Multidimensional Static Code Analysis Method for Automotive Software

Qiujun Zhao
China Automotive Technology & Research Center Co. Ltd;
CATARC Software Testing (Tianjin) Co. Ltd
Intelligent Communication Research Department
Tianjin, China
zhaoqiujun@catarc.ac.cn

Shuhua Zhou
China Automotive Technology & Research Center Co. Ltd;
CATARC Software Testing (Tianjin) Co. Ltd
Intelligent Communication Research Department
Tianjin, China
zhoushuhua@catarc.ac.cn

Ziwei Tian
China Automotive Technology & Research Center Co. Ltd;
CATARC Software Testing (Tianjin) Co. Ltd
Intelligent Communication Research Department
Tianjin, China
tianziwei@catarc.ac.cn

Yu Su
China Automotive Technology & Research Center Co. Ltd;
CATARC Software Testing (Tianjin) Co. Ltd
Intelligent Communication Research Department
Tianjin, China
suyu@catarc.ac.cn

2024 IEEE 4th International Conference on Data Science and Computer Application (ICDSCA) | 979-8-3503-6823-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDSCA63855.2024.10859504

***Abstract*—The accelerated development of automotive intelligence and connectivity has raised higher requirements for the quality and security of automotive software code. In this paper, a multi-dimensional static code analysis method is proposed by integrating lexical analysis with rule verification. This method employs Abstract Syntax Tree (AST) technology, data flow and control flow analysis techniques, combined with parallel analysis technology, to achieve comprehensive, accurate, and efficient static analysis of source code. Through comparative experimental validation, the static code analysis tool designed in this paper exhibits excellent performance in detection accuracy, efficiency, compatibility, and reliability, providing strong support for quality control and risk management in the software development process.**

Keywords—AST, Data Flow, Control Flow, Multidimensional Static Code Analysis, Test tool, Automotive Software

I. Introduction

With the rapid advancement of automotive intelligence and networking, the quality and safety of software code have become crucial considerations.[1] Although traditional code test and analysis tools can assist developers in identifying code defects to some extent, their built-in static code analysis methods are often limited to single dimensional checks, such as syntax validation or security vulnerability scanning. This single perspective analysis is difficult to comprehensively and profoundly reveal the overall quality of the code, especially in the face of complex and hidden non functional issues such as memory leaks, buffer overflows, null pointer references, etc.[2] These tools are often inadequate and difficult to effectively detect these problems.[3] In the current market, automotive code checking tools such as Code Climate, Codacy, FindBugs, and Checkstyle each have their own advantages: Code Climate provides comprehensive code quality assessment and visual display, compatible with multiple programming languages and IDE extensions; Codacy focuses on checking code quality, syntax compliance, and functional availability, but SaaS service integration has limitations; FindBugs used to exist as a lightweight Java static analysis tool, but it has now ceased maintenance;[4] Checkstyle, on the other hand, focuses on code formatting and style specifications, with relatively limited ability to detect bugs.[5]

To enhance the accuracy and efficiency of automotive code inspection, this paper proposes a multi-dimensional static code analysis method that integrates lexical analysis with rule verification. This method utilizes Abstract Syntax Tree (AST)[6] technology to trace the flow of data within programs, precisely capturing issues such as data overflow. Simultaneously, it combines control flow analysis[7] to deeply dissect program execution paths, effectively revealing potential logical errors like null pointer references. Furthermore, this method innovatively introduces parallel analysis technology, employing dedicated parsers to process different MISRA C rules in parallel, significantly reducing the analysis cycle. Additionally, it integrates test tools to generate comprehensive test reports. This parallel processing mechanism not only accelerates the detection process but also enhances the capability to detect deep-seated and complex logical errors, providing more detailed and accurate code quality feedback for quality control and risk management in the software development process, thereby laying a solid foundation.

II. Static Code Analysis Method

A. Multidimensional Static Code Analysis Method

In order to improve the efficiency of code detection, this article proposes a method that can screen out code errors, defects, or non compliant issues in the system without the need for comprehensive research on all details of the system. This method integrates AST, data flow, control flow, and parallel processing technologies, and constructs multiple unit modules to optimize the detection efficiency of code analysis. The multidimensional static code analysis method is divided into four units: code compilation, text syntax analysis, basic parser verification, and sub-parser detection. The relationship between the units is shown in Figure 1.

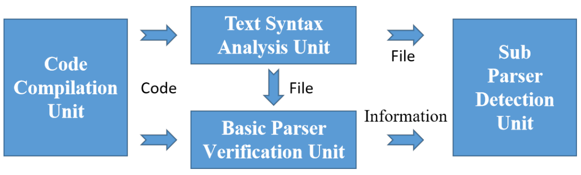
Code Compilation Unit
Code
Text Syntax Analysis Unit
File
Basic Parser Verification Unit
File
Information
Sub Parser Detection Unit

Figure 1. Schematic diagram of the units structure of multi-dimensional static code analysis method

(1) Code Compilation Unit

Compile the source code to check for syntax errors. If the compilation fails, the error messages will be generated and the errors must be corrected before attempting to recompile. Upon successful compilation, proceed to the subsequent static analysis steps. The specific steps are shown in Figure 2.

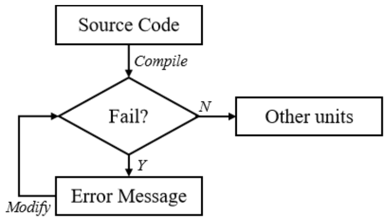

Figure 2. Flowchart of the code compilation unit

(2) Text Syntax Analysis Unit

Perform syntax analysis on successfully compiled source code to generate code structure files.[8] Figure 3 shows the implementation process and explains it in detail:

Step1: Read the successfully compiled source code text from the input buffer. Use predefined lexical rules to identify word boundaries and split the source code into lexical units (Tokens).

Step2: Classify the Tokens and identify their respective categories (such as keywords, identifiers, constants, operators, etc.).

Step3: Assign corresponding tags to each classified Tokens and construct a linear sequence of Tokens. During this process, irrelevant characters such as comments and spaces will be removed from the source code to ensure the accuracy of the analysis.

Step4: If any grammar errors are found during the analysis process, summarize the error messages and provide corresponding error prompts and repair suggestions.

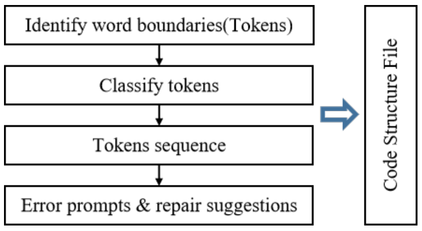
Identify word boundaries(Tokens)
Classify tokens
Tokens sequence
Error prompts & repair suggestions
Code Structure File

Figure 3. Flowchart of the text syntax analysis unit

(3) Basic Parser Verification Unit

Based on the code structure file, perform parser verification on the source code to discover potential code logic errors. The specific implementation process is shown in figure 4 and explained below:

Step1: Organize classified Tokens into an AST according to predefined syntax rules. The formula is as follows:

$$\mathrm{AST} = \mathrm{f}(\mathrm{Tokens}) \tag{1}$$

Where f represents the function that converts Tokens into the corresponding AST.

Step2: Traverse the AST and perform data flow and control flow analysis. Data flow analysis checks the definition, usage, and assignment of variables to ensure that data flow meets expectations.[9] Formula (2) encapsulates the core concept of data flow analysis.

$$\mathrm{Dflow}(\mathrm{AST}) = \mathrm{Def}(\mathrm{AST}) \cup \mathrm{Use}(\mathrm{AST}) \tag{2}$$

Where Def(AST) represents the set of definitions in AST, Use(AST) represents the set of uses for AST, and DFlow(AST) denotes the result of the data flow analysis.

Control flow analysis determines the program execution path, checks the correctness of control structures (such as loops and conditional statements), and detects issues such as dead code and unreachable paths. The structure of the control flow graph is delineated by the following formula:

$$\mathrm{CFG} = (\mathrm{Nodes}, \mathrm{Edges}) \tag{3}$$

Where CFG stands for the Control Flow Graph, Nodes represent the basic blocks or statements in the code, and Edges depict the control flow between these nodes.

Step3: Pass the analyzed AST (including data flow and control flow analysis results) to the symbol table for symbol analysis. Note that this step typically involves using the analysis results to update or query the symbol table, rather than directly outputting to it.

Step4: Handle error situations, provide error prompts and repair suggestions.

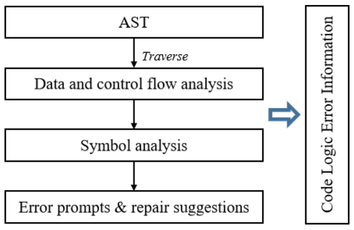
AST
Traverse
Data and control flow analysis
Symbol analysis
Error prompts & repair suggestions
Code Logic Error Information

Figure 4. Flowchart of the basic parser verification unit

(4) Sub Parser Detection Unit

Based on code structure files and code logic error information, develop corresponding sub parsers for different MISRA C rules[10] to detect violations of source code rules. Figure 5 shows the implementation process of this unit, which is detailed below:

Step1: Write corresponding sub parsers for each MISRA C rule, with each sub parser inheriting from the base parser.

Step2: The sub parser provides a unified interface to receive source code files and configuration files as inputs, and output analysis results and reports. The configuration file can specify the MISRA C rule entries to be checked.

Step3: The sub parser module summarizes the analysis results, collects information on rule violations, violation frequency, and severity, and displays the analysis results and recommendations.

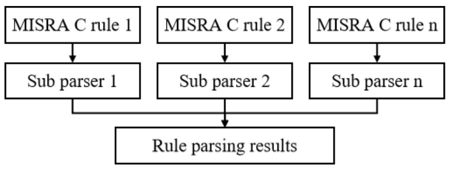

Figure 5. Flowchart of the sub parser detection unit

B. Tool Integration

In addition, based on the above static code, a memory and processor device are provided, wherein the memory stores a computer program, and the processor executes the steps of the multi-dimensional static code testing and analysis method when the computer program is executed. And a visual report generation unit is integrated into the test tool, which can summarize the final parsing results based on code logic error information and rule parsing results to generate visual reports. The report includes the following contents:

①Line number: Display the specific line number where the rule was violated or a defect was discovered.

②Violation rule number: Display the specific rule number violated.

③Defect severity level: such as fatal, severe, warning, etc.

④Defect information: Provide a detailed description of the defect and possible causes.

⑤ Context information: Provide the context of code snippets that violate rules or discover defects.

⑥Suggested repair plan: Provide suggested repair plans or best practices for each defect.

⑦ Scope of Impact Analysis: Analyze the impact of defects on system functionality, performance, or security.

⑧ Historical comparison: Compare the differences between current analysis results and historical results.

⑨Source code file path: Display the complete path of the source code file.

C. Evaluation Metrics

In order to fully assess the detection effectiveness of static code analysis tools, the following key metrics are used in this study:

(1) Violations Number

This metric measures the total number of potential problems or violations detected by the tool in the code base. It reflects the tool's sensitivity and ability to detect code quality.

(2) False Positives

False positives refer to instances of code that the tool incorrectly flags as violations but are not actually problematic. The level of false positives has a direct impact on developer trust and the efficiency of the tool.

(3) False Negatives

False Negatives are the actual offending code that the tool fails to detect. The level of False Negatives is directly related to the accuracy and completeness of the tool's detection capability.

By considering these metrics comprehensively, this study is able to comprehensively assess the performance of static code analysis tools in practical applications and provide data support for their improvement and optimization.

III. EXPERIMENTAL VERIFICATION

A. Experimental conditions

To verify the effectiveness and practicality of the multidimensional static code analysis method proposed in this paper, a detailed experimental plan was designed. The experiment selected C language program sets from two automobile manufacturers as the experimental objects, covering the logic code of automobile transmission ECU control and automobile inverter INV control, respectively, as code sets one and two. The specific information is shown in Table I.

TABLE I EXPERIMENTAL SUBJECTS

Code set	. c files	. h files	Code's lines
Code set 1	186	44	102671
Code Set 2	421	152	228544

The experimental environment configurations are shown in Table II.

TABLE II EXPERIMENTAL ENVIRONMENT

Configuration	Name	Model
hardware	processor	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 1.69 GHz
	Memory	16.0 GB
software	operating system	Windows 11 Professional, 64 bit

B. Experimental Results and Comparison

In the same experimental environment, a comparative analysis was conducted on the experimental subjects using the static code analysis device designed in this article and three other code detection tools (C++Test, Codacy, and FindBugs) for the above code set. The code set 1's experimental results are shown in figure 6 and table III.

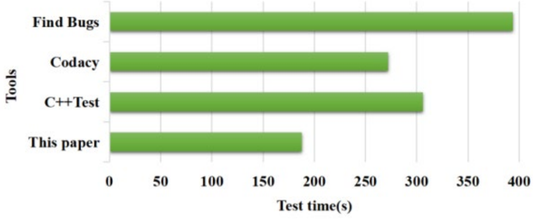

Figure 6. Code Set 1 Test Time

TABLE III CODE SET 1 TEST RESULTS

Tools	**This paper**	C++Test	Codacy	Find Bugs
Violations number	**677**	593	509	668
false positives	**15**	18	20	30
false negatives	**64**	151	237	88

Observing the test results of Code Set 1 in Table III, it was found that the total number of violations detected by the tool in this paper is 677, slightly lower than Find Bugs, but its false positives and false negatives were the lowest, proving that the device in this article has high detection accuracy in static code analysis. Comparing the test time in figure 6, it was found that the parsing of the device in this article only takes 204 seconds. When the total number of detected violations is similar, Find Bugs takes 394 seconds, which further proves that the detection device in this article has high detection efficiency.

The code set 2's experimental results are shown in figure 7 and table IV.

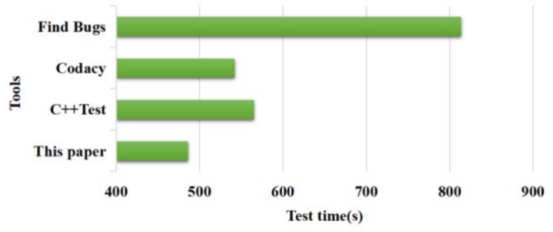

Figure 7. Code Set 2 Test Time

TABLE IV CODE SET 2 TEST RESULTS

Tools	This paper	C++Test	Codacy	Find Bugs
Violations number	**1202**	986	934	1067
false positives	**56**	124	166	208
false negatives	**300**	696	790	699

As shown in Table IV, the test tool designed in this paper performs best on Code Set 2, with a total of 1202 violations detected. The number of false positives and false negatives is as low as 56 and 300, respectively. Additionally, it boasts the shortest test time (figure 7). These fully demonstrates that the detection tool designed in this paper possesses high accuracy and efficiency.

Besides, to verify the compatibility and reliability of the device, tests are conducted on current mainstream operating systems and processor architectures, and the test results are shown in Table V.

TABLE V EQUIPMENT COMPATIBILITY AND RELIABILITY TEST RESULTS

Operatin g system	X86 architecture	X86 architecture	ARM architecture	ARM architecture
Windows	compatible	reliable	compatible	reliable
Linux	compatible	reliable	compatible	reliable
macOS	compatible	reliable	compatible	reliable

The detection results show that the test tool designed in this paper can be used normally in different operating systems and processor architectures, indicating that the tool has strong compatibility and reliability.

IV. CONCLUSION

The multidimensional static code analysis method and its application scenarios proposed in this paper have significant advantages and practical value. This method integrates lexical analysis and rule checking, AST technology, data flow and control flow analysis technology, and parallel analysis technology, which can deeply, comprehensively, and efficiently detect source code. The experimental results show that the static code analysis device designed in this paper outperforms traditional tools such as C++Test, Codacy, and FindBugs in terms of detection accuracy, efficiency, compatibility, and reliability. In the code sets test provided by two car manufacturers, this tool not only significantly reduced the number of false positives and false negatives, but also greatly shortened the test time. In addition, the tool demonstrates good compatibility and reliability across different operating systems and processor architectures, providing solid quality assurance for software development. Looking ahead to the future, we will continue to optimize and improve this method and its equipment to enhance the level of intelligence, conduct cross platform optimization, and expand functionality, in order to better meet the needs of software development in the context of automotive intelligence and networking, and provide more comprehensive, accurate, and efficient support for quality assurance and risk control in the software development process.

REFERENCES

[1] Fei Ding, Nan Zhang, Shengbo Li, Yougang Bian, En Tong, Keqiang Li. Overview of the Architecture and Key Technologies of Intelligent Connected Vehicle Road Cloud Collaboration System [J]. Journal of Automation, 2022, 48 (12): 2863-2885

[2] Rezvina S. Keep Code Review from Wasting Everyone's Time: Code Climate[J]. 2019.

[3] Gautam S. Comparison of Java Programming Testing Tools[J]. International Journal of Engineering Technologies and Management Research, 2020, 5(2): 66-76.

[4] Hovemeyer, D.; Pugh, W. Finding bugs is easy. ACM SIGPLAN Not. 2004, 39, 92–106.

[5] Yiu C C. Checkstyle for Legacy Applications[J].Itestra De. 2023.

[6] Mou, L., Jin, Z. (2018). TBCNN for Programs' Abstract Syntax Trees. In: Tree-Based Convolutional Neural Networks. SpringerBriefs in Computer Science.

[7] Springer, Singapore. Schaumont, P.R. (2013). Analysis of Control Flow and Data Flow. In: A Practical Introduction to Hardware/Software Codesign. Springer, Boston, MA.

[8] Wang Z ,Jabar A A M ,Jalis M M F .Cross-Disciplinary Analysis of the Syntactic and Lexical Features of Chinese Master Thesis Titles[J]. Theory and Practice in Language Studies,2024,14(9):2760-2772.

[9] Mogensen, T. (2017). Data-Flow Analysis and Optimisation. In: Introduction to Compiler Design. Undergraduate Topics in Computer Science. Springer, Cham.

[10] Bagnara, R., Bagnara, A., Hill, P.M. (2018). The MISRA C Coding Standard and its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software. In: Podelski, A. (eds) Static Analysis. SAS 2018. Lecture Notes in Computer Science, vol 11002. Springer, Cham.