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.

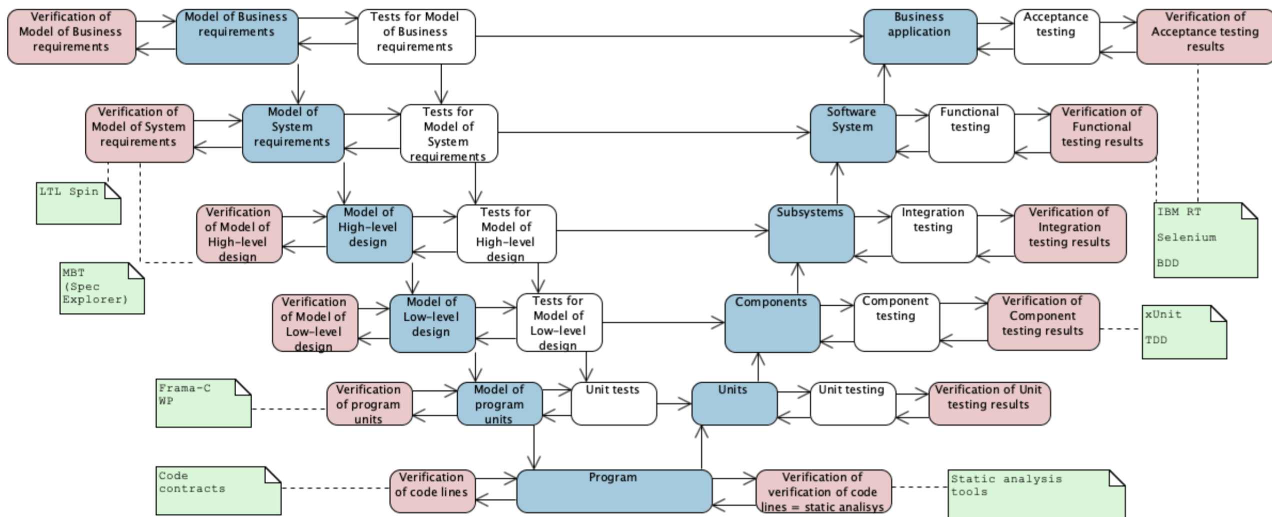

Fig. 1. Triple-V model and tools

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.

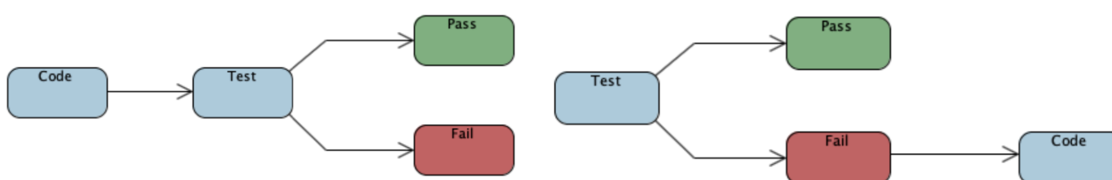

Fig. 2. On the left is the unit testing process (the code encourages us to write a test), on the right is the TDD development process (the test encourages us to write new code)

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).

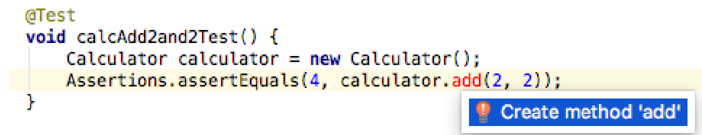

Fig. 3. Creating a method during a test writing

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

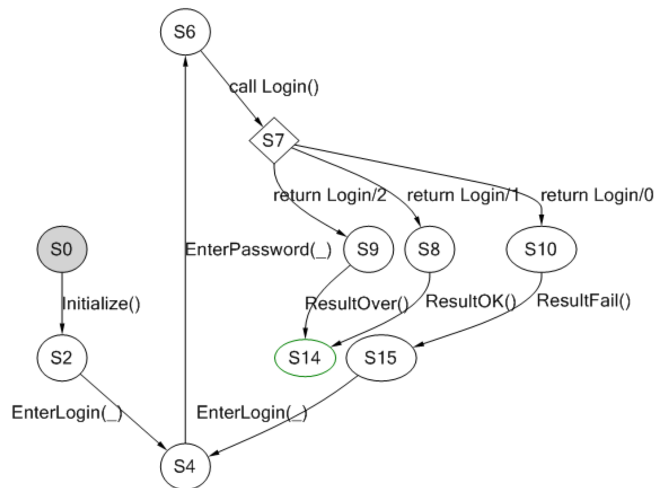

Fig. 4. A login model

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.

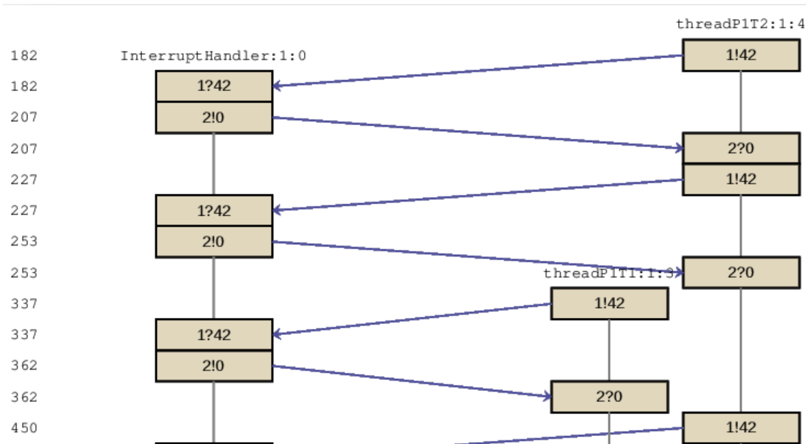

Fig. 5. Simulation of an OS model in iSpin

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.

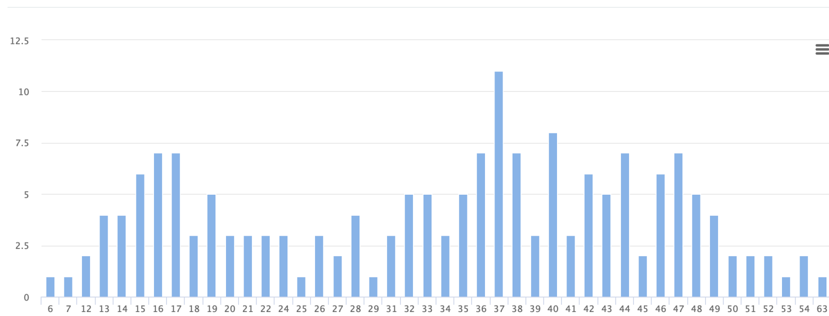

Fig. 6. Distribution of correct answers in the final test

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.SpecExplorer2010VisualStudioPowerTool-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.

УДК 004, 929

Enn Tyugu: a Deported Estonian and a Soviet Academician

Irina Krayneva (A.P. Ershov Institute of Informatics Systems, SB RAS),

Merik Meriste (Tallinn University of Technology)

Killu Sanborn (Oxford Finance LLC)

This work is dedicated to an Estonian scientist in Computer Science, Enn Tyugu (1935–2020). The two landmark events of his biography are his deportation in 1941 and his interest in computers. The topic appears relevant since in the post-Soviet (the same as in the USSR) environment research on the life paths of the representatives of deported nations was scarce; we know little about their life and the life of their progeny; there are no studies or ego-documents shedding light on the everyday aspect of their lives in deportation. We will not elaborate on the issue of access of science and technology specialists (technocrats) to political power and administrative decision-making and will limit our interest to their socio-professional identities.

Keywords: *history of informatics, Enn Tyugu, deportation, programming, STEM, PRIZ, Start*

1. Introduction

Though the focus of our study is Enn Tyugu, we are also going to dwell on the stories of two of his colleagues in Computer Science, the children of Baltic deportees born in the 1950s, who acquired their key competencies in the USSR/Russia. They are Irina Virbitskaite (born 1956) [22] and Algirdas Pakstas (born 1958) [3]. Their stories are the stories of success. Moreover, in his memories, Tyugu mentions several people of the same age as his elder brother Ants (1921–1996) [42]. Hopefully, that the story of the person who, as many his compatriots [14], managed to overcome the absence of freedom and deprivation when forced to accept the rules of other

people's game, will be informative and educational. It should shed more light on the system of the late-Soviet society, eloquently presented by A. Yurchak [49].

Several items shaping the context of our study are as follows. The first is the degree of liberty of a Soviet person and scientist, striving for self-improvement. Isaiah Berlin, discussing the correlation of "positive" and "negative" freedoms[1], came to the conclusion that "… pluralism, with its demand for a certain amount of "negative" freedom, is a truer and more humane idea than the aspirations of those who try to find the ideal of the "positive" self-realization of classes, peoples and entire humanity in large authoritarian and strictly disciplined societies. It is truer at least because it acknowledges the diversity of human goals, many of which are incompatible with each other and are in the state of eternal competition" [5].

The approach followed by A. Yurchak, whose heroes – Soviet people – neither dissidents nor supporters of the Soviet regime, were the so-called "normal"[2] people, demonstrates that they managed to find the ways to exist and coexist that were different from the aforesaid types of freedoms, namely in communities and among the "fellow public".[3] By studying the mentality of a small group of scientists, we will discover ways of existence discovered by Yurchak as well as some phenomena similar to a combination of Berlin's "positive" and "negative" freedoms, which also takes the mentality of scientists beyond the binary description (i.e. for or against the Soviet system).

In contrast to Yurchak, we reveal the stratum of "fellow people" basing on socio-professional principles, i.e. independent of biographical facts, such as being a deportee or a dissident/advocate.[4] Underlying this stratum were common professional interests and strive to become proficient in the new and then exotic field - computers. Moreover, being part of a broader stratum of scientific workers whose outstanding role in the industrial society is noted in the works of sociologists, futurologists and philosophers highlights Enn's belonging to a certain community

---

[1] Negative freedom (characteristic for liberal societies, according to Berlin) is the "freedom from", i.e. freedom from external intrusion, especially coming from the state bureaucracy. Violence is the main threat for freedom. This means that understanding "politics" is reduced to attempts of establishing a "peaceful order", and settle conflicts between individuals, groups, and institutions. Positive freedom is the "freedom to", i.e. freedom of self-development and self-expression. This freedom requires power (state or other) guaranteeing the process of personality formation and setting its parameters. Berlin considers this type of freedom to be typical for socialist societies.

[2] By "normal", A. Yurchak means people of the late-Soviet period, whose life was relatively free from state control and ideology, and was not necessarily perceived as juxtaposition to socialism or the state. See A. Yurchak, *Everything was forever* …, [49, p.193].

[3] Communities and the "fellow public" – in Yurchak's works, contexts of dominant ideology and authoritative discourse identified the "fellow people" not based on the same social origin or belonging to the same class, but by the perception of authoritative discourse (i.e. ideological slogans). [49, p. 249].

[4] The children of Baltic deportees were not subjected to repressions in their study, work and career. This is different from what happened to those who ended up on occupied territories during the Great Patriotic War. See refer. 24, p. 13–15.

[47]. It stemmed from the technological determinism typical of the Soviet society, technocracy, and absolutization of scientism. Anthropologically, these ideas formed the image of the "technocratic man", armed with knowledge and capable of modifying nature and society on the basis of scientific and rational approaches [34]. These phenomena formed in the wake of modernization, which was accompanied by excessive expectations of the society addressed to science and technology. We will not elaborate on the issue of access of science and technology specialists (technocrats) to political power and administrative decision-making, and will restrict ourselves to their socio-professional identities; however, we must remark that this stratum failed to avoid the so-called "public addresses" (slogans) urging the workers of science to become active participants in building the communist society – this goal was included in the Program of the Soviet Communist Party of 1961. The community of those involved in software development formed during the Cold War and atomic projects. Though the development of computers in the USSR was not directly connected to the national atomic project, it shortly became a key customer [23]. Computer development in the USSR in the fifties was yet another major Soviet project, on a par with the atomic project, albeit on a lesser scale. Understaffing, as pointed out many times by academician Andrei Ershov (1931–1988), the informal leader of Soviet programmers [6], indicated the limitations of this project, and Ershov did his best to handle this problem. Among other things, the Department of Programming, Computing Center, SB AS USSR, in Novosibirsk, which he led from 1957 to 1988, trained programmers of the highest qualification as the national departments of the Soviet Academy of Sciences did not have Dissertation Councils specializing in system programming or software engineering. Few programmers chose to defend dissertations: their priority was keeping pace with the development of program systems. Nevertheless, Ershov encouraged defending dissertation theses in programming and hand-picked the most talented programmers. His extensive ties in the academic world and the reputation of the Novosibirsk programming school helped him to solve this task.

This paper is based on the memories of E. Tyugu published in Estonian [42], memories of other programmers, as well as materials from the Academician A. P. Ershov Electronic Archive [2]. The memories of Enn Tyugu can be divided into several periods: first, from his childhood to deportation in June, 1941; second, being in Bashkiria as a deportee until April 1946, when he managed to return to Estonia thanks to the efforts of his father's sister, aunt Amanda. His memories of this period provide rare evidence of the life of the representatives of the Baltic peoples deported to Russia. The third period is his life in Haabersti, a suburb of Tallinn, which lasted until 1959, when he began his two-year training course in computing sciences in the Leningrad Polytechnic Institute. The longest period of his life in the Soviet Union (1959-1991) is

closely connected with computers. The final, post-Soviet period, began on January 1, 1992, when Tyugu went to Sweden as a professor of software engineering to work in the Royal Swedish Technological Institute (Kungliga Tekniska Högskolan, KTH), and lasted until 2000. Afterwards, he returned to Estonia, to the Tallinn Technological University, and combined his tenure with working in the Center of Collective Cyber-defense Competency until his retirement in 2016. While working abroad, Tyugu maintained a close connection with his Estonian colleagues through joint European scientific research projects.

2. How it begun

Enn Tyugu was born in a family of telegraph workers. His father, Harald Tyugu (1891–1942), came from a family of farmers. He moved to the city of Paide, where he studied telegraphing. Here, he met his future wife, Elfride Nael (1899–1945). Her mother was the manager of a ham and sausage factory in Põltsamaa. In Paide, Elfride also learned telegraphing (she was ambidextrous and could operate the Morse key equally well with both hands). This was probably the very beginning of the establishment of the national scientific and technical intelligentsia in Estonia.

When she was 16, Elfride left home with her sweetheart, who had been ordered to work in Tambov, a town southwest from Moscow. While they were in Tambov, the Revolution happened and the Civil War began, which made the return to Estonia problematic. Telegraph operators were in high demand by the Reds, the Whites and the Greens alike. There is a family legend that Harald tried to stop the use of telegraph poles for firewood by one of the war parties; as a result, he was arrested and sentenced to execution. His wife saved him by bribing the guards with two loaves of bread. When they returned to Estonia in 1920, they saw that the hair of Enn's father, aged 29, had gone completely grey. Soon, their first son, Ants, fourteen years older than Enn, was born. Back in his home country, Harald completed a correspondence course and got a degree in Law at Tartu University. He became a lawyer in the Main Post Office; he made laws and resisted the attempts of illegal connection to the radio.

Enn’s memories of his early childhood were joy-filled and calm; they had plenty of food; his nanny, and elder brother took good care of him. The family spent winters on the Kunder Street in Tallinn, and summers in their summerhouse, then under construction, in the suburbs of Haabersti. The area is now part of the city. The wooden house in Tallinn, with a shared hall and stone staircase, was a product of the pre-war construction boom. Enn recollected: “Our apartment was small by modern standards, but back then, my mother and father said, somewhat proudly, that we had a “two-room apartment with commodities”. This meant that we had tap water and our own

toilet; the house was heated by a wood stove. There was a shared bathroom and laundry room in the basement" [42, p.21]. Both the city apartment and summer house had telephones installed (sic!). The first stage of Enn's life was coming to its end.

Estonia was annexed to the Russian Empire in 1721, according to the Treaty of Nystad after the Great Northern War as part of two provinces – Estland and Livonia. In February 1918, Estonia became an independent parliamentary republic.[5] It remained one until the Soviet occupation of 1940, according to a secret supplementary protocol of the Molotov-Ribbentrop Pact of 1939 [41]. On September 28, 1939, the USSR forced Estonia to sign a mutual assistance treaty, which allowed the USSR to place its army, navy and air force bases and troops on the Estonian territory; the contingent of the troops was later increased. From June through August 1940, the state executive organs, police, army, financial and economic systems of the Estonian Republic were dismissed; education institutions were reorganized according to the Soviet model, and all public organizations were dissolved. On June 14 and 15, 1940, the"elections" of new parliaments, according to the directions of the USSR representatives, were held simultaneously in the three Baltic republics. At the simultaneous sittings of these "parliaments," the three Baltic nations were declared Soviet Socialist Republics. They petitioned for acceptance into the USSR. The land became state-owned, banks and industrial enterprises were nationalized [15]. Tyugu recalled:

> "That winter (1940) there was trouble in the air. My parents were fluent in German and Russian. Their speaking foreign languages, something that had not happened before, irritated me. Now I realize that they did not want to discuss the unpleasant events occurring in Estonia and other places with me around» [42, p.27].

His favorite children's magazine, *Play and Enjoy*, was renamed into *Work and Enjoy*, and began to publish stories about pioneers and *kolkhozes*. Harald Tyugu was offered a job – head of a communications department in Western Estonia, but he declined.

By the summer of 1941, Estonia was completely sovietized. The process was accompanied by arrests, executions and deportations of citizens: like in the other two Baltic republics, it was the elite that got prosecuted: local and national-level politicians, prominent figures in economics and finance, military officers, active members of the Kaitseliit (Estonian Self-defense Union), wealthy peasants, professionals, etc.[6] Then came June 14, 1941. Over 10,000 people, whole families, were

---

[5] On February 2, 1920, Russia acknowledged Estonia's independence. On September 22, 1921, Estonia became a member of the League of Nations.
[6] The same is true about Soviet Russia/USSR in pre-war years [33].

deported from Estonia. About 3,000 men and 150 women wore separated from the rest and placed into camps, where most of them were executed or perished. The remaining women and children were sent to special settlements in the Urals and in Siberia. According to the White Book, over 53,000 people were repressed in Estonia in June, 1941 [19, p.14–15].

3. Deportation

When a truck with soldiers and their commanding officer drove into their yard, Enn was alone with his mother. Elfride called her husband at work and her older son in his city apartment. They decided to stay together; some other families decided to disperse or hide. Enn and his mother were brought to the city, where the family reunited for a short period. Later, they were separated: men, youngsters and elderly people were placed into different trucks:

> "My parents had a good idea of what life was like in Russia based on their experience, we took as many clothes as we could, and some food for the journey... My mother was happy because she found a place on the upper shelf, next to a barred window, where the air was fresh, and I could stay on the shelf and look outside through the bars" [42, p. 29].

Cars with men were separated midway. Enn never got to see his father again. Harald Tyugu died on March, 17, 1942, in the Sosva division of the North-Urals Camp (Sevurallag) in the Sverdlovsk Oblast: the high mortality of prisoners in the Sevurallag, especially in 1941–1942, was the result of harsh working (logging) and living (unsanitary) conditions, as well as of poor nutrition [37, p. 41].

The second stage of their travails began, but the little boy was oblivious to the tragedy. He enjoyed traveling by train, on a steamboat, and on a cart pulled by horses. His new friend, the 14-year old Karl Tiidus, carved wooden chess figures for him. Enn remembered the unfamiliar taste of the food that they were given during a stop in a school building. At last, they reached their deportation destination – the town of Urzhum in the Kirov Oblast, Bashkiria. His older brother Ants and other young men were taken to a logging camp on the river Vyatka, about 500 miles away from Urzhum. Enn and his mother settled in a shoemaker's house. The population of the town was about 10,000; there were some stone houses and churches used as warehouses; roads, dusty in the summer, dirty in fall and spring, and snowy in winter. *Lapty*, starvation, lice, a vodka factory that supplied its product both to the nearby and remote villages.

Enn's mother was lucky to get a job at a ski factory: she knew Russian. Enn had to spend plenty of time alone. Elfride worked long hours, and every evening brought something to eat in a half-liter mason jar, her worker's lunch. Soon, Enn went to kindergarten. By learning poems by heart for the New Year show, he got his first Russian lessons. This was important, because he did not speak or understand Russian and so could not communicate. Food was becoming more and more difficult to procure. Elfride sold most of the things she had brought to the town and decided to move to a village. They stayed in the house whose owner had been arrested for taking wood planks from the *kolkhoz* yard to repair his cart. Their landlord and her daughter Galka, who was four years older than Enn, got on well. Enn quickly adapted to the new environment and made friends with local kids. They went to the forest to pick berries and to the little river Urzhumka, where he learned to swim; he herded the village cows. While the parents worked in the field, the children were on their own. They ate potatoes baked in embers with salt and garlic, young sprouts of horsetail, onions, and sorrel, and stole peas from the *kolkhoz* field: "We knew all the edible plants in the forest. I still know which herbs are edible and which are poisonous" [42, p. 44].

Elfride asserted her status among the villagers by helping illiterate women read and answer letters from the front. Sometimes, they went to see Ants. He was working as a lumberjack with other Estonian young men, and spent 16 years in Russia. The story of these men testifies to their perseverance and resourcefulness and to the lack of qualified male workers during the wartime. Ants was a blacksmith, an electrician, and a radio operator; le learned to drive a truck. He got a degree from the Sverdlovsk Polytechnic Institute (correspondence course) and worked as an automation engineer in Estonia. Ants' peers from the Jacob Westholm boys' school in Tallinn were with him in the logging camp. Later, they became known in various fields: Uno Kopvillem, a physics professor [7, p.36]; Juhan Tuldava (Haman),[7] a linguist, Doctor of Philological Sciences and professor of the Tartu State University; Juhan Zimmermann, a figure skating coach and a civil court judge [17] – almost all of them were educated remotely in the logging camp. On September 1, 1942, Enn went to school. Russian was a difficult subject for him, while mathematics was a success. Enn borrowed books from the school library. Despite the hardships, his mother did her best to educate her son: she invited an elderly Estonian lady to teach him German. Her strategy was understandable: she valued education and chose the language she herself knew well.

[7] Juhan Tuldava (1922–2003, pseudonym Arthur Johan Haman) was an Estonian linguist and Soviet spy. Graduated from the Kirov Pedagogical Institute in 1948 with a degree in English language and literature. Started cooperation with the USSR National Security. The MGB and KGB gave him the agent names Voronin and Skvortsov. He published books in the Estonian language and memoirs under the name Arthur Haman "*Sõbrad ja vaenlased : mälestuskilde*" (Tallinn : "Kodumaa" väljaanne, 1967).

Enn was adapting well: he completed his third school year with distinction and was awarded an honorary certificate. Life was getting back on track. In the spring, they planted the garden, like all other villagers: potatoes, rutabaga and onions, valuable sources of food in winter. The fall harvest was excellent, and the mother was optimistic about the upcoming winter, as she wrote in her letter to Amanda in October 1944. This was when she learned that the Red Army had occupied Tallinn once again. In winter, Elfride's main chore was providing firewood. Then, the irreversible happened: she contracted a strep infection. Despite all efforts of the doctor, E. Tuldave, to save her by blood transfusions, she died of sepsis in Urzhum on April 5, 1945, aged 45 [42, p.45].

Enn was left under the care of his older brother, who was away most of the time. He continued going to school in Urzhum. His Russian teacher fed him, and he helped her correct homework. By that time, Enn had become good at Russian grammar; moreover, he read a lot: *Tom Sawyer*, *Treasure Island*… The pain of his loss was gradually going away… Soon, Enn did a very manly thing: he persuaded an employee of the city council to sign up him and his brother for a labor camp, like other Estonians; soon, they had moved to a new place. Here, lots of Estonians worked in the fields to supply the army. Ants got a job at the radio station, and Enn went to the neighboring villages to buy potatoes, bravely covering long distances. The Estonians established correspondence with their homeland and even got parcels with Christmas gifts. The head of the labor camp, colonel Schwartz, tried to arrange decent living conditions for his subordinates. A sort of a foster home was organized for the children who had lost their parents: it was warm, and there was food.

4. Coming home

In the spring of 1946, Enn returned home: "What happened was that a quiet man named Stepan Shubin came to us, with compliments from aunt Amanda. He met colonel Schwarz and they agreed that the latter would take me to his regiment in Estonia upon his return from vacation. Stepan had a document stamped by the NKVD, stating that Enn Tyugu, born on May 20, 1935, was granted a permit to reside in the Estonian Soviet Socialist Republic" [42, p. 49]. Two girls who had also lost their parents, Airi Airing and Maret Looderaud, joined them. Enn and his aunt settled in the summer house built by his father. Every winter, the house had to be insulated. The parents' house in Tallinn had burned down in a bombing raid in 1944 during the advance of the Red Army.

Those were difficult times. In 1941, an anti-Soviet guerilla movement, the so-called "Forest Brothers" ("metsavennad") emerged in the Baltic States. In Estonia, over 40,000 people joined.

The movement remained active until the mid-1950's. At the same time, the national economy of the ESSR was being restored according to the Action Plan of 1944. The plan for the 4th *Pyatiletka* (five-year industrial plan), 1946–1950, provided an investment of 3.5 billion rubles into the economy of Estonia, which was 1.7–2.9 times greater than the amounts allocated to other republics, including Latvia and Lithuania with their bigger populations. Per capita investment into the Estonian economy exceeded the average for the USSR by around 30% in 1940-1950 and by 17% in 1951-1955. In the 1960s through 1990s, despite the general standoff with the West, Estonia had the most intensive economic connections with other countries, especially in the area of Computer Science [18]. It had good scientific connections with Denmark and Sweden, and good educational and trade connections with Finland. The Soviet-Finnish economic cooperation in the area of IT went through a joint Soviet-Finnish enterprise called Elorg Data, founded in 1974. 58% of the share capital belonged to Elektronorgtekhnika, an enterprise of the USSR Ministry of Foreign Trade [36]. Estonian specialists in Elorg Data were able to learn foreign programming technologies. Thanks to the Finns, by the late 1980s, some elements of modern computer architecture, such as e-mail, appeared in the AS ESSR institute of Cybernetics. Scientific exchange benefited the process of training specialists in both countries: when in the late 1980s and early 1990s Finnish universities lacked teaching staff, they invited specialists from Estonia [18, p. 115].

Computing sciences got a boost in the Soviet Baltic in the late 1950s and early 1960s within the development of cybernetics; in a broader context, it was part of the scientific and technological modernization of the Soviet economy and rehabilitation of cybernetics. Cybernetics was perceived in the Soviet Union as the engine driving the building of the Communist society, as declared by the Communist Party Program in 1961 [20, p. 312]. This is yet another telling evidence of the technological determinism of the Soviet society of the time, facilitated by many eminent scientists [4]. A number of institutes of cybernetics were formed within the Soviet Academy of Sciences, including the institute in Tallinn in 1960; in 1976, a Special Design Bureau of Computing Machines was formed within the Institute.[8] Such institutions emerging in the system of the Academy of Sciences and higher education of the Baltic republics became basic

[8] After the fall of the USSR, the AS ESSR Institute of Cybernetics became a semi-autonomous research institute of the Tallinn Technical University, and was closed after a structural reform. From January 1, 2017, specialists in phonetics and speech technologies and the laboratories of control systems and software were reassigned to the Department of Scientific Software at the New School of Information Technologies; specialists in wave technology, nonlinear dynamics, photoelasticity and from the Laboratory of Systems Biology were reassigned to the Department of Cybernetics of the New School of Science.

organizations for many computational sciences, programming, network technologies, and related special education.

The emergence of these institutions in the Baltics had internal economic reasons. The establishment of the Institute of Cybernetics in Estonia was dictated by explosive development of the chemical and energy industries (based on slate mining), where automation and controlling tools played a key role [16, p. 63]. Also, reforms aimed at the decentralization of the Soviet economy in 1957–1965 stimulated the development of the Baltic republics. New scientific and technical areas required new skills. A number of specialists in computer science for the Baltic republics were trained in the 1960s in the Leningrad Polytechnic Institute (LPI) and Moscow Energy Institue (MEI). Initially, there were about 25 students in Moscow and Leningrad. Specialists of the highest qualification were trained in Novosibirsk, Kyiv, and Minsk [12].

As mentioned above, Enn Tyugu had excelled in math in elementary school, which may have affected his choice of the Tallinn Polytechnic Institute as a higher education facility. Upon his graduation in 1958, he became a designer-engineer at the Tallinn Excavator Factory. According to his memoirs, it was then that he had his first experience with computers, which defined his future life and scientific career [43]. By the end of 1959, what Tyugu had believed to be impossible happened: he was accepted to the Leningrad Polytechnic Institute for a two-year course in computing, without leaving work. Ever after, he was grateful to Professor Aleksander Voldek,[9] who helped him to get to the Institute where he received a “fantastic” education in computer science. During this period, Tyugu’s work was connected with the Scientific Research and Technological Design Institute (1959–1976), where he grew from a staff researcher to the department head [10].

5. Science and a bit of politics

In the first half of the 1960s, the STEM microcomputer (Specialized Technological Electronic Machine) was designed and built in the Scientific Research and Technological Design Institute. Extremely reliable for the time, it was used in the technology department of the Kirov Factory in Leningrad (not in the computing center, though, as it required round-the-clock maintenance) [43, p. 13]. Similar computers were built for other major factories in the USSR. In 1967, the design received the State Award of the Estonian SSR. In 1966, Tyugu defended his thesis for the degree of the Candidate of Technical Sciences (supervised by Georgiy Konstantinovich Goranskiy, who

[9] Aleksandr Ivanovich Voldek (1911–1977) – electrical engineer, Doctor of Technical Sciences (1957), Academician of the Estonian AS (1969), from 1950 through 1961, worked at the Tallinn Polytechnic Institute (currently Tallinn Technical University), then at the Leningrad Polytechnic Institute (now Peter the Great’s St. Petersburg Polytechnic University).

was the director of the Institute of Technical Cybernetics of the Belorussian SSR Academy of Sciences in Minsk in 1965–1970).

Eventually, Tyugu received a recommendation to enroll for the Doctorate studies of the SB AS USSR Computing Center in Novosibirsk Akademgorodok, to the Programming department led by Andrei Petrovich Ershov [28]. Tyugu had read the work on parallel programming written by V. Kotov and A. Narinyani in the early 1960s [21], and came to Novosibirsk Akademgorodok to get to know them closer and to speak at a seminar. In 1970–1971, he became a researcher of the Computing Center, and Ershov became his scientific consultant in his work on his Doctorate thesis. Ershov spoke highly of Enn's progress.

Tyugu came to Akademgorodok when the Khruschov Thaw (*Ottepel'*), ambiguous as it was, ended with the suppression of the Prague Spring in 1968 and arrest of the protesters against the military operation in Czechoslovakia on the Red Square. In the context of the Czech events, Akademgorodok became one of the centers of the opposition movement. It was caused by the Process of the four trial in January 1968, when several "dissidents" were prosecuted, including the journalist A. I. Ginzburg, poet Yu. T. Galanskov, activist A. A. Dobrovolskiy, and typist V. I. Lashkova. 46 researchers of the SB AS USSR and the NSU signed a letter protesting the lack of *glasnost* during the trial. On March 23, the letter was published in the New York Times, and on March 27, it was broadcast by the Voice of America [25, p. 7]. Valeriy Menschikov, a member of Andrei Ershov's team, was among those who signed the letter. The reaction of various academic leaders to the actions of those who signed the letter was not unanimous: Menschikov got out of the turmoil virtually unscathed – Ershov vouched for him and accepted him as his postgraduate student.

On March 8-9, 1968, a bard festival was held in Akademgorodok. Alexander Galich, who was invited to the festival, performed his songs "Goldminer's Little Waltz" (*Staratel'skii valsok*)[10] and "In the memory of Pasternak". As a result, the Under the Integral club (*Pod integralom*),[11] which organized the festival, was closed. Galich's socio-political satire was condemned by the Communist Party officials as "food for our ideological enemies". Vladimir Davydov, a friend of Tyugu's and an active participant of the club's meetings, compiled a remarkable photographic gallery showing the life of the club members [38].

[10] This ballad is a bitter reproach to the indifference of the Soviet society towards the repression of political freedoms and nonconformists. The majority basically approved the persecution of dissidents with their silence… "Be silent and you will become a rich man… be silent and you will become a hangman".

[11] Café-club *Pod integralom* was a discussion club in Novosibirsk Akademgorodok hosting informal talks and meetings of scientists of different generations. One of the symbols of the Khrushchev's Thaw; closed in 1968.

There were some problems of scientific nature, too. The team of Ershov's Programming Department was working on the BETA Multilanguage translating system.[12] The bottleneck was developing an internal language for the system, which upset the timeframe of the project. Another obstacle was conflicts between the people responsible for the task. Tyugu observed it all at BETA seminars [24, p. 44–51]. However, his own work progressed successfully, despite some problems with the living conditions: it was difficult for doctorate students from other cities to get decent housing in Akademgorodok. Tyugu made friends with his colleagues and discovered a passion for hiking and hunting, fostered by Vladimir Davydov's enthusiasm.

During his Doctorate studies in the SB AS USSR Computing Center, Tyugu presented his research at a number of high-ranking national and international conferences. In particular, Tyugu and his co-authors presented the paper called "A system of modular programming for the Minsk-22 computer" [40] at the Second All-Union Programming Conference (VKP-2) in Novosibirsk in 1970. The same material served as the basis for his talk at the IFIP-71 Congress in Ljubljana in Yugoslavia. The Computing Center gave a truly royal gift to Tyugu: he was included in the Soviet delegation to the Congress though the number of people allowed to participate in the Congress as the Institute's employees was strictly limited. The trip, formally qualified as "scientific tourism,"cost 350 rubles, with a 10 rubles participant fee [27]. The head of the delegation, Academician A. Dorodnitsyn, signed a permit for the preparation of the necessary documents.

In 1973, Tyugu defended his Doctorate thesis in the Leningrad Electrotechnical Institute, entitled "Application of computational models in the software for machine-assisted design." His opponents were S. S. Lavrov, B. G. Tamm, and N. G. Bondarev. Tyugu remained engaged with Ershov's department. In 1979, he was invited to participate in a legendary event of the time – scientific pilgrimage to Urgench, the native city of Al-Khorezmi [46]. Eventually, Andrei Ershov, who was the vice-chairman of the Programming committee at the IFIP-1980 Congress and leader of the Software section, put considerable effort into making sure that the USSR would be properly represented at the Congress. Vadim E. Kotov and Enn Tyugu were invited speakers. As a member of the Program Committee, Erhsov helped Tyugu with his talk, both in terms of content and stylistics. He asked R. M. Berstall, a professor of the University of Edinburgh, "to assist in assuring the high quality of style and content of Tyugu's talk [29].

In 1979, when the AS USSR Coordinating Committee on Computing Machines formed the Commission on System Mathematical Support, headed by Andrei Ershov, Tyugu became part of its bureau [30] and head of the Workgroup on Program Synthesis [45]. Moreover, he was a

---

[12] BETA, which was not even an abbreviation, was sometimes caustically explained as Big Ershov's Translator Adventurism (***B****olshaya* ***E****rshovskaya* ***T****ransliatornaya* ***A****vantiura).*

member of the committee on the distribution and use of computing machines in the AS USSR [31]. In 1981, Tyugu was elected a Corresponding Member of Academy of Sciences of the Estonian SSR, and in 1985, he became a full member of the Academy and Secretary Academician of the AS ESSR Department of Informatics and Mechanics (1985–1991).

In 1976–1986, Tyugu headed the Laboratory of Software in the AS ESSR Institute of Cybernetics. The research program of the laboratory was aimed at problems of programming automation with applications to engineering calculations. Tyugu suggested an approach to developing instrumental systems for packaged applications based on automatic program synthesis (which later became known as "semantic computing networks" and "conceptual programming"). The idea was further developed by S. S. Lavrov (Leningrad) [26]. Tyugu's approach was implemented in the PRIZ program [Russian abbreviation for *Program for Solving Engineering Problems*] [35]. Grigoriy Efroimovich Mints (1939–2014), a Soviet dissident mathematician, was part of the team; upon the termination of his tenure at LOMI (A. V. Steklov Leningrad Division of Institute of Mathematics), he was accepted to the AS ESSR Institute of Cybernetics (1980–1991), where he collaborated with Tyugu's laboratory [44]. An important feature of the approach implemented in the PRIZ system was the possibility of integrating various software suites into a single system.

At the end of 1960s, Tyugu suggested organizing winter software schools in Viljandi. Its participants were the representatives of computer factories from Minsk, Zagorsk, Kyiv, and Kazan, as well as the masterminds of programming from the Lebedev Institute of Precision Mechanics and Computer Engineering, Keldysh Institute of Applied Mathematics, V. M. Glushkov Institute of Cybernetics, Dubna, etc. The agenda included the discussions of the participants' own projects as well as the reviews of new foreign software: IBM OS, IBM DOS; by that time, it had been decided that the IBM 360 computer (1967) was to be copied in the USSR. The scope of the topics discussed at the schools expanded to include the automata theory (M. A. Gavrilov, Corresponding Member of the AS USSR) and artificial intelligence (G. Yakobson, Candidate of Technical Sciences,[13] and A. D. Pospelov, Doctor of Technical Sciences). Active participants of the schools was the team led by V. I. Varshavskiy,[14] specializing in mathematics, biology, automata theory, collective behavior, image recognition, information transfer, and creation of asynchronous electronic devices and systems, took active part in the schools.

[13] Gabriel Yakobson is currently in the organizing committee of the CyCon conference devoted to cyber defense and is the honorable chairperson of the annual conferences CogSiMa (Cognitive Situation Management) in Estonia.

[14] Viktor Ilyich Varshavskiy (1933–2005) – cyberneticist, professor, Doctor of Technical Sciences, played a major role in the establishment of cybernetics and artificial intelligence studies in the USSR. From 1993, he worked in Japan and Israel.

In 1985–1988, the team led by Tyugu joined Start, – a Soviet project aimed at creating a 5th generation computer [9]. This was an attempt of the Academy of Sciences to regain Soviet positions in the development of computing machines, mainly by large-scale copying of American computers.[15] Participation in this project provided the Estonian team with good financial support for developing their own ideas. Enn was an active member of the working group that developed the concept of the project. His responsibilities included the choice of high-level tools for the creation of intellectual software for the programming system that would enable the creation of user-friendly user applications. Start teamed up with a large group of researchers and engineers from the Special Design Bureau of the Institute of Cybernetics. Its tasks included the implementation of the professional intellectual object-oriented workstation PIRS as part of a series of high-output modules unified by the Multibus-2 bus and an input-output machine. The modules of the workstation were the KRONOS processor, data filter, high-resolution raster display controller, specialized name processor and object memory control processor [39].

During the *Perestroika*, Tyugu took active part in the social and political life of Estonia. In 1989–1991, he participated in the Congress of People's Deputies of the USSR. Together with other Estonian deputees, he advocated the rescindment and condemnation of the secret protocols of the Molotov-Ribbentrop Pact. The People’s Front of Estonia in support of Perestroika became increasingly insistent on granting independence to Estonia; in the summer of 1988, the Supreme Soviet of Estonia reinstalled the blue-black-and-white national flag. In November, the Declaration of Sovereignty was signed, establishing the prevalence of the laws of the Estonian SSR above the laws of the USSR. On August 20, 1991, the Supreme Soviet of Estonia declared its independence, legally reinstalling the Estonian Republic. This resolution was followed by the restoration of diplomatic relationships and recognition of the Estonian Republic by many of the world’s states [48]. In 1996, Tyugu ran for president, on suggestion of his colleagues from Estonian Business School and Tallinn Polytechnic Institute in addition, but did not receive enough votes. According to an interview, becoming a president was not a priority for him; he did not have a specific program, believing that “the role of the president is to find balance, compromise and moderation” (exactly the role of the Estonian Republic president today) [32]. Later, he admitted that there had been some degree of adventurism in his presidential campaign, assuming that his political ambitions could have interfered with teaching in Sweden. In 2001, he was awarded the White Star Order of the Third Degree [11]. Along with such masterminds of informatics as Ia. Penjam, M.

---

[15] On December 30, 1967, the Soviet Communist Party Central Committee and the Cabinet of Ministers adopted the resolution “On further development and production of computing technology tools”. It installed the strategy of copying the technologies of IBM and DEC as the official technological policy [1].

Meriste, U. Pruuden et al., Enn Tyugu is one of the founders of informatics in Estonia, a country which has become the leader in IT applications in the Baltic region.

5. Conclusion

Estonians are usually characterized as calm, good-natured, substantial and business-like people. This is what Enn Tyugu was, judging by his ego document and memories. As a child, he suffered a serious trauma: lost his parents, home, and motherland. However, even if there is bitterness in his memories, it is well-concealed. The very form of the memories, written in several acts, like a play, as well as the name "Life as a Show" move the losses and bitterness into the background, both structurally and stylistically, and create an illusion of distance between the main character and the events. Details provided by Enn in his memories, as well as the skills acquired in his childhood, indicate that the experience was very deep-rooted. He put what nature had given him to the best possible use, and was persistent in pursuing his goals. Evidence left by Enn about his childhood is unique to the history of the Soviet multinational intelligentsia, even though the multinationality was not always by choice.

A look at the history of the study of the relationship between the Soviet scientific elite and government demonstrates the limitations of its factography and theory specifically in relation to national peculiarities. The approaches and metaphors used for the "Soviet scientific intelligentsia" do not apply here. We know that already in the pre-war period, the state became the only employer of scientific projects, the only source of their material support and the only customer of practical and scientific results. This situation drew them to the Big Deal of the post-war time – the reorientation of the Soviet regime towards satisfying the needs of the Soviet intelligentsia and middle-class bourgeoisie and the implicit return of middle-class values into the Soviet life in return for loyalty [8, p. 3–5]. The ambiguity of the relationship between the state and society allowed J. Hellbeck to formulate the concept of "Soviet subjectivity", i.e. the external loyalty of Soviet citizens as a cover-up of their "personal core" and private life [13, p.17]. This idea is supported by the heterogeneity of the scientific elite, a key support of the modernization of the Soviet society. The relationship of the Soviet scientists and the state was influenced not only by the awareness of the former that they were in demand in but also by the memory of the trauma inflicted by the occupation, deportation, destruction of their statehood, and repressions against their loved ones – the private life component that had been temporarily hidden. As Laozi said, there is no need to avenge evil – just sit by the river, and eventually you will see the corpse of your enemy floating by. The resistance of the "Forest Brothers" was suppressed in mid-1950s. Life in the Baltic States went back to normal, and they became a sort of a showcase region of the

Soviet Union. They flourished, they became richer and more educated. But the memory of the past remained. As soon as there appeared an opportunity to regain independence, the Baltic States used it. Enn Tyugu, whose family had experienced the dread of the steamroller of the Soviet history, became a parliamentary deputy for the period when the urgent problems of the very existence had to be solved, but later returned to science. After all, it was his lifetime calling.

References

1. Abramov R., "Soviet Technocratic Mythologies Myth as the Form of Lost Chance Theory: on the Case of the History of the Cybernetics in the USSR", *Sociology of Science and Technology* (2017), 8, no. 2, 61–73. – In Russian.
2. Academician A. Ershov's Digital Archive. [Online]. Available: http://ershov.iis.nsk.su/ru/folders
3. Algirdas Pakstas, CV [Online]. Available: https://prabook.com/web/algirdas.pakstas/263873
4. Berg A. I., Kitov A. I., and Lyapunov A. A. "O vozmozhnostyakh avtomatizatcii upravleniya narodnym hoziaistvom", *Problemy Kibernetiki (*Moscow.: Fizmatgiz, 1961*)*, 6, 83–100. – In Russian.
5. Berlin Is. Two Concepts of Liberty [Online]. Available: URL: http://kant.narod.ru/berlin.htm
6. Boulionkov M. and oth. eds., *Andrei Petrovich Ershov: uchenyi i chelovek* (Novosibirsk: SB RAS Publ. House, 2006), 24–25. – In Russian.
7. Chudnovsky V.M. "Uno Hermanovich Kopvillem – 90 let", in *Abstracts, Second Sci. Conf. Oceanography of Peter the Great Bay and Adjacent Area of the Japan Sea* (Vladivistok: Dal'nauka, 2013), 36. – In Russian.
8. Danham Vera S. *In Stalin's time: Middleclass values in Soviet Fiction* (Durham, NC: Duke University Press Books, 1990), 3–5.
9. Draft resolution of the Plenary Session of the AS USSR Coordinating Committee on Computers, discussing proposals on creating a national 5th generation computer (Academician A. Ershov's Digital Archive, f. 602, l. 1–3).
10. Enn Tyugu's CV [Online]. Available: https://www.eris.ee/user.cv.preview.php?id=542
11. Enn Tyugu's Obituary [Online]. Available: URL: https://rus.err.ee/1071469/skonchalsja-odin-iz-osnovatelej-informatiki-v-jestonii-akademik-jenn-tyugu
12. Gorodnyaya L.V., Krayneva I.A., and Marchuk A.G., "Computing in the Baltic Countries (1960–1990)", in *Select. Papers SoRuCom-2017. Fourth Intern. Conf. on Computer Technology in Russia and in the Former Soviet Union* (Zelenograd: IEEE PS, 2018), 97–108. [Online] Available: https://ieeexplore.ieee.org/document/8400356
13. Hellbeck J. *Revolution in my Mind: Diaries of the Stalin Era* (Revoliutciya ot pervogo litca. Dnevniki stalinskoi epohi). Moscow: Novoe Literaturnoe Obozrenie, 2017.
14. Helle-Liis Help, Toomas Kodres, Olvi Kuusik, & 14 more. Stolen Childhoods: Stories of Estonian Children Deported to Siberia. Lakeshore Press,135 pp. August 30, 2014.
15. Hiio T., Maripuu M., Paavle I., eds., *Estonia 1940-1945 International Commission Investigation of Crimes against Humanity* (Publisher: Estonian Foundation for the Investigation of Crimes against Humanity, Jan. 1, 2005).
16. Hogselius P. *The Dynamics of innovation in Eastern Europe. Lessons from Estonia, New Horizons in the Economics of Innovation series*. (Cheltenham, UK: Edward Elgar Publishing Ltd., 2005).

17. Juhan Zimmermann [Online]. Available: URL: http://entsyklopeedia.ee/artikkel/zimmermann_juhan
18. Kaataja S. “Expert groups closing to divide. Estonian–Finish computing cooperation since the 1960th”, *Beyond the divide: entangled histories of Cold War Europe*. Eds. S. Mikkonen and P. Koivunen. (Berghahn Books, 2015), 101–120.
19. Kangilaski J., Salo V. Okupatsioonide Repressiivpoliitika Uurimise Riiklik Komisjon, *The White book: losses inflicted on the Estonian nation by occupation regimes, 1940-1991* (Tallinn: Estonian Encyclopaedia Publishers, 2005), 14–15.
20. *Kibernetiku – na sluzhbu kommunizmu*. Ed. acad. A. I. Berg, (Moscow-Leningrad: Gosenergoizdat, 1961), 1, 312. – In Russian.
21. Kotov V. E. and Narinyani A. S. “Asinkhronnye vychislitelnye protcessy nad pamiatiu”, *Kibernetika*, 3 (1966), 63–72. – In Russian.
22. *Krayneva I. Savelova O*. Female programming fase (mid 1950s–erly 21st sentury) // HISTELCON 2021. Selected papers. Ed. By V. Burov. 2021, 1–6.
23. Krayneva I.A., Pivovarov N.Yu., and Shilov V.V., “Soviet Computing: Developmental Impulses”, in *Selected Papers SoRuCom-2017. Fourth Intern. Conf. on Computer Technology in Russia and in the Former Soviet Union (SoRuCom)*, 2018, 13–22.
24. Krayneva Irina and Cheremnykh Natalia, *Put’ programmista* (Novosibirsk: Nonparel’, 2011), 44–51. – In Russian.
25. Kuznetsov I. S. *Novosibirskii Akademgorodok v 1968 g.: pismo soroka vosmi. Dokumentalnoe issledovanie*, (Novosibirsk: “Ofset-TM”, 2nd ed., 2015) – In Russian.
26. Lavrov S. S. “Sintez programm (v chastnosti, sistema SPORA), *Kibernetika* (1982) 6, 11–16. – In Russian.
27. Letter. G. I. Marchuk – A. A. Dorodnitsyn 04.12.1971 (Ibid., f. 437, l. 26.) Workers without a scientific degree earned 100 to 120 rubles at the time; those with a degree earned up to 250 rubles.
28. Letter. O. Terno – G. I. Marchuk, 03.06.1970 (Academician A. Ershov’s Digital Archive, f. 328, l. 106).
29. Letter. A.P. Ershov – R.M. Burstall. 19.11.1979 (Academician A. Ershov’s Digital Archive, f. 112, l. 508).
30. Letter. E.H. Tuygu – A.P. Ershov. 25.12.1978 (Ibid., f. 260, l. 80).
31. List of members, Commission on System Mathematical Support (Ibid. f. 260, l. 52).
32. Lõhmus Alo. Professor E. Tyugu is running for President of Estonia. *Postiemees*, (1996) Sept, 06.
33. Makarov V.G. *Vysylka vmesto rasstrela: deportatciya intelligentcii v dokumentakh VCHK-GPU: 1921–1923*, (Moscow: Russkiy Put’, 2005). – In Russian.
34. Makeev S.V. *Kontceptcii tekhnokratizma: istoriko-filosofskii analiz*, (PhD dissertation. Deprt. of Philosophy: Moscow State Regional Univ., 2008). – In Russian.
35. Mints G. and Tyugu E. The programming system PRIZ, *Journal of Symbolic Computation* (1988) 5, 3, June, 359–375.
36. Paju P. “Finlandized computing or business as usual? Computer trade between Finland and the Soviet bloc in the 1970s.” *Proc. 24th Intern. Congr. of History of Sci., Technology and Medicine* [Online]. Available: http://www.ichstm2013.com/programme/guide/p/1036.html
37. Pazhit Iu.Iu. “Severo-Uralskii lager NKVD SSSR v gody Velikoi otechestvennoi voiny”, *Voennyi commentator*, 2 (2002), 41. – In Russian.
38. Photo-collection of the Café-club *Pod integralom* // SB RAS Photographic Archive [Online]. Available: http://www.soran1957.ru/?id=w20070417_3_6323
39. Project START, *Communications of the ASM* (1991), 34, no. 6, June, 30–67.
40. Speeches, section “G” on 2-ya Vsesoyuznaya Konferentciya po Programmirovaniyu – VKP-2 (Academician A. Ershov’s Digital Archive, f. 553, l. 6).

41. Stöcker L.F. "*Brining the Baltic Sea: Networks of Resistance and Opposition during the Cold War Era*". Thesis submitted for assessment with a view to obtaining the degree of Doctor of History and Civilization, (Florence: European University Institute, 2012).
42. Tõugu Enn. *Elu nagu etendus: Siberi ja Kremli kaudu Rootsi Kunigriiki* [Life as a Show: Through Siberia to Kremin and the Kingdom of Sweden], (Tallinn: Kirjastus Varrak, 2017).
43. Tyugu E., "Beginning of Computing in the Soviet Baltic Region", in Proceedings, *Third Intern. Conf. on the History of Computers and Informatics in the Soviet Union and Russian Federation: History and Prospects*, (Kazan: KNRTU-KAI, 2014), 12–17.
44. Tyugu E. "Grigori Mints and computer science" [Online] Available: URL: http://kodu.ut.ee/~varmo/tday-kaariku/GMe.pdf/
45. Tyugu E. (1982) Works on system software at the Academy of Sciences of the EstSSR. [Online]. Available: A. Ershov archive. F. 275. Ll. 11–26.
46. Tyugu E.H. The Structural Synthesis of Programs, in *Algorithms in Modern Mathematics and Computer Science. Proc.*, Urgench, Uzbec SSR, September 16–22, 1979 (LNCS: Springer, 1981), 122, 290–303.
47. Veblen T.B. *The Engineers and the Price System*, ed. A. Smirnov. (M.: Izd. dom NIU «VSHE», 2018), 43.
48. Vosstanovlenie nezavisimosti v 1985–1991, *The Estonia Encyclopedia*. [Online]. Available: URL: http://www.estonica.org/ru/ – In Russian.
49. Yurchak A., *Everything was forever, until it was no more. The last Soviet Generation*, (Princeton University Press, 2006).

УДК 81’33:004.822

Генерация лексико-синтаксических паттернов онтологического проектирования на основе вопросов оценки компетенции

Овчинникова К. А. (Новосибирский национальный исследовательский государственный университет),

Сидорова Е. А. (Институт систем информатики СО РАН)

Работа посвящена исследованию проблем автоматизации создания онтологий научных предметных областей с применением методов автоматического анализа текстов на естественном языке. Целью работы является разработка методов автоматической генерации лексико-синтаксических шаблонов для извлечения информации и пополнения онтологий на основе анализа содержательных паттернов онтологического проектирования для научных областей знаний, разрабатываемых в рамках концепции Semantic Web. Паттерны онтологического проектирования представляют собой структурированной описание понятий верхнего уровня в терминах классов, атрибутов и отношений, а также включают вопросы оценки компетенции на естественном языке, служащие для понимания и корректной интерпретации свойств и связей понятия пользователями. В статье предложен подход к генерации лексико-синтаксических паттернов на основе вопросов оценки компетенции. Процесс генерации лексико-синтаксических паттернов включает генерацию предметного словаря, выделение сущностей онтологии и формирование структуры паттернов на основе свойств Data Property и Object Property, и генерацию семантических, грамматических и позиционных ограничений. Вопросы оценки компетенции используются для выявления грамматических и позиционных ограничений, необходимых для поиска онтологических отношений в текстах. Для эксперимента использовалась онтология «Поддержка принятия решений в слабоформализованных областях» и корпус научных текстов той же предметной области. В ходе эксперимента получены следующие результаты: степень неоднозначности сгенерированных шаблонов - 1.5, F1-мера оценки качество поиска атрибутов и отношений объектов - F1-мера составила 0,77 для атрибутов и 0,55 для отношений соответственно. Сравнение результатов, полученных для шаблонов без грамматических ограничений, и результатов, полученных для шаблонов с

грамматическими ограничениями, показало, что добавление ограничений существенно улучшает качество извлечение объектов онтологии.

Ключевые слова: *лексико-синтаксический паттерн, генерация паттернов, вопросы оценки компетентности, пополнение онтологии, онтология научной деятельности, паттерны онтологического проектирования, извлечение информации.*

1. Введение

Под извлечением информации понимается процесс автоматического извлечения полезного материала из текстов некоторой предметной области, его обработка и использование. Интерес к этой процедуре повышается благодаря большому объему неструктурированной информации в Интернете.

Извлечение информации из текстов узкоспециализированных научных областей представляет наибольший интерес из-за проблемы недостаточного количества размеченных данных. Это усложняет использование методов машинного обучения и глубокого обучения и является причиной использования других методов. Альтернативным подходом является использование методов на основе знаний и их дальнейшая интеграция с методами машинного обучения.

Семантическая паутина (англ. Semantic Web) [6] — часть глобальной концепции развития сети Интернет, целью которой является реализация возможности машинной обработки информации, доступной во Всемирной паутине. Основной акцент концепции делается на работе с метаданными, однозначно характеризующими свойства и содержание ресурсов. Онтология и язык ее описания является одним из способов стандартизации представления знаний и информации, поддерживающем машинную обработку. Развитие этих инструментов может быть объяснено востребованностью онтологий [8, 16] как способа стандартизации знаний о предметных областях, хранения, навигации и поиска хорошо структурированных данных. Унификация средств представления онтологий и создание банка готовых решений [15], включающих стандартные образцы сущностей онтологий, ставит перед исследователями новые задачи, а именно необходимость создать механизмы использования образцов готовых решений для проектирования и разработки пользовательских онтологий, а также инструменты для ее автоматизированного пополнения.

В задачах пополнения онтологий используются разные методы: методы машинного обучения (правила кластеризации или ассоциации) [9, 10], итерационные методы с использованием графов [18] и на основе шаблонов (паттернов) [5, 12, 14].

Для упрощения процесса разработки и дополнения онтологий в некоторых исследованиях [7, 19] уже более десяти лет используется подход, основанный на использовании паттернов онтологического проектирования (ОП). Они представляют собой задокументированные описания проверенных решений общих проблем онтологического моделирования. Одним из видов паттернов ОП являются паттерны содержания, описывающие фрагменты онтологии предметной области. Авторы методологии XD (eXtreme Design methodology) [6] предлагают добавлять в паттерн содержания не только описание одного онтологического класса, его атрибутов и отношений, но и вопросы оценки компетенции (ВОК). Вопросы оценки компетенции (ВОК) - это выраженные на естественном языке вопросы к структурным элементам класса, служащие для понимания и корректной интерпретации свойств и связей понятия пользователями. Еще одним типом паттернов ОП являются лексико-синтаксические шаблоны (ЛСП). Эти паттерны представляют собой структурные образцы конструкций языка, отражающие их лексические и поверхностные синтаксические свойства. ЛСП определяют отображение языковых единиц текста в онтологические структуры. Исследователи могут использовать ЛСП при решении задачи автоматического построения онтологий на основе корпуса текстов на естественном языке.

Подход представленный в работе [14] описывает использование лексико-синтаксических паттернов, соответствующих онтологическим паттернам проектирования, для пополнения онтологии. В отличие от паттернов Херст [11], они носят более общий характер, что позволяет охватывать большее количество вхождений сущностей в текстах. Такие паттерны показывают высокую полноту и низкую точность, тогда как паттерны Херст, наоборот, демонстрируют высокую точность извлечения информации, но низкую полноту. Разработка таких шаблонов является достаточно кропотливой работой, поэтому существует необходимость автоматизации данного процесса.

В связи с частым отсутствием размеченных данных одним из активно развиваемых подходов является генерация закономерностей на основе небольшого количества информации, представленной в справочных материалах, толковых словарях или непосредственно в онтологиях. В данной работе для автоматического формирования лексико-синтаксических моделей предлагается использовать вопросы оценки компетентности и научный словарь. Экспериментальное исследование проводится на материале, представленном на портале «Поддержка принятия решений в слабоформализованных областях» (https://uniserv.iis.nsk.su/rdms/), и корпусе научных текстов той же предметной области.

2. Подход к автоматизации пополнения онтологии

В ходе работы [19] были разработаны паттерны содержания для некоторых понятий, характерных для большинства научных предметных областей: *Объект исследования, Предмет исследования, Метод, Задача, Раздел науки, Научный результат, Деятельность, Проект, Персона, Организация, Публикация, Информационный ресурс и др*. Также был определен набор вопросов оценки компетенции для каждого из этих паттернов. С помощью них были выявлены ограничения для паттернов и описаны требования к ним, которые были представлены в виде аксиом и ограничений. Для каждого паттерна, представляющего концепт научной предметной области (НПО), мы определили набор ключевых атрибутов, однозначно идентифицирующих конкретный экземпляр класса.

Для реализации автоматического пополнения онтологии для каждого паттерна содержания строится набор лексико-синтаксических шаблонов, который описывает различные способы представления информации в научных текстах на основе извлеченной информации.

Рассматривая ЛСП как инструмент пополнения онтологии, мы определили две ключевые задачи для достижения поставленной цели. Первой задачей является извлечение имен объектов (в том числе не входящих в словарь) и значений их атрибутов. Во-вторых, это создание объектов на основе структуры классов онтологий. В соответствии с этими задачами были выделена два типа ЛСП: терминологический и информационный. В работе [4] предложена архитектура системы для пополнения онтологии на основе лексико-синтаксических паттернов, реализующая алгоритм пополнения. Предлагается использовать следующие технологии: система извлечения предметной лексики из текстов и построения словарей KLAN [3], система анализа текстов на основе шаблонов PatTerm [17], система анализа текстов FATON [1]. Взаимодействие с онтологией обеспечивает специально разработанный модуль, использующий средства поддержки онтологически-ориентированного программирования из библиотеки owlready2 [13].

ЛСП автоматически строятся на основе словарей общенаучной и предметной лексики и актуальной версии онтологии научной предметной области. На Рис. 1 представлена схема взаимосвязей компонентов системы, участвующих в генерации ЛСП.

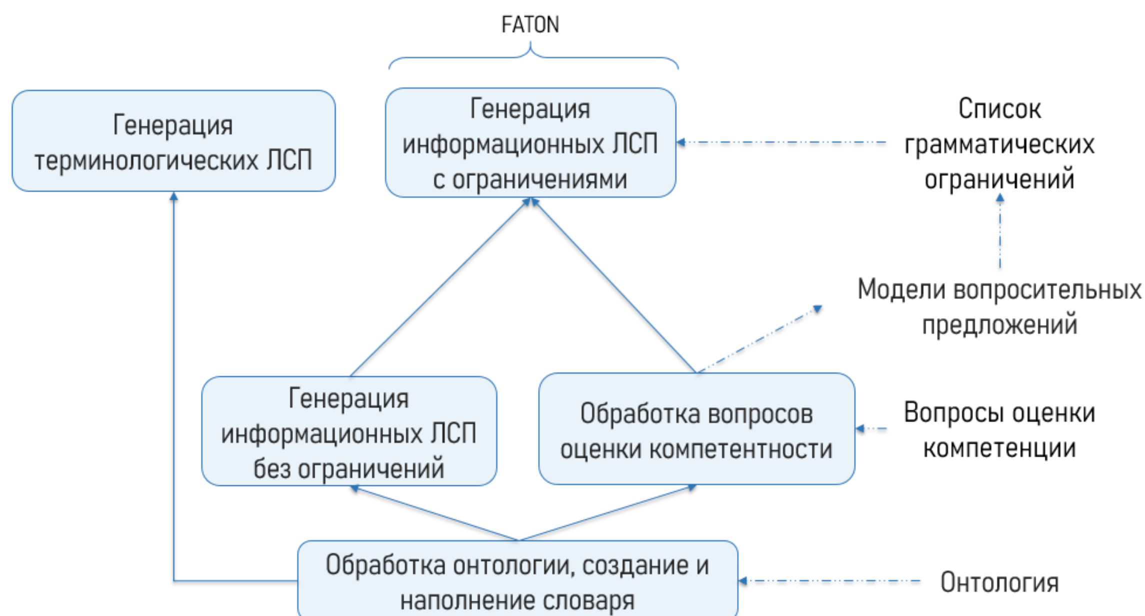

Рис. 1. Схема взаимосвязей компонентов системы, участвующих в генерации ЛСП.

На первом этапе генерации происходит обработка онтологии, создание и наполнение словаря. После чего происходит извлечение имен классов объектов, атрибутов и отношений из онтологии.

Данный этап подробно рассмотрен в работе [3]. На нем формируются Т-ЛСП с использованием индикаторных термов, полученных на основе онтологии, содержащие переменные с заданными свойствами. Означивание таких переменных конкретными фрагментами текста позволяет получить новые значения. В результате создаются многословные термины для словаря и шаблоны для извлечения терминов.

На следующем этапе происходит обработка вопросов оценки компетентности: удаляются лишние символы, строятся модели вопросительных предложений и извлекаются граммемы. Затем генерируются И-ЛСП, для которых требуется информация о ключевых атрибутах классов онтологий, особенно для объектов, создающих шаблоны (например, атрибут Название для Метода исследования), и отношениях с другими классами.

2.1. Словарь научной лексики

Для автоматического пополнения онтологии с помощью ЛСП важно обеспечить извлечение из текста специфических терминов данной научной предметной области. Для этого предлагается использовать словарь научной лексики, который включает не только общенаучную лексику, но и предметную.

Для создания общенаучного словаря было необходимо собрать тематически-нейтральный корпус научных текстов. С этой целью тексты распределялись по 5 научным коллекциям, соответствующим гуманитарным, естественным, техническим, общественным и точным наукам. Всего для проведения исследования было собрано 100 русскоязычных научных публикаций, относящихся к списку ВАК (Высшая аттестационная комиссия) или библиографической базе научных публикаций RSCI (Russian Science Citation Index) или базе научной периодики Scopus. Общий объем корпуса составляет 370,8 тыс. терминов.

Для проведения исследования был создан корпус, включающий 100 русскоязычных научных публикаций. Каждая из них относится к списку ВАК (Высшая аттестационная комиссия), библиографической базе научных публикаций RSCI (Russian Science Citation Index) или базе научной периодики Scopus. Общий объем корпуса составляет 370,8 тыс. терминов.

Следующий этап заключается в создании и обработке общенаучного словаря, который автоматически строится на основе онтологии и корпуса текстов. Для разметки научных терминов на основе анализа семантических значений, представленных в корпусе, были выделены 8 универсальных классов: *Восприятие, Ментальные, Существование, Сущность, Создание, Деятельность, Результат* и *Применение*. Такое разделение основано на семантических значениях, выраженных в предложениях. Например, универсальный класс *Деятельность* включает в себя глаголы несовершенного вида, которые определяют разворачивающееся действие, совершаемое с помощью, например, *Метода исследования*, а класс *Результат* – действие с акцентом на результат, получаемый, например, с помощью применения *Метода исследования*. Универсальные классы могут использоваться в паттернах для нахождения синонимов или в случае стандартных (универсальных) способов выражения отношений.

Предметный словарь создается как расширение словаря общенаучной лексики и включает в себя слова и словосочетания (термины), относящиеся к конкретной предметной области. Система предметно-ориентированных классов основана на структуре онтологии НПО, отражая иерархию ее объектов и отношений. Имена классов терминов генерируются на основе названий онтологических элементов в соответствии с шаблоном: <название_класса.название_отношения> или <название_класса. название_аттрибута>. Например, отношение *Метод исследования.используется_в*.

В словарной статье хранится вся информация, необходимая для извлечения термина из текста и поддержки следующих этапов анализа текста. Каждый термин словаря, найденный в

тексте, снабжен морфологической и семантической информацией, которые впоследствии используются при создании и применении ЛСП.

Научный словарь представляет собой интеграцию двух словарей: универсального и тематического. Поэтому он включает в себя две самостоятельные иерархии лексико-семантических классов: универсальную иерархию из словаря общенаучной лексики и предметно-ориентированную иерархию, основанную на онтологии (Рис. 2).

Универсальный класс	***Список лексем***	***Предметно-ориентированный класс***
ментальные	объяснять, определять, трактовать, расценивать, рассматривать, …	Метод.представлен_на
создание	предложить, ввести, разработать, описать, создать	Метод.создан_в Метод.имеет_автора
применение	применять, применяться, использовать, использоваться	Метод.используется_в Метод.применяется_к

Рис. 2. Фрагмент лексико-семантических классов созданного словаря.

Все словарные термины отмечены признаками из предметной и/или универсальной иерархии. Лексико-семантические признаки словаря используются при описании ЛСП как способа обозначения терминов предметной области науки с определенной семантикой.

2.2. Вопросы оценки компетентности

Анализ вопросов оценки компетентности (ВОК) позволяет выявить начальные синтаксические свойства языковых выражений, описывающих связи между понятиями предметной области, которые могут быть впоследствии уточнены на основе корпуса текстов [18]. Таким образом, вопросы оценки компетентности могут быть использованы при генерации лексико-синтаксических паттернов для уточнения синтаксических и позиционных ограничений.

Можно выделить 5 видов вопросов оценки компетентности:

- вопросы, не содержащие вопросительных слов (*«Применяется ли метод к объекту исследования?»*);
- вопросы, не содержащие значимых вопросительных слов (*«**Какой** объект исследования исследуется в деятельности?»*);
- вопросы, содержащие вопросительные слова, напрямую не обладающие семантикой, относящей его к атрибуту класса онтологии (*«**Когда** была дата начала проекта?»*);

- вопросы, содержащие вопросительные слова, напрямую не обладающие семантикой, относящей его к классу онтологии (*«**Кто** использует метод?»*);
- вопросы, содержащие вопросительные слова, не несущие семантической нагрузки, которую можно связать с некоторым классом онтологии (*«**Как** называется задача?»*).

Был составлен отдельный список вопросительных слов третьей и четвертой группы и добавлено соотношение с атрибутами или классами онтологии:

где: Географическое место.Название,

когда: Информационный ресурс.Дата, Публикация.Дата

кто: Персона.Фамилия, Организация.Название.

Для каждого отношения в онтологии были разработаны 1-3 вопроса оценки компетентности. Например, для отношения Метод.имеет автора были предложены следующие вопросы:

Кто придумал метод?

Кем предложен метод?

Кто является автором метода?

В данном случае все вопросы являются вопросами третьего типа, т.к. для обозначения неизвестного субъекта в русском языке обычно используется местоимение *кто* (и производные от него).

Для отношения *Задача.решается* был предложен только один вопрос:

«Какая задача решается в разделе науки?»

Наличие только одного варианта можно объяснить тем, что данное отношение не предполагает вариативности названия актантов в предложении, а данный предикат полно отражает отношение между ними.

Для каждого вопроса оценки компетентности строится модель вопросительного предложения.

Модель вопросительного предложения состоит из объектов, для которых известны соотнесенность с названием онтологического класса, атрибута или отношения и необходимые грамматические категории.

В нашем подходе предложено две модели. Формально их можно представить в следующем виде:

$$M1=\langle O1, Rel, O2\rangle \qquad (3.1)$$

– для связи Rel между объектами O_1 и O_2.

$$M2=\langle O,D\rangle \quad (3.2)$$

– для добавления атрибута D или создания объекта O.

Поясним некоторые обозначения:

O, O_1 и O_2 представляют собой наборы (Name, Grammatic), где Name – название онтологического класса, Grammatic – множество граммем,

D представляет собой набор (DataProperty, DataProperty_type, Grammatic), где DataProperty– название атрибута класса, DataProperty_type – тип атрибута, Grammatic – множество граммем,

Rel представляет собой набор (PWord, ObjectProperty, Grammatic), где PWord – конкретный предикат, ObjectProperty – онтологическое отношение, а Grammatic – множество граммем.

Анализ вопросов оценки компетентности позволяет выделять некоторые грамматические ограничения на паттерны.

2.2.1. Грамматические ограничения

Анализ вопросов оценки компетентности проводится по трем направлениям: анализ предиката и анализ его первого и второго актантов.

Была составлена таблица, в которой были описаны грамматические ограничения для вопросов оценки компетентности. Под грамматическими ограничениями мы понимаем морфологические свойства аргументов и синтаксические связи между ними.

Анализ вопросов оценки компетентности проводится для каждого вопроса с точки зрения граммем каждого релевантного слова. К релевантным словам относятся те, которые имеют непосредственное отношение к объектам онтологии: названия классов и атрибутов и предикаты. Каждому предикату сопоставляются его грамматические категории и название отношения в онтологии, а каждому актанту – грамматические категории и название класса или атрибута онтологии (Таблица 1).

Все морфологические категории слов рассматриваются с точки зрения релевантности и нерелевантности. Во время анализа выделяются нерелевантные грамматические категории. Например, было решено не учитывать ***лицо*** глагола. В вопросе *«Кто предложил метод исследования?»* глагол стоит в 3 лице, как и в любом другом предложении, в котором подлежащее выражено именем собственным (или именами собственными). Компонент *Персона* определяет объект одноименного класса, в котором подразумевается наличие имени собственного, поэтому, если подлежащее в предложении, из которого будет извлекаться

информация, будет выражено чем-то другим, то предложение не будет обрабатываться конструкцией <*Персона*, *Метод.имеет автора*, *Метод исследования*>, поэтому в данном случае указание на лицо глагола можно опустить.

Рассмотрим некоторые релевантные граммемы. Граммема ***времени*** релевантна для тех случаев, когда однозначно определено время их действия. Так, например, для атрибута *Дата* (создания) класса *Метод* в паттерн будет добавляться прошедшее время глагола. Граммема ***падежа*** у существительных и местоимений считается релевантной для всех вопросов.

Таким образом, был составлен список релевантных граммем для каждой части речи:

- Глагол: аспектуальность, переходность, число, время;
- Причастие: аспектуальность, залог, число, время
- Существительное: падеж, число, род
- Местоимение: падеж

2.3. Генерация лексико-синтаксических паттернов

Процесс генерации И-ЛСП можно разделить на несколько этапов:

- предобработка вопросов оценки компетентности;
- морфологический анализ вопросов оценки компетентности;
- сопоставление с элементами онтологии;
- построение моделей вопросительных предложений;
- генерация разных видов И-ЛСП на основе моделей.

На первом этапе происходит приведение вопроса оценки компетентности к нужному формату для проведения дальнейших этапов. В частности, удаление знаков препинания.

«В какой деятельности участвует Организация?» =>

«В какой деятельности участвует Организация»

На этапе морфологического анализа выделяются релевантные граммемы для каждого релевантного слова. Далее происходит сопоставление этих слов с классами, атрибутами и отношениями в онтологии*:*

Деятельность Class: Персона, Grammemes: П.п.

Организация Class: Организация, Grammemes: И.п.

участвует Rel: Организация.участвует в деятельности, Grammemes: [невозврат., несов. вид, неперех., ед. ч., н.в.]

В конце обработки ВОК каждое слово будет сопровождаться информацией о нем:

- часть речи;
- грамматические значения;
- название онтологического класса или атрибута;
- тип атрибута (инициализирующий, Data Property, Object Property).

После обработки вопросов оценки компетентности строятся модели вопросительных предложений. Для каждого вопроса в модель добавляется:

- сопоставленные с релевантными словами объекты онтологии (классы и атрибуты);
- предикат в его изначальной форме;
- указывается тип атрибута (если он есть);
- список грамматических ограничений.

Пример модели вопросительного предложения:

M= ⟨ (Деятельность, [П.п.]) (участвует, Организация.участвует в деятельности, [невозврат., несов. вид, неперех., ед. ч., н.в.]) (Организация, [И.п.]) ⟩ (3.5)

На последнем этапе на основе созданных моделей осуществляется генерация И-ЛСП.

Scheme

arg1: Object::Деятельность (Падеж: ‘пр’)

arg2: Term::Организация.участвует в деятельности (Число: ‘ед’)

arg3: Object::Организация (Падеж: ‘им’)

Condition Contact (arg1, arg2) = Contact_Object,

Contact (arg2, arg3) = Contact_Object

⇨ arg3::Организация (участвует в деятельности: arg1.Название)

На схеме показан простой вариант связывания объекта класса *Метод* с объектом класса *Персона*. Для этого рассматриваются уже три аргумента: объект класса *Персона*, термин лексико-синтаксического класса *Метод.имеет автора* и объект класса *Метод исследования*. Для указания на возможность аргументов быть разделенными используется контактность (Contact) в разделе условий (Condition). Такой паттерн будет обрабатывать случаи вида: *«Метод “мозгового штурма” был разработан в 1953 г. американским консультантом Осборном»*.

Предложенные этапы позволяют формировать информационные лексико-синтаксические модели на основе вопросов оценки компетентности с учетом грамматических значений каждого из них.

3. Оценка качества генерации

Во время генерации из 209 вопросов оценки компетентности было сформировано 200 неповторяющихся моделей И-ЛСП. Уменьшение количества моделей по сравнению с количеством вопросов можно объяснить тем, что, несмотря на разные вопросы для отношений, ограничения на объекты могли совпадать.

Была выбрана онтология «Поддержка принятия решений в слабоформализованных предметных областях» (https://uniserv.iis.nsk.su/rdms/) [21], на основе которой был создан словарь предметной области (129 объектно-ориентированных классов и 689 терминов) и корпус, состоящий из 31 научного текста той же предметной области, для проведения эксперимента по извлечению информации с использованием сгенерированных шаблонов. Эксперименты проводились в системе FATON.

Были использованы стандартные метрики точности, полноты и F_1-меры для анализа результатов извлечения информации.

В Таблице 1 представлены результаты генерации паттернов с грамматическими ограничениями (они выделены серым) и без грамматических ограничений.

Таблица 1. Результаты генерации паттернов

Степень неоднозначности		**Точность**		**Полнота**		**F_1**	
-	1,5	0,97	0,70	1,0	0,96	0,98	0,81

Из таблицы видно, что генерация паттернов с грамматическими ограничениями дала результаты ниже, чем без грамматических ограничений.

Ожидалось получение полноты, равной 1.0, поскольку вопросы оценки компетентности охватывают все отношения в онтологии. Было подсчитано, что 45 вопросов оценки компетентности не сформировали паттерны. 9 из них оказались избыточными, что привело к уменьшению количества неповторяющихся моделей по сравнению с вопросами. В оставшихся же не были обнаружены отношения, чему способствовал ряд причин.

Во-первых, невозможность сравнения некоторых онтологических отношений со словами в ВОК. К таким отношениям можно отнести *являетсяЧастьюРесурса*, *являтьсяЧастью* и *автор-Персона*. Во-вторых, были замечены опечатки в названиях онтологических

отношений и в связях в базовой онтологии. Так, например, отношение *Организация.участвует в деятельности* связывает не два онтологических класса *Организация* и *Деятельность*, а онтологический класс *Организация* и онтологическое отношение *Организация.участвует в деятельности.*

4. Эксперименты на корпусе научных текстов

Для проведения экспериментов по извлечению информации с помощью сгенерированных паттернов были выбраны онтология «Поддержка принятия решений в слабо формализованных областях» и корпус, состоящий из 31 научного текста той же предметной области. Эксперименты проводились в системе FATON.

Для формализации полученных данных для каждого текста были автоматически созданы таблицы, в которых описаны созданные онтологические объекты, добавленные атрибуты и связи. Они выводятся в каждой новой строчке в той последовательности, в которой встречаются объекты.

Для созданных онтологических объектов строка содержит название онтологического класса, атрибут *Название* и конкретную лексему. Для добавленных атрибутов объектов строка включает название онтологического класса, название типа атрибута и конкретную лексему его наименования. Для созданных связей между объектами указывается класс и идентификатор первого объекта, название отношения, класс и идентификатор второго объекта.

Эксперименты были проведены для паттернов без грамматических ограничений и паттернов, в которые в процессе генерации были добавлены грамматические ограничения. Результаты приведены в таблице ниже (Таблица 2), где серым цветом выделены значения для паттернов без ограничений.

Таблица 2. Результаты извлечения сущности онтологии

/	**Точность**		**Полнота**		**F_1**	
Attributes	0,34	0,83	0,97	0,72	0,50	0,77
Relations	0,09	0,38	1,0	1,0	0,17	0,55

Все ожидаемые объекты извлекаются из текстов, поэтому они не были внесены в таблицу. Полнота добавления атрибутов и отношений также оказалась высокой, что говорит об извлечении большей части (или всех) ожидаемых атрибутов и отношений, однако низкая

точность указывает на извлечение большого количества неправильных атрибутов и отношений.

Полученные результаты показывают, что паттерны с ограничениями значительно увеличивают точность добавления атрибутов, но при этом уменьшают полноту по сравнению с паттернами без ограничений.

Увеличение точности извлечения отношений было достигнуто сначала добавлением грамматических ограничений, а затем регулированием добавления связи с самим собой. Количество полученных объектов в онтологии будет совпадать с ожидаемым.

Анализ ошибок помог выявить причины ошибочных результатов:

- слишком строгие ограничения на падеж аргументов для паттернов, добавляющих атрибуты объектам онтологии;
- недостаточно строгие ограничения на позицию аргументов (расположение относительно друг друга) для паттернов, создающих связи между объектами онтологии;
- списки рассматриваются как одно предложение, из-за чего формируются ненужные связи;
- объектам добавляется связь с самим собой.

Увеличение полноты можно достичь с помощью добавления паттернов, в которых не будет грамматических ограничений, но будет ограничение на контактность.

Для улучшения точности можно добавить предварительный анализ, который позволил бы разбить список на фрагменты текста, чтобы избежать ненужной связи между частями списка. Также необходимо убрать возможность аргументов создавать связи с самим собой.

5. Заключение

Данная работа является продолжением цикла работ, посвященных методам автоматической генерации лексико-синтаксических паттернов онтологического проектирования, предназначенных для извлечения информации из текстов и пополнения онтологии. В работе предложен подход к генерации лексико-синтаксических паттернов на основе онтологических паттернов НПО. Особенностью подхода является использование вопросов оценки компетентности для извлечения грамматических свойств языковых выражений, используемых для представления понятий в текстах.

Во время проведения экспериментов были получены следующие результаты. При генерации степень неоднозначности паттернов составила 1,5. При извлечении информации F1-мера составила 1,0 для объектов, 0,77 для атрибутов и 0,55 для отношений. В целом, полученные результаты показывают, что паттерны с грамматическими ограничениями значительно увеличивают точность извлечения атрибутов и отношений, но при этом уменьшают полноту по сравнению с паттернами без ограничений. Проведенный анализ ошибок позволяет сделать предположение, что добавление новых паттернов, которые будут содержать только ограничение на взаиморасположение терминов в текстах, позволит увеличить полноту. Что касается точности, то предлагается проводить предварительный анализ, который позволит, в частности, разбивать список на фрагменты текста во избежание неправильных связей между частями списка.

Дальнейшие исследования будут связаны с а) апробацией подхода на других предметных областях, б) расширением обучающего корпуса ВОК утвердительными предложениями (например, определениями из энциклопедических источников знаний), что может дать более полную картину возможных ограничений, в) рассмотрением других типов грамматических ограничений (например, согласование в числе и роде), г) применением для генерации терминологических паттернов.

Список литературы

1. Гаранина, Н. О. Мультиагентный подход к извлечению информации из текстов и пополнению онтологии / Н. О. Гаранина, Е. А. Сидорова // Материалы Всероссийской конференции с международным участием «Знания – Онтологии – Теории» (ЗОНТ–2015), 6 — 8 октября 2015 г., Новосибирск. – Новосибирск: Институт математики им. С. Л. Соболева СО РАН, 2015. –Т.1. — С. 50-59.
2. Кононенко И.С., Сидорова Е.А. Методика разработки лексико-семантических паттернов для извлечения терминологии научной предметной области // Системная информатика. 2022. № 20. С. 25-46.
3. Сидорова, Е. А. Комплексный подход к исследованию лексических характеристик текста / Е. А. Сидорова // Вестник СибГУТИ, №3, 2019. – С. 80-88.
4. Загорулько Ю.А., Сидорова Е.А., Загорулько Г.Б., Ахмадеева И.Р., Серый А.С. Автоматизация разработки онтологий научных предметных областей на основе паттернов онтологического проектирования // Онтология проектирования. – 2021. – Т.11, №4(42). - С.500-520. – DOI: 10.18287/2223-9537-2021-11-4-500-520.

5. Aguado de Cea, A. Using Linguistic Patterns to Enhance Ontology Development / G. Aguado de Cea, A. Gomez-Perez, E. Montiel-Ponsoda, M. C. Suarez-Figueroa // In: Proc. Int. Conf. on Knowledge Engineering and Ontology Development (KEOD 2009) (Funchal - Madeira, Portugal, October 6-8, 2009), 2009. – P. 206–213.
6. Blomqvist, E. Engineering Ontologies with Patterns: The eXtreme Design Methodology / E. Blomqvist, K. Hammar, V. Presutti // Ontology Engineering with Ontology Design Patterns. Studies on the Semantic Web, 2016. – P. 23 – 50.
7. Gangemi, A. Ontology Design Patterns / A. Gangemi, V. Presutti // Handbook on Ontologies. Springer, 2009. – P. 221–243.
8. Ganino G., Lembo D., Mecella M., Scafoglieri F. Ontology population for open-source intelligence: a GATE-based solution // Software: Practice and Experience. 2018. V. 48. Is. 12.
9. Gnminger, M. Methodology for the design and evaluation of ontologies / M. Gnminger, M. Fox // Workshop on Basic Ontological Issues in Knowledge Sharing. Montreal, Canada, 1995. – 10 p.
10. Guarino, N. OntoSeek: content-based access to the Web / N. Guarino, C. Masolo, G. Vetere // IEEE Intelligent Systems, 1999. – P. 70-80.
11. Hearst, M. Automatic acquisition of hyponyms from large text corpora / M. Hearst // Conference on Computational Linguistics (COLING'92), Nantes, France, Association for Computational Linguistics. 1992. – P. 539-545.
12. Ijntema, W. A lexico-semantic pattern language for learning ontology instances from text / W. Ijntema, J. Sangers, F. Hogenboom, F. Frasincar // Journal of Web Semantics, 2012. – P. 37–50.
13. Lamy, J.-B. Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies / J.-B. Lamy // Artificial Intelligence In Medicine, 2017. – P. 11-28.
14. Maynard, D. Using Lexico-Syntactic Ontology Design Patterns for ontology creation and population / D. Maynard, A. Funk, W. Peters // Proceedings of the 2009 International Conference on Ontology Patterns, vol. 516, 2009. – P. 39-52.
15. Ontology Design Patterns // Ontology Design Patterns URL: http://ontologydesignpatterns.org (дата обращения: 20.10.2022). (44)
16. Petasis G., Karkaletsis V., Paliouras G., Krithara A., Zavitsanos E. Ontology Population and Enrichment: State of the Art. In: Paliouras G., Spyropoulos C.D., Tsatsaronis G. (eds). Knowledge-Driven Multimedia Information Extraction and Ontology Evolution. LNCS, V. 6050. Springer, Berlin, Heidelberg.
17. Rosenberg, G. Handbook of Formal Language / G. Rosenberg, F. Salomaa, 1996. – 450 p.
18. Roux, C. An ontology enrichment method for a pragmatic information extraction system gathering data on genetic interactions / C. Roux, D. Proux, F. Rechenmann, L. Julliard // Proceedings of the First Workshop on Ontology Learning (OL-2000) in conjunction with the 14th European Conference on Artificial Intelligence (ECAI 2000), Berlin, Germany, 2000.

19. Zagorulko Y. A. Using a System of Heterogeneous Ontology Design Patterns to Develop Ontologies of Scientific Subject Domains / Y. A. Zagorulko, O. I. Borovikova // Programming and Computer Software, 2020. – P. 273-280.
20. Zagorulko Y. A., Sidorova E. A., Akhmadeeva I. R. and Sery A. S . Approach to automatic population of ontologies of scientific subject domain using lexico-syntactic patterns // International Conference «Marchuk Scientific Readings 2021» (MSR-2021) 4-8 October 2021, Novosibirsk, Russian Federation. Journal of Physics: Conference Series, 2021, vol.2099, p. 012028. doi:10.1088/1742-6596/2099/1/012028)
21. Zagorulko Y., Zagorulko G. Application of Ontology Design Patterns for Building an Ontology of Decision Support in Weakly Formalized Domains // Proceedings of Selected Contributions to the Russian Advances in Artificial Intelligence Track at RCAI 2021, collocated with the 19th Russian Conference on Artificial Intelligence (RCAI 2021). – Vol. 3044. – P. 108–116. – CEUR Workshop Proceedings, CEUR-WS.org, 2021.