UDK 004.054

P. Serdyuk, PhD.,
O. Nytrebych

DEVELOPMENT OF SOFTWARE USAGE MODEL FOR DOMAIN-ORIENTED TESTING BASED ON SYNTAX AND LEXICAL ANALYSIS OF PROGRAM SOURCE

Abstract. *The new approach of software usage model development with syntax and lexical analysis are proposed in this article. Code analysis is used to develop software usage model represented as a graph of possible application flow and set of domain program variables value that are changing during execution flow.*

Keywords: *software usage model, domain-oriented testing, lexical analysis, control flow graph*

П. В. Сердюк, канд. техн. наук,
О. О. Нитребич

ПОБУДОВА МОДЕЛІ ВИКОРИСТАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ДОМЕННОГО ТЕСТУВАННЯ НА ОСНОВІ ЛЕКСИЧНОГО ТА СИНТАКСИЧНОГО АНАЛІЗУ ПРОГРАМНОГО КОДУ

Анотація. *У цій статті розглянуто новий підхід до побудови моделі використання програмного забезпечення за допомогою синтаксичного та лексичного аналізу коду. Аналіз коду використовується для побудови графу потоків виконання та змін доменів значень змінних програми у моделі використання програмного забезпечення.*

Ключові слова: *модель використання програмного забезпечення, доменне тестування, лексичний аналіз коду, граф потоку управління*

П. В. Сердюк, канд. техн. наук,
О. О. Нытрэбыч

ПОСТРОЕНИЕ МОДЕЛИ ИСПОЛЬЗОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ДОМЕННОГО ТЕСТИРОВАНИЯ НА ОСНОВЕ ЛЕКСИЧЕСКОГО И СИНТАКСИЧЕСКОГО АНАЛИЗА ПРОГРАММНОГО КОДА

Аннотация. *В этой статье рассмотрен новый подход к построению модели использования программного обеспечения с помощью синтаксического и лексического анализа кода. Анализ кода используется для построения графа потоков выполнения и изменений доменов значений переменных программы в модели использования программного обеспечения.*

Ключевые слова: *модель использования программного обеспечения, доменное тестирования, лексический анализ кода, граф потока управления*

Introduction

The most costly software failures are a consequence of a fact that such failures are usually caused by complicated scenarios, and are found at late stages of software development, such as regression testing, alpha and beta testing, software deployment, etc. These scenarios are difficult to cover by the automated or manual testing due to the fact that each stage of a complex scenario can have a number of degrees of freedom, i.e. the possible subsets of the variables values set it uses [1]. The corresponding number of all scenarios is the product of all degrees of freedom of each stage, which can be quite a large number.

For building long complex software testing scenarios, which contribute significantly to the software cost, software usage model is used [2 – 3]. State-based software usage model is represented as a graph of transitions and a set of variables with respective sets of equivalence classes [4]. The process of software execution can be modeled as a transaction through paths of the graph, each node of which can be presented as a method or operation that changes the value of the set of variables values.

One important issue in usage modeling that helps to simplify the model and decrease testing cost is domain testing [5]. The essence of domain testing is stratified sampling of a few tests from a group of candidate test cases by identification of a finite number of equivalence classes of input variable value intervals so that each test that is representative of a class of test cases, was equivalent to any other test in this class within given range of input values. Software failures in a variety of domains can be caused by combinations of relatively few conditions, thus all faults in a system can be triggered by a combination of few parameters [6], but testing all combination of parameters is technically impossible in application with large number of variables. To effectively decrease valuable combination of parameters software complex behavior should be analyzed.

The verification of highly concurrent systems is a challenging task, as their state space grows exponentially with the number of processes. Generating a effective test suite usually needs a lot of manual work and expert knowledge. In a model-based process, among other subtasks, test construction and test execution can also be partially automated.

Such techniques of solving this problem like partial order reduction [7] or using binary decision diagrams for combinatorial test design [8] that automatically construct tests so that it covers all valid value combinations of parameters have received a lot of attention recently.

Model-based testing can have different definition; in classical meaning it is testing that relies on models specifying the intended behavior of software [9], automation of the design of black-box tests [10], etc. In this article we consider a slightly different meaning of the term in sense grey-box model that is developed automatically based on the whitebox fuzzing[11]. Whitebox fuzzing is another form of automatic dynamic test generation, based constraint solving and symbolic execution. Due to the enormous number of control paths in early processing stages, whitebox fuzzing effectiveness is still limited when testing applications with highly-structured inputs.

Nowadays, this approach needs further investigation and improvement due to different trends in software development:

- *Multi-threaded applications*. Threads dependencies can involve appearance of new equivalence classes set. Identification equivalence class should include cases that another thread can make changes in our test, accessing and changing important variables, moreover such impact sometimes could be hardly found as it could occur occasionally during running tests.
- *Graphics*. More and more applications are developed for touch-screens. Naturally, the input field to be replaced by graphic elements which can easily push, move, etc. For graphic elements, which are built on functional equivalence classes is extremely difficult or expensive to find equivalence classes.
- *Third party frameworks and libraries*. Nowadays applications are so actively using frameworks and third-party libraries, that majority of the internal logic and the underlying code is used just inside the external libraries that radically breaks all the plans for the use of equivalence classes.

Usually domain is also tested at the boundaries and near-boundary values, to identify that domain are correctly defined at the boundaries. But it does not give confidence that error can be passed somewhere inside of domain when equivalence class is not correctly defined, because in fact it may be more than one equivalence class inside of another and errors will not be found in such cases during testing.

Domain model

Correct definition of domain knowledge plays a very critical role while testing domain-specific work. Equivalence class is identified as the basis because of some criteria, like specifications, code, opinions, analyzes, etc. This knowledge is based on graph structure that connects the causes of errors with the expected responses. During testing all information about errors is collected and used in assessing the reliability and quality of the software.

In this article we are constructing domain model with the following criteria:

D_T – domains that are identified by typical testing value for “black box” model. For example, for string input values we should check next domains for this value: null value, empty string value, strings that contain special symbols “~`!”@’#$;%:^&?*()[]{},.\/+=-_”, string that exceeds 255 characters and other types of string.

D_F – domains that are identified by applications flow, i.e. by cycle or conditional operators, for example predicate condition (k>100) result to two classes of equivalence $k \leq 100$ and k>100, or even to three classes of equivalence k<100, k>100 and k= 100 for boundary approach;

D_E – domains that are identified by possible exceptions in applications flow, for example, division by zero or null reference exception or other changes can impact on failures;

D_R – domains that are identified by requirement specification. D_R can also contains limitation for input values integer, for example value k is within range [0, 200]. That splits domain of value k to three regions: negative numbers, values between 0 and 200 and values over 200.

Thus, domains set can be identified by the set $D = \{ D_T, D_F, D_E, D_R \}$

For example: requirements descriptions allows input integer value Quantity in range from 1 to 99. Up to requirement to specification we have 2 domains for value $D_R = \{ [1, 99] \cap Z,\ [-\infty, 1) \cup [100,-\infty] \} \}$

Thus, we should provide different test cases with different value for Quantity value

1. Enter value in the middle 50. Positive result expected according requirements.
2. Enter boundary values 1 and 99, positive result expected.
3. Enter external boundary values 0 and 100, failure expected.
4. Enter floating point number 50.5, failure expected.
5. Enter letter, special symbols, like: ~`!”@’#$;%:^&?*()[]{},.\/+=-_., failure expected.

For domains D_F and D_E that are difficult to evaluate manually we propose to identify them with usage model to automatically build it. To build graph structure of usage model we create lexical and syntax analyzer that could work in wide ranges of programming languages and technologies. Model is simplified view of software application that contains graph structure of application and possible changes in variable domain values and possible failures (Fig.1).

Thus lexical syntax for usage model is usually a regular language, with the grammar rules consisting of command separators, assignment operator, mathematical and logical operators, conditional and cycle operators, functions and classes keywords and variables. Developed lexical analyzer neglects other features, like database operations, connection to different service, UI displaying, etc. and transfer code to the list of tokens that contain information of changes in variables value and application flowThe lexical analysis algorithm is optimized according assumptions mentioned above: based on command-separator regular expression search it identifies information tokens, it does not require a lot of computational resource comparing to compilation.

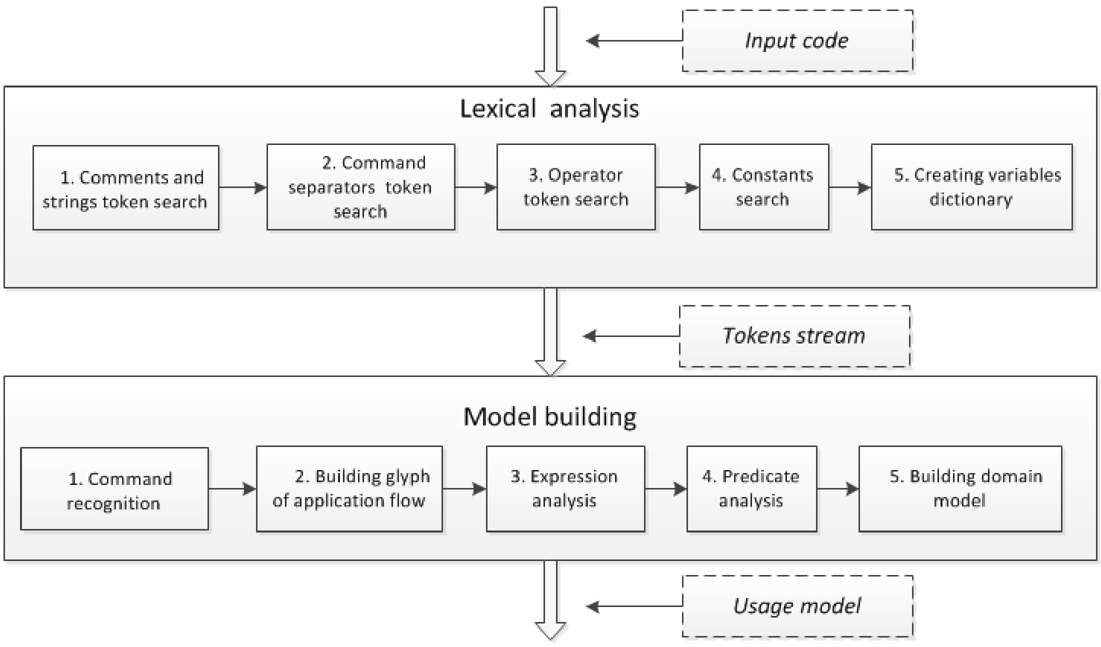

Fig. 1. Algorithm of building usage model through lexical and syntax analysis

After lexical analysis that converts code to the list of tokens syntax analysis is performed, that have three main stages:

- Commands analyzer – this analyzer is core analyzer that recognizes the flow of application, including –conditional and cycle operators, assignment of new value, call of external and internal methods and thread management. This analyzer is used to make first step of analysis, that don't recognize conditions of flow and expression that are assigned to the variables.
- Predicate analyzer – analyzer of logical expressions that contains condition in logical and conditional operators. This analysis will identify the condition of application flow through different branches of execution. Domains obtained by description of predicate can depend on values of few variables, and cannot be tied to fixed values. For example, in expression *if (quantity*price < discountLimit){...}* flow depends on value of multiplication of two variables: *quantity* and *price.* As a result, these domains are defined by predicate expression of few variables.
- Expression analyzer – analyzer of mathematical and other expression that can be assigned to variable. This type of analyzer is used to distinguish possible faults in expressions: null reference/not initialized value usage, mathematical faults like zero division, index out of array, etc. This analyzer also determines the possible domain of new value of variables. Unfortunately, it was possible only for simple expressions, that don't refer to any external libraries functions. More complex expressions were analyzed statistically by collecting possible changes in the variables values domains.

Developed model is represented in the form of a directed graph G = { *O, T* }, where *O* – set of software operations, *T* – set of transitions between the respective operations. Each operation contains one or more variable V^i that belongs to the set of variables used in the model V and is characterized by two or more of equivalence classes. An example of such a representation is shown in Fig. 2. It should be noted that there are two types of equivalence classes: the correct equivalence classes representing the correct input values for variable and incorrect equivalence classes corresponding to all other possible states of the environment (i.e. wrong input values).

The process of software execution can be modeled by the passage paths of the graph, which edges will be responsible for the sequence of method call and each node of which can be presented as a operator that can:

- Start new execution thread due to user actions like pressing keys or buttons;
- Exit current thread;
- Other thread operations like pause/resume, locking on variable, waiting to execution for other threads, etc;
- Change flow of execution in accordance with conditional/cycle operators;
- Changes the value of the set of variables values (method arguments as well as global variables);
- Result to failure in case set of variables is containing values from failure domain.

Each operation that would fit the node has the following properties:

Each operation O^i can contain the following properties:

1. List of variables used by the method – O^i_{used} .
2. List of variables changed by the method O^i_{change} , and appropriate change domains.
3. List of variables and corresponding incorrect equivalence classes O^i_{err} that can cause failure in this operation.

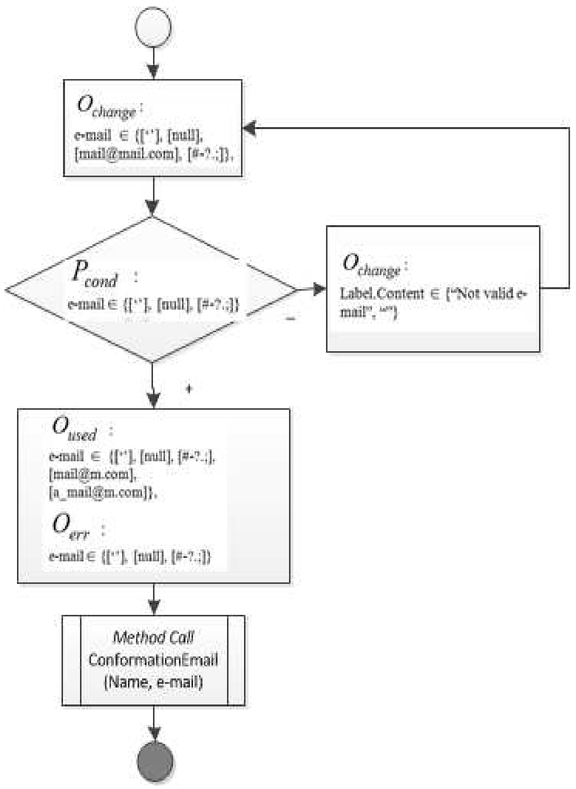
O_{change} :
e-mail ∈ {['], [null], [mail@mail.com], [#-?.;]},
P_{cond} :
e-mail ∈ {['], [null], [#-?.;]}
–
+
O_{change} :
Label.Content ∈ {"Not valid e-mail", ""}
O_{used} :
e-mail ∈ {['], [null], [#-?.;], [mail@m.com], [a_mail@m.com]},
O_{err} :
e-mail ∈ {['], [null], [#-?.;]}
Method Call
ConformationEmail
(Name, e-mail)

Fig. 2. Sample of part of usage model build by lexical and syntax analysis

Developed usage model can be used during automation testing process to build and execute complex scenario and find failures that depend on many factor and are difficult to reproduce [5].

Conclusion

With the use of lexical and syntax analysis of source code we developed usage model of software that contains description of possible changes in variables, condition of changing flows execution and failures condition. This model have some limitations as it does not cover all possible failures like flow errors (endless cycling, wrong conditions of recursion end) or UI display errors, but together with statistical analysis application flows it can become powerful tool in discovering failures that occurs in software that contains complex algorithms.

References

1. Mohd Ehmer Khan, (2010), Different Forms of Software Testing Techniques for Finding Errors, *IJCSI*, Volume 7, Issue 3, pp. 11 – 16 (In English).

2. Siegl S., Winfried D., Reinhard G., and Gerhard K., (2009), Model-Driven Testing based on Markov Chain Usage Models in Automotive Domain, *Proc. of the 12th European Workshop on Dependable Computing* (In English).

3. Trammell C., (1995), Guantifying the reliability of software: statistical testing based on a usage model, *Software Engineering Standards Symposium*, pp. 208 – 218 (In English).

4. Fedasyuk D., Yakovyna V., Serdyuk P., and Nytrebych O., (2014), Variable State-based Software usage Model, *Econtechmod: an International 19 Quarterly Journal on Economics in Technology, new Technologies and Modeling Processes*, Volume 3, Issue 2, pp. 15 – 20 (In English).

5. Kaner S., Folk D., and Nguen E., (2000), Software Testing, *"DiaSoft"*, 544 p. (In English).

6. Kuhn D.R., Wallace D.R., and Gallo A.M., (2004), Software Fault Interactions and Implications for Software Testing, *IEEE Transactions on Software Engineering*, Volume 30, Issue 6, pp. 418 – 421 (In English).

7. Cormac Flanagan, (2005), Patrice Godefroid. Dynamic Partial-order Reduction for Model Checking Software, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* Volume 40, Issue 1, January 2005, pp. 110 – 121.

8. Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi, (2011), Using Binary Decision Diagrams for Combinatorial Test Design, *ISSTA '11 Proceedings of the 2011 International Symposium on Software Testing and Analysis,* pp. 254 – 264.

9. Pretschner. Model-based Testing in Practice. In: Formal Methods. Lecture Notes in Computer Science, Vol. 3582, pp. 537–541.

10. Utting M., and Legeard B., (2006): Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, San Francisco.

11. Patrice Godefroid, and Adam Kiezun. Grammar-based Whitebox Fuzzing, *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation,* pp. 206 – 215.

Received 26.05.2015

Serdyuk
Pavlo Vataliyovych,
PhD., Associate professor at Software department,
National University Lviv Polytechnic.
Ukraine, Lviv, S. Bandery st. 12,
+380984574543.
E-mail: pavlo.serdyuk@lp.edu.ua

Nytrebych
Oksana Oleksandrivna,
Assistant at Software department,
National University Lviv Polytechnic.
Ukraine, Lviv, S. Bandery st. 12,
+380969440305.
E-mail:
ksenija.volynj@gmail.com