Identifying Change in Dynamic Link Library COTS Components When Source Code is Not Available

Jiang Zheng[1], Laurie Williams[1], Brian Robinson[2], Karen Smiley[2]
[1] *Department of Computer Science, North Carolina State University, Raleigh, NC, USA*
{jzheng4, lawilli3}@ncsu.edu
[2] *ABB Inc., US Corporate Research, Raleigh, NC, USA*
{brian.p.robinson, karen.smiley}@us.abb.com

Abstract

Software products are often configured from commercial-off-the-shelf (COTS) components. When new versions of COTS components are available, efficiently regression testing component-based applications is challenging. Component vendors often do not provide source code or change information. Based on our prior work, we are studying the solution to regression testing COTS-based applications that use components of dynamic link library (DLL) files. A binary change identification tool that analyzes DLL component binaries is being developed. The tool was used to examine differences in four releases of an internal ABB software component of DLL file format written in C. The current tool identified all changes but had false positives when two versions were not built by the same linker.

1. Introduction

An increasing number of companies incorporate a variety of commercial-off-the-shelf (COTS) components in their products. Users of these components often need to conduct regression testing to determine if a new component or new version of a component will cause problems with their existing systems. However, users of COTS components often have access only to the binary files and reference documents. Most existing regression test selection (RTS) processes rely on source code [1-3], and therefore are not suitable when source code is not available for analysis.

In our prior research, we have proposed an effective RTS process called *Integrated - Black-box Approach for Component Change Identification* (I-BACCI) for COTS-based applications by static binary change identification and the firewall analysis [5] RTS technique [6]. Tool support for library (.lib) components has been developed [6]. We are continuing this research to *evolve a stable and efficient RTS process with supporting tools for COTS-based applications that use DLL components*.

A key step of our RTS process is to identify changes between the binary code of the new release and the previously-tested version. Due to the significant difference between .lib and .dll file formats, a new tool is being developed to identify the changes between DLL binaries. Using this information, code-based RTS techniques can then be applied to select regression tests. We present our DLL binary change identification algorithm and report on the results of a case study.

2. DLL Binary Change Identification Tool

The inputs to the tool are the binary files of the old and new DLL components. The output of the tool includes: (1) the call-graph of each exported component function; (2) a full binary code representation of each exported component function, including all sub-functions and data that might be called by that exported function; and (3) a differencing report on the two releases of the DLL.

Compared to BMAT [4], Microsoft's internal tool, the purpose of our tool is for regression test selection instead of stale profile propagation [4]. Our tool is able to generate call-graphs for exported component functions, such that users of the component know which exported component functions are affected by which function and/or data changes. Also, unlike .lib file, which is in Common Object File Format (COFF), a DLL binary is in the Portable Executable (PE) file format [1]. Some characteristics of the PE format make the change identification and call-graph generation more complex and difficult.

The algorithm examines the DLL binaries from coarse to fine granularity step by step. First, the tool invokes DUMPBIN[1] Version 8 to translate the illegible binary library files into readable plain text files. This file-level granularity step assumes that file names do not change between releases. Then a file reader automatically scans the DUMPBIN output and loads useful information, such as instructive information in file header, section headers, export table and import table, into a predefined data structure which is constructed according to the PE file format specification. File and section information is ready to facilitate future lookup after this section-level step.

[1] MSDN Library - Visual Studio .NET 2003

The next finer granularity is in function/data-level. Binary code of functions and data are stored consecutively in *.text* section and data sections (*.rdata*, *.data*, *.idata*, etc.), respectively. However, only names of exported component functions are available. Other functions and all data have to be labeled by their *start virtual addresses* (SVA). We abstract function and data as the same class (`DLLFunctionData`) with the following attributes: SVA, *end virtual address* (EVA), raw binary code, and relocation list. The relocation table is read from the *.reloc* section and then converted into a Hashtable called *relocation index*. For each key-value pair in the relocation index, the key is a *calling virtual address* (CVA) where the control flow jumps to another function or data, and the SVA of the function or data being called (a.k.a. *target virtual address* (TVA)) is stored as the value. Function CVA and TVA can also be calculated according to the position of each *call* instruction and the address offset following each *call* instruction, respectively. Because only binary code is available instead of assembly code, the tool searches opcode `E8` and `E9` which represent "call near" in the Intel instruction set[2] to locate the position of each function call. After finding all functions and data SVAs, an array in `DLLFunctionData` type is constructed and the raw code of the *.text* and data sections is decomposed into separate functions or data. Then function/data call-graphs and full code representation for all exported component functions can be generated recursively following the calling track. A few steps that remove trivial bytes are also conducted during processing of this level. For example, most raw code of functions/data is followed by a few useless bytes (e.g. `90`, `CC`) for the purpose of alignment.

Further instruction-level comparisons can be conducted after the function/data level if the full code representations of an exported component function in two releases are still different. For example, false positives may be caused by register allocation changes from build to build. After all of the above steps, a report on differencing of exported component functions will be generated. We can use this report to identify affected application code and then select proper regression test cases, similar to our prior work [6]. This tool has several limitations, similar to our prior tools [6].

3. Formative Case Study

We are in the process of assessing the efficacy of the tool. The case study is an internal ABB software component of DLL file format written in C. Each release contains one DLL file with size of about 100 kilobytes. The results of applying our tool on this component are shown in Table 1. The tool worked well when comparing two releases built by same linker. However, two compared releases have significant binary differences if they are built by different compilers or linkers.

Table 1: Case Study Results

Metrics	Comparisons		
	1 vs 2	**2 vs 3**	**3 vs 4**
Built by same linker?	No	Yes	Yes
True positive ratio [6]	100%	100%	100%
False positive ratio [6]	60%	0%	0%

4. Conclusions and Future Work

This work in progress reports on development of a tool that identifies the changes and change impact between releases of DLL binaries. The result of the case study shows the potential of this tool. The tool can be most accurate when there are small incremental changes between revisions. However, validation of both the tool and RTS process will require more industrial case studies and data collection and further RTS analysis. We will keep on improving the accuracy of binary change identification, including investigating more complex situations, such as interrelated components and dynamic jump/call issue. Eventually, end-to-end automation will be implemented to enhance the efficiency and convenience of the whole process.

Acknowledgements

This research is supported by a research grant from ABB Corporate Research.

References

[1] Bible, J., Rothermel, G., and Rosenblum, D., "A Comparative Study of Course- and Fine-Grained Safe Regression Test-Selection Techniques," *ACM Transactions on Software Engineering and Methodology*, 10(2), 2001, pp. 149-183.

[2] Graves, T. L., Harrold, M. J., Kim, Y. M., Porter, A., and Rothermel, G., "An Empirical Study of Regression Test Selection Techniques," *ACM Transactions on Software Engineering and Methodology*, 10(2), 2001, pp. 184-208.

[3] Rothermel, G. and Harrold, M., "Analyzing regression test selection techniques," *IEEE Trans. on Software Engineering*, 22(8), 1996, pp. 529-551.

[4] Wang, Z., Pierce, K., and McFarling, S., "BMAT: A Binary Matching Tool for Stale Profile Propagation," *The Journal of Instruction-Level Parallelism*, Vol. 2, 2000.

[5] White, L. and Abdullah, K., "A Firewall Approach for the Regression Testing of Object-Oriented Software," in *Software Quality Week*, San Francisco, 1997.

[6] Zheng, J., Robinson, B., Williams, L., and Smiley, K., "Applying Regression Test Selection for COTS-based Applications," Proceedings of the 28th IEEE International Conference on Software Engineering (ICSE'06), Shanghai, P. R. China, May 2006, pp. 512-521.

[2] http://developer.intel.com/design/pentium4/manuals/index_new.htm