2013 International Conference on Engineering of Complex Computer Systems

Software Internationalization and Localization: An Industrial Experience

Xin Xia*[♭], David Lo[†], Feng Zhu[‡], Xinyu Wang*, and Bo Zhou*[§]
*College of Computer Science and Technology, Zhejiang University
{xxkidd, wangxinyu, bzhou}@zju.edu.cn
[†]School of Information Systems, Singapore Management University
davidlo@smu.edu.sg
[‡]State Street Corporation
FZhu3@statestreet.com

***Abstract*—Software internationalization and localization are important steps in distributing and deploying software to different regions of the world. Internationalization refers to the process of reengineering a system such that it could support various languages and regions without further modification. Localization refers to the process of adapting an internationalized software for a specific language or region. Due to various reasons, many large legacy systems did not consider internationalization and localization at the early stage of development. In this paper, we present our experience on, and propose a process along with tool supports for software internationalization and localization. We reengineer a large legacy commercial financial system called PAM of State Street Corporation, which is written in C/C++, containing 30 different modules, and more than 5 millions of lines of source code. We propose a source code ranker that recovers important source code to be analyzed. Based on this code, we extract general patterns of the source code that need to be reengineered for internationalization. We divide the patterns into 2 categories: convertible patterns and suspicious patterns. To locate the source code that need to be modified, we develop an automated tool I18nLocator, that consumes these patterns and outputs the locations that match the patterns. The source codes matching the convertible patterns are automatically converted, and those matching the suspicious patterns are converted by developers considering the context of the corresponding codes. For localization, we extract hard-coded strings, translate them, and store them into resource data files. Out of the 504 thousands of lines of source code that are modified using our proposed approach, we can automatically modify 79.76% of them, saving much valuable developers' time. The quality of the resultant system is also good. The number of bugs per lines of code modified found during user acceptance test and deployment to the production environment is 0.000218 bugs/LOC.**

Keywords*—Software Internationalization, Software Localization, Reengineering, Industry Experience

I. Introduction

Nowadays, software applications always need to be distributed to different countries and regions over the world. It is common to see that a popular software has many local versions, for example, Windows 7 has US version, French version, simplified Chinese version, etc. The local versions of software help local users better understand and use it, attract more users, and increase software sales. Unfortunately, not all software products are originally designed with multi-language support. Some legacy systems developed between 1980s and 1990s did not take software internationalization and localization into consideration, and they still provide service around the world due to various reasons.

Reengineering a legacy system to support multiple languages and providing local versions require modification of considerable amount of source code. We need an effective process and efficient tools in order to reduce development effort. One naive way to internationalize and localize software is to locate all the places in the source codes that require modifications manually, and modify those codes. But in practice, this is ineffective since there are millions of lines of source code in legacy systems. It is hard to cover all the source code by manual inspection. This work requires hundreds of human resources, which will significantly increase the software development and maintenance cost.

At present, a couple of books have been published on software internationalization and localization [1], [2]. These books are good sources of information for software internationalization and localization, and introduce many technologies and tools. However, in a real commercial project, the process of software internationalization and localization is a bit different, and no industrial experience is reported in these books. Generally speaking, software internationalization and localization refer to two different tasks to reengineer legacy systems. Internationalization is the process of making a software application adapts to various languages and regions without engineering modifications; Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text [2].

In this paper, we present our experience in software internationalization and localization. We propose a process and tool supports for software internalization and localization. We reengineer PAM, which is a large-scale legacy financial system of State Street Corporation, containing more than 5 millions of C/C++ source codes, and 30 different modules grouped into clusters. The whole project lasts for 2 years, from 2009-2011, and 25 developers and 8 quality assurance personnel (QAs) are involved. We iterate the reengineering work from cluster to cluster. For each iteration, we use our source code

[♭]The work was done while the author was visiting Singapore Management University.
[§]Corresponding author.

978-0-7695-5007-7/13 $26.00 © 2013 IEEE
DOI 10.1109/ICECCS.2013.40

ranker *IRanker* to find the most important source files. We extract some patterns from these files, and use them in our automated code search tool *I18nLocator*. These patterns are divided into 2 categories: convertible patterns and suspicious patterns, and *I18nLocator* automatically converts the source code located by convertible patterns, and highlights the source code located by suspicious patterns. By considering the context of source codes located by suspicious patterns, developers will reengineer them. There might be some patterns that we miss by just analyzing the important files highlighted by IRanker. We run unit tests and perform other internal quality assurance (QA) activities to find bugs caused by the missing patterns. Additional patterns to address these bugs are then identified. After we internationalize PAM, we extract hard-coded strings and put these in a separate resource data file.

The main contributions of this paper are as follows:

1) We present our experience in reengineering a large financial legacy system consisting of more than 5 million lines of code to support internalization and localize it to Chinese language. To our best knowledge, it is the first time an industrial experience on software internationalization and localization is reported. We reengineer an actual legacy system, propose a framework, and present the experience we learned from the project. We believe our experience and proposed process can help others perform other internationalization and localization projects.
2) We propose a source code ranker *IRanker* to select a small proportion of source code files from which important patterns are inferred by manual inspection. We extract two types of patterns: convertible patterns, and suspicious patterns.
3) We develop an internationalization tool *I18nLocator* to locate and convert source codes. It processes the convertible and suspicious patterns.

The remainder of the paper is organized as follows. In Section II, we present a brief introduction of PAM. In Section III, we elaborate our framework for internationalizing and localizing PAM. In Section IV, we present our software internationalization methodology. In Section V, we present our software localization methodology. In Section VI, we report our assessment on the effectiveness of our proposed methodologies. In Section VII, we briefly introduce the related work. In Section VIII, we conclude and present future work.

II. PAM: A Brief Introduction

PAM is developed in 1990s, and it has a traditional client/server architecture. The architecture of PAM is presented in Figure 1. PAM uses Microsoft Foundation Class (MFC) technology to build its client, and socket communication to transfer messages between client and server. The server contains three parts: application server handles messages from client, and decides whether to communicate with data servers to generate the required data; Data servers mainly fetch data from a database or execute some operations on the database (such as add, modify, remove, and delete records) according to the requests from the application server; Database stores and persists data. The four layers of our client/server architecture

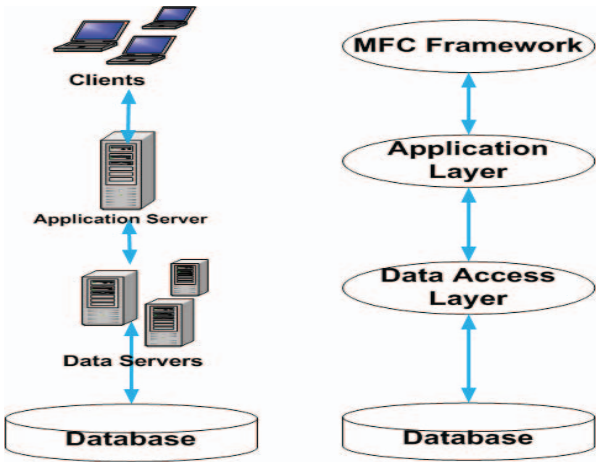


Fig. 1. The Architecture of PAM

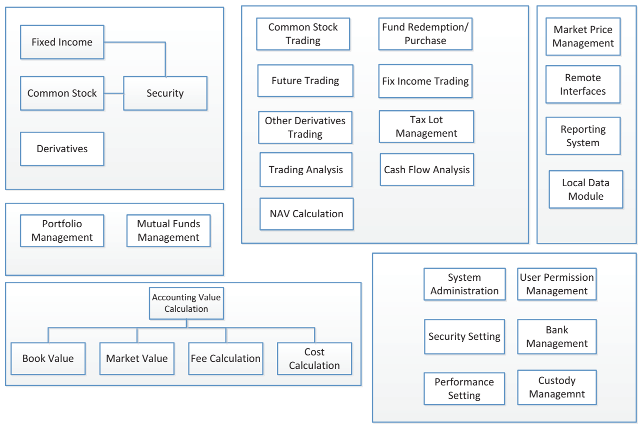


Fig. 2. PAM Modules

could be mapped to MFC Framework layer, application layer, data access layer, and database layer, as shown in Figure 1.

Figure 2 presents the modules of PAM, which contains 30 different modules. According to the functionalities of these modules, we divide them into 6 clusters as grouped in boxes.

III. Overall Framework

Software internationalization and localization refer to two different processes in reengineering PAM. Localization depends on the internationalized version of PAM. Figure 3 presents our proposed internationalization and localization framework. We believe that this framework could be used for other internationalization and localization projects.

We first extract important source code files using our source code ranker *IRanker* (Step 1). Then, we analyze the important source codes to extract convertible patterns and suspicious patterns (Step 2 and 3). Next, we input these patterns into

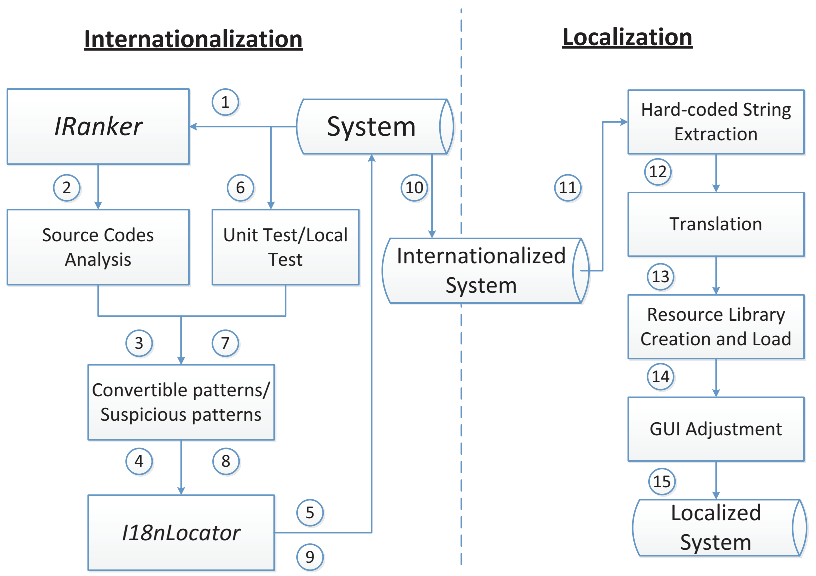


Fig. 3. The Overall Framework of PAM Internationalization and Localization

I18nLocator (Step 4), an automated tool to locate the source codes which match these patterns. *I18nLocator* automatically converts the source codes matching convertible patterns, and highlights the source codes matching suspicious patterns to the developers for manual conversion (Step 5). After the above reengineering steps, we also perform unit testing and local quality assurance checks (Step 6). We analyze the failures captured by the unit tests and quality assurance checks, discover the patterns we missed, and remove/edit the incorrect patterns (Step 7). We again locate these patterns using I18nLocator (Step 8). By iterating the above steps many times for each cluster of PAM modules, we finally internationalize PAM.

For PAM localization, we first extract the hard-coded strings from the source codes (Step 11). Then we translate these hard-coded strings to Chinese language (Step 12), and for each of the different languages (i.e., English and Chinese), we provide different resource library files to cleanly separate resource data from source codes (Step 13). Finally, we adjust the GUI widgets to ensure that they have enough space to display texts in different languages (Step 14).

IV. Software Internationalization Methodology

In this section, we first present the general idea of our software internationalization process in Section IV-A, then we elaborate *IRanker* which extracts important source code files for manual analysis in Section IV-B. After manual analysis, we extract convertible and suspicious patterns from the extracted codes. We describe some of these patterns in Section IV-C. Finally, we present *I18nLocator*, an automated tool that locates the codes matching the patterns in Section IV-D.

A. General Idea

The main task for system internationalization is Unicode transformation, i.e., replace the types and functions in source codes which do not support Unicode standard to Unicode compatible ones. Concretely, Unicode transformation includes the transformation of the following 5 program element types:

1) **Character Variable:** We transform regular character variables to Unicode compatible character variables. For example, we need to replace *char* or *char** with *TCHAR* or *TCHAR** respectively, and replace *LPSTR* with *LPTSTR*.
2) **Function:** We replace the invocations of functions that perform unsuitable string-related operations with appropriate ones. These functions mainly include string processing functions in standard C library and Win32 API. For example, we need to replace the invocations of function *strcpy* defined in the standard C library with function *_tcscpy* defined in the same library. We need to replace the invocations of function *SetWindowTextA* defined in the Win32 API with function *SetWindowText* defined in the same API.
3) **Windows Macro:** In standard C language, macro *_T* and *_Text* are used to represent constant characters and constant strings respectively [3]. These two macros are compatible with both Unicode and ASCII. If a system follows ASCII standard, *_T* and *TEXT* will use ASCII characters and strings, i.e., *char* type which takes one byte in Windows. Otherwise, they will use Unicode characters and strings, i.e., *TCHAR* type which takes 2 bytes in Windows. We need to extract all these constant characters and constant strings, and replace them with macro *_T* and *TEXT*, respectively. For example, for a constant string *"Hello"*, we need to replace it with *TEXT("Hello")*. *TEXT("Hello")* will correspond to *"Hello"* in ASCII standard if a system follows ASCII standard, or *L"Hello"* in Unicode standard otherwise.

4) **Database Fields:** We transform database fields to wide character type, and codes that access the database to support Unicode. Since PAM was developed in 1990s, at that time there is no wide character type and Unicode support. Fortunately, currently ODBC provides Unicode compatible functions. Thus, we need to modify types and functions for source codes related to database access such that they support Unicode.
5) **Text Processing:** We need to support writing and reading from Unicode files. In non-Unicode project, it is easy to write to and read from files. But when we would like to make the project internationalized, the input and output files can be in various formats, i.e., ASCII or Unicode. Even Unicode files have various formats, such as UTF-8, UTF-16 BE (Big Endian), and UTF-16 LE (Little Endien).

It seems the transformation of the above 5 types of program elements involve only replacement: we just need to replace the source codes based on some defined patterns. For example, we can simply replace all *char* with *TCHAR*. However, we meet the following two problems when we internationalize PAM:

1) What are the simple transformation patterns? Since PAM is a legacy system, there is not many documents that describe the system. We need to read source codes to extract the patterns. However, it is impossible to read all the source codes since there are more than 5 million lines of codes. Thus, we need to find a small yet representative set of codes that allows us to extract these patterns.
2) Are these simple transformation patterns enough for internationalization? We find that simple replacement patterns do not handle all cases. There are complicated code patterns. For example, in ASCII standard, one character occupies one byte which make developers not differentiate character and byte: character arrays are actually used as byte arrays, and byte arrays are used to store characters. For internationalization, we need to convert *char* into *TCHAR* which no longer occupies one byte. Also, in some cases, null pointer *void** are used as a parameter to pass string pointer to another function for writing or reading operations. Thus, we need to consider the context of the source code whenever we meet a null pointer *void** before performing any transformation. We refer to simple transformation patterns as convertible patterns. We automatically perform transformations for these convertible patterns. We refer to patterns that highlight codes that require more complex transformations as suspicious patterns. We need to locate codes matching these patterns and refer them to developers for manual inspections and modifications.

B. *IRanker: Ranking Important Codes*

Large-scale legacy systems, such as PAM, do not have many documents that describe them, and as discussed in Section IV-A, we need to find the transformation patterns from a small yet representative set of source codes. In this section, we propose *IRanker* which is used to detect a representative set of codes for manual identification of convertible and suspicious patterns.

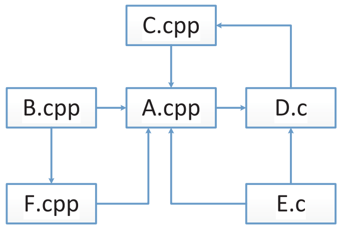


Fig. 4. An Example Dependency Graph which Contains 6 Source Files

We build a dependency graph [4], [5] at source code file level. We use a directed graph to represent the dependency relationships among source files, and for any two source files A and B, if A uses any of the methods in B, then A and B have a dependency relationship and there is an edge from A to B. In C/C++ language, the dependency is defined at the top of source files, by statements like "include <stdio.h>". We use these "include" statements to build a dependency graph among source files, and we use the corresponding source code files to replace the header files referred to in the "include" statements. For example, in source file "A.c", it includes "B.h", and in our dependency graph, we just use "B.c" to replace "B.h". We remove the standard C/C++ libraries and other libraries (e.g., Oracle libraries, MFC libraries) as we do not want to re-engineer them.

Figure 4 presents an example dependency graph which contains 6 source files. We notice several source files such as B.cpp, C.cpp, E.c and F.cpp depend on A.cpp, and A.cpp depends on D.c. To extract a subset containing representative files from these 6 source files, we consider the relative importance of each of the source files. The general rationales behind our measurements of how important a file is are: (1) Important source files will depend on important source files; (2) The more times one source file is depended by other source files, the more important it is. Based on the above two rationales, we leverage PageRank [6], [7] to discover the set of representative files. PageRank is used to measure the importance of a webpage in the world wide web and has been deployed in web search engines.

Our PageRank based approach follows the following scenario: We start by reading a source file A. When we decide to read the next source files, we have two choices: (1) we read the source files which A depends on with the probability α; (2) we randomly choose another source files in the system with probability $1-\alpha$. By iterating the above steps enough times, we can get convergence values for the probability of each source file to be read. The source files can then be ranked or sorted based on their probabilities. For example, we begin by reading source file B.cpp in Figure 4, and we choose the next source file. Since B.cpp only depends on A.cpp and F.cpp, we may choose A.cpp or F.cpp as the next source file. Otherwise, we may decide after reading B.cpp to randomly choose a source file to be read.

Formally, suppose there are n source files, let us denote $Rank_i$ as a vector containing the ranks of source files at iteration i, and denote B_u as a collection of source files which u depends on. Moreover, let us denote an $n \times n$ matrix P being the transition matrix with P_{uv} being the probability that a random surfer moves from source file u to source file v, and P_{uv} is defined as follows:

$$\begin{cases} P_{uv} = 1/|B_u|, if \ v \in B_u \\ P_{uv} = 0, else \end{cases} \tag{1}$$

The $Rank_i$ matrix is updated at each iteration by applying the following equation:

$$Rank_i^T = Rank_{i-1}^T * (\alpha P + (1-\alpha)ee^T/n) \tag{2}$$

In the above equation, $Rank_i^T$ represents the transpose of vector $Rank_i$, and $e = \{1, 1, ...1\}$ represents a unit vector. After we iterate Equation (2) enough times, we will get the final ranking scores of each source file. We set the transfer probability α as 0.85, which is the same α value used in the original PageRank paper [6], [7].

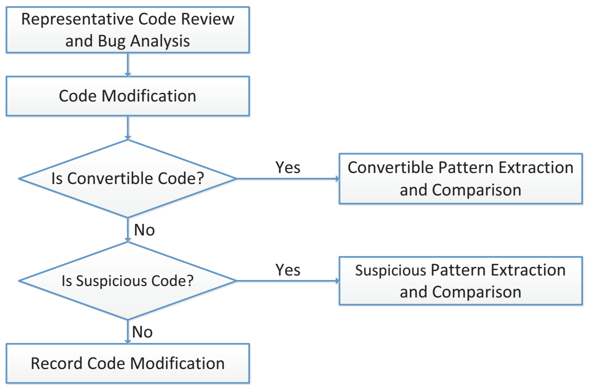
After we get the ranking scores for the source files, we choose the top 2% files based on the scores as the set of representative files, and we read these source files to detect common patterns which are used for PAM internationalization. As we described in the previous section, there are more than 5 million lines of source code, and if we consider top 2% source files, there are approximately $5,000,000 \times 2\% = 100,000$ lines of source code. Although this is a lot of lines of code, reading these lines requires much less effort than reading the original 5 million lines of code.

C. Convertible Patterns vs. Suspicious Patterns

We represent these patterns by regular expressions. We refer to the patterns that identify codes that can be automatically converted to their corresponding Unicode compatible counterparts as convertible patterns. We refer to the patterns that identify codes that require developers' judgement as suspicious patterns.

Figure 5 summarizes the process of convertible and suspicious patterns extraction. We first review representative set of codes collected by *IRanker* or analyze the bugs identified by unit tests and local checks, and modify these codes for internationalization. We analyze the modified codes one by one. If a modified code contains convertible code patterns, we extract the code patterns and represent them with regular expressions, and check whether these patterns have already existed in the convertible pattern repository. If these patterns are not in the convertible pattern repository, we add these patterns into the repository. If a modified code does not contain convertible code patterns, we further check whether it contain suspicious code patterns. If it does, we extract the suspicious code patterns and add them into the suspicious pattern repository if they have not been added there before. Finally, if we could not decide whether a modified code contains either convertible or suspicious code patterns, we record it, and re-analyze it when more of such modified codes are analyzed.

Suspicious patterns require developer inspection as the modifications of the corresponding codes need to consider the



Representative Code Review and Bug Analysis
Code Modification
Is Convertible Code?
Yes
Convertible Pattern Extraction and Comparison
No
Is Suspicious Code?
Yes
Suspicious Pattern Extraction and Comparison
No
Record Code Modification

Fig. 5. Suspicious and Convertible Patterns Extraction Process

```
1: char str1[12] = "Hello, World";
2: char str2[12], str3[12];
3: strncpy(str2,str1, 12); //A
4: size = strlen(str1) + 1;//B
5: memcpy(str2, str1, size); //C
```

Fig. 6. An Piece of Source Code where its Context Needs to be Taken into Consideration for Internationalization.

context of the codes. Often the root cause of the modifications corresponding to the suspicious patterns is that the size of a character increases from 1 byte to 2 bytes, which makes the memory space requirement of related data structures and operations changes. For example, Figure 6 presents a piece of source code for which we need to consider its context. In ASCII, the size of a *char* is one byte. The source code in Figure 6 makes this assumption. If we directly convert Lines 1 and 2 as

```
1: TCHAR str1[12] = "Hello, World";
2: TCHAR str2[12], str3[12];
```

Then at Line 3, we just copy half of *str1* into *str2*; similarly for Line 5. These kinds of code patterns are difficult to be automatically converted, since we need to consider the contextual information to get the true meaning of "12" at Line 3. The right way to modify the codes in Figure 6 is presented in Figure 7. We consider five suspicious pattern types, i.e., string pointer, function, COM component, file operation, and third-party component.

1) String Pointer: In C/C++ language, we use pointer to store the memory address of a variable. The source codes

```
1: TCHAR str1[12] = "Hello, World";
2: TCHAR str2[12], str3[12];
3: _tcsncpy(str2,str1, 12*sizeof(TCHAR));
   //A
4: size = _tcslen(str1) + 1;//B
5: memcpy(str2, str1, size *
   sizeof(TCHAR)); //C
```

Fig. 7. The Correct Way to Modify the Code Shown in Figure 6

which compute the address using pointer and its offset address are suspicious codes. For example, the following source code is deemed suspicious:

```
char buffer[MAX_SIZE];
memcpy(pCache, (LPSTR)buffer + 1,
count);
```

In ASCII standard, developers did not differentiate *char* and *byte*. We need to read the context of these pieces of source codes to decide whether the array *buffer* is a *byte* or a *char* array. For different usages of the array *buffer*, there are different ways to modify them. If it is used as a *byte* array, we do not need to modify it, i.e.,

```
char buffer[MAX_SIZE];
memcpy(pCache, (LPBYTE)buffer + 1,
count);
```

And if it is used as a *char* array, we need to modify it as follows:

```
TCHAR buffer[MAX_SIZE];
memcpy(pCache, (LPTSTR)buffer + 1,
count*sizeof(TCHAR));
```

2) Function: There are 3 types of suspicious patterns related to function definitions and invocations:

1) **No corresponding Unicode-compatible function:** Not all functions have their Unicode compatible version. For example, the functions _fcvt and _gcvt in standard C library which transform from double type to string type do not have their corresponding Unicode versions. For these kinds of functions, we need to locate them, and write a new function to make it compatible with Unicode.
2) **Parameter difference between corresponding Unicode and ASCII functions:** The corresponding ASCII and Unicode compatible versions may accept different parameters. For example, the ASCII and Unicode versions of memset accept different parameters:

   ```
   void *memset(void *dest, int c,
   size_t count);
   wchar_t *wmemset(wchar_t *dest,
   wchar_t c, size_t count);
   ```

   For these kinds of functions, we need to locate the invocations of these functions, and modify them accordingly.
3) **Parameter Misuse:** Many functions have the same parameters for their ASCII and Unicode versions. However, for some function invocations, although the arguments being passed in work for the ASCII version, these arguments need to be modified when invoking the Unicode version. We need to consider the contextual information and modify them accordingly. Figure 6 shows a typical example of parameter misuse. The correct way to modify the code is shown in Figure 7.

3) COM Component: Some COM related API functions only accept wide characters, and in the original PAM, developers use “MultiByteToWideChar” and “WideCharToMultiByte” functions to do the transformations between wide characters and single byte characters. However, to internationalize PAM, we no longer need to transform from a wide character to a single byte character. To deal with these kinds of COM codes, and to ensure that PAM is both compatible with ASCII and Unicode, we check for token “UNICODE”. If token “UNICODE” is not defined, the code follows ASCII standard, and we keep the original code; if token “UNICODE” is defined, we run the code which does not transform a wide character to a single byte character. For example, for the following piece of code:

```
COMServer::GetPrice(INT2 ID, BSTR NAME,
unsigned char* a_cpResult)
{
char L_NAME[MAX_NAME_SIZE] = 0;
wcstombs(L_NAME, NAME, sizeof(L_NAME));
*a_cpResult = ::GetPrice(ID, L_NAME);
}
```

We modify it as:

```
COMServer::GetPrice(INT2 ID, BSTR NAME,
unsigned char* a_cpResult)
{
#ifndef UNICODE
  char L_NAME[MAX_NAME_SIZE] = 0;
  wcstombs(L_NAME, NAME,
sizeof(L_NAME));
  *a_cpResult = ::GetPrice(ID, L_NAME);
#else
  *a_cpResult = ::GetPrice(ID, NAME);
//B
#endif
}
```

4) File Operation: To internationalize PAM, we need to make all file operation related codes support Unicode (mainly UTF-8, UTF-16) and ASCII. We develop a new file operation class UniFileOperation which supports Unicode and ASCII, locate source codes related to file operations, and replace these codes with the invocations of appropriate functions in the UniFileOperation class. The internal process of UniFileOperation is presented in Figure 8.

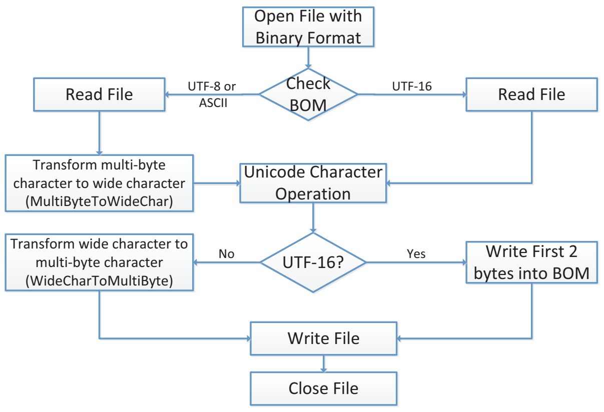
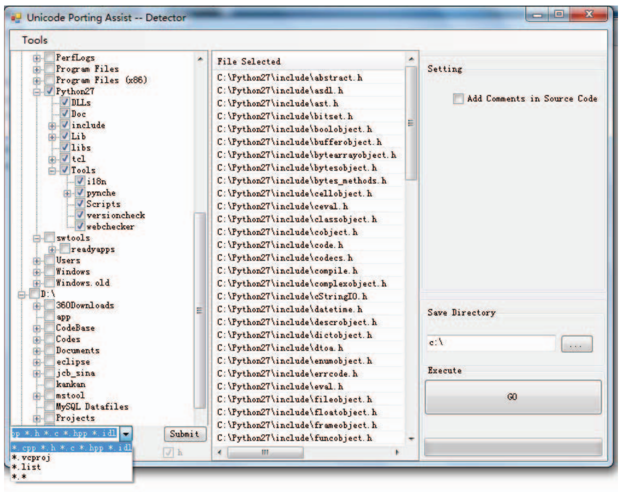

Open File with Binary Format
Check BOM
UTF-8 or ASCII
UTF-16
Read File
Read File
Transform multi-byte character to wide character (MultiByteToWideChar)
Unicode Character Operation
Transform wide character to multi-byte character (WideCharToMultiByte)
No
UTF-16?
Yes
Write First 2 bytes into BOM
Write File
Close File

Fig. 8. File Operation Process of UniFileOperation Class



```
File Selected
C:\Python27\include\abstract.h
C:\Python27\include\asdl.h
C:\Python27\include\ast.h
C:\Python27\include\bitset.h
C:\Python27\include\boolobject.h
C:\Python27\include\bufferobject.h
C:\Python27\include\bytearrayobject.h
C:\Python27\include\bytesobject.h
C:\Python27\include\bytes_methods.h
C:\Python27\include\cellobject.h
C:\Python27\include\ceval.h
C:\Python27\include\classobject.h
C:\Python27\include\cobject.h
C:\Python27\include\code.h
C:\Python27\include\codecs.h
C:\Python27\include\compile.h
C:\Python27\include\complexobject.h
C:\Python27\include\cStringIO.h
C:\Python27\include\datetime.h
C:\Python27\include\descrobject.h
C:\Python27\include\dictobject.h
C:\Python27\include\dtoa.h
C:\Python27\include\enumobject.h
C:\Python27\include\errcode.h
C:\Python27\include\eval.h
C:\Python27\include\fileobject.h
C:\Python27\include\floatobject.h
C:\Python27\include\frameobject.h
C:\Python27\include\funcobject.h
```

```
pp *.h *.c *.hpp *.idl
*.cpp *.h *.c *.hpp *.idl
*.vcproj
*.list
*.*
```

```
c:\
```

Fig. 9. GUI of Detector Component of *I18nLocator*

UniFileOperation first checks the file type (whether it is UTF-8, UTF-16 or ASCII). UTF-16 files could be identified by the first two bytes in its header: if the first two bytes are "FEFF", then it is a Big Endian file; and if the first two bytes are "FFFE", then it is a Little Endian file. For a UTF-8 file, the first three bytes are "EFBBBF" in Windows. We refer to the first several bytes of a file header as byte order mask (BOM). When reading and writing UTF-8 and ASCII file, we need to transform the characters from single byte characters to wide characters. Functions "MultiByteToWideChar" and "WideCharToMultiByte" perform such transformations.

5) Third-party Component: In PAM, there are many third-party components. For these components, we do not have any source codes. One way to internationalize them is to update them to the latest versions. However, only a few of them can be updated, due to economic reasons (they are expensive) or some software vendors stop developing new versions. To internationalize third-party components in PAM, we build an adapter and place it in between PAM and these third-party components. We first locate all the source codes interfacing with third-party components. Then we modify the codes which fetch information from third-party components and pass arguments to third-party components and involve the invocations of "MultiByteToWideChar" and "WideCharToMultiByte" functions.

D. I18nLocator

I18nLocator is an automated tool which we use to locate codes matching suspicious patterns and convert codes matching convertible patterns. It contains two components: replacer which is used to locate and convert convertible codes, and detector which is used to locate suspicious codes. Figure 9 presents the GUI of the detector component in *I18nLocator*.

1) Replacer: Replacer component mainly converts source codes matching convertible patterns. We use regular expressions to represent convertible patterns. These convertible patterns are not simple patterns. For example, although we need to replace *char* with *TCHAR* for internationalization, there are three scenarios in which we cannot directly replace *char* with *TCHAR* as shown in Table I. The regular expression for this convertible pattern is:

TABLE I. THREE SCENARIOS WHERE *char* CANNOT BE REPLACED BY *TCHAR*

Scenarios	Description	Example
char in **#define** statements	Macro definition, no modification is required	#define A char;
char in **typedef** statements	Macro definition, no modification is required	typedef char A;
unsigned char	Cannot be replaced	unsigned char a='0';

```
!(^(unsigned\s+|typedef \s+|#define.*))
\bchar\b
```

The above regular expression refers to occurrences of keyword *char* in source code lines which do not begin with "unsigned", "typedef" or "#define".

2) Detector: Detector component is used to locate suspicious codes matching suspicious patterns. These suspicious patterns are also represented by regular expressions. Developers use I18nLocator to locate suspicious source codes in PAM. Whenever suspicious codes are detected, I18nLocator will add a comment behind these lines of source codes. The format of the comment is: //SUSPICIOUS(OPEN): {Corresponding Regular Expressions}. For example, the following "memset" function is located based on a suspicious pattern:

```
memset((void*)str, '\0', 9);
//SUSPICIOUS(OPEN): { \bmemset\b
(?!(.*,\s*0\s*,))}
```

When developers complete the modification, they are required to modify the status of the comments as: //SUSPICIOUS(CLOSED): {Corresponding Regular Expressions} {Developer Name and Modification Time}. The above example would then become:

```
#ifdef UNICODE
wmemset((void*)str, '\0', 9); //
SUSPICIOUS(CLOSED): { \bmemset\b
(?!(.*, \s*0\s*,))} { by Xin Xia
2012/11/2 20:19 }
#endif
```

When *I18nLocator* is run again, it would ignore source codes with comments "SUSPICIOUS(CLOSED)".

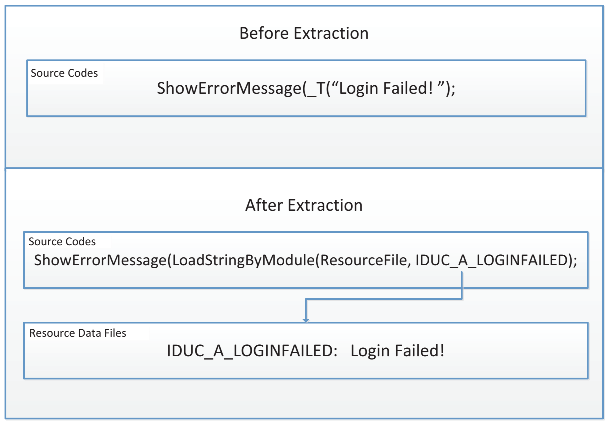
V. SOFTWARE LOCALIZATION METHODOLOGY

Software localization requires us to adapt internationalized software for a specific region or language. In this section, we propose our steps for PAM localization. We first present hard-coded string extraction and resource file creation in Section V-A, then we address the translation of PAM in V-B.

A. Hard-coded String Extraction

Hard-coded strings refer to the strings which can be displayed in the GUI, and they are directly written in source codes. An example is shown below:

```
CString loginFailedMsg = "the password
is wrong";
```



```
ShowErrorMessage(_T("Login Failed! ");
```

```
ShowErrorMessage(LoadStringByModule(ResourceFile, IDUC_A_LOGINFAILED);
```

```
IDUC_A_LOGINFAILED:   Login Failed!
```

Fig. 10. Hard-coded String Extraction and Resource Data Creation Process

For hard-coded string, we need to extract it, use a variable to replace it, and store the mapping relationship between the variable and hard-coded string in a resource data file. Different from codes targeted by our internationalization methodology, hard-coded strings have a single pattern, i.e., begins with a " or a ', ends with a " or a ', and does not appear in macro definition statements. We use a regular expression to represent this pattern. A typical process of hard-code string extraction, and storage in a resource data file is presented in Figure 10. Before extraction, the hard-coded string (i.e., "Login Failed! ") exists in the source code. After extraction, we replace the hard-coded string with a variable (i.e., IDUC_A_LOGINFAILED). Then, we load the variable value from a resource data file (i.e., ResourceFile). And in the resource data file, we store the mapping between the variable and its value (i.e., IDUC_A_LOGINFAILED: Login Failed!). Different languages have different resource data files. To localize PAM, we configure the language setting in a configuration file.

B. Translation

Translation includes GUI text translation and online help document translation. We outsource these need-to-translate materials (i.e., hard-coded string and documents) to a third-party local language translation provider. Four types of objects need to be translated, i.e., resource data file which is used to store hard-coded strings, online help document, constant strings in database and other text files in PAM. Translation may cause various bugs in PAM, and these bugs are hard to fix.

1) Translation Causing Functional Bugs: Translation may cause the lengths of strings to change. The length of a translated string may be longer than the original one, which causes the string buffer to overflow. This phenomenon is common for English abbreviation. For example, a financial terminology "SEC" which represents "Securities and Exchange Commission", when translated into Chinese becomes "美国证券交易委员会". In this case, the length of the string increases, and the string buffer overflows. This is a functional bug.

Another situation that causes functional bugs due to translation happens when we translate strings which should not be translated. For example, consider the source code below:

```
1: #define RED_COLOR "RED"
2: color = RED_COLOR
3: if(color == "RED")
4: {
5: //DO WORK
6: }
```

String "RED" which appears in "#define" statement is not a hard-coded string since it will not appear in the GUI. Note that if we replace "RED" at Line 1 with a method invocation (to load the corresponding string from a resource file), it would cause a compilation error. The string "RED" at Line 3 is a hard-coded string though. We may extract the hard-coded string "RED" at Line 3 and translate it to Chinese "红", which would cause a functional bug since the condition in the "if" statement will never be met.

2) Translation Causing GUI Bugs: After translation, the lengths of strings change, and these would potentially cause GUI bugs:

1) Widget alignment. The width of some widgets would change according to the length of the displayed texts. This causes the widgets to be badly aligned when the text is changed from one language to another.
2) Truncated text display in widgets. Some widgets have a fixed width. Thus, when the text length increases, it would cause a truncated text to be displayed
3) Unreasonable translation which causes shortcut keys to be hidden. For example, a menu item "&New" has two meanings: (1) it means that this menu can open a new file; (2) "&N" also means that there is a shortcut key "CTRL+N" which will open a new file in the MFC standard. Thus, the right way to translate it to Chinese should be "新建(&N)". But if we translate it as "新建", then the shortcut key "CTRL+N" would be disabled.

VI. Experiments

In this section, we report our assessment on the effectiveness of our proposed methodologies for PAM. We first present the effectiveness of *I18nLocator* in Section VI-A, and then we present the test results for PAM internationalization and localization in Section VI-B.

A. Effectiveness of I18nLocator

Since *IRanker* and unit tests/local checks both discover convertible and suspicious patterns, we divide the effectiveness of *I18nLocator* into 2 parts: *IRanker* and unit tests/local checks. Table II presents the results of I18nLocator by *IRanker* and unit tests/local checks. With *IRanker*, we detect most of the convertible patterns and suspicious patterns, i.e., 30 convertible patterns and 25 suspicious patterns respectively, which cover 78.95% of all the convertible patterns and 71.43% of all the suspicious patterns.

Due to the security confidentiality policy of State Street Corporation, we can only list the lines of source codes in

TABLE II. THE NUMBER OF CONVERTIBLE PATTERNS (#CON.PAT), NUMBER OF SUSPICIOUS PATTERNS (#SUS.PAT), NUMBER OF TOTAL PATTERNS (#TOTAL.PAT), NUMBER OF LINES OF SOURCE CODES CONVERTED BY CONVERTIBLE PATTERNS (#CON.LINES), NUMBER OF LINES OF SOURCE CODES LOCATED BY SUSPICIOUS PATTERNS (#SUS.LINES), AND NUMBER OF TOTAL SOURCE CODES (#TOTAL.LINES) MODIFIED. ALL NUMBERS OF LINES OF CODE ARE ROUNDED DOWN TO THE NEAREST THOUSANDS.

Type	#Con.Pat	#Sus.Pat	#Total.Pat	#Con.Lines	#Sus.Lines	#Total.Lines
IRanker	30	25	55	350,000	80,000	430,000
Unit tests/Local checks	8	10	18	52,000	22,000	74,000
Total	38	35	73	402,000	102,000	504,000

TABLE III. THREE TYPES OF INTERNATIONALIZATION AND LOCALIZATION TEST FOR PAM

Type	**Test Aspects**	**Description**
Functional Test	General functionality	Test general functionality. Internationalization and localization may cause logic error in source codes.
	Resource data files	Test whether correct language resource data files are loaded.
	File processing	Test whether the resultant internationalized and localized system supports different file formats, such as ASCII, UTF-8 and UTF-16 files.
	Non-Unicode program transformation	Test whether source codes support Unicode. For example, input Chinese character, and read output character.
GUI Test	Widgets layout	Test whether the widgets are aligned, and the lengths of texts in the widgets are acceptable.
	Text display	Test the display of text, whether it is displayed in the right way.
	Shortcut key usage	Test whether the shortcut keys are changed due to localization.
	TAB key sequence	Test whether the sequence of widgets highlighted when the TAB key is pressed multiple times is correct.
Translation Test	Online help documents	Test whether the online help documents are translated, and whether online help documents can be switched from one language to another.
	GUI translation	Check whether all the texts in the GUI are translated, i.e., no other languages appear in a localized version.
	Translation correctness	Review the quality of the translation, whether it is correct and acceptable. We mainly focus on the texts displayed in the GUIs and online help documents.

approximate format, i.e., we need to truncate the number to the nearest 1,000. Using the patterns identified by analyzing the source code selected by *IRanker*, *I18nLocator* automatically converts around 350,000 lines of source codes matching convertible patterns, and locates around 80,000 lines of source codes matching suspicious patterns, which cover 87.06% and 78.43% of the total source codes modification work. *IRanker* helps PAM internationalization significantly since it extracts a relatively small proportion of the code from which general patterns can be inferred. These patterns cover most of the changed code. This reduces the testing and analysis work needed.

With the help of *IRanker* and unit tests/local checks, we discover 73 patterns and modify 504,000 lines of source codes. Among 504,000 lines of source codes, we automatically convert 402,000 lines of source codes which covers 79.76% of the reengineering work. The remaining lines of code are flagged by the suspicious patterns. Although they do not automatically re-engineer the codes, they significantly aid developers locate these codes. Locating these codes is not an easy job, due to the complexity of PAM.

B. Internationalization and Localization Test

The test for the quality of PAM internationalization and localization should consider two aspects: first, we should not affect the original functionality of PAM; Second, we should support multi-language input and have local language GUI. The test work should cover these two aspects, and we divide test work into 3 types: functional test, GUI test and translation test. The details of these 3 types of test are presented in Table III.

Due to software management strategy of State Street, we have 3 environments, i.e., development environment (DEV), user acceptance test environment (UAT) and product environment (PROD). DEV is used by local QAs and developers; we perform local checks and unit tests in DEV. We designed 3,014 test cases to test PAM. After we pass all these test cases, we deliver PAM to UAT and further to PROD. Only the bugs discovered in UAT and PROD are recorded in the bug tracking system. There are 110 bugs reported. Considering that we modify 504,000 lines of code, the number of bugs per lines of code modified found during user acceptance test and deployment to the production environment is 0.000218 bugs/LOC. This shows that the quality of the resultant internationalized and localized system is good.

TABLE IV. TYPE, TEST, AND NUMBER OF BUGS (# BUGS) FOR PAM

Type	**Test Aspects**	**# Bugs**
Functional Test	General functionality	4
	Resource data files	4
	File processing	4
	Non-Unicode program transformation	8
GUI Test	Widget layout	8
	Text display	10
	Shortcut key usage	4
	TAB key sequence	1
Translation Test	Online help document	10
	GUI translation	25
	Translation correctness	26
Miscellaneous	Other Bugs	6

The bug distribution is presented in Table IV. The number of bugs detected by functional tests, GUI tests and translation tests are 20, 23 and 61 respectively, which are 18.2%, 20.9% and 55.5% of the total number of bugs respectively. Considering all 110 bugs, it is interesting to note that translation bugs are the majority of all bugs detected when PAM is deployed in UAT and PROD. Moreover, many functional bugs and GUI bugs are also related to translation.

From Table IV, we also notice that internationalization related bugs are relatively few. File processing and non-Unicode program transformation are related to internationalization, but there are only 12 bugs in these categories, which is 10.9% of the total number of bugs. These show that our proposed software internationalization and localization process is effective.

VII. Related Work

To our best knowledge, there is limited research work on software internationalization and localization. Wang et al. propose a method which automatically locates need-to-translate constant strings for software internationalization [8], [9]. They first collect APIs related to GUI, and based on a string-taint analysis, they search for need-to-translate constant strings. Finally, they evaluate the performance of their approach using 4 open source applications: RText, RISK, ArtOfIllusion, and Megamek. Wang et al. further extend their work by locating these need-to-translate constant strings for web application internationalization [10]. They propose a flag propagation based approach which distinguishes strings visible at browser side from non-visible strings, and evaluate the performance of their approach using 3 PHP based web applications.

Our work is different from the work of Wang et al. in the following aspects:

1) In our paper, we propose a *holistic and end-to-end* process for software internationalization and localization, along with tool supports. Wang et al. solve ONE problem in the software internationalization process, which is the locating of constant strings. We use a more lightweight solution to locate constant strings. We found our simple solution to be sufficient, effective, and fast. In the future, we could also use their proposed approach to locate constant strings. Our approach considers a complete picture of software internationalization and localization including: extraction of convertible and suspicious patterns for software internationalization, locating of these patterns in code, automatic conversions of convertible patterns, and many more.
2) We evaluate our proposed process on a commercial system called PAM, which contains 5 million lines of source code. The largest system evaluated by Wang et al. only contains 110,000 lines of source code. Also, the system that we analyze and re-engineer is currently deployed and used in the industry.

There are many industrial studies reported in the literature. We highlight some of them below. Gegick et al. report an industrial study on bug report classification in Cisco [11]. They propose an approach that can automatically label a bug report as security-related or not. Port et al. describe experiences on mining a large collection of textual software artifacts in Jet Propulsion Laboratory (JPL), NASA [12]. Dang et al. describe experiences on clone mining in Microsoft using their tool named XIAO [13]. In this paper, we also describe an industrial experience. However, we focus on a different problem domain namely software internationalization and localization.

VIII. Conclusion and Future Work

In this paper, we describe our experience on software internationalization and localization. We study PAM, a large-scale commercial system, which contains 5 million lines of source codes. We propose a holistic and end-to-end process for software internationalization and localization, and also propose supporting tools: IRanker and I18nLocator. We build *IRanker* to select a representative set of code to extract convertible and suspicious patterns. Using *I18nLocator*, we locate source codes which match suspicious patterns and convert source codes which match convertible patterns. *IRanker* and *I18nLocator* saves much time and cost. For software localization, we extract hard-coded strings, translate these strings and related documents, and store them into resource data files. Our case study shows that our proposed process and tool support for software internationalization and localization are effective. Out of the 504 KLOC that are modified using our proposed approach, we can automatically modify 79.76% of them, saving much valuable developers' time. The quality of the resultant system is also good. The number of bugs per lines of code modified found during user acceptance test and deployment to the production environment is 0.000218 bugs/LOC.

In the future, we plan to do more internationalization and localization work on other large-scale commercial systems, and learn more from these experiences.

Acknowledgment

This research is sponsored in part by NSFC Program (No.61103032) and National Key Technology R&D Program of the Ministry of Science and Technology of China (No2013BAH01B03). The author would also thank the other developers and QA personnels of State Street Corporation that are involved in the internationalization and localization of PAM.

References

[1] B. Esselink, *A practical guide to localization*. John Benjamins Publishing Co, 2000, vol. 4.

[2] E. Uren, R. Howard, and T. Perinotti, *Software internationalization and localization: an introduction*. Van Nostrand Reinhold New York, 1993.

[3] C. Petzold, *Programming Windows®*. Microsoft Press, 2010.

[4] F. Balmas, "Using dependence graphs as a support to document programs," in *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*. IEEE, 2002, pp. 145–154.

[5] F. Balmas, "Displaying dependence graphs: a hierarchical approach," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 3, pp. 151–185, 2004.

[6] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." Technical Report. [Online]. Available: http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

[7] A. Langville and C. Meyer, *Google's PageRank and beyond: The science of search engine rankings*. Princeton University Press, 2009.

[8] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating need-to-translate constant strings for software internationalization," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 353–363.

[9] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Transtrl: An automatic need-to-translate string locator for software internationalization," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 555–558.

[10] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating need-to-translate constant strings in web applications," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 87–96.

[11] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *MSR*, 2010, pp. 11–20.

[12] D. Port, A. P. Nikora, J. Hihn, and L. Huang, "Experiences with text mining large collections of unstructured systems development artifacts at jpl," in *ICSE*, 2011, pp. 701–710.

[13] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie, "Xiao: tuning code clones at hands of engineers in practice," in *ACSAC*, 2012, pp. 369–378.