Mathematical Problems of Computer Science 59, 45–56, 2023.
doi: 10.51408/1963-0101

UDC 004.891.3

Expert Knowledge-Based RGT Solvers for Software Testing

Mane P. Buniatyan[1], Sedrak V. Grigoryan[2] and Emma H. Danielyan[3]

[1]Synopsys Armenia, Yerevan, Armenia
[2]Institute for Informatics and Automation Problems of NAS RA,Yerevan, Armenia
[3]EPAM Systems Inc., Yerevan, Armenia
e-mail: buniatyanmane@gmail.com, addressforsd@gmail.com, emma_danielyan@yahoo.com

Abstract

Program testing is a way of assessing the quality of software and reducing the risk of software failure in operation [1]. Quality issues can cause as financial loss as well as harm to human lives (e.g., when the bug is in medical instruments, cars, etc.). So, it is very hard to underestimate the importance of testing.

There are multiple testing techniques, which are split into 3 major categories. One of them includes experience-based techniques. Test cases and scenarios used in experience-based testing are derived from the tester's knowledge and intuition, as well as their experience with similar applications and technologies. These techniques can be helpful in identifying tests that are not identified easily by other more systematic techniques. Depending on the tester's approach and experience, experience-based techniques may achieve widely varying degrees of coverage and effectiveness [1].

We propose a method for automation of experience-based testing via a class of combinatorial problems (RGT class). A Solver is developed for the class. It acquires expert knowledge and elaborates effective strategies for RGT problems [2]. The proposed method generates test cases dynamically based on the response of the program. The adequacy of the method is being experimented for "blender" open-source application, which has Python API allowing to experiment with testing and analyze test results.

Keywords: RGT class, RGT Solver, Software testing, Expert systems.
Article info: Received 25 September 2022; sent for review 11 October 2022; accepted 07 February 2023.
Acknowledgement: The authors express their deep gratitude to Dr. Edward Pogossian for his contribution and constructive comments to the work.

1. Introduction

Software Testing is an approach to assess the quality of software and to reduce the risk of its failure in operation [1].

In [1], testing techniques are divided into 3 groups: black-box, white-box and experience-based techniques. In the case of the last one, test cases are based on the testers' knowledge and intuition, on experience with similar applications and technologies. These techniques are efficient in identifying tests that are not identified easily by other more systematic techniques as well as when there is a limited testing time or incomplete specifications [1].

According to the World Quality Report 2021-2022 [3], one of the current trends in quality assurance and software testing is test automation. Test automation has the following benefits [1]:

- saves time by reducing repetitive manual work
- provides greater consistency and repeatability
- allows to evaluate the situation more objectively based on static measures, coverage reports, etc.
- provides more accurate information about the current state of testing based on gathered statistics, test progress, defect rates and performance.

There is a way to automate test case generation, known as the model-based testing (MBT). MBT is a technique for generating a test suite from requirements [4]. Instead of individual tests creation, testers create models that allow generating test cases automatically. These methods can be used in regression testing and are especially useful when the system changes frequently. In this case, the test suite can be regenerated easily by adjusting the model instead of readjusting each test case separately.

MBT has three important components [4]:

- a model (requirement, information, workflow, architectural, behavioral, configuration, deployment, performance, risk, environment, and usage models [5])
- a test-generation algorithm
- tools generating a supporting infrastructure (including the expected output).

MBT tools are meant to generate test suites by manipulating either with input data or behavior without handling both simultaneously. Generated test cases do not provide ways to test the system dynamically (the choice of modules to testing depends on the previous test results).

Software Testing can be considered as a combinatorial problem between a tester and states of a program. Hence, testing can be also considered as a representative of Reproducible Game Tree (RGT) class problems. RGT is a class of combinatorial problems, for which the space of solutions is a reproducible game tree. These problems meet the following requirements [6]:

- there are interacting actors (players, competitors, etc.) performing identified types of actions in specified moments of time and specified types of situations
- there are identified benefits for each actor
- there are descriptions of situations in which actors act in and are transformed after actions.

For such problems with a given arbitrary situation x and an actor A, who is going to act in x, we can generate a corresponding game tree GT(x, A) comprising all the games started from x. Games represent all possible sequences of legal actions for players and situations that they can create from the given initial, or the root situation x. In our consideration, the games are finite and end with one of the goal situations of the problem [6].

Assuming that A plays according to a deterministic program, a strategy, the GT(x, A) represents, in fact, all possible performance trees of the strategies from x. In that sense, the GT(x, A) determines the space of all possible solutions from the situation x. With the given criterion K to evaluate the quality of strategies, we can define the best strategy S*(x, A) and the corresponding best action of A from x [6].

RGT class includes important problems like computer networks intrusion protection, optimal management and marketing strategy elaboration in competitive environments, testing of programs, defense of military units from various types of attacks, communication problems, certain types of teaching, chess and chess-like games [2].

One of the advantages of RGT class is that these problems are reducible to the standard kernel problems K. K- methodology multiplies the achievements for particular problems of this class. Distributed development of this methodology is possible. K-methodology enhances the effectiveness of RGT Solvers providing answers to urgent RGT questions including the following ones [2]:

- measurement of the effectiveness of Solvers
- analysis and typification of combating knowledge
- construction of knowledge-based Solvers
- regular acquisition of RGT expert knowledge and enhancing the effectiveness of Solvers.

The validity of K-methodology was proved for certain RGT problems including Chess, Network Intrusion Protection, Navy Defense from Attacks, Management, Marketing etc. [2].

RGT Solver is a software that acquires expert knowledge and elaborates effective strategies for RGT problems [2]. It is a universal tool for solving RGT-class problems.

Strategy searching and game tree. As already mentioned, the space of solutions for RGT problems is a reproducible game tree, and with the given criteria, we can evaluate and choose the best possible actions in given situations for the given actor.

As the combinatorial complexity of the mentioned problems is huge, we need to reduce the game tree. Otherwise, the computer's computational resources (memory and storage) will not be enough to solve them. C. Shannon suggested reducing the tree by building it until the resources are expired. It is not an effective way because we waste our resources to compute steps that will not improve the current situation. Another approach, suggested by M. Botvinnik, is to consider only those steps that have potential benefit in the current case, i.e., we should not examine the steps that have no meaning. We can evaluate the possible usefulness of an action with the knowledge (without reviewing the opponent's answers) and choose the most profitable one. Then, by checking the opponent's potential actions, we can build the game tree and choose the best move in a given situation [7].

The Solver builds the game tree, evaluates situations with the knowledge, then chooses the best action using the minimax algorithm.

The purpose of this paper: Testing of programs can be considered as an RGT problem, and RGT Solver can be used for experience-based testing as an expert system when the corresponding knowledge is available.

In this work, we aim to provide a definition of testing problems as RGT problems, a way of formulating knowledge, and an approach for proper assessment of tested programs, which also covers the drawbacks of model-based testing approaches (in particular, combining different behaviors and input data, running both functional and non-functional tests at the same time, and generating tests dynamically). Thus, the following open questions are addressed:

1. What kind of knowledge are we going to use, who are the actors as well as what are their possible actions?

2. How to evaluate each situation, what kind of goals each actor has, etc.?

Overall, this leads to proposing a model for representing an experience-based testing as an RGT problem.

2. Reduction of Program Testing to RGT Class

In RGT problems, it is essential to define the situations, the actors, the actions, and benefits for each of them. Let's define these terms for program testing.

The actors in software testing are the system under test (i.e., the program) and the tester. Note, that unlike some other problems in the RGT class (e.g., like chess), where the opponent tries to make counteraction, in testing the program just responds to the tester's actions.

The actions are any valid elementary operations that can be performed with the program. While building the "game" tree, the Solver dynamically combines these actions, creates test cases and executes them depending on the response of the program. Note, that not all combinations of the elementary operations are meaningful from the perspective of the user (e.g., actions that have no effect or are not connected with each other). That is why we need to find a way to control these combinations. The actions of the program are actually only responses to the tester's actions.

The situations are the current states of the program. We can estimate the current situations with [0;1] numbers, where 0 means that no bugs are found, 1- that the program is in a critical state and is not usable. The numbers in-between 0 and 1 are intermediate values, and situations with values closer to 1 are worse than situations with values closer to 0. We suggest the following criteria for evaluating the current state of programs (these criteria can be expanded later):

- Existence of bugs (difference between expected and observed results): different bugs have different importance; when the main functionalities of the program do not work as expected, the program becomes useless (e.g., if the user is not able to log into a social network, save the result of the accomplished job, do a transfer in the banking system, etc.).

- Performance degradation: we all would like to have fast, high performing programs, but unfortunately it is not always possible. Performance degradation in a part of the program that is used frequently will cause to slowdown the work, but if it is in a part

that can be done without human interaction and/or is performing rarely, then it can be acceptable.

- Security: this is essential for some programs (e.g., banking system, strategic information storing, transfers, etc.).
- Crashes and hangovers: this is always bad, and in some cases, they can even cause to a fatal problem, like losing the whole work performed. In most situations, this is not acceptable.

We need to take into account the number of problems, as well as their severity and importance, the sequence of actions causing the problem (i.e., how frequently the problem occurs in "real life"). A bug in a very important functionality is worse than a crash that users might not even encounter, but, on the other hand, having lots of "minor" issues in the program is also not acceptable. When one of the main functionalities does not meet the requirements mentioned above, the program is in a critical state, and it cannot be delivered to customers. The importance of each functionality is considered as a multiplier for the appropriate criterion.

The current state of the program can be measured with the following evaluation function:

$$st = mc * c + mb * b + mp * p + ms * s \quad \{1\},$$

where $mc, mb, mp, ms \in [0; 1]$, $c,\ b,\ p,\ s = \{0 \mid 1\}$. C, b, p and s are Boolean variables, that show the existence of crashes/hangovers, bugs, performance degradations or security problems respectively (1 if the mentioned problems occurred, otherwise - 0). Mc, mb, mp and ms are multipliers for the occurred problems (they show the importance of the broken functionality). Any occurred problem is counted only once, so if, for example, a crash occurs, even if it relates to a security problem or it is a bug (obviously, it is not an expected result) we will consider c = 1, b = 0, s = 0 and p = 0. If the current state of the program is bigger than 1, we consider it as 1.

3. RGT Expert Knowledge Formatting for Testing

Error guessing, exploratory testing and checklist-based testing are representatives of experience-based techniques [1].

Considering the characteristics of each of these techniques, we propose the following usage of the Solver: by reviewing issues occurred before, the usage of the program and its main functionalities, we create checklists. In the Solver, checklists are represented as plans, and the checklists' actions as goals. Based on the coverage reports, the source files responsible for each action can be defined. These connections help to prioritize the created checklists. The user can also define priorities depending on the module he/she is most interested in.

Checklists lead to the creation of a game tree. Each branch in the tree is a test case. It is important to mention that actions in the checklists are general, i.e., many elementary actions can correspond to one action in the checklist. It allows you to combine multiple actions and build a tree. Checklists define if it still needs to proceed to the next steps or not in case of a defect occurrences in the current step.

Multipliers in formula {1} are also given as knowledge for the Solver. They show the importance of user action. Note, that multipliers should be defined for both elementary

and checklist actions. The same elementary action in different situations can have different importance, e.g., if the user tries to save a text file it is more important to save the text than the style. We multiply both multipliers to get one for the action. Imagine that in the example below, mb = 0.8 for the elementary action "save" and for the following checklists of actions "open the program, add text, save", "open an existing text file, change the style, save". Let's say we have mb = 1 for the "save" in the checklist1 and mb = 0.6 for the "save" in the checklist2. In this case, if the program is not able to save the text, we will have mb=1*0.8=0.8 and for the second case: mb=0.6*0.8 = 0.48. Thus, the first case will be considered worse than the second one.

In the case of performance degradation, we need to multiply mp with the coefficient showing how many times the performance was slowed down or how much longer it takes to perform the same action. E.g., if the performance is 2x slower than expected, we need to multiply mp with 2.

The testing continues until a. the given time is expired, b. all/chosen checklists are checked or c. if the program gets into a critical state.

4. RGT Solver Experiments in Program Testing

We have chosen the Blender program as a system under test. It is an open-source 3D modeling program with a Python interface that can be used for testing. In order to understand how the program testing Solver works and how the knowledge and checklists can be represented, let's study an example.

To understand how the knowledge and checklists can be represented, let us review an example.

The checklist below checks some of the main functionalities of the program:

```
1 # basic_operations; x.cpp
2
3 Open the program; mc=1, mp/5s/=0.04;
4 Move 3d cursor; mb=0.7, mc=0.9, mp=0.06, nextStep=1;
5 Add object; mb=0.9, mc=0.9, mp=0.07;
6 Change geometry; mb=0.8, mc=0.9, mp=0.07;
7 Transform object; mb=0.8; mc=0.9, mp=0.07
```

Fig. 1. Checklist Example

To keep it simple, we just added a few basic operations, but this list can be enlarged if needed. The operations in this checklist can be independent, like lines 6 and 7. But if this was a checklist based on the previous failures or a user story, then all steps would depend on each other. This checklist could be used if we had limited testing time and could only check the main operations to make sure that there were no critical issues (like a smoke test). The first line of the checklist (i.e., the comment) represents the name of the checklist and the source file which is associated with the checklist (here, as we don't know the corresponding

source file, we put x.cpp just to show the structure of the checklist. The source file is not mandatory). If some multipliers are absent in the checklist, we assign 0 to them (e.g., ms=0 for all actions in checklist below, because they could not lead to security problems). The variable nextStep is used to determine whether the next step should be performed or not in case of bug in the current step (e.g., if the user is not able to move the 3D cursor it is still somewhere in the scene and the user can add objects). In line 3 we open the program. If it crashes it is a critical state for the program, thus mc=1.

Next to mp there is the expected time the operation should take (mp/5s/). If it takes 25 seconds, we multiply mp by 5. As this operation is not repeatable and happens only once, when the work starts, its performance is not very important, but yet the user cannot wait for about 10 minutes to start working. As the performance depends on the users' computer, the performance parameters are defined for minimum system requirements of the program. In the example above, we just used values based on local resources.

In line 4, we need to move the 3D cursor. 3D cursor position defines where the object is being added. It can also be used as a 3D view orientation to define where to move objects, to move the pivot point to the 3D cursor, as the rotation point in the spin tool, etc. So, it is a quite important feature, but in case it does not work users can still find workarounds. Note that there is no expected time next to mp for this action. It is because this action should work simultaneously with the click (i.e., should not take noticeable time). Like other actions in the checklist, this is one of the basic operations, so crash is unacceptable here, thus mc = 0.9. Note that all the multipliers here are conditional and this is just an example. In real world example, probably, multipliers should be chosen more thoroughly. nextStep is 1 here, because even if the 3D cursor cannot be moved, we are still able to add an object. To perform this step using the Python API we do the following:

```
1  # Move 3d cursor, m=1
2
3  import random
4  import bpy # python module for blender
5  x = random.randint(0, 100)
6  y = random.randint(0, 100)
7  z = random.randint(0, 100)
8
9  bpy.context.scene.cursor.location = (x, y, z)
10
11 assertEqual(bpy.context.scene.cursor.location.x, x)
12 assertEqual(bpy.context.scene.cursor.location.y, y)
13 assertEqual(bpy.context.scene.cursor.location.z, z)
```

Fig. 2. Elementary Operation: Move 3D Cursor

This is an elementary operation for moving the 3D cursor. The first line comment shows the corresponding general operation (in the checklist) and the multiplier. As in this case only 1 elementary operation corresponds to the checklist operation, its multiplier is 1. Note that the case is not always the same (the coordinates are randomly generated) and the test also checks if the operation was performed successfully or not.

In the 5th line of the checklist, we have the "Add object" operation. Many elementary operations correspond to this operation (see Fig. 3): there are lots of groups of objects, and each group itself contains various objects.

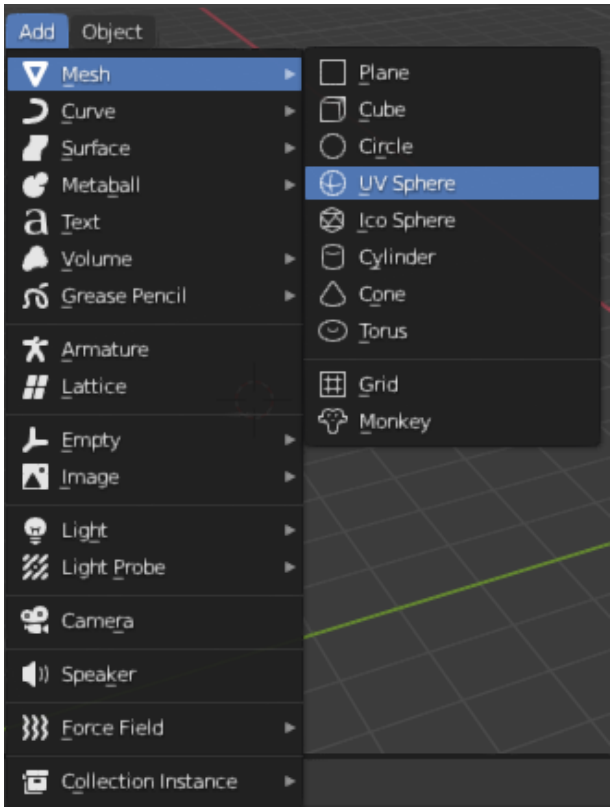

Fig. 3. Add Object

The Python code below is an example of the "Add object" operation. It adds a cube in the current location of the cursor. As all objects can be used for creating different 3D models, and their importance is dependent on what exactly the user tries to create m=1 for all objects. Note that if the object is not added then we cannot perform the next action, i.e., we cannot change its geometry.

The last command in the checklist is "Change geometry". First of all, the user should switch to the edit mode in order to change the object's geometry, i.e., move the object's vertices, edges and faces, and then perform the corresponding operations. For this general action, there are 3 possible elementary actions (move vertices, edges, faces). All of them are important while creating a 3D model, but considering the fact that if a user is not able to move the edge, he/she can choose vertices of the edge and move them together (so that the edge will be moved), or choose all edges/vertices of a face and move it. The most important one in those operations is moving vertices, and then edges, then surfaces. Thus, in this case,

```
# Add object, cube, m=1
import bpy

initial_count = len(bpy.context.scene.objects)
c1 = bpy.ops.mesh.primitive_cube_add(enter_editmode=False,
align='WORLD',
location=bpy.context.scene.cursor.location, scale=(1, 1, 1))

count_after_add = len(bpy.context.scene.objects)
assertEqual(initial_count, (count_after_add - 1))
```

Fig. 4. Elementary Operation: Add Object

the multiplier for each operation will be different:

```
# Change geometry, vertices, m=0.9
# . . .
```

```
# Change geometry, edges, m=0.8
# . . .
```

```
# Change geometry, faces, m=0.7
# . . .
```

Fig. 5. Elementary Operation: Change Geometry

For the given example, the Solver moves the 3D cursor to different positions, adds different objects, changes their geometry, and makes sure that these operations work as expected for different objects (i.e., checks that the Python tests are passing). To check how the Solver behaves if the operation does not work, we can simply use assertNotEqual function instead of assertEqual (e.g., instead of "assertEqual(bpy.context.scene.cursor.location.x, x)" we can write "assertNotEqual(bpy.context.scene.cursor.location.x, x)"). The Solver will combine different elementary tests together, create test cases and run them.

To run tests, we use the following command:

In order to use the Solver for different programs, we use a configuration file, which defines how to run tests (e.g., paths to test cases, checklists and elementary operations).

```
ctest -R <test_name> -C Release --output-on-failure
```

Fig. 6. Command For Running a Test

5. Conclusion

We propose a new approach for test automation and test results evaluation considering the testing of programs as a RGT-class problem. In this work:

1. tools defining the types of knowledge for testing the target application are described. The described knowledge is being integrated into RGT Solver and being used to run test cases, test scenarios with later evaluation of test results.
2. An approach for evaluating the state of the program during the testing is proposed.
3. The adequacy of the proposed approach is being experimented with the open-source Blender application.
4. The proposed approach solves drawbacks of the model-based testing approach, namely, allows to generate test cases dynamically.

The described solution is generic for the RGT Solver and can be used for testing various applications.

References

[1] K. Olsen and M. Posthuma and S. Ulrich, " Certified Tester Foundation Level Syllabus", *International Software Testing Qualifications Board*, pp. 56–62, 2019.

[2] E. Pogossian, *Constructing Models of Being by Cognizing*. Yerevan, pp. 150–159, 2020.

[3] *World Quality Report*, Capgemini, Sogeti, Micro Focus, pp 16–37, 2021

[4] D. Rakhi, J. Ashish, N. Karunanithi, J. Leaton, C. Lott, G. Patton and B. Horowitz, "Model-based testing in practice", *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, Los Angeles, CA, USA, 1999, pp. 285-294, doi: 10.1145/302405.302640.

[5] I. Schieferdecker and A. Hoffmann, *Model-Based Testing*, IEEE Software 29.1, pp. 14–18, 2012.

[6] E. Pogossian, V. Vahradyan A. Grigoryan, On competing agents consistent with expert knowledge, *Proceedings of Second International Workshop, AIS-ADM 2007,Autonomous Intelligent Systems: Multi-Agents and Data Mining*, St. Petersburg, Russia, pp. 229–241, 2007.

[7] M. Botvinnik, *Computers in Chess: Solving Inexact Search Problems*, Springer-Verlag, New York, 1983.

Փորձագիտական գիտելիքների վրա հիմնված RGT SOLVER-ի կիրառումը ծրագրային ապահովման թեստավորման խնդրում

Մանե Պ. Բունիաթյան[1], Սեդրակ Վ. Գրիգորյան [2] և Էմմա Հ. Դանիելյան[3]

[1]Synopsys Հայաստան, Երևան
[2]ՀՀ ԳԱԱ Ինֆորմատիկայի և ավտոմատացման պրոբլեմների ինստիտուտ, Երևան, Հայաստան
[3] EPAM Հայաստան, Երևան
e-mail: buniatyanmane@gmail.com, addressforsd@gmail.com, emma_danielyan@yahoo.com

Ամփոփում

Թեստավորումը ծրագրի որակը գնահատելու և շահագործման մեջ ծրագրային ապահովման ձախողման ռիսկերը նվազեցնելու միջոց է: Ծրագրում սխալների առկայությունը կարող է բերել ինչպես ֆինանսական կորուստների, այնպես էլ մարդկային զոհերի (օրինակ, բժշկական սարքավորումների կամ մեքենաներում առկա սխալները): Այսպիսով, բարդ է թերագնահատել թեստավորման կարևորությունը: Թեստավորման մոտեցումները կարելի է բաժանել 3 հիմնական խմբերի, որոնցից մեկը փորձի վրա հիմնված (experience-based) թեստավորումն է: Այս պարագայում թեստերը ստեղծվում են՝ հիմնվելով թեստավորողի գիտելիքների և ինտուիցիայի, ինչպես նաև նախկինում նմանատիպ ծրագրերի հետ ունեցած փորձի վրա: Փորձի վրա հիմնված մոտեցումներն օգնում են բացահայտել այնպիսի սխալներ, որոնք շատ բարդ է հայտնաբերել ավելի համակարգված մոտեցումներով: Այս աշխատանքում մենք առաջարկում ենք փորձի վրա հիմնված թեստավորման ավտոմատացում՝ օգտագործելով կոմբինատոր խնդիրների RGT դասը: RGT դասի խնդիրների լուծման համար մշակվում էRGT Solver-ը՝ ծրագրային փաթեթ, որը կուտակում է փորձագիտական գիտելիքներ և ստեղծում է արդյունավետ ռազմավարություններ RGT դասի խնդիրների լուծման համար: Առաջարկում ենք RGT Solver-ն օգտագործել ծրագրերի թեստավորման խնդրում: Solver-ը ստեղծում է թեստային իրավիճակներ՝ կախված ծրագրի արձագանքից/պատասխանից և գնահատում է դրանք ըստ նախապես սահմանված չափանիշների: Այս մոտեցման աղեկվատությունը փորձարկվում է եռաչափ մոդելավորման եBlenderե ծրագրի միջոցով:

Բանալի բառեր՝ RGT դաս, RGT Solver, ծրագրային ապահովման թեստավորում, փորձագիտական համակարգեր:

RGT SOLVER на основе экспертных знаний для тестирования программного обеспечения

Мане П. Буниатян[1], Седрак В. Григорян[2] и Емма Г. Даниелян[3]

[1]Synopsys Армения, Ереван
[2]Институт проблем информатики и автоматизации НАН РА, Ереван, Армения
[3]EPAM Армения, Ереван
e-mail: buniatyanmane@gmail.com, addressforsd@gmail.com, emma_danielyan@yahoo.com

Аннотация

Тестирование программ-это способ оценки качества программного обеспечения и снижения риска отказа программного обеспечения в работе. Очень трудно недооценить важность тестирования: проблемы с качеством программ могут привести как к финансовым потерям, так и нанести ущерб здоровью людей (например, когда ошибка находится в медицинских приборах, автомобилях и т. д.).

Методы тестирования можно подразделить на 3 основные группы. Одна из них - это методы, основанные на опыте. Здесь тестовые примеры создаются на основе знаний и интуиции тестировщика, а также на его опыте работы с аналогичными приложениями и технологиями. Эти методы могут быть полезны при определении тестов, которые не легко идентифицировать другими более систематическими подходами к тестированию. В зависимости от подхода и опыта тестировщика, эти методы могут обеспечивать широкую степень покрытия и эффективность тестирования. В данной статье мы предлагаем метод тестирования на основе опыта (автоматизация тестирования) через класс комбинаторных задач (RGT класс). RGT класс включает такие важные задачи, как защита от вторжений в компьютерные сети, разработка оптимальной стратегии управления и маркетинга в конкурентной среде, тестирование программ, защита воинских частей от различных типов атак, проблемы со связью, отдельные виды обучения, шахматы и шахматоподобные игры. RGT Solver - это программа, которая накапливает экспертные знания и разрабатывает эффективные стратегии для решения задач класса RGT. В качестве экспертной системы для тестирования, основанного на опыте, предлагается использовать RGT Solver. Solver генерирует тестовые ситуации на основе ответа/реакции программы и оценивает их по ряду заранее определенных критериев. Адекватность метода показана на примере приложения с открытым исходным кодом "Блендер".

Ключевые слова: RGT класс, RGT Solver, тестирование программного обеспечения, знания, экспертные системы.