Section 4. Engineering sciences

Anton Bukarev,
National Research University of Electronic Technology
Applicant, the Faculty of Informatics
and Software Computing Systems

A COMPREHENSIVE EXAMINATION OF SOFTWARE VERIFICATION METHODS: COMBINING STATIC AND DYNAMIC APPROACHES

Abstract: The domain of software design and development confronts substantial impediments in efficaciously addressing the verification process. This investigation endeavors to devise a classification framework for software verification methodologies, facilitating the scrutiny of extant techniques and their corresponding merits and demerits within software applications. By examining and categorizing these methodologies, this research aspires to generate an exhaustive set of criteria and proposals for further progress in automated testing execution on cloud-based apparatuses. The article delves into three salient categories of software verification methods: empirical, formal, and dynamic, and expounds on their disparate degrees of automation, extending from manual to entirely automated approaches. Through this comprehensive assessment, the study aims to augment the continual refinement and optimization of software verification techniques in a progressively cloud-oriented computing.

Keywords: software verification, automation, cloud-based devices, software testing.

1. Introduction

In the realm of software design and development, the verification process remains a critical challenge. Verification methodologies are devised to identify errors, susceptibilities, improperly executed attributes and specifications, as well as to ascertain the conformity

of the final software product with the stipulated prerequisites. The development of a novel classification system for software verification methodologies is an imperative undertaking, as it enables the analysis of extant techniques and their software applications, in addition to discerning their merits and demerits. Investigating and categorizing these methodologies facilitates the formulation of a set of criteria and suggestions for subsequent exploration and enhancement of an automated testing execution approach on cloud-based devices.

Software verification methodologies can be broadly classified into three distinct categories: empirical, formal, and dynamic. Furthermore, with respect to the degree of automation, verification techniques may be characterized as either manual or automated.

2. Software verification

A primary objective of software validation is to ensure that the implemented code adheres to the terms of reference and fulfills functional requirements. This is achieved through the utilization of expertise, which facilitates the assessment of documentation and code for compliance with established norms and design standards, as well as software verification, which may encompass symbolic program execution and model checking methodologies. Formal verification relies on the mathematical representation of the program and does not necessitate its tangible implementation.

Symbolic execution is a technique that enables the emulation of a program's execution with symbolic input variable values. This is tantamount to executing the program on specific test values of input variables, yet it minimizes the requisite number of tests. The semantics of symbolic execution are delineated for a programming language in which data objects are symbolically represented and are defined by augmenting the language's fundamental constructs for interaction with symbolic values.

3. Software verification methods

A paramount phase in software verification involves ensuring that the software aligns with the stated quality attributes, such as correctness (conformity of the system to its intended purpose), security, resilience against indeterminate environmental fluctuations, efficiency in terms of time and memory resource utilization, adaptability to environmental alterations, as well as portability and compatibility.

This article examines merely a fraction of the numerous software verification methodologies, specifically: symbolic execution, model

validation, and dynamic and static verification techniques, which are currently deemed to be the most efficacious. Investigating the algorithms and operational principles of these methodologies may serve as a foundation for enhancing software testing procedures in cloud-based solutions.

4. Classification of software verification methods

Software verification methodologies are classified according to diverse criteria, including the level of automation, functional aptness, precision, types of errors detected, efficiency, scope, execution duration, and the approach to attaining the outcome. During software verification, the primary considerations are system stability in the event of non-deterministic environmental behavior and the effective utilization of time and memory resources.

Expertise stands as one of the most prevalent methods of software verification. This approach entails a software assessment conducted by a subject matter expert. The expert may either be the software product's creator or an external individual (or group of individuals) invited to provide an impartial evaluation of the software product's attributes.

Software expertise can be categorized as either general or specialized, with general expertise further subdivided into distinct types: technical expertise, end-to-end control, inspection, and audit. Technical expertise is directed toward verifying a software product's conformity with its specifications and standards. End-to-end control entails the analysis and evaluation of a program through a sequential examination of artifact characteristics by a group of experts who identify potential errors and vulnerabilities. Inspection involves an analysis where the detection of errors and vulnerabilities adheres to a well-defined plan. Lastly, an audit constitutes an analysis of a program executed by individuals who are not members of the project team.

Specialized software expertise encompasses several types. Organizational expertise is targeted at supervising the project's status by management personnel. Usability examination is conducted by the client and users to evaluate the developed software's user-friendliness. Security expertise is performed by information security professionals to gauge the security level of the software under development. Architecture property analysis focuses on appraising and categorizing software interaction scenarios with users, in addition to scrutinizing the properties of the software architecture.

The software examination method, conducted by qualified experts, is non-automatable and facilitates the resolution of a broad array of

tasks. It boasts high functional suitability and is applicable at any stage of project development. The accuracy of the examination hinges on the expertise of the specialists carrying it out and can detect up to 90% of errors and vulnerabilities. The completion time is contingent upon the software's intricacy and the expert team's experience. Conversely, formal verification methods rely on the scrutiny of the program's mathematical model rather than its source code, examining the practicability of specification requirements within the program model.

Formal software verification methods can be classified into several types based on the approach employed: deductive analysis, model verification, consistency checking, and abstract interpretation. Contrary to expertise, formal methods are amenable to automation but necessitate skilled specialists for constructing mathematical program models. Nonetheless, formal methods exhibit high functional suitability and accuracy, provided an appropriate formal model is devised. These methods can identify diverse error classes, such as undefined program behavior, uninitialized variables, format string errors, standard library usage errors, and others.

Formal software verification methodologies exhibit certain constraints in addressing software verification challenges, as constructing a comprehensive and suitable mathematical model is not always feasible. Nevertheless, these methods can prove efficacious in industrial projects when applied to testable domains that can be incorporated into a formal model. A range of techniques are employed for constructing mathematical models, including the Kripke structure and temporal logic, alongside model-based approaches such as finite state machines and Petri nets.

A primary advantage of the model verification methodology is its capacity for automating the processes of verification and model construction. The development of a formal model enables the representation of program code as logical expressions and facilitates the examination of program properties articulated in the form of a specification. Nonetheless, it is crucial to acknowledge that devising the most comprehensive and appropriate mathematical model necessitates the expertise of highly qualified professionals.

5. Static software analysis

Static program analysis constitutes an evaluation performed without the actual execution of the program, typically conducted on the basis of source code. Static analysis enables the examination of all

potential program execution paths, identifying errors and potential vulnerabilities. This method is frequently employed in conjunction with specialized automated tools. Two prominent groups of static verification methods are widely used: deductive program analysis methods and model verification methods. Deductive analysis techniques are utilized to substantiate a program's compliance with its specification, generally provided in the form of preconditions and postconditions. However, these tools are ill-suited for the analysis of large-scale programs, as they necessitate manual annotation of functions and loops within the program text. Model validation approaches involve the creation of a mathematical program model, typically employing a Kripke model, which is subsequently analyzed for adherence to established conditions and constraints.

In static verification, the program is scrutinized without actual execution, typically through parsing the program text and its internal representation. The generation of the internal representation transpires during the parsing process, preserving the program's original structure. Subsequent to the analysis, a control flow graph is constructed, enabling the examination of all potential program execution paths. The accuracy of the analysis is contingent upon the quality of the tools employed and the capacity to analyze the program's internal representation.

Moreover, static analysis can aid in pinpointing potential performance concerns, suboptimal resource utilization, flawed flow control, and specific security vulnerabilities, such as SQL injection and XSS attacks. Nevertheless, it is important to acknowledge that static analysis cannot ensure the total absence of errors within the program, as covering all conceivable execution paths may be unattainable. Furthermore, static analysis cannot supplant comprehensive program testing on real data, which can reveal errors associated with the program's interaction with the external environment.

Undoubtedly, static analysis-based verification is most impactful during the software design phase, as it facilitates the early detection of numerous errors and defects, substantially mitigating project costs and risks. Concurrently, automated verification tools employing static analysis cannot entirely supplant expert assessment and dynamic program testing, given their inability to account for all potential program execution scenarios and real-time component interactions. Consequently, static analysis utilization should be supplemented with alternative verification methodologies to attain optimal results.

6. Dynamic software verification methods

Dynamic software verification methods encompass the analysis of a program during actual execution. In simulation modeling, the program itself is not executed; rather, a program that simulates it is employed. Program inputs can provoke nondeterministic behavior, facilitating the detection of vulnerabilities and bugs. Dynamic analysis comprises several types, including testing, monitoring, simulation testing, and profiling.

Monitoring is an approach wherein the software's operation is observed, documented, and evaluated, with the capacity to procure data on program operation through instrumentation. Instrumentation can be executed in diverse manners, such as manual, compiler, binary-translation-based, runtime injection, or simulator monitoring. Monitoring techniques can be event-based or static. The most exhaustive method of dynamic analysis is testing.

Software testing methods fundamentally aim to identify nondeterministic, erroneous, or non-compliant program code behavior. Testing is typically conducted based on known, predefined scenarios, which involve monitoring and creating a controlled program execution environment. This permits experimentation with various test sets and documentation of the results obtained. The quality of testing is determined by explicitly defined testing objectives, comprehensive test coverage, and established criteria.

In contrast to static analysis, dynamic software verification methodologies are predicated on the actual program execution and can be automated. These methods facilitate the creation of a controlled environment for testing and monitoring, and identify various defects, encompassing temporal and quantitative software characteristics, such as execution time and resource usage. Dynamic analysis can uncover memory leaks, errors in multithreaded applications, and other faults that solely transpire during actual program execution. Nevertheless, the efficacy of dynamic verification approaches is directly contingent upon the quality and volume of input data. Dynamic verification techniques are typically employed in domains where response time, resource consumption, and reliability are paramount, including database servers and real-time systems.

7. Conclusion

Upon analyzing the classification of software verification methods, it can be inferred that examination methods, while unable to be automated,

enable the detection of numerous errors. Conversely, formal verification methods, albeit more time-consuming, possess the capacity to identify a vast array of errors and are readily automated. However, static methods no longer ensure comprehensive testing due to the employment of dynamically generated code, which is impervious to static method verification. Dynamic methods can only detect a specific set of errors, which precludes the guarantee of exhaustive testing.

Consequently, to effectuate efficient software testing, it is prudent to employ various verification techniques at distinct project stages. When utilizing static methods, it should be acknowledged that enhancing analysis accuracy results in heightened resource consumption. To improve static analysis accuracy, dependencies between variables in the program code can be identified. During the initial development stages, it is advisable to apply dynamic methods only if functional software components exist. Their implementation necessitates the establishment of a test or monitoring system to regulate program behavior. Generally, dynamic methods are more efficacious and contemporary, as they can detect a greater number of vulnerabilities.

The findings of this study will contribute to the development of a system for automating test launches on cloud devices.

References

1. Schütte J., Fedler R., Tetze D. ConDroid: targeted dynamic analysis of Android Applications. AINA'15 Proceedings of IEEE 26th international Conference on Advanced Information Networking and Applications, Gwangui, South Korea, March 24-27, 2015.
2. Kim T., Park J., Kulinda I., Jang Y. Concolic Testing Framework for Industrial Embedded Software. APSEC'14 Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference, volume 2, Jeju, South Korea, December 01-04, 2014, pp. 7-10.
3. Gerasimov A.Y., Kruglov L.V., Ermakov M.K., Vartanov S.P. An approach of reachability confirmation for static analysis defects with help of dynamic symbolic execution. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-7.
4. Miller, C., Valasek, C.: A survey of remote automotive attack surfaces. Black Hat USA (2014).
5. Huuck, R.: Technology transfer: Formal analysis, engineering, and business value. Sci. Comput. Program. 103 (2015) 3–12.

6. Mateo Tudela, F.; Bermejo Higuera, J.-R.; Bermejo Higuera, J.; Sicilia Montalvo, J.-A.; Argyros, M.I. On Combining Static, Dynamic and Interactive Analysis Security Testing Tools to Improve OWASP Top Ten Security Vulnerability Detection in Web Applications. Appl. Sci. 2020, 10, 9119.
7. Barabanov, A.; Markov, A.; Tsirlov, V. Statistics of software vulnerability detection in certification testing. In International Conference Information Technologies in Business and Industry 2018; IOP Publishing: Tomsk, Russia, 2017.
8. Bermejo, J.R.; Bermejo, J.; Sicilia, J.A.; Cubo, J.; Nombela, J.J. Benchmarking Approach to Compare Web Applications Static Analysis Tools Detecting OWASP Top Ten Security Vulnerabilities. Comput. Mater. Contin. 2020, 64, 1555–1577.
9. Nunes, P.; Medeiros, I.; Fonseca, J.C.; Neves, N.; Correia, M.; Vieira, M. Benchmarking Static Analysis Tools for Web Security. IEEE Trans. Reliab. 2018, 67, 1159–1175.
10. Mohino, J.D.V.; Higuera, J.B.; Higuera, J.-R.B.; Montalvo, J.A.S.; Higuera, B.; Mohino, D.V.; Montalvo, J.A.S. The Application of a New Secure Software Development Life Cycle (S-SDLC) with Agile Methodologies. Electronics 2019, 8, 1218.
11. Al-Amin, S.; Ajmeri, N.; Du, H.; Berglund, E.Z.; Singh, M.P. Toward effective adoption of secure software development practices. Simul. Model. Pr. Theory 2018, 85, 33–46.