Title	保守性の高いファームウェア開発を支援するプログラミング環境の構築［課題研究報告書］
Author(s)	栗林，大樹
Citation	
Issue Date	2016-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/13609
Rights	
Description	Supervisor: 田中 清史，情報科学研究科，修士

Construction of Programming Environment to Assist in Developing Highly Maintainable Firmware

Hiroki Kuribayashi (1210904)

School of Information Science,
Japan Advanced Institute of Science and Technology

February 6, 2016

Keywords: Static code analyzer, maintainability, readability, C language, preprocessor, firmware.

1 Introduction

In developing embedded systems, multi-functionality and short-term development are required to provide competitive products. Therefore, the embedded software is getting more sophisticated and is becoming enlarged. To shorten the development period, existing firmware is reused with a slight modification or additional functions. In the modification, macros including conditional branches/switches which can cope with differences in versions of the products are used. This degrades the maintainability and readability of the source codes. Therefore, some analyzing tools of source codes, especially ones with macros before preprocessing, are desired to detect code fragments which are against coding rules.

In this research, I proposes a method of analyzing source codes before preprocessing and builds a static code analyzer implementing the method. Then, the tool is applied to actual device drivers for embedded systems and the results are shown and discussed.

2 Related Work

This chapter first describes several standards of C-language (C89, C90, C99, C11, and ANSI), and then shows several coding rules (ISO/IEC 25010, JIS X 012901, MISRA-C:{1998, 2004, 2012}, SEI CERT C Coding Standard, Recommended C Style and Coding Standards, and GNU coding standards) which are widely accepted in embedded systems development. Next, several existing software analysis tools are introduced (QA·C, Coverity, CODESONAR, etc.) Then, well-known lexical analysis tools and parsing tools are shown (lex/yacc, flex/bison, ML-yacc/ML-lex, Happy, JavaCC, and Parsec). These are used in implementing static code analysis tools.

3 Design of Static Code Analyzer

In this research, I implement a checking tool for source codes written in C language. This chapter illustrates the design of the tool. The tool consists of four functions: lexical analyzer, processor for preprocessor directives, parser, and checker. The functions are explained individually in each section. In the section for lexical analyzer, rules for tokens including preprocessor directives are given. The section for preprocessor directives describes how to extract preprocessor directives from a series of tokens and then add preprocessor information to the remaining tokens. The section for the syntax parser deals with means of parsing C-language codes with an LL-parser and restrictions about macro statements which the parser can treat. In the section for carrying out tests, the ways of testing token, syntax, and types is described.

4 Test Items

This chapter describes test items of the static code analyzing tool designed in this research. The analyzer mainly targets warnings for three types of primary factors, that is, token, syntax, and types. Test items corresponding to these three types are explained in detail. The first factor, token, is categorized into eight items: unstandardized escape sequence, carriage return in a literal, goto/continue statements, octal constant number, #line macro, #undef macro, relative path name for a file in #include macro, and absolute path name for a file in #include macro. As for the factor, syntax, twenty-two items are considered: declaration or definition of names starting by an underscore, omission of parentheses for conditional expression, mixed binary operators, use of comma operators, etc. Finally, for the factor, types, nine items are defined: inconsistency of types of results in conditional expression, inconsistency of types between a loop counter and a compared variable, application of unary minus operator to unsigned integers, inconsistency of types between left and right expressions of binary operators, etc. All these items are shown with concrete examples.

5 Evaluation

This chapter reports the results of the analysis for source codes by the static code analyzer developed in this research. The target source codes of the analysis are from the drivers in Linux kernel I^{2}C. These drivers are used in the embedded Linux. As the results of the analysis for 128 source files consisting of totally 67,960 steps, 1,383 warnings are reported. Among them, warnings about usage of basic types and ones about not-parenthesized bodies of if, for or while structure are most frequent. The details about the macros that cannot be analyzed by conventional tools and the detected warnings are shown with the corresponding code fragments in this chapter.

6 Conclusion

In this research, for static code analysis of C-language codes, a method of analyzing source codes before preprocessing is proposed. The proposed method is implemented as a static code analyzer which supports programmers in describing source codes with high maintainability and readability. The analyzer is applied to device drivers which are actually used in embedded systems. As the results, it is confirmed that the code fragments impeding maintainability and readability can be detected.

To enhance the analyzer, it is necessary to implement additional test items. In addition, there is room for improving the analysis method, for example, checking the functions' or variables' values in compile time by expanding macros which do not include branches. Furthermore, although the current tool analyzes source codes per file, it would be useful to check the consistency between files based on the obtained information from each file.