2020 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)

Development of a System for Static Analysis of C ++ Language Code

Yu.V. Adamenko
Senior lecturer at the department of «Computer Software for Automated Systems»
Polytechnic Institute, Kurgan State University (KSU)
Kurgan, Russian Federation
jul_adamenko@mail.ru

A.A. Medvedev
Associate professor, Department «Computer Software for Automated Systems»
Polytechnic Institute, Kurgan State University (KSU)
Kurgan, Russian Federation
ark_medv@mail.ru

D.A. Karpunin
Bachelor's degree in Computer science
Polytechnic Institute, Kurgan State University (KSU)
Kurgan, Russian Federation
groupm20416@gmail.com

***Abstract*—The main goal of the system is to make it easier to standardize the style of program code written in C++.**

Based on the results of the review of existing static analyzers, in addition to the main requirements, requirements for the structure of stylistic rules were identified.

Based on the results obtained, a system for static analysis of the C++ language has been developed, consisting of a set of modules.

The system is implemented using the Python 3.7 programming language. HTML and CSS markup languages were used to generate html reports. To ensure that rules can be stored in the database, the MongoDB database management system and the pymongo driver module were used.

Keywords— analyzers, static analysis, style, regular expressions, argument keys, algorithm for finding stylistic errors

I. General Provisions

Static code analysis is software analysis performed without actual execution of the programs under study [1].

Static source code analysis is a set of methods to extract various information from the source code of a program for further analysis [2]. Static analysis allows to detect various functional and non-functional errors in the source code and optimize it [3, 4].

Studying the style of program code written by a programmer can help them improve their skills in writing high-quality and easy-to-read code [5, 6].

Correctly and well-formatted program code becomes more understandable both for the developer and for other users who will have to work with this code in the future. Based on this, the system can be implemented by groups of developers that need a common style of writing code in C++ [7 - 10].

Maintaining a user profile allows you to create sets of verification rules used for static analysis of projects.

Creating a profile with requirements for the code style and distributing it to developers will help you bring the code to a uniform style by checking the source codes of the project.

Static analysis allows you to find the main functional and non-functional errors and it is able to check compliance with rules and standards in writing a software code [11]. At the same time, static analyzers have acceptable requirements for computing resources of computers. Static analyzers can be used to search for errors related to logical and syntactic errors, or to analyze and study the programming style.

The relevance and importance of using static analyzers increases over time. This is due to the constant growth in the size of the source code of most modern applications [12, 13]. Programs become larger and more complex, the support of such developments also becomes more complex, and the amount of errors in the code directly depends on its size [14]. The larger the project, the more errors per 1000 lines of code it contains (table 1). This table is based on the source: "Program Quality and Programmer Productivity" (Jones, 1977), "Estimating Software Costs" (Jones, 1998) [15, 16].

TABLE I. Project size and typical error density

Project size in lines of code	Typical error density per 1000 lines
Less than 2 thousand lines	from 0 to 25 errors
From 2 to 64 thousand lines	from 0.5 to 50 errors
From 64-512 thousand lines	From 2 to 70 errors
512 or more thousand lines	From 4 to 100 errors

Static analysis is a widely used method in industrial development of software systems. It allows you to check automatically whether the code conforms to pre-defined programming rules [17-20].

II. Input and Output Documents

An input document is a document that has a specific form and contains data to be entered into the system.

The output document is a document that contains the final results of data processed by the system.

Projects and/or source codes in the C++ programming language, rules for collecting metrics, and rules for checking the code style are used as input documents to the program.

978-1-7281-6951-4/20/$31.00 © 2020 IEEE

Information about the analyzed source codes is passed as lists of paths to the project files in the user's file system.

The output documents in this system are:

1) final reports in the format supported by the analyzing system. Reports are generated on basis of the lists of stylistic comments found in the code during the static analysis of the project. Reports are generated on basis of the results of the static analysis of the project source code in C++;

2) visual representation of the unweighted oriented graph of the project is an output document formed if needed by the user. Visual representation is made by creating a graphical image with a representation of the structure of the relationship of source code files in the project analyzed by the static analysis system. The vertices (or nodes) of the graph are the project files, the edges connecting the vertices are directives to connect external header files and source code files. After the image is generated, it is saved in a folder with the static analysis system.

III. Main Functional Requirements

The system allows you to collect source code metrics according to the specified rules. Rules for collecting metrics can be stored as files in the user's file system, or in a database. The rules have the general form: "rule name, regular expression, < collection mode>, <description>". The definition of the metrics collection rules is given below:

1) the rule name defines the key in the dictionary that will be used to locate the container with metrics. If you need to collect metrics for individual files, but not for the entire project, the Manager will write the path of the file, from which the metrics were obtained, to the key using the " - " symbol;

2) a regular expression in Python serves as a rule for collecting metrics;

3) collection mode determines the type of metric container. You can collect metrics by quantity or collect instances of the necessary metrics. Collection mode is not a mandatory attribute and, according to the standard, the system must collect instances of the metrics. A metric container with instance collection can be easily converted to an iterable container for quantitative metric collection. The operation of converting an iterated container to an instance container is not possible after the metrics are collected;

4) the rule description is an optional attribute and is used to inform the user in detail about the type and kind of collected metric.

Collected metrics are exported to files in the folder "metrics". The system allows the user to create rules for collecting metrics easily by using the system itself. The system informs the user with examples of metrics, design rules, and provides an opportunity to test the operation of a rule created by the user on a test example. It provides the user with an opportunity to import and export rules to collect metrics.

The system generates reports based on the results of the analyses. The form of reports is selected by the user out of the list of possible forms. Possible formats for export are defined by the tuple of formats and the availability of corresponding export modules. You can generate reports in the form of an html page. The html page is formed according to a template that contains a table of the test results.

Exporting modules store various static methods of converting the reported information which is transmitted to them in accordance with the norms and standards inherent in a particular data storage format.

For example, when exporting a report to an html page, the exporting module gets information from reports and directs it in the necessary parts of the page, turning them into HTML language markup and forming a document based on a specific html page template. The template creation is accompanied by embedding CSS styles and JavaScript functions in the page template.

The system does not make changes in the analyzed projects and project source code files.

The system provides an opportunity to use various argument keys passed when the static analysis system is started. This allows the user to speed up their work with the console by calling the program with the necessary set of argument keys known to them, and then get a ready-made report on the static analysis of the project, minimizing the interaction with the system in the console. The system itself implies getting the value of argument keys and their processing by using special methods and classes.

The following argument keys are provided:

1) key to disable the system usage of the user’s profile;

2) argument key to use the user’s name at system startup (if there is no profile with this name, a temporary profile with the specified name will be created);

3) argument key of the full path to the analyzed project;

4) key to use the rules from the database without using the standard set of rules and rules from files;

5) key to use the metrics Manager and to collect metrics of the program code of the analyzed project;

6) argument key for the format of the final report on the conducted static analysis;

7) key to use the debugger and to call debugging messages from the static analysis system modules.

Other keys and argument keys that users need can be added by third-party developers.

The system provides for possibility of easy integration with the GUI. This requirement is partially met by using the argument keys described above. In this case, the GUI can be developed by third-party developers and users, and not only in Python, but also in other languages, such as C++. If you need a stronger modification of the static analysis system and integration with the GUI, you can use Python with various modifications of the source code static analyzer.

978-1-7281-6951-4/20/$31.00 © 2020 IEEE

The system organizes the static analysis of the projects and source code files passed to it. The static analysis focuses on checking the style and compliance with C++ code writing standards. This happens by reading files into memory and passing lines of its code to test modules, classes, and their methods. To determine the structure of the code and the used operators, methods, and functions, the analysis tools and regular expression-based rules are used to extract the desired structures from source code files.

IV. Description of the Algorithm

The static analysis system startup paths affect the further operation of the argument Manager, which monitors the parameters passed at system startup and detects indefinite parameters. All indefinite parameters will be detected later by using the system's console communication with the user.

The algorithm of creating a list of source code files and header files of the analyzed project consists in checking the directory containing the C++ project and in forming a list of paths to files in the cpp, hpp, and h formats. When the list is created, the paths are indexed and the file contents are superficially analyzed to find out connections of other files with the project source code. Having indices of files and lists of other files connected to them, you can compare them with indices of existing ones to get an unweighted oriented graph.

The algorithm of classifying names of variables, classes, and methods in C++ considers the use of English letters in upper- or lowercase, of figuers, and of the underscore character. When classifying the ID, the following things are important:

1) where the underscore character is located, at the beginning of the name, in the middle or at the end of the identifier;

2) whether the name begins with a capital letter, whether the entire name consists of capital letters, whether there are upper-case characters in the middle of the name;

3) the use of figures in the name identifier and their number.

After getting the list of files, the stylistic rules are loaded, and the search for stylistic errors in the code begins. During the stylistic analysis, a list of stylistic comments and errors is generated, and then it is added to the General list of analysis results.

The report generating algorithm converts the obtained results to the appropriate format that is inherent in a particular format for storing data in the user's file system. If an html report is generated, an html table is created to record all comments about stylistic errors and errors in the code. After the table with information for the report is created, it will be passed as an argument to the html page collector, which can also get arguments for the CSS style, html markup text, and JavaScript scripts. All this will eventually be written to an html file in the “reports” directory. The name of the report is set according to the current date and time.

Collecting metrics is the process parallel to the analysis, which reduces the time required to re-read source code files. Here the metric information, extracted during the static analysis according to the specified collection rules, is stored. When analyzing the project source code, lines are extracted from the code and passed to the metrics Manager. Before the analysis, the Metrics Manager loads the rules of collecting metrics. After collecting metrics, you can export them. Due to the large volume of collected metrics, the export is performed to a separate subdirectory of the "metrics" folder. The information from metric containers will be written to separate files, where the container key will act as the file name in the file system.

The algorithm of finding stylistic errors uses the developed structures, which can be presented in three variants:

- rules based on regular expressions;
- rules based on tags and commands for analyzing and defining the code style;
- hybrid rules based on the two previous rules.

Each variant of the listed structures has three common fields: the rule name, event category, and rule description. The rule name does not have to be unique. The event category defines the type to which an entry about the current error or comment will be assigned. The rule description contains the essence of this stylistic rule. Proceeding from the general structure of the rules, you need to develop a base class with common fields and methods that correspond to all the used rule categories. During the analysis the system iteratively applies active style rules. Depending on the used structure, the methods of working with the source code file and the algorithm of processing fields from the resulting rule structure change.

As an example, here is the structure of rules based on regular expressions:

- the name of the rule;
- the type of the rule;
- the regular expression of the general construction;
- the regular expression that matches the correct code style;
- the description of the rule.

Here is an example of writing a rule structure from the first category:

```
CommaRule, Style Warning," *, *", ", {1}", Comma should not be preceded by whitespace, but should be followed by one.
```

This rule specifies the requirement to use a single space after the comma and the exception of spaces before the comma sign. This is the rule in Vera++ in TCL, which is described as follows:

```
foreach f [getSourceFileNames] {
    foreach t [getTokens $f 1 0 -1 -1 {comma}] {
```

978-1-7281-6951-4/20/$31.00 © 2020 IEEE

```
        set line [lindex $t 1]
        set column [lindex $t 2]
        set preceding [getTokens $f $line
0 $line $column {}]
        if {$preceding == {}} {
        report $f $line "comma should not
be preceded by whitespace"
        } else {
        set lastPreceding [lindex [lindex
$preceding end] 3]
        if {$lastPreceding == "space"} {
report $f $line "comma should not be
preceded by whitespace"}
        }
        set following [getTokens $f $line
[expr $column + 1] [expr $line + 1] -1 {}]
        if {$following != {}} {
            set firstFollowing [lindex
[lindex $following 0] 3]
            if {$firstFollowing != "space"
&& $firstFollowing != "newline" &&
                !($lastPreceding ==
"operator" && $firstFollowing ==
"leftparen")} {
report $f $line "comma should be followed
by whitespace"
} } } }
```

The structure of the rules based on tags and commands:

- the name of the rule;
- the type of the rule;
- the list of tags;
- commands for checking lines of code;
- the description of the rule.

The structure of hybrid rules based on the previous two rule groups:

- the name of the rule;
- the type of the rule;
- the regular expression of the general construction;
- commands for checking lines of code;
- the description of the rule.

Style rules can be loaded from files stored in the user's file system, or from the database. Loading rules is controlled by the system control classes.

V. Conclusion

Initially developed software provides only a console interface, which, however, does not impose restrictions on integration into the GUI system. The graphical user interface can be developed within the modules of the static analyzer system itself by modifying it or creating other modules that implement the GUI.

The implementation of argument keys for starting a static analysis system allows you to build a graphical user interface independently from the system itself and does not impose restrictions on the choice of programming language for implementing the GUI. You can develop a graphical interface that allows the user to specify parameters of static analysis, and a program with a graphical interface will generate keys and launch the static analysis system, passing the generated arguments to it.

The development of graphical interface prototypes provides an opportunity to visually display possible graphical interfaces of a static analysis system by using graphical images and form layouts.

Here you can place fields to enter the project path, user selection (if necessary), and items for selecting system startup parameters.

References

[1] Definition of static code analysis, https://ru.wikipedia.org/wiki/Static_analysis_code (free access).

[2] S. V. Yulianto and I. Liem, "Automatic grader for programming assignment using source code analyzer," 2014 International Conference on Data and Software Engineering (ICODSE), Bandung, 2014, pp. 1-4.

[3] Q. Ashfaq, R. Khan and S. Farooq, "A Comparative Analysis of Static Code Analysis Tools that check Java Code Adherence to Java Coding Standards," 2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE), Islamabad, Pakistan, 2019, pp. 98-103.

[4] R. Szalay, Z. Porkoláb and D. Krupp, "Towards Better Symbol Resolution for C/C++ Programs: A Cluster-Based Solution," 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), Shanghai, 2017, pp. 101-110.

[5] H. Prähofer, F. Angerer, R. Ramler and F. Grillenberger, "Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application," in IEEE Transactions on Industrial Informatics, vol. 13, no. 1, pp. 37-47, Feb. 2017.

[6] Q. Deng and D. Jin, "Static analysis intermediate file analysis optimization strategy," 2015 8th International Conference on Biomedical Engineering and Informatics (BMEI), Shenyang, 2015, pp. 705-709.

[7] S. M. Alnaeli, M. Sarnowski, M. S. Aman, K. Yelamarthi, A. Abdelgawad and H. Jiang, "On the evolution of mobile computing software systems and C/C++ vulnerable code: Empirical investigation," 2016 IEEE 7th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON), New York, NY, 2016, pp. 1-7.

[8] D. Guamán, P. A. Quezada-Sarmiento, L. Barba-Guaman and L. Enciso, "Use of SQALE and tools for analysis and identification of code technical debt through static analysis," 2017 12th Iberian Conference on Information Systems and Technologies (CISTI), Lisbon, 2017, pp. 1-7.

[9] G. Díaz and J.R. Bermejo "Static analysis of source code security: Assessment of tools against {SAMATE} tests" Inf. Softw. Technol. vol. 55 no. 8 pp. 1462-1476

[10] Y. Takhma, T. Rachid, H. Harroud, M. R. Abid and N. Assem, "Third-party source code compliance using early static code analysis," 2015 International Conference on Collaboration Technologies and Systems (CTS), Atlanta, GA, 2015, pp. 132-139.

[11] A. Razzaq K. Latif H. F. Ahmad A. Hur A. Anwar and P. C. Bloodsworth "Semantic security against web application attacks" Information Sciences vol. 254 pp. 19-38.

[12] A. Braga, R. Dahab, N. Antunes, N. Laranjeiro and M. Vieira, "Understanding How to Use Static Analysis Tools for Detecting Cryptography Misuse in Software," in IEEE Transactions on Reliability, vol. 68, no. 4, pp. 1384-1403, Dec. 2019.

978-1-7281-6951-4/20/$31.00 © 2020 IEEE

[13] A. Imparato, R. R. Maietta, S. Scala and V. Vacca, "A Comparative Study of Static Analysis Tools for AUTOSAR Automotive Software Components Development," 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Toulouse, 2017, pp. 65-68.

[14] R. Haas, R. Niedermayr, T. Röhm and S. Apel, "Recommending Unnecessary Source Code Based on Static Analysis," 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Montreal, QC, Canada, 2019, pp. 274-275.

[15] J. Capers, “Estimating Software Costs”, 2007, p. 644 стр

[16] J. Lowell, “Program Quality and Programmer Productivity”,1978, p. 292.

[17] G. Carrozza, M. Cinque, U. Giordano, R. Pietrantuono and S. Russo, "Prioritizing Correction of Static Analysis Infringements for Cost-Effective Code Sanitization," 2015 IEEE/ACM 2nd International Workshop on Software Engineering Research and Industrial Practice, Florence, 2015, pp. 25-31.

[18] A. F. Maskur and Y. Dwi Wardhana Asnar, "Static Code Analysis Tools with the Taint Analysis Method for Detecting Web Application Vulnerability," 2019 International Conference on Data and Software Engineering (ICoDSE), Pontianak, Indonesia, 2019, pp. 1-6.

[19] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia and M. Vieira, "Benchmarking Static Analysis Tools for Web Security," in IEEE Transactions on Reliability, vol. 67, no. 3, pp. 1159-1175.

[20] A. Vetrò, "Using automatic static analysis to identify technical debt," 2012 34th International Conference on Software Engineering (ICSE), Zurich, 2012, pp. 1613-1615.

978-1-7281-6951-4/20/$31.00 © 2020 IEEE