2 depicts a simple example of a String Flow Graph. In visualization of the String Flow Graph, we use a square with a string label to denote a node in the string flow graph. The string label of the squire presents the string form of the node's corresponding string element in the code. Furthermore, we use an edge with an arrow to denote an edge in the String Flow Graph. The direction of the arrow indicates the direction of the data flow, and the shape of the arrowhead indicates the type of the data flow. Specifically, we use a triangle arrowhead to indicate an assignment and we use a diamond arrowhead to indicate the participation in a string operation.

However, in some cases, a String Flow Graph may be very complex and contain a large number of nodes. Fig. 3 shows the String Flow Graph for a located need-to-externalize constant string (i.e., "You have") in the Risk application. The graph contains more than 800 nodes. In such a case, visualizing the whole String Flow Graph can hardly provide substantial help for the developers due to the complexity of the String Flow Graph. Therefore, TransVis provides the second kind of visualization support to visualize the String Example Graph, which is a subgraph of the String Flow Graph.

Given a constant string (denoted as *Str*) that may flow to an *initial output string* (denoted as *Output*). We define the example value set (denoted as *EVS*) of *Str* and *Output* as all possible values of *Output* that contain *Str* (i.e., each value in *EVS* has *Str* as one of its data origins). Each element in *EVS* is an example value (denoted as *EV*) of *Str* and *Output*. A

9. TransVis is downloadable from http://sourceforge.net/projects/transtrl/files/TransVis/.

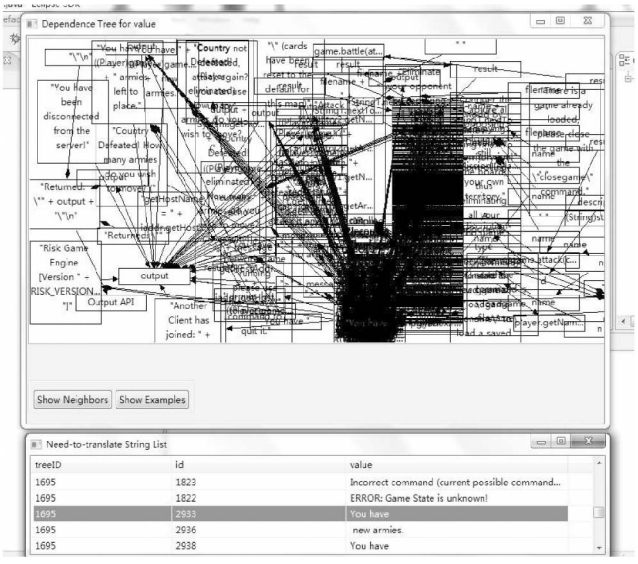



Fig. 3. A complex string flow graph.

String Example Graph (denoted as *EG*) of *EV* is the subgraph of the String Flow Graph that is related to the generation of *EV*. Thus, *EG* demonstrates how *Str* is concatenated with other strings to form a value of *Output*. Actually, a String Flow Graph can be viewed as the visual presentation of the CFG with string operations generated by the generalized string-taint analysis for *Output*, while a String Example Graph can be viewed as the visual presentation of a deduction tree of the CFG with string operations, making sure that the deduction has *Str* as one of its leaves. The rationale behind String Example Graph is that the concatenation pattern of one constant string should typically be consistent in different examples. Thus, String Example Graphs can still provide a large part of the context information *Str*, compared to String Flow Graphs that provide the whole context information.

Given the String Flow Graph (i.e., *G*) of *Output*, TransVis calculates one String Example Graph (i.e., *EG*) of *Str*, *Output* in the following way: First, TransVis calculates the path from *Str* to *Output* in *G*. Second, for each node in the path, TransVis further traces the origins of the node in *G*. For any node (denoted as *A*) during this tracing, if the strings from other nodes flow into *A* in the first way (i.e., through a flow with a solid triangle), we randomly trace one node if none of the nodes have been traced, and trace none of them if one node has already been traced. If the strings from other nodes flow into *A* in the second way (i.e., through a flow with a solid rhombus), we further trace each of these nodes. The String Example graph generated by TransVis for the need-to-externalize constant string "You have" is shown in Fig. 4. From the figure, we can observe that the String Example Graph is much simpler than its corresponding String Flow Graph, and much easier for the developer to explore. Furthermore, for the convenience of the developers, we add quick links from each node in the String Example Graph to its corresponding code elements in the code (e.g., the corresponding code of the left node "output" in Fig. 4 is highlighted after a click on it).

To further study the effectiveness of building String Example Graphs, for each subject in our empirical study, we acquire a String Flow Graph and a String Example Graph for each located need-to-externalize string. Therefore, for each subject, we have two sets of graphs. Then, for each graph set, we derive three measures: the maximal number of nodes in a single graph in the graph set, the average number of nodes in the graph set, and the number of large graphs (with more than 50 nodes) in the graph set. Finally, we compare these measures to observe whether String Example Graphs are effective in reducing the sizes of String Flow Graphs for better usability. We present the results in Table 10.

From Table 10, we can observe that for many need-to-externalize strings (1 in JFreeChart, 17 in RText, 43 in ArtofIllusion, 209 in Risk, more than 1,000 in Megamek), String Flow Graphs can be quite large (with more than 50 nodes) and may not be easy for developers to understand. By contrast, among all the String Example Graphs for the need-to-externalize strings in all the subjects in our empirical evaluation, there are only 7 String Example Graphs in Megamek that have more than 50 nodes. Therefore, for a significant number of need-to-externalize constant strings that correspond to large String Flow Graphs, String Example Graphs provide a good way for developers to understand about how the externalized string flows to the GUI and how it is concatenated with other strings.

## 6 Discussion

In this section, we discuss several important issues that are related to our research.

### 6.1 Limitations of Our Approach

The basis of our approach is string-taint analysis, which is based on data-flow analysis. Since string-taint analysis is not path-sensitive (assuming all paths are feasible), a data source *D* of an *initial output string S* determined by string-taint analysis may not be a real data source of *S* as the data flow from *D* to *S* may be based on an infeasible path. Furthermore, for a string operation whose inputs cannot be determined statically, we can build only an approximate FST for it that generates an overestimation of the output of the string operation. In both of the above cases, string-taint analysis may have to concede with inaccuracy, which then may result in some false positives.

One possible way to further reduce this kind of false positives is to use dynamic analysis [2]. The main disadvantage of using dynamic analysis for locating need-to-externalize constant strings is that dynamic analysis requires a set of test data to cover possible usages of the software application under analysis. In fact, developers seldom use variables of other types to determine whether a particular value of an *initial output string* is output to the GUI. Thus, this disadvantage of string-taint analysis may not result in many false positives in practice.

Another limitation is about transmitted strings, where 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. Currently, our string-transmission analysis is able to deal with the situation of string transmission via objects through sockets. However, there

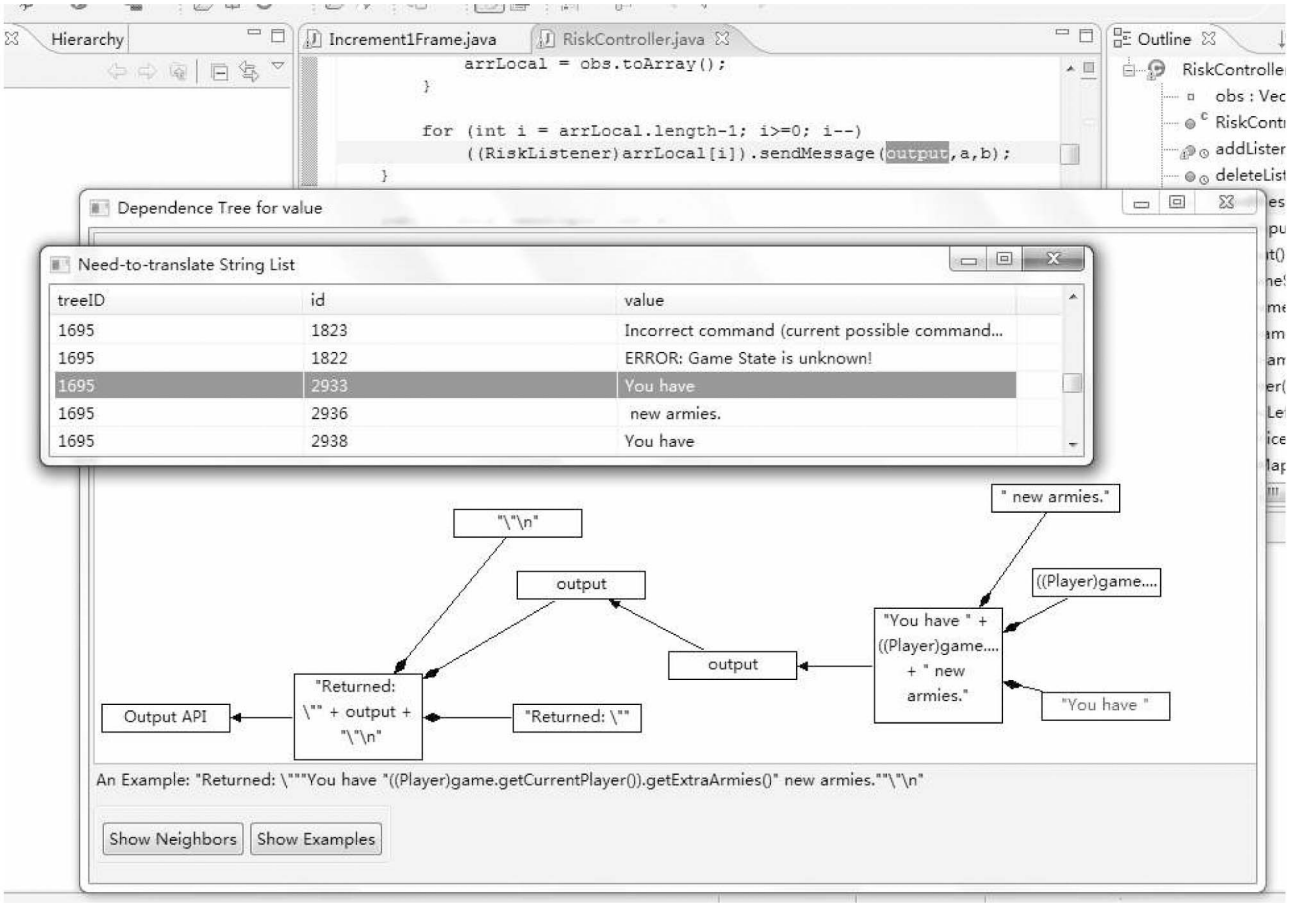



Fig. 4. A string example graph for the need-to-externalize constant string "You have."

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 may require more specific techniques.

### 6.2 String-Taint Analysis versus Data-Flow Analysis

In our approach, we leverage string-taint analysis to trace the flows of string data from string constants to output API methods. As indicated by the example in Section 2, compared to the basic data-flow analysis, string-taint analysis is able to precisely handle data-flows going through string operations and thus reduce some false positives (e.g., the "CARD" in the example in Section 2). In the meantime, string-taint analysis is also more expensive because it requires the computation of possible values of strings and FST simulation when handling string operations. According to the performance results in Section 4.3.2, the cost of string-taint analysis is acceptable for many real-world software systems. However, it is still an open problem whether string-taint analysis is more cost-effective than the basic data-flow analysis for the target problem. To investigate this problem, we need to implement an approach based on the basic data-flow analysis and quantitatively compare the cost and effectiveness of these two analyses for the target problem. Furthermore, it is also unclear whether it is feasible to add some lightweight string-operation-handling mechanisms to the basic data-flow analysis to achieve both high precision and high efficiency for the target problem.

TABLE 10
The Size Reduction from String Dependence Graphs to String Example Graphs

| Application | String Dependence Graph | | | String Example Graph | | |
|---|---|---|---|---|---|---|
| | Max | Avg | #Large | Max | Avg | #Large |
| ArtOfIllusion | 201 | 12 | 43 | 21 | 2 | 0 |
| JFreeChart | 73 | 2 | 1 | 8 | 2 | 0 |
| Megamek | 9576 | 3691 | 1080 | 78 | 6 | 7 |
| Risk | 1249 | 369 | 209 | 48 | 4 | 0 |
| RText | 80 | 10 | 17 | 16 | 3 | 0 |
| StoryBook | 35 | 7 | 0 | 13 | 2 | 0 |
| TV-Browser | 31 | 7 | 0 | 14 | 2 | 0 |

### 6.3 String Splitting

One outstanding issue related to locating need-to-externalize constant strings is string splitting. This issue comes from the situation when a need-to-externalize constant string is also involved in some internal processing logic in the software. In such a situation, developers need to refactor the code before actual translation of the constant string. An illustration of typical string splitting cases are as below.

```
String item = "Header";
String header = database.fetch(item);
panel.append(item+":"+header);
```

The preceding code portion tries to create a panel with the label of "Header:xxx," in which "xxx" is the actual header retrieved from the database. The developer uses the constant string "Header" as both part of the panel label and a key to fetch the actual header. Thus, if we translate "Header" to another language, the statement `String header = database.fetch(item);` may not execute properly.

In our empirical evaluation, string splitting cases are not common, and for most of the need-to-externalize constant strings, the developers simply replace the constant string with a method call that read a string from supporting files. But we do find string splitting cases. For example, the 17 strings that are reported to and verified by the developers of Megamek are of string splitting cases. That is to say, these 17 strings are used elsewhere besides being presented on the GUI.

To address string splitting, we may need additional analysis to locate those need-to-split constant strings. Furthermore, it might also be possible to automate code refactoring related to need-to-split constant strings for software internationalization.

### 6.4 Soundness of Our Approach

The factors that will affect the soundness of our approach are as follows.

First, for the part of generalized string-taint analysis, we use the same underlying technique to propagate annotations through FSTs as string-taint analysis, which is proven to be sound [25]. Furthermore, our propagation rule on productions is to update the annotation of the left-hand-side nonterminal as the union of the annotations of all nonterminals in the production, which is conservative. Therefore, given the set of output API methods, this part of our approach is sound, and is able to locate all the constant strings and external inputs (user inputs, file inputs, network inputs, etc.) that will flow to the GUI.

Second, string-transmission analysis can detect all the network transmissions through explicit socket function invocations. Our approach assumes that only objects are transmitted through the network, so we may miss some transmitted values of primitive type. When deciding whether a transmitted string flows to the GUI, we conservatively exclude only the strings that are impossible to flow to the GUI. However, if network transmission techniques other than sockets are used in the application or the transmission through sockets are implicit[10] (as we discussed in Section 6.1), our approach will not be able to detect such transmissions and the soundness of the string transmission analysis part in our approach will be compromised.

Third, for the part of string-comparison analysis, our approach also leverages a conservative strategy that locates all the string comparisons in the application and reports a constant string to be need-to-externalize whenever it is compared with a known need-to-externalize string. Therefore, if our string-taint analysis part and string-transmission analysis part both generate sound results, our string-comparison analysis is also sound and is able to locate all the constant strings that are compared with need-to-externalize strings.

Fourth, the soundness of our approach is related to our basic assumption that only constant strings flowing to GUI are need-to-externalize strings. From our empirical evaluation, we discover that this assumption is generally correct, but is not always sound. For example, in the Risk project, we find that the filenames of those language-specific maps are also need-to-externalize.

Finally, our filtering strategies also affects the soundness of our approach. Although our approach does not use aggressive filtering strategies, it is still difficult to ensure that these strategies cannot filter out some need-to-externalize constant strings.

To sum up, the three analysis parts of our approach are sound if all the network transmissions are implemented with sockets. Our basic assumption is not always sound due to some project-specific issues, but the developers may add new project-specific output API methods to make it sound to their project. The soundness of filtering strategy is also project-specific. Actually, developers can define project-specific filtering strategies or even remove the filtering step to ensure soundness.

Furthermore, due to the existence of string-splitting cases, it may not always be safe to directly replace a need-to-externalize constant string located by our approach with a method call that reads the string from support files. To detect string-splitting cases, it may be required to further analyze destinations (e.g., database queries) other than the GUI. If a constant string flows to both the GUI and some other destination, there may be a string-splitting case. Due to the rarity of string-splitting cases and expensiveness of analyzing all the destinations besides the GUI, we do not do systematic investigate string-splitting in this paper. Note that the false positives reported in our empirical evaluation are not related to string-splitting and may be easy for developers to identify in practice.

### 6.5 Dealing with Other Programming Languages

Although our approach currently focuses on Java, it should be principally feasible for us to extend our approach to other programming languages, such as C#, C++, and C. In those languages, it should be also be feasible to locate need-to-externalize constant strings through tracing back from output arguments. However, some adjustments may be required. For example, in the C language, strings are represented as arrays of characters, and more advanced techniques to handle arrays may be needed. Furthermore, as different languages have different library methods to handle string operations, different techniques dealing with string operations may also be needed.

## 7 Related Work

In this section, we introduce research efforts that are related to our work. These research efforts fall into the following three categories: support for software internationalization, research on string analysis, and dependence-based code-element localization.

### 7.1 Support for Software Internationalization

Our work is an extension of the first reported effort directly focusing on automatically locating need-to-externalize

10. Implicit transmissions through sockets are transmissions through socket-based techniques and libraries such as jRMI.

constant strings [22]. In this extension, we formulate our adaptation of string-taint analysis as generalized string-taint analysis, add new experimental subjects, and introduce the visualization support of the approach. Based on our initial work on software internationalization, we also proposed a tool demo [23] which mainly focus on the structure and implementation of the TranStrL tool, while in this paper, we focus on the technical extensions on the approach, the empirical study and the visualization. Another recent work on locating need-to-externalize constant strings in software internationalization [24] tries to tackle the special difficulty of locating need-to-externalize constant strings in web applications, which is based on the technique in this paper, but its technical contribution is about determining the appearance order of the constant strings in the HTML texts, which is totally different from this paper.

There have been a couple of books published on how to internationalize a software application [5], [19]. These books provide some guidelines on how to find need-to-externalize 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-externalize strings [11], [4]. However, none of them provides any automatic approach to locating need-to-externalize strings.

There exist tools (e.g., GNU gettext,[11] Java internationalization API[12]) to help developers externalize need-to-externalize constant strings after developers locate them. Other tools such as KBabel[13] help developers edit and manage resource files (called PO files in KBabel) containing 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 4 show that in the real-world software applications studied, less than half of the constant strings need translation. Thus, it will be a waste of time for translators to translate all the constant strings. 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.

## 7.2 Research on String Analysis

There are extensive research efforts on both the techniques and the applications of string analysis.

### 7.2.1 Techniques

String analysis is a recent advance in static data-flow analysis [13]. Christensen et al. [1] first proposed string analysis, which is an approach for obtaining possible values of a string variable. Minamide [15] suggested to simulate string operations in an extended CFG with FSTs, and implemented a string analyzer for PHP code to check contents of dynamically generated web pages. Based on Minamide's work, Wassermann and Su developed string-taint analysis [25] to determine whether the value a string variable may come from insecure source. Most recently, Tateishi et al. [18] further developed path-sensitive and index-sensitive string analysis based on monadic second-order logic to more precisely predict the possible values of string variables. Veanes et al. [20] proposed symbolic finite-state transducers and related algorithms which may provide the basis for further enhancement on string analysis. Compared to these efforts, the work in our paper is most related to the string-taint analysis. Specifically, we extend string-taint analysis conceptually to allow non-Boolean annotations so that we can determine the locations of all the data sources of a string variable.

### 7.2.2 Applications

The existing applications of string analysis mainly fall into two categories: software security enhancement and understanding dynamically generated strings.

In the category of software security enhancement, there have been a number of efforts trying to detect SQL-injection vulnerabilities [10], [29], [25], [14], and cross-site scripting vulnerabilities [26], [14]. More recently, Yu et al. used string analysis to summarize the signatures of possible vulnerabilities [30] and provide patches to fix them [31]. Hooimeijer et al. [12] developed BEK, which is able to check whether a given santinizer can effectively block illegal string values.

In the category of understanding dynamically generated strings, Gould et al. [8] used string analysis to check the correctness of dynamically generated SQL query strings. Geay et al. [7] proposed to use string analysis and slicing to precisely acquire access-control permissions of software components. Their approach also used annotations to label the program locations related to the generation of string variables. However, their approach annotated both string variable/constants and string operations and did not use the taint-propagation technique of string-taint analysis to propagate annotations through FSTs.

Compared to these research efforts on the applications of string analysis, we apply string-taint analysis on a new problem (i.e., locating need-to-externalize constant strings) and we further develop techniques to cope with practical complications in the problem.

## 7.3 Dependence-Based Code-Element Localization

Our approach can also be viewed as determining a subset of constant strings that have certain data dependencies with the GUI. From this perspective, our approach is also related to the research efforts on dependence-based code-element localization. Program slicing techniques try to locate a number of code elements that have data or control dependencies with a certain variable [27]. O'Callahan and Jackson [17] proposed a technique called abstract type determination to decide the semantic role of a variable in the code from its dependence on other variables. Guo et al. [9] further improved the approach to the same problem using dynamic data-flow analysis. More recently, Gabel et al. [6] proposed dependence-based code clone detecting techniques which can locate code element groups with similar dependence structure. Wang et al. [21] proposed the

11. http://www.gnu.org/software/gettext/manual/gettext.html.
12. http://java.sun.com/docs/books/tutorial/i18n/index.html.
13. http://kbabel.kde.org/.

Dependence Query Language to describe dependence-related constraints and proposed an approach to locate all the code elements that satisfy given constraint. Our approach differs from all these preceding approaches in two main aspects. First, our approach targets at a specific task in software development and deals with only dependencies related to the task, while all these approaches are general-purpose approaches and may be applicable for different tasks in software development. Second, our approach is based on various techniques to analyze strings in source code, while these approaches do not focus on analyzing strings.

## 8 Conclusions and Future Work

In this paper, we present an approach to automatically locating need-to-externalize constant strings. Our approach is based on generalized string-taint analysis, and includes three practical techniques to cope with complications in the targeted problem. We evaluated our approach on seven real-world open-source applications: ArtOfIllusion, JFreeChart, Megamek, Risk, RText, StoryBook, and TV-Browser. The empirical results demonstrate that our approach is able to locate most of the constant strings externalized by the developers, with a small number of false positives and false negatives. We also demonstrate that it is feasible to visualize the results in our analysis to provide further help for developers to internationalize software applications.

In future work, we plan to extend our approach to address the following research issues. First, as indicated in Section 6.2, we plan to quantitatively compare string-taint analysis and data-flow analysis, and investigate techniques specific to the target problem to achieve better cost-effectiveness. Second, 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. Third, we plan to extend our approach to support other ways of string transmission across the network. Fourth, we plan to further automate the collection of the *output API methods* required by our approach. In particular, we plan to mine the list of *output API methods* from existing internationalized software applications in which we can trace forward from the externalized strings to the methods that eventually send the strings to the GUI. Fifth, as 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 also plan to systematically investigate the string splitting issue.

## Acknowledgments

The authors from Peking University are sponsored by the National Basic Research Program of China (973) No. 2009CB320703, the Science Fund for Creative Research Groups of China No. 61121063, and the National Science Foundation of China No. 90718016. Tao Xie's work is supported in part by US National Science Foundation (NSF) grants CNS-0720641, CCF-0725190, CCF-0845272, CCF-0915401, CNS-0958235, and US Army Research Office grant W911NF-08-1-0443. This research is supported by the National Science Foundation of China No. 61228203.

## References

[1] A. Christensen, A. Møller, and M. Schwartzbach, "Precise Analysis of String Expressions," *Proc. Static Analysis Symp.*, pp. 1-18, 2003.

[2] J.A. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," *Proc. Int'l Symp. Software Testing and Analysis* , pp. 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. Programming Languages and Systems,* vol. 13, no. 4, pp. 451-490, Oct. 1991.

[4] V. Dagiene and R. Laucius, "Internationalization of Open Source Software: Framework and Some Issues," *Proc. Int'l Conf. Information Technology: Research and Education,* pp. 204-207, 2004.

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

[6] M. Gabel, L. Jiang, and Z. Su, "Scalable Semantic Code Clone," *Proc. Int'l Conf. Software Eng.*, pp. 321-330, 2008.

[7] E. Geay, M. Pistoia, T. Tateishi, B. Ryder, and D. Julian, "Modular String-Sensitive Permission Analysis with Demand-Driven Precision," *Proc. Int'l Conf. Software Eng.*, pp. 177-187, 2009.

[8] C. Gould, Z. Su, and P.T. Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications," *Proc. Int'l Conf. Software Eng.*, pp. 645-654, 2004.

[9] P. Guo, J.H. Perkins, S. McCamant, and M.D. Ernst, "Dynamic Inference of Abstract Types," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 255-265, 2006.

[10] W.G.J. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks," *Proc. IEEE/ACM Conf. Automated Software Eng.*, pp. 174-183, 2005.

[11] J. Hogan, C. Ho-Stuart, and B. Pham, "Current Issues in Software Internationalisation," *Proc. Australian Computer Science Conf.*, 2003.

[12] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and Precise Sanitizer Analysis with BEK," *Proc. USENIX Conf. Security*, 2011.

[13] J. Kam and J. Ullman, "Global Data Flow Analysis and Iterative Algorithms," *J. ACM,* vol. 23, no. 1, pp. 158-171, Jan. 1976.

[14] A. Kieżun, P.J. Guo, K. Jayaraman, and M. Ernst, "Automatic Creation of SQL Injection and Cross-Site Scripting Attacks," *Proc. Int'l Conf. Software Eng.*, pp. 199-209, 2009.

[15] Y. Minamide, "Static Approximation of Dynamically Generated Web Pages," *Proc. Int'l Conf. World Wide Web,* pp. 432-441, 2005.

[16] F. Nielson, H. Nielson, and C. Hankin, *Principles of Program Analysis.* Springer, 1999.

[17] R. O'Callahan and D. Jackson, "Lackwit: A Program Understanding Tool Based on Type Inference," *Proc. Int'l Conf. Software Eng.*, pp. 338-348, 1997.

[18] T. Tateishi, M. Pistoia, and O. Tripp, "Path- and Index-Sensitive String Analysis Based on Monadic Second-Order Logic," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 166-176, 2011.

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

[20] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner, "Symbolic Finite State Transducers: Algorithms and Applications," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 137-150, 2012.

[21] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and Y. Jeffery, "Matching Dependence-Related Queries in the System Dependence Graph," *Proc. IEEE/ACM Conf. Automated Software Eng.*, pp. 457-466, 2010.

[22] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating Need-to-Translate Constant Strings for Software Internationalization," *Proc. Int'l Conf. Software Eng.*, pp. 353-363, 2009.

[23] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "TranStrL: An Automatic Need-to-Translate String Locator for Software Internationalization," *Proc. Int'l Conf. Software Eng.*, pp. 555-558, 2009.

[24] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating Need-to-Translate Constant Strings in Web Applications," *Proc. Int'l Symp. the Foundations of Software Eng.*, pp. 87-96, 2010.

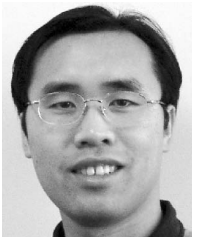
[25] G. Wassermann and Z. Su, "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities," *Proc. ACM Conf. Programming Language Design and Implementation,* pp. 32-41, 2007.

[26] G. Wassermann and Z. Su, "Static Detection of Cross-Site Scripting Vulnerabilities," *Proc. Int'l Conf. Software Eng.*, pp. 171-180, 2008.

[27] M. Weiser, "Program Slicing," *Proc. Int'l Conf. Software Eng.*, pp. 439-449, 1981.
[28] R. Wilson and M. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 1-12, 1995.
[29] Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," *Proc. USENIX Security Symp.*, pp. 176-192, 2006.
[30] F. Yu, M. Alkhalaf, and T. Bultan, "Generating Vulnerability Signatures for String Manipulating Programs Using Automata-Based Forward and Backward Symbolic Analyses," *Proc. ACM/IEEE Conf. Automated Software Eng.*, pp. 605-609, 2009.
[31] F. Yu, M. Alkhalaf, and T. Bultan, "Patching Vulnerabilities with Sanitization Synthesis," *Proc. Int'l Conf. Software Eng.*, 2011.

**Xiaoyin Wang** received the BS degree from the Department of Computer Science at the Harbin Institute of Technology in July 2006. Since September 2006, he has been working toward the PhD degree in the School of Electronic Engineering and Computer Science at Peking University. His research interests include the area of software engineering, including software maintenance, software mining, and software refactoring.

**Lu Zhang** received the BS and PhD degrees in computer science from Peking University in 1995 and 2000, respectively. He is a professor in the Institute of Software, School of Electronics Engineering and Computer Science, Peking University, P.R. China. He was a visiting postdoctoral researcher in the School of Computing and Mathematical Sciences, Oxford Brookes University, United Kingdom, from September 2000 to February 2001. From April 2001 to January 2003, he worked as a postdoctoral researcher in the Department of Computer Science, University of Liverpool, United Kingdom. His research interests include testing of software components and component-based software, program comprehension, software maintenance and evolution, software reuse, and component-based software development.



**Tao Xie** received the BS degree from Fudan University in 1997, the MS degree from Peking University in 2000, and the PhD degree from the University of Washington in 2005. He is an associate professor in the Department of Computer Science at North Carolina State University. His primary research interest is software engineering, with an emphasis on software testing, program analysis, and software analytics. He is a senior member of the IEEE and the ACM.

**Hong Mei** received the BA and MS degrees in computer science from the Nanjing University of Aeronautics and Astronautics, in 1984 and 1987, respectively, and the PhD degree in computer science from Shanghai Jiaotong University in 1992. From 1992 to 1994, he was a postdoctoral research fellow at Peking University. Since 1997, he has been a professor and PhD advisor in the Department of Computer Science and Engineering at Peking University. He has also served as dean of the School of Electronics Engineering and Computer Science and the Capital Development Institute at Peking University, respectively. His current research interests include software engineering and software engineering environment, software reuse and software component technology, distributed object technology, software production technology, and programming language. He is a senior member of the IEEE.

**Jiasu Sun** received the BS and MS degrees from Peking University. He is a professor in the Institute of Software, School of Electronics Engineering and Computer Science, Peking University, P.R. China.