UDC 004.05, 372.8

Teaching the Discipline “Software Testing and Verification” to Future Programmers

Sergey Staroletov

(Polzunov Altai State Technical University)

An introduction to the study of quality assurance methods is essential to understand the development of complex and reliable software. Nevertheless, the modern software industry requires the earliest possible launch of a product to the market, and the methods of formal specification and verification of programs do not find much interest among the broad mass of future programmers. In this article, the author proposes to organize a dedicated discipline and conduct seamless training in testing, test-driven development and formal verification using various methods for writing program specifications and using software tools for program checking. The purpose of discussing discipline is to redefine the attitude of future developers towards software quality, its specification and automatic checking. Within the framework of this article, the author considers his own discipline, which combines two courses – software testing and formal verification. The proposed approach of teaching is primarily practice-oriented and includes teamwork. In accordance with the current curriculum, the discipline is held in the last semester for undergraduate students (4th course). The material of the article is based on the author’s five-year experience of teaching the subject to students of the Software Engineering specialty. The article offers rather voluminous and descriptive examples of specifications and programs in model languages.

Keywords: *teaching, software models, testing, specification, formal verification.*

1. Introduction

The development of reliable programs is inconceivable without the organization of the industrial testing process in accordance with the latest established trends in software engineering. In order to obtain a holistic picture by future software engineers, it is advisable to introduce a special discipline in which quality assurance methods in software development are studied. At the same time, this course should be expanded with formal verification methods, which have recently been developed as one of the directions of modern software engineering devoted to proving the behavior of program models for the operation of software systems in conditions with increased reliability requirements. The combination of testing, test driven development, static analysis, and formal verification allows the software engineers to seamlessly move from tests to models and choose the right quality assurance method based on labor costs and system requirements.

Setting up an author's new course is always fraught with difficulties. In this article, the author presents a plan for conducting classes in his discipline related to testing, formal specification and verification methods in order to take care of the quality of software by future programmers. Since the course was made from scratch, there were some reasons for creating it.

After the transition to a two-level education system in our country (undergraduate student+master, 4+2 years) from a 5-year education system (specialist or engineer), the question arose of revising the curriculum with the development of new disciplines. Note, we use a competency-based approach, which means that the Ministry of Education lowers the expected competencies from above, and then the University itself decides which disciplines are needed for covering them. At the same time, the current elaborations by the departments were mostly used and, as a rule, the existing disciplines were updated. The same cannot be said about some competencies that required the creation of new disciplines. For example, for the discipline under discussion, the competency "the ability to apply testing and verification methods" was set (however, it does not imply which methods and whether they should be formal – this should be decided locally depending on the qualifications of lecturers). At the same time, this competency belongs to the professional type, and not general, which implies the obligatory study of the discipline by all groups of the specialty.

Since the author had a background both in the fields of formal verification and model based testing, had industrial experience and understood that formal methods are primarily applicable to study the latest achievements in the field of software engineering (nevertheless, their applicability in industry is limited by time and complexity), then he enthusiastically started creating this course. The main goal was to introduce as much specifications as possible into the development process and show that program correctness is only possible if there exist additional artifacts that the future developer should pay attention to. If these artifacts are expressed formally, then this opens the door to prove the correctness of systems with complicated quality requirements. If informally, then it offers at least manual or automatic regression check, which allows taking into account incremental changes in constantly changing software.

The resulting course for the 4th year of the specialty "Software Engineering" [56] has the following structure: 8 lectures, 8 labs for 17 weeks, the assessment includes three equally weighted components: the attendance, semester work (weighted by 8 labs) and a final test, which will be discussed later. The author already has five years of teaching experience of the discipline, so some modules have been updated due to the feedback. The course has also served as the basis for more streamlined courses for "Computer Science" specialty as well as for college students with less lab work and almost no

formal verification material. The author created a textbook in Russian [57] for the course, but it is not available in the public domain, however, this article describes the main points from there. Preliminary requirements for students are the following: (1) knowledge of mathematical logic as well as theory of algorithms and (2) practical skills in writing programs in modern IDEs. The skills obtained as a result of mastering the discipline are used in the implementation of the graduation projects by students and (possible) in their future work.

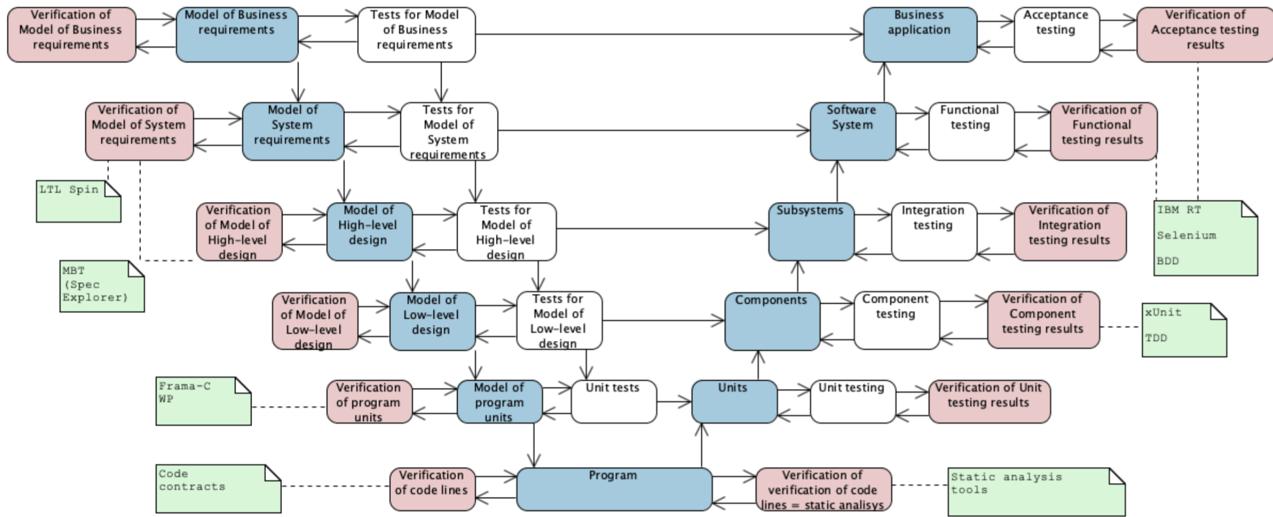

Fig. 1. Triple-V model and tools

The author would like to highlight two key points on which the approach to this discipline is based: (1) the theorem on the undecidability of testing within the framework of the existing theory of algorithms and (2) the Triple-V model, indicating levels and specific tools for the application in the course.

Let us assume that the program works in a simple single-tasking environment, has a set of control states, and after each step of transition to the next state, the values of variables are output. If after the completion of the program, the output is exactly equal to the expected one, then the program is recognized as a correct one. The problem of testing is whether it is possible to make such a check for any program P according to its description, or formally, for the input X, the output Y and the Gödel number $N_g(P)$ to make a translation:

$$X * N_g(P) * Y \Rightarrow \Big\{true, false\Big\}$$

Then the given problem is undecidable, since it is a more general case of the translatability problem, which is closely related to the *CircuitSAT* and to the *Halting problem* which are undecidable problems [23]. Thus, testing (using current assumptions about algorithms on current computer architectures)

cannot show the correctness of a program only using data from its code. This all implies either a transition to the search for errors or counterexamples (that is, violations of the correctness of work), or to the use of artifacts (models and specifications) additional to the code. Also, if it is impossible to prove the complete correctness of the program, then one should use as many projections of the programs into different models and apply different methods to check different aspects of the programs, which is called the *principle of methodological diversity*.

To solve problems with the impossibility of proving programs by testing, formal verification methods are began to develop, which are according to Clarke *"mathematically based languages, techniques, and tools for specifying and verifying systems that operate reliably despite this complexity"* [12]. A prospective review on the occasion of the fortieth anniversary of formal methods was published in the work [8]. Also of note are the annual workshops: International Symposium on Formal Methods [31], NASA Formal Methods [43], and Model Checking Software [32].

In [20], the Triple-V (triple-waterfall) model of software development is considered. The Triple-V scheme is an idealized scheme with maximum quality control for systems that are decomposed into modules, components and subsystems. Here at each development phase the result is a model and the phases of verification of model tests (left) as well as verification phases of test results (right) are added. We consider a modified model of the Triple-V, indicating the possible tools and methods for some levels, which are studied in the discussed discipline (Fig. 1).

As for related works, today the trend is to create open courses and present them at specialized international workshops. In particular, in the field of formal methods, the community holds a Formal Methods Teaching Workshop (FMTea). For example, in 2021, ten significant works were published [19] with the presentation of their courses, including the use of formal methods in games, working with Dafny and Isabelle tools. In 2023, a lot of papers were submitted for the outgoing workshop [21], it is due to the large number of online seminars and the dissemination of information about the workshop during the pandemic.

In the next section with eight subsections, we consider all the components of the discipline plan, and in conclusion, we discuss the results of conducting final test for the discipline. It should also be noted that we use a language-agnostic approach: by the 4th year, students already code in various programming languages. Therefore, we present examples in different languages that are most convenient for showing each new approach in specification, testing, and verification.

2. Structure of the discipline

2.1. Software specification and black box testing

The purpose of the first module is to prepare students for writing natural language specifications and black box testing them. To begin with, the concept of a software bug is introduced as a violation of the specification (after all, as it was shown before, just a program in any form is not enough). Consequently, students become technical writers and manual testers. However, it is proposed to write not just textual descriptions, but rather formal specifications according to the famous *Hoare triples* approach [24] (if it is understood in a worldly sense):

$$\{Precondition\}Program\ action\{Postcondition(result)\}$$

By the time of this course, the trainees already know how to create interactive programs or web applications in different languages. In addition, during four years of training, everyone has a number of programs passed as a result of laboratory works. These programs, obviously, were made in the last moments and have bugs. Such programs are supposed to be described and tested.

An example specification for an engine ECU simulation program is shown in Table 1. There is also a related publication [59] on demonstration of using industry standards and the Hoare triples approach to check non-trivial cyber-physical models.

Since we also promote the importance of describing the software architecture in some other courses, in addition to the text specification, it is proposed to draw a diagram on the software in the form of a UML use-case diagram, where the actor is a window or program mode which is associated with other actors and transition conditions between them. As a result, the specification will be in two formats – textual and graphical, and a potential tester can think over test scenarios in advance. All students are divided into minigroups of about three people, choose its software and distribute work within the group.

When the artifacts are ready, the minigroup submits them along with the program to a GitHub/GitLab repository and waits for the submitted programs from other minigroups to appear. Then they start writing about the bugs found by others in a bug tracker, in this case using the “Issues” tool. To learn how to describe correct bug reports, the author suggests getting acquainted with public bug trackers (for example, Google Chrome [33], etc.) and taking into account the 4W+1H principle [1]. As a result, we will get the involvement of students in the specification and finding bugs from friends, especially from those who did not specify their software in enough detail.

Table 1

Specifications in tabular form for an engine model

	Precondition / Action	**Postcondition**
1	The ambient temperature is greater than -30 and less than 0, the battery voltage is greater than or equal to 12V, or the ambient temperature is greater than or equal to 0, the battery voltage is greater than or equal to 11V / Start button pressed	A message about the successful start of the engine is shown, the output parameters panel displays the current parameters of the engine, the slider “Battery voltage” and “Current value, V” on the input parameters panel increases over time until it reaches 14, the “Current state” on the launch panel changes from “Engine is not running” to “Engine is running”
2	The engine is not started, the ambient temperature is below -30, or the ambient temperature is above -30 and less than 0, the battery voltage is less than 12V, or the ambient temperature is greater than or equal to 0, the battery voltage is less than 11V / Start button pressed	A message is shown stating that the engine cannot start
3	Engine is running / Start button pressed	A message is shown stating that the engine has been started before
4	Engine is running / Stop button pressed	“Engine was stopped” message is displayed, “Outputs” set to zero, “Current Status” on Launchpad set to “Engine is not started”
Invariants: the program is not closed, there are no messages about exceptional situations, the information in the program window corresponds to its internal state		

The work of the minigroup is judged on the details of specifications and the quality of bug reports from others. This topic also explains testing standards [29] and provides links to the International Software Testing Qualifications Board training syllabuses [30] for those who are interested in the professional work of a tester.

2.2. Code level. Unit testing and project documentation

After testing systems without knowledge of their internal organization, we move on to unit testing the code. Such testing can be formally represented as:

$$\bigwedge_{M_i \in Testcase} M_{act_i}(x_1..x_n) == Ret_{expect}(M_i)$$

where $M_{act_i}(x_1..x_n)$ is the result of running the tested method (function) M_i with the specified parameters $x_1..x_n$, Ret_{expect} is the expected return value of the method. Methods (functions) are grouped into test cases. Conjunction means that if one of the tested values does not correspond to the expected, the operation of the entire test suite is considered incorrect. While the pioneering standard for such type of testing was proposed in 1987 [9], the commence of industrial applicability of this method was done around 1998 by the famous people in Java software engineering, Kent Beck and Erich Gamma, who invented JUnit [6].

During the labs, students are required to understand that it is necessary to build the code in such a way that it becomes checkable by the unit testing method. For arbitrarily written software, where I/O calls are mixed with program logic, such checking is difficult to perform without refactoring the code, so here the author reminds students of the need for organization of architecture with separation of responsibility [37] and design patterns [17]. Students should also understand that writing unit tests is on the developers' duty and the unit testing is primarily a development methodology, not a pure testing methodology. However, writing such artifacts as tests today is the preferred way to organize a project by default. Now starting the project from $main()$ is not practical until the logic has been well tested (by running only the tests, not the project as a whole). The lecturer can also make some technical introductions about xUnit frameworks as well as testing support by modern IDEs (for example, [34]), and about the need for a CI process in a project when tests are run when they are committed to the version control repository [7].

This all allows us to get rid of regression errors, i.e. errors associated with new changes that are correct in themselves but lead to the inoperability of already written code. Thus, with the support of previously written tests, it is possible to conduct regression testing at the code level. Therefore, with

such testing, we can solve a very important problem for modern software, which is distinguished by its incremental structure.

To support teamwork when writing specifications, the following techniques are suggested:

- Tests for a code should not be written by the person who wrote the code.
- To make it easier for others to write tests, the developer is advised to make specifications for the methods that are supposed to be tested.
- The specifications at this stage should not be formal but should be sufficiently capacious, in particular, each parameter and return value should be described.
- Documentation is proposed to be made in the JavaDoc annotation format, or rather, in a more generalized form for all major programming languages processed by the DoxyGen tool [14].

An example of such a specification for a matrix processing library:

```
/**
* Bring the matrix to a triangular form
* @param matrix - nonzero original matrix
* @param col - complement of matrix, column vector of
size as number of rows of original matrix (may be null)
* @param useMainElement - the parameter indicating whether
to use the main (maximum modulo) element and rearrange
rows, @see maxPosInColumn
* @return Returns the number of row permutations, while
changing the original matrix
* @throws Exception when original matrix is degenerate
*/
public static int toTriangleMatrix(double[][] matrix,
 double[] col, boolean useMainElement) throws Exception {
  ...
}
```

It was noticed that when writing specifications and writing tests based on them, students begin to have questions about their completeness. So, we can think that the approach of creating informal (but nevertheless at least some) specifications and tests in accordance with them is the first step towards creating future formal specifications.

2.3. *DD development methodologies. Test Driven and Behaviour Driven Development

Considering the methodologies in software engineering over time, we can state that by now the process has moved from "code writing" programming methodologies named *OP (like OOP and AOP), to development methodologies (something more significant than just writing code) named *DD, with examples will be considered here: MDD, TDD and BDD.

Remembering that the unit testing approach is a development methodology and that we consider teaching it for student programmers. Therefore, it is required to dive into programming at this stage, but with the use of modern methodologies that assume that the code is not a primary, but a secondary stage of the development. The article by Harry Robinson [53] presents the application of graph theory to the tests generation. The program here is given by a transition system, that is, the graph is a model, and then the code and tests can be obtained from it. In the modern development of cyber-physical systems (see our already mentioned article [59]), engineers (1) build a physical model based on diagrams using formulas according to physical laws, (2) simulate its operation (analogous to testing with the analysis of graphs) and (3) finally, based on this model, the code for a microcontroller and workpieces for tests can be generated. All of the above corresponds to MDD – the Model Driven Development approach, where there is an initial abstract model and everything else in the development comes from it.

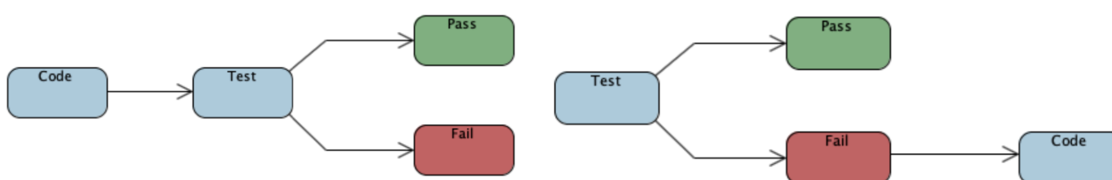

Fig. 2. On the left is the unit testing process (the code encourages us to write a test), on the right is the TDD development process (the test encourages us to write new code)

If we consider a development process, where the initial attribute is a test, then we move on to the test driven development (TDD) process [5]. The author of the methodology is Kent Beck, one of the developers of JUnit [6]. It should be noted here that this approach was borrowed (as Beck notes) from the development of programs on punched cards, when the developer saw the input and output sequences and thought about what transformations should be applied to obtain the latter. This methodology, in addition, expects the writing of program code (changes) only after the discovery of the fact of not passing any test (Fig. 2) and assumes to use the laziness of the human being.

Therefore, these changes should be minimal and it is sufficient to pass only the failed test. Thus, for further implementation, we need to think about tests and evolve the code according to them. In this case, when developing in the IDE, we immediately have the opportunity to create the code templates we need, which speeds up the development (see Fig. 3).

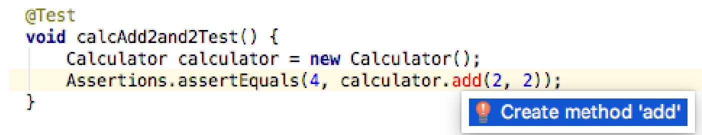

Fig. 3. Creating a method during a test writing

As for the sufficiency of the method for passing the test, it is initially written trivially:

```
public double add(double a, double b) {
    return 4;
}
```

Further, when other tests appear, the implementation can turn into a set of switches by parameters of the method and return the expected constants, which can later be generalized into formulas. However, this code can be used at any stage (it is already in the working state on given inputs according to the tests), can be transferred to other team members to develop other modules, or demonstrated to customers. At the same time, some generalization can be done later if there is time.

The process of performing work on the TDD part by the students consists in receiving a task for the implementation of a computer system or game (which can be completed in a short time), and proceeding to individual implementation according to the task using the methodology, starting with tests. In the time of the development, each step (written test or added/changed code) is committed to the student's private git repository, and when submitting the assignment, the project is shown to the teacher in the form of a sequence of changes.

However, the current applicability of the TDD process is limited due to the following things:

- the developer thinks too much about the code;
- the customer is not involved into the process;
- managers think that the process is costly;
- small amount of companies still use it because of lack for experience how to setup this process.

The BDD (Behavior driven development) methodology was proposed to involve customers and domain experts in the development and testing process. These people write specifications (work scripts) for the system, how they expect the system to work correctly, then the specifications are tied

to program objects and become unit tests. At the same time, the developer can set up IDE and see which specification (and not the test!) is being executed and what are the results of its checking. The implicative Gherkin language [54] with constructions of the form *given, where, and, then* is used here as a specification language. It creates the illusion of writing specifications in natural (controlled) language. The approach was first implemented for Ruby in the Cucumber framework, ported to major programming languages, and, at least the author knows, it is actually used in the industry in web development companies, where specifications come from real customers. So, an example of BDD scenario:

```
Scenario:
    Given I have my software calculator
    When I have entered 2 as first operand
    And I have entered 2 as second operand
    And I press 'Add'
    Then The result should be 4
```

In the case of Java, a Step Definition file for the scenario is then generated using IDE tools. It contains ready-made methods for the given natural language scenario (*iHaveMySoftwareCalculator*, *iHaveEnteredAsFirstOperand*, *iHaveEnteredAsSecondOperand*, *iPressAdd*, *theResultShouldBe*). The developer's task is now to implement the code for creating an environment for a test object (its creation and passing parameters), and then check the expected and actual values:

```
public class MyStepdefs {
 private Calculator calc;
 int operand1, operand2, result;

 @Given("^I␣have␣my␣software␣calculator$")
 public void iHaveMySoftwareCalculator() {
  this.calc = new Calculator();
 }
 @When("^I␣have␣entered␣(\\d+)␣as␣first␣operand$")
 public void iHaveEnteredAsFirstOperand(int number) {
  this.operand1 = number;
 }
```

```
 public void iHaveEnteredAsSecondOperand(int number) {
  this.operand2 = number;
 }
 @And("^I␣press␣'Add'$")
 public void iPressAdd() {
  this.result = calc.add(operand1, operand2);
 }
 @Then("^The␣result␣should␣be␣(\\d+)$")
 public void theResultShouldBe(int expected) {
  Assert.assertEquals(expected, this.result, 1e-5);
 }
}
```

Thus, we got a mapping of texts in controlled natural language into unit test-style calls. The initial specification can be written by a non-programmer, and then the developer should take care of the code so that this specification is executed correctly. Here one can also use all the assumptions of the TDD approach about the minimum code for passing tests. In addition, current implementations support tabular data values for working with datasets. The task for students here is also to implement their previous exercise, but now with the Cucumber-like environment set up and the specifications (not tests) written in advance than the code.

2.4. Functional automated testing

Testing programs with user interaction requires writing automation scripts that replace manual testers. If there is a means of recording and reproducing such scripts, then, it allows us to set the initial conditions for the test (by influencing the controls), perform the necessary actions (button click, for example), and read the resulting state of the program from its user interface. When conducting this course, the author all the time defends the approach in modern testing – what the teacher can enter with his own hands and check with his own eyes, it could be done automatically and this should be done on each commit of a change. Therefore, we are here learning how to manage programs and do "assert" to check the expected value against the actual one, but in this case, at the functional level by managing a ready-made application with its user interface.

When conducting practical tasks, we are primarily interested in an automation tool that allows

us to record scripts in the form of programs in some programming language. Here one can fully control the setting of the initial values for the program, make different loops with different data, and compare the results. Historically, automation has been well implemented for programs under MacOS from Apple. In fact, all MacOS programs have some kind of interface for "listening to what they tell us from the outside" and in the Automator tool [35], one can record and play scripts in a special language. Further, a good support for UI scripts was implemented [4], and a functional test for the designed solution in the XCode environment is launched as a unit test. For Windows and Linux, the IBM Rational tester [28] has historically been a good tool, allowing the users to write scripts in Java, support tests by datasets and other program management. It does not require source code, all operations are carried out according to a user interface model that the program can access. In this approach, it is also possible to express functional tests as unit tests. However, times and technologies are changing, and the most popular automation tool in the world today is Selenium [55]. It targets for modern web-based applications that work in the browser, so it is advisable to focus on it.

Selenium consists of three parts: Selenium IDE for writing and playing scripts, Selenium WebDriver for controlling the browser programmatically, and Selenium Cloud for running scripts on the server. Selenium IDE operates as a browser plugin that works with the DOM model of the current document and intercepts events when clicking on links, submitting forms, and so on. Also, it provides a context menu where the user can choose, for example, which element on the page can be verified now. For each action, a certain log is generated in the form of Selenium commands, which can be then replayed. Manual testers usually limit themselves to recording sequences of working with the web application under test, where they click on elements and make sure that the required element is on the page. For example, when the user successfully logs in, an element to edit user's data appears – its presence can show us that the user is logged in:

```
open                    /blog/login
clickAndWait            link=Registration
type                    id=inputEmail serg_soft@mail.ru
type                    id=inputPassword password
clickAndWait            //button[@type='submit']
verifyElementPresent    link=Edit my data
clickAndWait            link=Exit
```

The considered log is a specification, according to which it is possible in the future to generate a program code for different programming systems using a set of Selenium WebDriver libraries. This

allows testers to write initial scripts, and developers integrate them into their development tools to control the browser from programs.

An example of the generated code from the given log:

```
driver = new FirefoxDriver();
...
driver.Navigate().GoToUrl(baseURL + "/blog/login");
driver.FindElement(By.LinkText("Registration")).Click();
driver.FindElement(By.Id("inputEmail")).Clear();
driver.FindElement(By.Id("inputEmail")).SendKeys
    ("serg_soft@mail.ru");
driver.FindElement(By.Id("inputPassword")).Clear();
driver.FindElement(By.Id("inputPassword")).SendKeys
    ("password");
driver.FindElement(By.XPath("//button[@type='submit']")).
    Click();
Assert.IsTrue(IsElementPresent(
    By.LinkText("Edit my data"))); //assertion
driver.FindElement(By.LinkText("Exit")).Click();
```

One may notice that this code is the secondary artifact derived from the initial specification in the form of Selenium commands. Such specifications are easier to write, modify, and maintain.

A developer with some experience will be able to further learn WebDriver commands and integrate them into unit tests, as well as use such commands in writing programs in accordance with the discussed TDD and BDD methodologies.

Lab assignments in this module include writing various scripts for existing programs, checking their correct state on key interface elements, and developing a simple web application with authentication using BDD and WebDriver.

2.5. Static checks and dynamic program analysis

Static analysis is very important in the modern world of programming. A large number of vulnerabilities today arise from code written with incorrect assumptions or gross errors. Modern languages like Kotlin, Swift and Rust are come with built-in null-safety and type checking, while classic languages like C/C++ or Java are neither memory-safe nor type-safe. At the same time, a lot of modern

code is written (and is being written) in unsafe languages. Therefore, it makes sense to use tools that check the source code and identify typical instances of potentially unsafe behavior. Static checkers or linters analyze the abstract syntax tree of the program (this is shown for example in our work [60]) in order to find typical errors and vulnerabilities. They also build a limited control flow graph to find potential paths with erroneous behavior and use various heuristics. Now in industrial development processes, it is good practice to launch a static analyzer during the build of a project using a CI tool, so it is necessary to accustom future developers using such tools. We consider both the easy-to-use cppcheck [36] and PVS-Studio [52] built into the development environment, as well as the popular SonarQube [13].

For complex programs, especially those working in a multi-threaded environment and dealing with memory in a non-trivial way, static checking will not do much. For these purposes, dynamic analyzers are used, in particular the Valgrind tool [44]. One can execute a long-running tool like a server for a while and observe possible incorrect operation and memory leaks.

The labs consist of checking past code of the students and discussing the output of analyzer with the teacher to get feedback on their code. In this case, the students can learn something new about writing quality code based on messages from analysers.

2.6. Hoare triples. Deductive verification. Code Contracts

At this point, the students already have a good idea of what the specifications and Hoare triples are, and it is time to try to check them at the code level. Bertrand Meyer's Eiffel language was the first attempt to exploit Hoare's ideas in a general-purpose object-oriented programming language [16]. At the same time, the so-called contracts have been introduced as part of the syntax of the language: preconditions and postconditions have been added to methods, and invariants have been added to classes. Of the interesting things, Eiffel offers the generation of random tests under the contract and introduces its own multithreading model based on contracts [42]. However, for an average developer, learning new languages just because the contracts can be specified there does not seem to make sense. Therefore, it is better to learn how to write contracts for existing languages using syntactic extensions or annotations. The most successful state-of-the-art product for developers, according to the author, is the experimental MS Code Contracts tool by Microsoft Research, which integrates the contracts approach into the C# language [40].

An example of a contract for the "Student" class that can be checked right in the development environment [18] (code like *stud.age = 10* will violate the contract and get highlighted):

```
public Student(String name) {
 Contract.Requires(name != null,
  "Name␣should␣not␣be␣empty");
 Contract.Requires(name.Contains("␣"),
  "Name␣should␣have␣at␣least␣2␣words");
 this.name = name;
 this.age = 16;
 this.yearOfAdmission = DateTime.Now.Year;
}

[ContractInvariantMethod]
private void ObjectInvariant() {
 Contract.Invariant(this.name!= null && this.age >= 14
 && this.age <= 80 && this.yearOfAdmission > 2000,
 "Student's␣fields␣are␣not␣set␣correctly");
}
```

Let us now consider more complex contracts for the container class “Student group”. The method that adds the “Student” object to the list checks that the specified object is not empty and it is not in the current list. It is checked by using the lambda predicate in C#. The postcondition is that the number of elements in the list has increased by 1 and the list contains the added element:

```
protected List<Student> list { get; set; }

public GroupStudents() {
 list = new List<Student>();
}

public void AddStudent(Student stud) {
 Contract.Requires(stud != null);
 Contract.Requires(! this.list.Exists(x => x.number ==
 stud.number && x.name == stud.name));
```

```
 Contract.Ensures(list.Count ==
 Contract.OldValue(list.Count) + 1);
 Contract.Ensures(list.Contains(stud));
 list.Add(stud);
}
```

As for the class invariant, consider the code to ensure that there will never be two students with the same number in the list:

```
[ContractInvariantMethod]
private void GroupInvariant() {
 Contract.Invariant(Contract.ForAll(list, x =>
 Contract.ForAll(list,
  y => (x != y && x.number != y.number)
  || (x == y && x.number == y.number)
  )));
}
```

Thus, this approach allows us to embed correctness conditions inside classes and check them using contract library methods, C# language tools, and possibly auxiliary methods. At the same time, there is no talk of any sufficiency of such a check.

To specify contracts for code with improved reliability requirements, we consider the Frama-C approach [22] (an extensible platform for static and dynamic analysis) and its WP (Weakest Precondition) subsystem for specifying formal specifications for C code. This approach allows developers to specify contracts in the form of an ISO-standardized extension for C [2]. The place for the contracts is in code comments with special keywords (vs the informal specification we learned in Section 2.1). Here, in order for the contract to be proved automatically, it is necessary to set postconditions with all changing variables in the function, as well as invariants for loop, essentially turning an imperative program into a predicative functional one. This is all done by hand, and since we will have two representations of the same code, we can guarantee its quality according to our assumptions if the contracts are proven. An example of the applicability of the approach for standard library functions on the example of working with files is well considered in the article [51]. For a different example, consider the open-source C-code of the ArduPilot project for Arduino:

```
float get_i(PID *pid, float error, float dt) {
 if ((pid->ki != 0) && (dt != 0)) {
   pid->integrator += ((float) error * pid->ki) * dt;
   if (pid->integrator < -pid->imax) {
      pid->integrator = -pid->imax;
      } else
        if (pid->integrator > pid->imax) {
          pid->integrator = pid->imax;
        }
    return pid->integrator;
   }
 return 0;
}
```

In the next snippet, we show a specification for the function. Here *old* is a memory state before calling the function and *at(..., Post)* – after calling it:

```
ensures ((pid->ki != 0) && (dt != 0)) ==> \at(pid->integrator, Post
   )  ==
CheckUp((float) (\old(pid->integrator) + ((float) error * pid->ki)
   * dt), (int)pid->imax);
ensures !((pid->ki != 0) && (dt != 0)) ==> \at(pid->integrator,
   Post)  ==
 \old(pid->integrator);
ensures ((pid->ki != 0) && (dt != 0)) ==> \result ==  \at(pid->
   integrator, Post);
ensures !((pid->ki != 0) && (dt != 0)) ==> \result == 0;
```

Firstly, it can be seen that the function changes the value of pid->integrator and there are three cases:

- pid $\rightarrow$ integrator $<$ -pid $\rightarrow$ imax: it is limited to -pid $\rightarrow$ imax;
- pid $\rightarrow$ integrator $>$ pid $\rightarrow$ imax: it is limited to pid $\rightarrow$ imax;
- otherwise, that is, (pid $\rightarrow$ integrator $>=$ -max) and (pid $\rightarrow$integrator $<=$ max): do not change of pid $\rightarrow$ integrator.

At the same time, there must first be a change of *pid* $\rightarrow$ *integrator* to *error * (pid* $\rightarrow$ *ki) * dt*. Therefore,

the solution is to create a set of lemmas and an axiomatic that is used as a function in the ensures section. Secondly, it can be noted that the function returns 0 if the first condition does not hold and does not change the value of pid $\rightarrow$ integrator. To describe the postcondition, the description of the guard conditions in the form of implications can be performed.

```
axiomatic CheckAxiomatic {
logic float CheckUp{L}(float integrator, integer max);
 lemma CheckUpMin{L}: \forall float integrator, integer max; (
    integrator < -max) ==>
    CheckUp(integrator, max) == (float)-max;
 lemma CheckUpMax{L}: \forall float integrator, integer max;
    integrator >  max ==>
    CheckUp(integrator, max) == (float)max;
 lemma CheckUpNorm{L}: \forall float integrator, integer max; (
    integrator >= -max) && (integrator <= max) ==> CheckUp(
    integrator, max) == integrator;
}
```

In general, there is a good manual with a large number of discussed specifications of well-known algorithms from Fraunhofer [3].

When performing laboratory work on these topics, students in minigroups propose contracts for their existing code, analyze examples for Frama-C and try to specify some of the algorithms previously written on their own.

2.7. Model Based Testing. MS Specification Explorer

In this module, we move from code to formalized behavioral models and consider the model based testing approach. This approach is interesting in that it can automatically generate unit tests from specified behavioral automata. The author believes that the best tool that combines research and industry for this approach is Microsoft Spec Explorer [39], originally created by MS Research for internal purposes of testing Office and Internet Explorer (however, the tool does not work with the latest versions of MS Visual Studio). The approach is based on the ASM theory [10]. An overview of the approach in a pioneering version of the tool is done in the paper [46].

To demonstrate the approach, let us briefly consider an example of describing the behavior of a login-password application in the special CordScript language [38]:

```
machine LoginScenario():Main where ForExploration = true {
 Initialize; (EnterLogin; EnterPassword; call Login;
 ((return Login/0; ResultFail){0,1}))+;
 ((return Login/1; ResultOK) |
 (return Login/2; ResultOver))
}
```

Here we are modeling that Login() can return ResultOK on success, ResultFail on failure, and ResultOver if the number of login attempts has been exceeded. For such a model, SpecExplorer generates the automaton representation shown in Fig. 4. Actions here are not states, but arcs. Next, using the

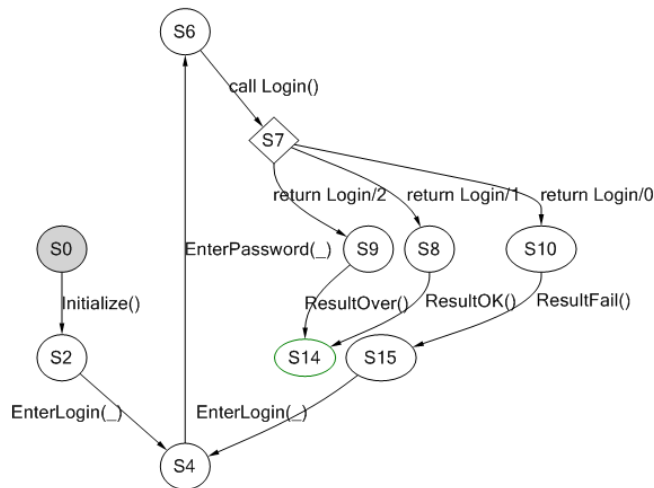

Fig. 4. A login model

parallel composition operation, we intersect the behavioral automaton with the system model:

```
machine LoginScenarioSliced():Main
where ForExploration = true {
 LoginScenario || ModelProgram
}

machine TestSuite():Main where ForExploration = true,
TestEnabled = true
{
 construct test cases for LoginScenarioSliced()
}
```

Here $ModelProgram$ is originally written in C# and presents a simplified model for implementation of the program under test with some special annotations. Spec Explorer itself builds an automaton based on this program. Ultimately, a suite of unit tests is built from the parallel composition and generated as suites that can be run like regular tests in Visual Studio.

Labs on this topic include studying ways to specify the behavior of programs in the form of automata, implementing simple model programs and generating unit tests for them, as well as introducing errors into specifications and models in order to check the generated unit tests.

2.8. Model Based Checking. SPIN tool

In this module, we move from automata models of specifications to their expression in the form of formulas of temporal logics. In accordance with the considered test undecidability theorem, additional artifacts are needed besides the code. In the case of the model based checking method, the program is replaced by its model (in the form of automata or some executable model), and in addition, the requirements for the model are set in the form of temporal formulas. If the executable model has the form of a program in a language with strict semantics, then it is possible to prove its correctness with respect to given formulas with requirements. Such a proof, however, does not guarantee the correctness of the original program in a real-world programming language, since such languages have very complex semantics that cannot be expressed formally, or program verification will be possible in this case only for simple programs due to inefficiency caused by the complexity.

This section discusses the SPIN model checker (or verifier) created by Gerard Holzmann [26]. The advantages of this product are that the model programs for it are expressed in the special Promela language, which corresponds to the CSP formalism [25] and is somewhat similar in syntax to the original EMC language [11] implemented by Clarke as the first model checking system. Requirements for programs are expressed in the LTL language [47] (in the form of predicates with boolean and temporal operators over the key variables of the program). The use of SPIN in teaching to software engineering students is especially useful, since here the model is expressed in code that they are able to understand. The ability to model interacting processes with the SPIN system allows us to simulate interactions between models of microservice programs, which is relevant today.

In the course of teaching, we learn the syntax of the Promela model language [48] and the syntax of LTL formulas for expressing various requirements patterns [15]. We also study some internals of model checking according to Clarke's works [41] and issues of their implementation in SPIN from Holzmann's articles [26]. We also touch the main problem of state explosion [41] in the model

checking and the optimization methods implemented in SPIN to somewhat bypass it.

Previously, the author created a sufficient number of good examples demonstrating the Promela language and the tasks solved on it. In a very basic example, we consider a service system as processes interacting in a given sequence [49]. Following this example, students can make models of the interaction of windows or screens in a program or models of interacting microservices. There is also a good and complex model of a partitioned operating system scheduler presented in article [61]. In Fig. 5, we show how the current implementation of the operating system model works: processes make system calls as messages and our scheduler encoded in Promela schedules them.

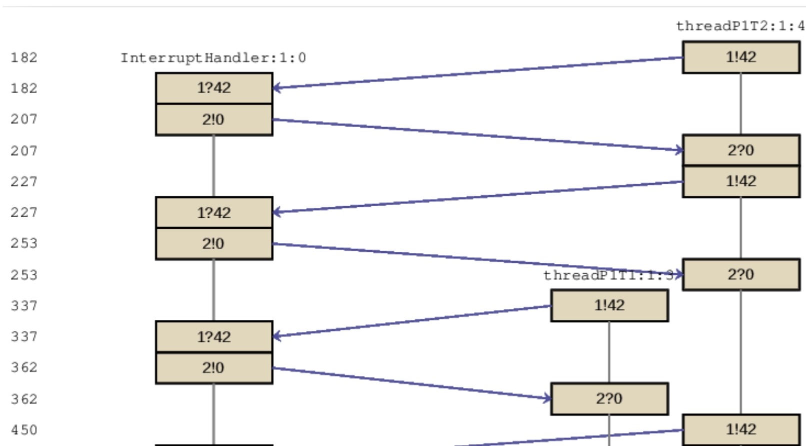

Fig. 5. Simulation of an OS model in iSpin

An interesting feature of the model checking is the generation of a counter-example when a requirement formula is violated. This provides a transition sequence that leads to a violation of the requirement. If we negate the LTL formula with the requirement, then the verifier will try to generate a path from the initial state to the accepted state, which can find a solution to the problem given as a transition system. Therefore, to solve search problems, one can encode behavior using all possible non-deterministic transitions and deny the requirement that the problem be solvable. An example of solving the *Hanoi Towers problem* is given below. Here we introduce arrays rod_i by the number of rods that store the numbers of disks on the rod ($count_i$ denotes the count of disks on the ith rod), and in the following Promela code we just try to non-deterministically move a disk from the top to the other rod or not move it:

```
do
:: count1 > 0 -> {
 disk = rod1[count1-1]; //get the top disk from the rod 1
 //and try moving it to 1st rod
 if  //here we try to move a disk...
  ::(count2==0 || (count2 < N && rod2[count2-1] > disk)) ->
  {
   printf("Disk␣%d␣from␣1␣to␣2␣\n", disk);
   rod2[count2] = disk;
   moves++;
   count1--;
   count2++;
  }
 //...or refuse to move it and try other branches
 ::(count2==0 || (count2 < N && rod2[count2-1] > disk)) ->
  skip;
 fi
}
//1->3; 2->1; 2->3; 3->1
od
ltl count_check  { [] (count3 != 5) }
```

During the proof of the negation that the problem can be solved, the verifier will try all the branches and find a solution to the Hanoi Towers problem. The full solution is given in [50]. This approach can solve difficult problems, especially when using the Swarm Model Checking technology [27].

As for laboratory work, students are invited in minigroups to describe models of their interacting programs in a simplified form and come up with requirements for them, showing the teacher the results of simulation and verification in the iSpin tool.

3. Results and Conclusion

In this discipline, the basics methods and tools for testing and verification are studied. The fact that the subject is compulsory suggests that it is aimed at the average student programmer. Nevertheless, the author believes that the majority of students, as a result of studying the course, have practically successfully mastered all of the listed methods, or at least understood what is intended for what. During the teaching of the course, a final test of 70 questions was prepared (the online test is available under the link [45]). It has already been passed by 175 students, the results are shown in Fig. 6.

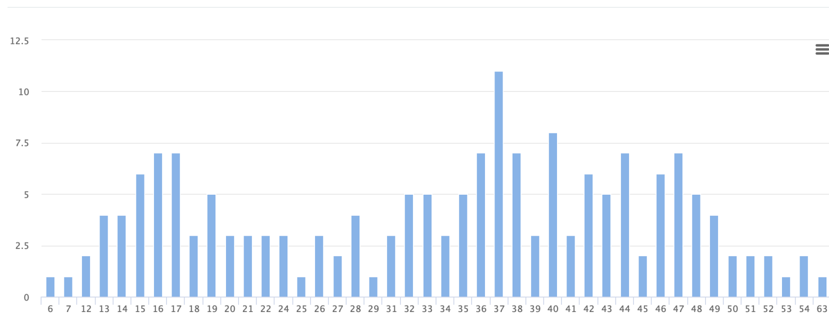

Fig. 6. Distribution of correct answers in the final test

The OX axis shows the number of correct answers, and the OY axis shows the number of passes for that correct answers. The maximum score is 63/70 or 90%. The accuracy of answers on the right side of the graph is greater, which indicates rather good mastering of the discipline by students.

Over the five years of teaching the discipline, it was necessary to change the distribution of laboratory work depending on the readiness of the audience (specifically, in the field of mathematical logic). In some years, testing methods occupied a significant amount of time, while there were also student groups that understood the examples well and made their own unique formal models. Teamwork in minigroups of three people eliminates the unpreparedness of some students and allows students to achieve basic mastery of the competencies considered.

It can be concluded that when training young developers, if from the very beginning they have an understanding of the methods and tools for testing and verification and the need to set an initial specification of the behavior of the software system, then in the future, they would be ready to produce software of a different quality level.

As for a further work, it is planned to expand the material of the manual and the course with an introduction to the verification of cyber-physical systems using formal methods. An example of specification and verification of stability properties for a continuous-time system has already been created [58].

Abbreviations. The following abbreviations are used in this paper:

ACSL	ANSI/ISO C Specification Language
BDD	Behavior Driven Development
LTL	Linear Time Logic
MBC	Model Based Checking
MDD	Model Driven Development
MBT	Model Based Testing
SPIN	Simple Promela Interpreter
Promela	Protocol meta-language
TDD	Test Driven Development

Bibliography

1. 4W1H & 5W1H with examples : 2022. URL: https://readandgain.com/2022/07/05/4w1h-5w1h-with-examples/.
2. ACSL: ANSI/ISO C Specification / Baudin P., Filliâtre J.-C., Marché C., Monate B., Moy Y., and Prevosto V. 2015. URL: https://frama-c.com/download/acsl.pdf.
3. ACSL by example, towards a verified C standard library / Burghardt J., Gerlach J., Gu L., Hartig K., Pohl H., Soto J., and Völlinger K. // DEVICESOFT project publication. Fraunhofer FIRST Institute (December 2011). 2016. URL: https://www.cs.umd.edu/class/spring2016/cmsc838G/frama-c/ACSL-by-Example-12.1.0.pdf.
4. Apple. UI Testing in Xcode : 2015. URL: https://developer.apple.com/videos/play/wwdc2015/406/.
5. Beck K. Test-driven development: by example. Addison-Wesley Professional, 2003.
6. Beck K., Gamma E. Test infected: Programmers love writing tests // Java Report. 1998. Vol. 3, no. 7. P. 37–50.
7. Beller M., Gousios G., Zaidman A. Oops, my tests broke the build: An explorative analysis of travis CI with github // 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR) / IEEE. 2017. P. 356–367.
8. Bjørner D., Havelund K. 40 years of formal methods // International Symposium on Formal Methods / Springer. 2014. P. 42–61.
9. Board I. IEEE standard for Software unit Testing. ANSI/IEEE Std 1008-1987 // IEEE Computer Society, New York, YK1987. 1987.
10. Börger E., Stärk R. F. Abstract state machines: a method for high-level system design and analy-

sis. Springer, 2007.

11. Clarke E. M., Emerson E. A., Sistla A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications // ACM Transactions on Programming Languages and Systems (TOPLAS). 1986. Vol. 8, no. 2. P. 244–263.
12. Clarke E. M., Wing J. M. Formal methods: State of the art and future directions // ACM Computing Surveys (CSUR). 1996. Vol. 28, no. 4. P. 626–643.
13. Code Quality and Code Security : 2022. URL: https://www.sonarqube.org.
14. Doxygen – Generate documentation from source code. 2022. URL: https://www.doxygen.nl.
15. Dwyer M. B., Avrunin G. S., Corbett J. C. Patterns in property specifications for finite-state verification // Proceedings of the 21st international conference on Software engineering. 1999. P. 411–420.
16. Eiffel: analysis, design and programming language / Bezault E., Howard M., Kogtenkov A., Meyer B., and Stapf E. // ECMA International, Tech. Rep. ECMA-367. 2006. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-367/.
17. Elements of Reusable Object-Oriented Software / Gamma E., Helm R., Johnson R., and Vlissides J. // Design Patterns. Massachusetts: Addison-Wesley Publishing Company. 1995.
18. Fähndrich M. Static verification for code contracts // International Static Analysis Symposium / Springer. 2010. P. 2–5.
19. Ferreira J. F., Mendes A., Menghi C. Formal Methods Teaching. LNCS 13122. 2021.
20. Firesmith D. Using V Models for Testing : 2013. URL: https://insights.sei.cmu.edu/sei_blog/2013/11/using-v-models-for-testing.html.
21. Formal Methods Teaching Workshop : 2022. URL: https://fmtea.github.io.
22. Frama-C: A software analysis perspective / Kirchner F., Kosmatov N., Prevosto V., Signoles J., and Yakobowski B. // Formal Aspects of Computing. 2015. Vol. 27, no. 3. P. 573–609.
23. Garey M. R., Johnson D. S. Computers and intractability. 1979.
24. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. 1969. Vol. 12, no. 10. P. 576–580.
25. Hoare C. A. R. Communicating sequential processes. 1985.
26. Holzmann G. J. Software model checking with SPIN // Advances in Computers. 2005. Vol. 65. P. 77–108.
27. Holzmann G. J., Joshi R., Groce A. Swarm verification techniques // IEEE Transactions on Software Engineering. 2010. Vol. 37, no. 6. P. 845–857.

28. IBM. IBM Rational Functional Tester : 2022. URL: https://www.ibm.com/products/rational-functional-tester.
29. IEEE/ISO/IEC International Standard for Software and systems engineering–Software testing–Part 3:Test documentation - Redline // ISO/IEC/IEEE 29119-3:2021(E) - Redline. 2021. P. 1–274.
30. International Software Testing Qualifications Board : 2022. URL: https://www.istqb.org.
31. International Symposium on Formal Methods, https://link.springer.com/conference/fm : 2021.
32. International Symposium on Model Checking Software : 2022. URL: https://link.springer.com/conference/spin.
33. Issues - Chromium : 2022. URL: https://bugs.chromium.org/p/chromium/issues/list.
34. Jetbrains. IDEA. Testing : 2021. URL: https://www.jetbrains.com/help/idea/testing.html.
35. Kissell J. Take Control of Automating Your Mac. Alt concepts, 2022.
36. Marjamaki D. Cppcheck – Online Demo : 2022. URL: http://cppcheck.net/demo/.
37. Martin R. C. SRP: The Single Responsibility Principle // Agile Software Development: Principles, Patterns, and Practices. 2003.
38. Microsoft. Cord Syntax Definition : 2013. URL: https://msdn.microsoft.com/en-us/library/ee691953.aspx.
39. Microsoft. Spec Explorer 2010 Visual Studio Power Tool : 2013. URL: https://marketplace.visualstudio.com/items?itemName=SpecExplorerTeam.SpecExplorer2010 VisualStudioPowerTool-5089.
40. Microsoft. Source code for the CodeContracts tools for .NET : 2015. URL: https://github.com/Microsoft/CodeContracts.
41. Model checking and the state explosion problem / Clarke E. M., Klieber W., Nováček M., and Zuliani P. // LASER Summer School on Software Engineering / Springer. 2011. P. 1–30.
42. Morandi B., Bauer S. S., Meyer B. SCOOP–A contract-based concurrent object-oriented programming model // Advanced Lectures on Software Engineering. Springer, 2007. P. 41–90.
43. NASA Formal Methods Symposium, https://link.springer.com/conference/fm : 2022.
44. Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation // ACM Sigplan notices. 2007. Vol. 42, no. 6. P. 89–100.
45. Online test on testing and verification : 2017. URL: https://onlinetestpad.com/t/testingverification.
46. Online testing with model programs / Veanes M., Campbell C., Schulte W., and Tillmann N. //

Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. 2005. P. 273–282.

47. Pnueli A. The temporal logic of programs // 18th Annual Symposium on Foundations of Computer Science (SFCS 1977) / IEEE. 1977. P. 46–57.
48. Promela grammar. URL: http://spinroot.com/spin/Man/grammar.html.
49. Promela samples – cafe : 2020. URL: https://github.com/SergeyStaroletov/PromelaSamples/blob/master/cafe.pml.
50. Promela samples – Hanoi Puzzle : 2020. URL: https://github.com/SergeyStaroletov/Promela Samples/blob/ master/HanoiPuzzle.pml.
51. Promsky A. V. C program verification: verification condition explanation and standard library // Automatic Control and Computer Sciences. 2012. Vol. 46, no. 7. P. 394–401.
52. PVS-Studio : 2022. URL: https://pvs-studio.com/en/.
53. Robinson H. Graph theory techniques in model-based testing // International Conference on Testing Computer Software. 1999. URL: http://www.harryrobinson.net/GraphTheoryInMBT.pdf.
54. Rose S., Wynne M., Hellesoy A. The Cucumber for Java book: Behaviour-driven development for testers and developers // The Cucumber for Java Book. 2015. P. 1–338.
55. Selenium automates browsers : 2022. URL: https://www.selenium.dev.
56. Software Engineering. Federal Standard [in Russian] : 2017. URL: https://fgos.ru/fgos/fgos-09-03-04-programmnaya-inzheneriya-920/.
57. Staroletov S. Basics of Software Testing and Verification [in Russian]. Lanbook, Saint Petersburg, 2020. P. 344. – EDN SGQVLL. URL: https://e.lanbook.com/book/138181.
58. Staroletov S. Automatic proving of stability of the cyber-physical systems in the sense of Lyapunov with KeYmaera // 2021 28th Conference of Open Innovations Association (FRUCT) / IEEE. 2021. P. 431–438.
59. Staroletov S. Modeling the Anti-Lock Braking System in Scilab and Its Checking for Compliance with Uniform Requirements // International Conference on Industrial Engineering / Springer. 2021. P. 413–424.
60. Staroletov S., Dubko A. A Method to Verify Parallel and Distributed Software in C# by Doing Roslyn AST Transformation to a Promela Model // System Informatics. 2019. Vol. 15. P. 13–44. URL: https://system-informatics.ru/files/article/staroletovdubko.pdf.
61. Staroletov S. M. A formal model of a partitioned real-time operating system in Promela // Proceedings of the Institute for System Programming of the RAS. 2020. Vol. 32, no. 6. P. 49–66.