УДК 004.45
DOI: 10.18523/2617-3808.2020.3.50-55

S. Sosnytskyi, M. Glybovets, O. Pechkurova

STATICAL AND DYNAMICAL SOFTWARE ANALYSIS

The development of software built with quality has become an important trend and a natural choice in many organisations. Currently, methods of measurement and assessment of software quality, security, trustworthiness cannot guarantee safe and reliable operations of software systems completely and effectively.

In this article statistical and dynamical software analysis methods, main concepts and techniques families are overviewed.

The article has an overview of why combination of several analysis techniques is necessary for software quality and examples how static and dynamical analysis may be introduced in a modern agile software development life cycle.

As a summary of techniques for software analysis, represented on Table 1, due to the computability barrier, no technique can provide fully automatic, robust, and complete analyses. Testing sacrifices robustness. Assisted proving is not automatic (even if it is often partly automated, the main proof arguments generally need to be human provided). Model-checking approaches can achieve robustness and completeness only with respect to finite models, and they generally give up completeness when considering programs (the incompleteness is often introduced in the modeling stage). Static analysis gives up completeness (though it may be designed to be precise for large classes of interested programs). Last, bug finding is neither robust nor complete. Another important dimension is scalability. In practice, all approaches have limitations regarding scalability, although these limitations vary depending on the intended applications (e.g., input programs, target properties, and algorithms used).

Already implemented code could be analysed in a continuous integration environment by a tool like SonarQube. Properly configured metrics and quality gates provide fast and detailed feedback on incremental changes starting from development machine till highload enterprise production environments. Software analysis helps to improve quality and development speed in Agile development life cycle with reasonable cost.

Keywords: Software Static Analysis, Software Dynamical Analysis, Testing, Quality Assurance, SQALE Mode, Continuous Code Analysis, SonarQube.

Introduction

Each engineering discipline has the main question to answer, it is whether the design will work as planned. This question is relevant whether we develop microchips, bridges or spaceships. The answer comes from the analysis of structures, using knowledge of nature about the projects. For example, when we design a bridge, we analyze how different forces (such as gravity, wind, and vibration) are applied and whether the structure is strong enough to withstand of nature events. The same question applies to computer software. We want to make sure that software application works as intended. We want to ensure that the designed software does not fail in the event of an unexpected termination. If the software operated in an internet environment we want to make sure that it will not be hacked and private data be espoused. Regarding functions requirements, we want to be sure that software realizes its functional purpose.

However, there is a significant difference between software analysis and analysis of other types of engineering structures. The computer executes the software according to the language values and not the nature. A computer is simply a tool that blindly executes software exactly as it is designed. Any execution behavior that deviates from our intent could lead to a failure. So, to answer the question of whether software design will work as we intended, we need knowledge on how we can analyze the meaning of a software source. There is a formal definition of software behavior, which is determined by the meanings, semantics of the language of its source [5].

1. Software Analysis Challenges

Software analysis can be applied wherever understanding program semantics is important or beneficial. Developers can use software analysis for quality assurance, errors allocation and better design

© *S. Sosnytskyi, M. Glybovets, O. Pechkurova,* 2020

decisions. Software maintainers can use program analysis to understand legacy software structures for safe changes introduction. System security can use program analysis to proactively monitor malicious code semantics. Software that operates with data can use software analysis for the performance improvements. Language processors such as compilers need program analysis to translate the input programs into optimized code [2].

Though the benefits of program analysis are obvious, building a cost-effective program analysis is not trivial, since computer programs are complex and could be very large. For example, the number of lines of smartphone applications frequently reaches over half a million, not to mention larger software such as web browsers or operating systems, whose source sizes are over ten million lines. With semantics, the situation is much worse because a program execution is highly dynamic [5]. Given that software is in charge of almost all infrastructures in life the need for cost-effective program analysis technology is greater than ever before. We have already experienced a sequence of accidents due to not identified mistakes in software. The long list includes accidents like the large-scale Twitter outage (2016), the Heartbleed bug (2014) and many more prominent software accidents.

However building error-free software may be unlikely feasible at least within reasonable costs and time, cost-effective ways to reduce errors numbers are on highest demand.

1.1. Families of Software Analysis Techniques

1.1.1. Testing: Checking a Set of Finite Executions

When trying to understand how a system behaves, often the first idea that comes to mind is to observe the executions of this system. In the case of a program that may not terminate and may have infinitely many executions, it is of course not feasible to fully observe all executions. Therefore, the testing approach observes only a finite set of finite program executions [5].

Testing has the following characteristics:

- Easy to automate
- Not robust
- Complete since a failed testing run will produce an execution that is incorrect

1.1.2. Assisted Proof: Relying on User-Supplied Invariants

This approach is followed by machine-assisted techniques. This means that users may be required to supply additional information together with the program to analyze [3].

Machine-assisted techniques have the following characteristics:

- They are not fully automatic and often require the most tedious logical arguments to come from the human user;
- In practice, they are robust with respect to the model of the program semantics used for the proof, and they are also complete up to the abilities of the proof assistant to verify proofs.

1.1.3. Model Checking: Exhaustive Exploration of Finite Systems

Another approach focuses on finite systems, that is, systems whose behaviors can be exhaustively enumerated, so as to determine whether all executions satisfy the property of interest.

Model checking has the following characteristics:

- Automatic;
- Robust and complete with respect to the model.

An important caveat is that the verification is performed at the model level and not at the program level.

1.1.4. Conservative Static Analysis: Automatic, Robust, and Incomplete Approach

Instead of constructing a finite model of programs, static analysis relies on other techniques to compute conservative descriptions of program behaviors using finite resources. The core idea is to finitely over-approximate the set of all program behaviors using a specific set of properties, the computation of which can be automated [4].

Static analysis approaches have the following characteristics:

- They are automatic;
- They are produces robust results, as they compute a conservative description of program behaviors, using a limited set of logical properties;
- They are incomplete because they cannot represent all program properties and enforce termination of the analysis even if the program has infinite executions.

While a static analysis is incomplete in general, it is often possible to design a robust static analysis that gives the best possible answer on classes of interesting input programs.

1.1.5. Bug Finding: Error Search, Automatic, Unsound, Incomplete

Some automatic program analysis tools sacrifice not only completeness but also robustness. The main motivation to do so is to simplify the design and implementation of analysis tools and to provide lighter-weight verification algorithms. The techniques

used in such tools are often similar to those used in model checking or static analysis, but they relax the robustness objective. Such tools are usually applied to improve the quality of noncritical programs at a low cost.

Bug-finding tools have the following characteristics:
- Automatic;
- Neither robust nor complete, instead.

Table 1

An overview of program analysis techniques

Techniques	Automatic	Robust	Complete	Object	Execution
Testing	Yes	No	Yes	Program	Dynamic
Assisted proving	No	Yes	Yes/No	Model	Static
Model checking of finite-state model	Yes	Yes	Yes	Finite Model	Static
Model checking at program level	Yes	Yes	No	Program	Static
Conservative static analysis	Yes	Yes	No	Program	Static
Bug finding	Yes	No	No	Program	Static

1.2. Software Metrics

A software metric is a measure of a property of software or its specifications. Quantitative measurements are important in all areas, computer scientists constantly trying to introduce similar approaches to software development. The goal is to obtain objective, reproducible and quantifiable indicators that can contain numerous valuable programs for budget planning and planning, cost estimation, quality assurance tests, software debugging, software performance optimization and optimal tasks for employees [5].

Developers found that metrics have become an integral part of the software development process.

1.3. The SQALE Model

SQALE (Lifecycle Quality Assessment) is a method to support the evaluation of software sources. This is a general method that is independent of language and source analysis tools.

The indicators of the model represent the costs. These costs can be calculated in a unit of work, in a unit of time, or in a unit of money. In all cases, the values of the indices are on a scale of the type of relationship. You can use them to perform all permissible operations for this type of scale. For each element of the source code artifact hierarchy, the cost of restoration associated with that characteristic can be estimated by adding all of the restoration costs associated with the characteristics of the characteristic. The indices of SQALE characteristics are the following [7]:
- Testability Index : STI
- Reliability Index : SRI
- Changeability Index : SCI
- Efficiency Index : SEI
- Security Index : SSI
- Maintainability Index : SMI
- Portability Index : SPI
- Reusability Index : SRuI.

The method also defines a global index: for each element of the hierarchy of source code artifacts, the restoration costs that relate to all characteristics of the quality model can be estimated by adding all the restoration costs that are associated with all requirements of the quality model.

This derived measurement is called: SQALE Quality Index [7].

2. Tools overview

2.1. iPlasma

iPlasma is an integrated environment for analyzing the quality of object-oriented software systems that includes support for all required analysis phases: from model extraction (including scalable analysis for C ++ and Java) to monitoring based on high-level metrics or code duplication. iPlasma has three main advantages:
- Extensibility of the supported analysis
- Integration with other analysis tools
- Scalability as used in the past to analyze large projects the size of millions of lines of code.

2.2. SonarQube as a platform for continuous analysis

SonarQube is an open-source platform for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities in 20+ programming languages. SonarQube offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security vulnerabilities.

SonarQube can record metrics history and provides evolution graphs. SonarQube provides fully automated analysis and integration with Maven, Ant, Gradle, MSBuild and continuous integration tools (Atlassian Bamboo, Jenkins, Hudson, etc.).

3. Projects Analysis

3.1. Project overview

Art of Illusion is a free, open source 3D modelling and rendering studio. Many of its capabilities rival those found in commercial programs. Highlights include subdivision surface based modelling tools, skeleton based animation, and a graphical language for designing procedural textures and materials [1].

3.2. Project metrics

Main directed metrics measured in iPlasma are shown in Figure 1.

Each line has a colored percentage. The percentage is derived from the ratio of the number in this line to the number below.

Table 2

Metrics definition

Code	Description
NDD	Number of direct descendants
HIT	Height of inheritance tree
NOP	Number of packages
NOC	Number of classes
NOM	Number of methods
LOC	Lines of code
CYCLO	Cyclomatic complexity
CALL	Calls per method
FOUT	Fan out (number of other methods called by a given method)

The numbers indicate the ratio. Colors indicate where the conditions fit into industry-standard areas (derived from numerous open-source projects). Each ratio is either green (inside the area), blue (below the area), or red (outside the area). For the Struts code base, NDD and CYCLO are outside of the industry standards for these values, and LOC and NOM are listed below.

Table 3

Industry ranging for metrics

	Low	Medium	High
CYCLO / Line	0.16	0.20	0.24
LOC / method	7	10	13
NOM / class	4	7	10
NOC / package	6	17	26
CALLS / method	2.01	2.62	3.20
FANOUT / call	0.56	0.62	0.68

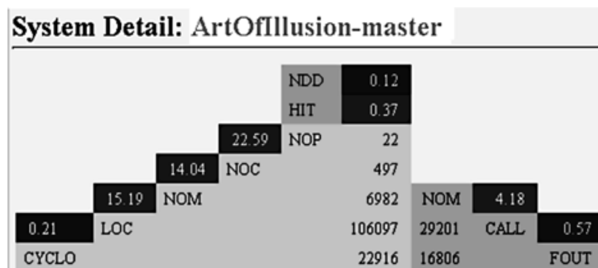

Figure 1. iPlasma Metrics Pyramid for ArtOfIllusion

Metrics Pyramid for ArtOfIllusion generated in iPlasma is shown on Figure 1.

The metrics on this pyramid for ArtOfIllusion project indicate that:

- Class hierarchies tend to be tall and narrow
- Classes tend to be:
  - rather large (i.e. they define many methods)
  - organized in rather fine-grained packages
- Methods tend to:
  - be rather long yet having a rather simple logic (i.e. few conditional branches)
  - call several methods from few other classes (i.e. low coupling dispersion)

3.3. Continuous Code Analysis with the SonarQube

The SonarQube uses an evolved SQALE model.

Bugs, Vulnerabilities and Code Smells are main metrics to measure.

- Bugs: Code that is demonstrably wrong or highly likely to yield unexpected behaviour.
- Vulnerabilities: Code that is potentially vulnerable to exploitation by hackers.
- Code Smells: Will confuse maintainers or give them pause. Not only ratings, but also approximate remediation efforts.

3.3.1. Three Lines of Analysis

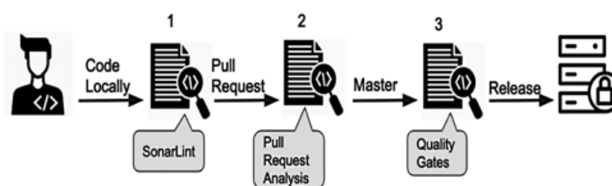

Figure 2. SonarQube Three Lines of Analysis

First line is a SonarLint, that runs in IDE and analyses new code and highlights issues if any.

Second, pull request analysis. When a pull request (PR) is submitted to the repository user comment the changed code with any new issues. After it is up to the manual reviewer to decide if the issues

are critical or accepted in the context before merging the PR.

Finally, quality gates and leak management. If previous prevention fails, there is an ability to set up a collection of go/no-go conditions that indicate whether or not the project is releasable – Quality Gate. In case the project fails the Quality gate then the tool automatically informs about it.

Security and Reliability ratings are based on the severity of the worst open issue in that domain [6]:

- E – Blocker
- D – Critical
- C – Major
- B – Minor
- A – Info or no open issues

For Maintainability the rating is based on the ratio of the size of the code base to the estimated time to fix all open Maintainability issues [6]:

- <=5% of the time that has already gone into the application, the rating is A
- between 6 to 10 % the rating is a B
- between 11 to 20 % the rating is a C
- between 21 to 50 % the rating is a D
- anything over 50 % is an E

3.3.2. SonarQube results for ArtOfIllusion project

The SonarQube provider comprehensive code analysis. The tool found 223 bugs, 125 vulnerabilities, 15 security hotspots. All this categories are marked with rating E (Figure 3.), which means Blocker and must be fixed before next release (Figure 4.).

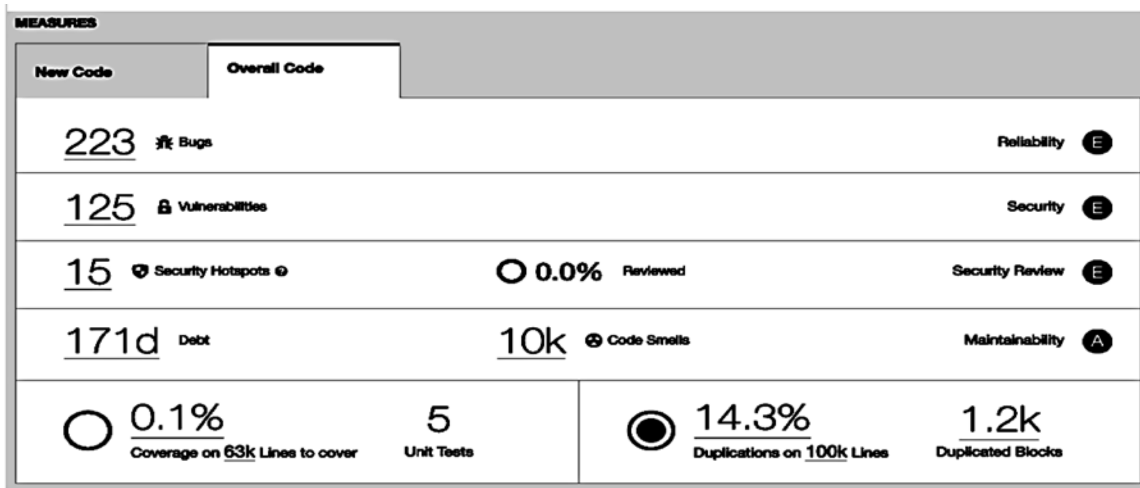

Figure 3. SonarQube Overall Code Analysis Results

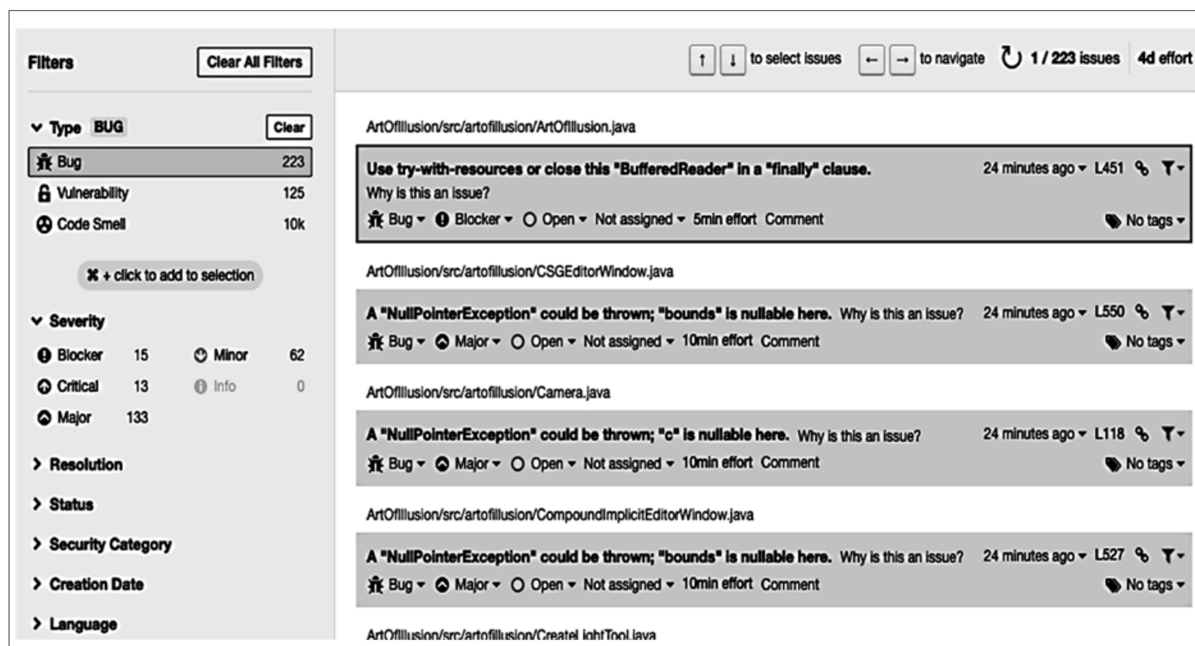

Figure 4. SonarQube Bug Analysis Results

Conclusions

As a summary of techniques for software analysis, represented on Table 1, due to the computability barrier, no technique can provide fully automatic, robust, and complete analyses. Testing sacrifices robustness. Assisted proving is not automatic (even if it is often partly automated, the main proof arguments generally need to be human provided). Model-checking approaches can achieve robustness and completeness only with respect to finite models, and they generally give up completeness when considering programs (the incompleteness is often introduced in the modeling stage). Static analysis gives up completeness (though it may be designed to be precise for large classes of interested programs). Last, bug finding is neither robust nor complete. Another important dimension is scalability. In practice, all approaches have limitations regarding scalability, although these limitations vary depending on the intended applications (e.g., input programs, target properties, and algorithms used).

Comprehensive software analysis needs usage of a number of techniques depending on software complexity and quality attribute requirements like security, performance, resilience and other. Analysis should be done on all stages during the development life cycle.

During the design phase, architecture design tools can easily find architectural bottlenecks in the software and prevent them cheaply and quickly before the start of development and save resources on costly fixes.

Already implemented code could be analysed in a continuous integration environment by a tool like SonarQube. Properly configured metrics and quality gates provide fast and detailed feedback on incremental changes starting from development machine till highload enterprise production environments. Software analysis helps to improve quality and development speed in Agile development life cycle with reasonable cost.

References

1. Art of Illusion. Retrieved from http://www.artofillusion.org.
2. Coquand, T., & Huet, G. (1988). The calculus of constructions. *Information and Computation, 76*, 95–120.
3. Godefroid, P., Klarlund, N., & Sen, K. (2005). DART: Directed automated random testing. *In Conference on Programming Language Design and Implementation (PLDI)* (pp. 213–223).
4. Patrick, C. (2020). *Principles of Abstract Interpretation*. MIT Press.
5. Rival, X. (2020). *Introduction to Static Analysis.* MIT Press.
6. SonarQube Metric Definitions. Retrieved from https://docs.sonarqube.org/7.1/MetricDefinitions.html.
7. SQALE Model. Retrieved from https://en.wikipedia.org/wiki/SQALE.

Сосницький С. О., Глибовець М. М., Пєчкурова О. М.

СТАТИЧНИЙ ТА ДИНАМІЧНИЙ АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розроблення програмного забезпечення зі вбудованою якістю стало важливою тенденцією і природним вибором у багатьох організаціях. У наш час методи визначення та оцінки якості, безпеки, надійності програмного забезпечення не можуть гарантувати безпечну й надійну роботу програмних систем повністю і ефективно.

У цій статті розглянуто статистичні й динамічні методи аналізу програмного забезпечення, основні поняття і методи сімейства. Досліджено, чому для якості програмного забезпечення необхідне поєднання декількох методів аналізу, і наведено приклади того, як статичний і динамічний аналіз може бути впроваджений у сучасний життєвий цикл розроблення гнучкого програмного забезпечення.

Ключові слова: статичний аналіз програмного забезпечення, динамічний аналіз програмного забезпечення, тестування, забезпечення якості, режим SQALE, безперервний аналіз коду, SonarQube.

Матеріал надійшов 16.05.2020

Creative Commons Attribution 4.0 International License (CC BY 4.0)