Syllabus
Formal Methods in Software Engineering
(10 ECTS)

Prof. Dr. I.A. Lomazova (ilomazova@hse.ru, https://www.hse.ru/staff/ilomazova)
Department of software engineering
Meeting Minute # ___dated _____ 20_

1. Course Description

a) Pre-requisites

- Informatics, mathematical logics, and theory of computation;
- Discrete mathematics;
- Software programming.

b) Abstract

In computer science and software engineering, **formal methods** are a particular kind of mathematically based techniques for the specification, development and verification of software and hardware systems. The use of formal methods for software and hardware design is motivated by the fact that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design.

Formal methods are best described as the application of a fairly broad variety of theoretical computer science fundamentals, in particular logic calculi, formal languages, automata theory, and program semantics, but also type systems and algebraic data types to problems in software and hardware specification and verification.

Formal methods can:

- be a foundation for describing complex systems.
- be a foundation for reasoning about systems.
- provide support for program development.

In contrast to other system design approaches, formal methods use mathematical proof as a complement to system testing in order to ensure correct behavior. As systems become more complicated, and safety becomes a more important issue, the formal approach to system design offers another level of insurance.

Formal methods differ from other design methods by the use of formal verification schemes, the basic principles of the system must be proven correct before they are accepted. Traditional system design uses extensive testing to verify behavior, but testing is capable of only finite conclusions. E. Dijkstra and others have demonstrated that tests can help to

find fails and bugs, but cannot guarantee their absence. In contrast, once a theorem is proven true it remains true.

It is very important to note that formal verification does not obviate the need for testing. Formal verification cannot fix bad assumptions in the design, but it can help identify errors in reasoning which would otherwise be left unverified. In several cases, engineers have reported finding flaws in systems once they reviewed their designs formally. Roughly speaking, formal design can be seen as a three-step process, following the outline given here:

1. **Formal Specification**: During the formal specification phase, the engineer rigorously defines a system using a modeling language. Modeling languages are special formalisms which allow users to model complex structures out of predefined types. This process of formal specification is similar to the process of converting a word problem into algebraic notation.
   In many ways, this step of the formal design process is similar to the formal software engineering technique developed by Rumbaugh, Booch and others. At the minimum, both techniques help engineers to clearly define their problems, goals and solutions. However, formal modeling languages are more rigorously defined. And the clarity that this stage produces is a benefit in itself.
2. **Verification**: As stated above, formal methods differ from other specification systems by their heavy emphasis on provability and correctness. By building a system using a formal specification, the designer is actually developing a set of theorems about his/her system. Verification is a difficult process, largely because even a very simple system usually has several dozen theorems, each of which has to be proven. Even a traditional mathematical proof is a complex affair. Given the demands of complexity, almost all formal systems use an automated theorem proving tool of some form. These tools can prove simple theorems, verify the semantics of theorems, and provide assistance for verifying more complicated proofs.
3. **Implementation**: Once the model has been specified and verified, it is implemented by converting the specification into code. As the difference between software and hardware design grows narrower, formal methods for developing embedded systems have been developed. LARCH, for example, has a VHDL implementation.

Formal methods offer additional benefits outside of provability, and these benefits do deserve some mention:

- **Discipline**: By virtue of their rigor, formal systems require an engineer to think out his design in a more thorough fashion. In

particular, a formal proof of correctness is going to require a rigorous specification of goals, not just operation. This thorough approach can help identify faulty reasoning far earlier than in traditional design. The discipline involved in formal specification has proved useful even on already existing systems. Engineers using the PVS system, for example, reported identifying several microcode errors in one of their microprocessor designs.

- **Precision**: Traditionally, disciplines have moved into jargons and formal notation as the weaknesses of natural language descriptions become more glaringly obvious. There is no reason that systems engineering should differ, and there are several formal methods which are used almost exclusively for notation.

For engineers designing safety-critical systems, the benefits of formal methods lie in their clarity. Unlike many other design approaches, the formal verification requires very clearly defined goals and approaches. In a safety critical system, ambiguity can be extremely dangerous, and one of the primary benefits of the formal approach is the elimination of ambiguity.

The purpose of this course is to learn how to specify behavior of systems and to experience the design of a system where you can prove that the behavior is correct. Students will learn how to formally specify requirements and to prove (or disprove) them on the behavior. The behavior of systems will be represented by such formalisms as

- finite state machines;
- process algebras;
- Petri nets;
- temporal logics.

With a practical assignment students will experience how to apply the techniques in practice.

The first part of this course is focused on studying the semantics of a variety of programming language constructs. We will study structural operational semantics as a way to formalize the intended execution and implementation of languages, axiomatic semantics, useful in developing as well as verifying programs, and denotational semantics, whose deep mathematical underpinnings make it the most versatile of all.

Then the special emphasis will be put on parallel and distributed systems modeling, specification and analysis. We consider two basic approaches to concurrent systems specification and analysis: process algebras and Petri nets.

Process algebra is a mathematical framework in which system behavior is expressed in the form of algebraic terms, enhancing the available techniques for manipulation. Fundamental to process algebra is a

parallel operator, to break down systems into their concurrent components. A set of equations is imposed to derive whether two terms are behaviorally equivalent. In this framework, non-trivial properties of systems can be established in an elegant fashion. For example, it may be possible to equate an implementation to the specification of its required input/output relation. In recent years a variety of automated tools have been developed to facilitate the derivation of such properties.

Applications of process algebra exist in diverse fields such as safety critical systems, network protocols, and biology. In the educational vein, process algebra has been recognized to teach skills to deal with complex concurrent systems, by representing and reasoning about such systems in a mathematically clear and precise manner.

Petri nets is another popular formalism for modeling, analyzing and verifying reactive and distributed systems. Their strength are their simple but precise semantics, their clear graphical notation, and many methods and algorithms for analysis and verification.

The course introduces Petri nets and their theory by the help of examples from different application domains. The focus, however, will be on traditional Petri net theory, in particular on Place/Transition-Systems and on concepts such as place and transition invariants, deadlocks and traps, and the coverability tree. The course also covers different versions and variants of Petri nets as well as different modeling and analysis techniques for particular application areas. Thus we consider an urgent topic of modeling and analysis of workflow processes in more details.

The fourth module covers a prominent verification technique that has emerged in the last thirty years – model checking. This approach is based on systematical check whether a model of a given system satisfies a property such as deadlock freedom, invariants, or request-response. This automated technique for verification and debugging has developed into a mature and widely-used industrial approach with many applications in software and hardware. It is used (and further developed) by companies and institutes such as IBM, Intel, NASA, Cadence, Microsoft, and Siemens, to mention a few, and has culminated in a series of mostly freely downloadable software tools that allow the automated verification of, for instance, C#-programs or combinational hardware circuits.

Subtle errors, for instance, due to multi-threading that remained undiscovered using simulation or peer reviewing can potentially be revealed using model checking. Model checking is thus an effective technique to expose potential design errors and improve software and hardware reliability.

This course provides an introduction to the theory of model checking and its theoretical complexity. We introduce transition systems,

safety, liveness and fairness properties, as well as omega-regular automata. We then cover the temporal logics LTL, CTL and CTL*, compare them, and treat their model-checking algorithms. Techniques to combat the state-space explosion problem are at the heart of the success of model checking.

We will show that model checking is based on well-known paradigms from automata theory, graph algorithms, logic, and data structures. Its complexity is analyzed using standard techniques from complexity theory.

2. Learning Objectives

The objective of the "Formal methods in software engineering" course delivery is to train students to treat the specification of software as a very important stage of software development, and also to appreciate the advantages and problems associated with this approach for future projects.

One of the important aspects of formal methods is that, even for quite simple problems, they force the students to think very carefully about the specification, and not to get involved in the coding too quickly. Even for students who have done a lot of programming before the ideas behind formal methods are likely to be completely new, and can draw their attention to problems of program correctness and reliability.

Another very important reason for teaching formal methods is that they are gradually being used in more industrial projects, and thus students should be familiar with at least the ideas associated with the approach, even if they have not learnt the specific formal specification language that their particular industry may require.

During the course, the students will:

- study the basic principles of using formal methods for specification and analysis of soft-ware systems;
- study basic notions and modes of formal semantics for sequential and concurrent programs.
- study formalisms, such as process algebras and Petri nets, and methods for modeling and analysis of concurrent and distributed systems.
- study methods and algorithms for model checking of concurrent systems;
- master methods and tools of software specification, analysis and verification;
- acquire practical skills in design, specification and analysis of model distributed systems examples.

3. Learning Outcomes

Upon the completion of the "Formal methods in software engineering" course, students should be able to:

- understand the language of studied formalisms;

- model various classes of systems using these formalisms;
- apply specific analytical techniques;
- prove properties of discrete systems using process algebras, Petri nets and appropriate specification formalisms.

4. Course Plan

The course is given to the students of the Master Program “System and Software Engineering”, Faculty of Computer Science, the National Research University - Higher School of Economics/HSE.

It is a part of general scientific curricula unit, and it is delivered in modules 1-4 of the first aca-demic year. The course length is 128 academic hours of audience classes divided into 48 lecture hours and 80 seminar hours and 252 academic hours for students’ self-study.

The covered number of credits is 10. Academic control forms are one home assignment, one written exam after module 2, and one written exam after module 4.

№	Topic name	Course hours, total	Audience hours		Self-study
			Lectures	Practical studies	
	Module 1 (80 hrs.)				
	Formal methods as a basis for software reliability.	8	2	2	4
	Floyd method for verification of sequential programs. Hoare axiomatic semantics for sequential and parallel programs.	32	4	8	20
	Finite state machines (FSMs): basic definitions, operational semantics. Categories of FSMs. Extended FSMs. Modeling concurrent systems with communicating FSMs.	22	2	4	16
	Petri nets: basic notions, definitions and classification. Modeling distributed systems with Petri nets.	30	4	6	20
	Module 1, totally:	92	12	20	60
	Module 2 (80 hrs.)				
	Petri nets analysis. Checking structural and behavioral properties.	24	4	6	14
	High-level Petri nets. Colored Petri nets and CPNTools.	22	2	4	16
	Modeling distributed and concurrent system with process algebras. Structured operational semantics and its formalization (SOS). Algebra CCS: syntax, semantics, modeling technique.	30	4	6	20
	The notion and properties of bisimilarity relation.	16	2	4	10
	Module 2, totally:	92	12	20	60

№	Topic name	Course hours, total	Audience hours		Self-study
			Lectures	Practical studies	
	Module 3 (100 hrs.)				
	Verifying reactive concurrent systems with CCS. Hennesy-Milner logic and temporal properties. The notion of fixed point and Tarski's fixed point theorem.	20	4	4	12
	Transition systems and program graphs. Nondeterminism, parallelism and communication. Peterson algorithm.	16	4	2	10
	Specifying distributed systems with Promela. Spin model checker.	30	2	8	20
	Temporal logics LTL and CTL for specification of behavioral properties of reactive systems.	26	4	4	18
	Module 3, totally:	92	14	18	60
	Module 4 (100 hrs.)				
	Automata-based approach for verification of LTL formulae.	74	6	16	52
	Model checking algorithm for verification of CTL formulae.	30	4	6	20
	Module 4, totally:	104	10	22	72
	TOTAL:	380	48	80	252

Topic 1. Formal methods as a basis for software reliability.

Topic outline:

- Why formal methods.
- Formal methods and software/hardware reliability.
- Formal methods: historical overview.
- How logic helps computer scientists.
- Formal methods vs. simulation and testing.
- Course overview.

Topic 2. Floyd method for verification of sequential programs. Hoare axiomatic semantics for sequential and parallel programs.

Topic outline:

- Partial and total correctness assertions.
- Floyd method for proving partial program correctness. The notion of invariant.
- The axiomatic approach for proving program correctness
- Hoare's assertion language: syntax and semantics.
- Partial correctness properties.
- Hoare's logic. Soundness and relative completeness of Hoare's logic.
- Weakest preconditions and their properties.
- Proving total program correctness. Soundness and relative completeness of total correctness.
- Equivalence of axiomatic and denotational/operational semantics.

- Hoare's logic for parallel programs. Semantics of parallel constructions. Rules for partial correctness assertions.

Topic 3. Finite state machines (FSMs): basic definitions, operational semantics. Categories of FSMs. Extended FSMs. Modeling concurrent systems with communicating FSMs..

Topic outline:

- Finite state machines (FSMs): informal introduction, formal definitions, case study.
- State transition diagrams.
- Deterministic and nondeterministic FSMs.
- Extended FSMs.
- Communicating mechanisms for concurrent systems. Specifying distributed systems with interacting automata.
- Proving protocol correctness with communicating FSMs.

Topic 4. Petri nets: basic notions, definitions and classification. Modeling distributed systems with Petri nets.

Topic outline:

- Motivation and informal introduction. Net formalisms for modeling distributed systems. Examples from different areas.
- Place/Transition systems: basic concepts. Places, transition, linear algebraic representation.
- Firing rule, interleaving semantics, occurrence graph, unboundedness.
- Variants of Petri nets: condition/event systems, contact-free nets, high-level Petri nets, colored Petri nets, nested Petri nets.
- Modeling basic control constructs with Petri nets: sequencing, nondeterministic choice, concurrency.
- Modeling causality relations and resource dependencies with Petri nets.

Topic 5. Petri nets analysis. Checking structural and behavioral properties.

Topic outline:

- Interleaving and concurrent semantics for Petri nets. Sequential and concurrent runs.
- Coverability tree.
- Propositional state properties of P/T nets: incidence matrix, state equation, place invariants.
- Positive place invariants and boundedness; transition invariants and deadlocks; siphons and traps.
- Analysis of behavioral problems for Petri Nets: Safeness; Boundedness; Conservation; Liveness; Reachability and coverability.
- Analysis techniques for State Machines, Marked Graphs, Extended Free Choice Nets.

Topic 6. High-level Petri nets. Colored Petri nets and CPNTools.

Topic outline:

- Expressibility of Petri nets. Extending Petri nets with reset and inhibitor arcs.
- Introducing colored tokens and types.
- Hierarchical modeling.
- Modeling multi-agent systems with nested Petri nets.
- Modeling case studies: producer/consumer system, sequential and parallel buffers, crosstalk algorithm, mutual exclusion, dining philosophers.

Topic 7. Modeling distributed and concurrent system with process algebras. Algebra CCS: syntax, semantics, modeling technique.

Topic outline:

- Reactive systems: main notions and examples.
- Flow diagrams of distributed systems. Ports and interactions.
- Interleaving semantics of concurrent systems. Labeled transition systems. Concurrency and nondeterminism.
- The Calculus of Communicating Systems (CCS) of R.Milner informally.
- Formal definition of CCS; semantics of CCS; transition diagrams; examples.
- CCS case studies.

Topic 8. The notion and properties of bisimilarity relation.

Topic outline:

- Trace equivalence; strong bisimilarity; bisimulation games; properties of strong bisimilarity.
- Weak bisimilarity; weak bisimulation games; properties of weak bisimilarity; example (a tiny communication protocol).
- Analysis of CCS behavior; examples.
- Value passing CCS.
- The language of Communicating Sequential Processes (CSP): brief overview.

Topic 9. Verifying reactive concurrent systems with CCS. Hennesy-Milner logic and temporal properties. The notion of fixed point and its and Tarski's fixed point theorem.

Topic outline:

- Syntax of Hennessy-Milner logic; semantics of Hennessy-Miler logic; examples.
- Correspondence between strong bisimilarity and Hennessy-Milner logic.
- Strong bisimulation as a greatest fixed point.
- Game semantics and temporal properties of reactive systems.

Topic 10. Transition systems and program graphs. Nondeterminism, parallelism and communication. Peterson algorithm.

Topic outline:

- Transition systems. Meanings of nondeterminism in transition systems.

- Transition systems and program graphs for sequential and parallel programs. Transition system semantics of a program graph.
- Guarded Command Language.
- Parallelism and communication. Interleaving for a transition system and a program graph.
- Mutual exclusion with semaphores.
- Peterson algorithm.

Topic 11. Specifying distributed systems with Promela. Spin model checker.

Topic outline:

- Sequential Programming in PROMELA specification language: data types, operators and expressions, control statements.
- Verification of sequential programs, assertions, guided simulation.
- Interactive simulation of concurrent programs.
- Synchronization and nondeterminism in concurrent programs.
- Deadlock verification.
- Verification with temporal logic LTL.
- Expressing and verifying safety properties.
- Expressing and verifying liveness properties.
- Case studies.

Topic 12. Temporal logics LTL and CTL.

Topic outline:

- Model and temporal logics: main consepts.
- Linear Temporal Logic LTL: syntax, semantics, main properties and case studies.
- Linear time properties: safety, liveness, decomposition.
- Fairness: unconditional, strong and weak fairness.
- Computational Tree Logic CTL: syntax, semantics, equational laws.
- Comparing LTL and CTL.

Topic 13. Automata-based approach for verification of LTL formulae.

Topic outline:

- Automata on finite words.
- Verifying regular safety properties. Product construction, counterexamples.
- Automata on infinite words. Generalized Büchi automata, ω-regular languages.
- Verifying ω-regular properties: nested depth first search.

Topic 14. Model checking algorithm for verification of CTL formulae.

Topic outline:

- Kripke structures.
- Semantics of CTL on computational trees.
- CTL model checking: recursive descent, backward reachability, complexity.
- Fairness, counterexamples/witnesses.
- CTL^{+} and CTL^{*}.
- Fair CTL semantics, model checking.

5. Reading List

c) Required

1. Singh, Arindama. Elements of Computation Theory / Arindama Singh. – Springer, 2009. – URL: https://link.springer.com/book/10.1007%2F978-1-84882-497-3 – ЭБС Springer eBooks (Complete Collection 2009).
2. Reisig, Wolfgang. Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies / Wolfgang Reisig. – Springer-Verlag, 2013. – URL: https://link.springer.com/book/10.1007%2F978-3-642-33278-4 – ЭБС Springer eBooks (Complete Collection 2013).
3. Jensen, Kurt. Coloured Petri Nets: Modelling and Validation of Concurrent Systems / Kurt Jensen, Lars M. Kristensen. – Springer-Verlag, 2009. – URL: https://link.springer.com/book/10.1007%2Fb95112 – ЭБС Springer eBooks (Complete Collection 2009).
4. Baeten J.C.M. Process Algebra: Equational Theories of Communicating Processes / J.C.M. Baeten, T. Basten, M.A. Reniers. – Cambridge University Press, 2010. – URL: http://ebookcentral.proquest.com/lib/hselibrary-ebooks/detail.action?docID=501280 – ЭБС Ebrary Purchased Textbooks.
5. Fisher, Michael. An Introduction to Practical Formal Methods Using Temporal Logic / Michael Fisher. – John Wiley & Sons, 2011. – URL: https://library.books24x7.com/toc.aspx?bookid=46822 – ЭБС Books 24x7 IT Pro Collection.
6. Milner, Robin. A Calculus of Communicating Systems / Robin Milner. – Springer-Verlag, 1980. – URL: https://link.springer.com/book/10.1007/3-540-10235-3 – ЭБС SpringerLink.
7. Baier, Christel. Principles of Model Checking / Christel Baier, Joost-Peter Katoen, Kim Guldstrand Larsen. – MIT Press, 2008. – URL: https://ebookcentral.proquest.com/lib/hselibrary-ebooks/detail.action?docID=3338793 – ЭБС ProQuest Ebook Central - Academic Complete.

a) Optional

1. Fokkink, Wan. Modelling Distributed Systems / Wan Fokkink. – Springer-Verlag, 2007. – URL: https://link.springer.com/book/10.1007%2F978-3-540-73938-8 – ЭБС Springer eBooks (Complete Collection 2007).
2. Nielson, Hanne Riis. Semantics with Applications: An Appetizer / Hanne Riis Nielson, Flemming Nielson. – Springer-Verlag, 2007. – URL: https://link.springer.com/book/10.1007%2F978-1-84628-692-6 – ЭБС Springer eBooks (Complete Collection 2007).
3. Ben-Ari, Mordechai. Principles of the Spin Model Checker / Mordechai Ben-Ari. – Springer-Verlag, 2008. – URL: https://link.springer.com/book/10.1007%2F978-1-84628-770-1 – ЭБС Springer eBooks (Complete Collection 2008).
4. Van der Aalst, W. Workflow Management: Models, Methods, and Systems / Wil van der Aalst, Kees van Hee. – The MIT Press, 2002. – URL:

https://library.books24x7.com/toc.aspx?bookid=7954 – ЭБС Books 24x7 Business Pro Collection.

6. Grading System

The evaluation is based on the ten-point scale:

10-point scale	5-point scale
10	Excellent
9	Excellent
8	Excellent
7	Good
6	Good
5	Satisfactory
4	Satisfactory
3	Fail
2	Fail
1	Fail

Student classwork is evaluated via quick tests and assignments given in the classroom. Each assignment is weighted in points. The amount of assignment points depends on the assignment complexity. The points of an assignment are given in the assignment description. Practical assignments are expected to be solved and submitted during the seminar on the day of their distribution.

These assignments form grades O_{accum1} and $O_{current2}$ used in the formulae below. Evaluation criteria are as follows:

- completeness of the solution (if all potential problems with correctness and performance are taken into account);
- analysis of the suggested solution (recognition of shortages and benefits of the suggested solution; diagnosis of the solution performance bottlenecks, or explanation, why the solution is free of them);
- argumentation of the suggested solution correctness;
- correct answers to theory-check questions.

In addition, an instructor evaluates proactive attitude of students in a class:

- proactive attitude of a student in solving offered assignments:
- suggesting alternative solutions and comparison of them with the submitted solution;
- demonstrating erudition in the field of study (in-depth knowledge beyond the bounds of the course);
- demonstrating erudition in adjacent fields of knowledge;
- ability to find defects in the submitted solution;
- be fluent in applying learned methods and algorithms.

Grades for practical and self-study work are written down in a worksheet. Cumulative grade for practical work or self-study is calculated at the end of each module before intermediate or final control.

The result grade after the 1st term is calculated by the formula:

$$O_{term\ 1} = 0{,}5 \cdot O_{accum1} + 0{,}5 \cdot O_{exam1},$$

where O_{accum1} is the accumulated grade composed of grades for the current work (quick tests, assignments, work in the classroom) during modules 1 and 2; O_{exam1} is the grade for the intermediate exam. Rounding is done by "round half up" rule.

The final course grade is calculated by the following formulae:

$$O_{accum2} = 0{,}5 \cdot O_{current2} + 0{,}3 \cdot O_{h\_task} + 0{,}2 \cdot O_{term1} \text{ and}$$

$$O_{final} = 0{,}6 \cdot O_{accum2} + 0{,}4 \cdot O_{exam2},$$

where $O_{current2}$ is the grade for the current work (quick tests, assignments, work in the classroom) during modules 3 and 4; O_{h_task} is the grade for the home task, O_{term1} is the result grade after the 1st term, and O_{exam2} is the grade for the final exam after module 4. Rounding is done by "round half up" rule.

Control forms are as follows:

Type of control	Control form	1 year				Settings
		1	2	3	4	
Intermediate	Exam		*			Computer-based written test; 120 minutes
Mid-term (week)	Home assignment				*	Written report
Final	Exam				*	Computer-based written test; 120 minutes

Students may be exempted from the intermediate exam (the grade O_{exam1} will be set equal to the grade O_{accum1}) if the following requirements are met:

- the grade O_{accum1} is greater or equal to 8;
- no more than three seminar classes were missed in the 1st term.

Students may be exempted from the final exam (the grade O_{exam2} will be set equal to the grade O_{accum2}) if the following requirements are met:

- the grades $O_{current2}$, O_{h_task}, O_{term1} are greater or equal to 8;
- no more than four seminar classes were missed in the 2nd term.

7. Examination Type

Exam, intermediate control (module 2):

A 120-minute computer-based test covering topics of the first term.

Students should demonstrate:

- understanding of the basic formalisms and notions learnt in first two modules (communicating finite automata, Petri nets, coloured Petri nets, process algebra CCS);
- skills of modeling and analysis (reachability graph, coverability graph, S- and T- invariants, traps, siphons, strong/weak bisimulation, etc.) of distributed and parallel systems with the studied formalisms and algorithms.

Home assignment, mid-term control (module 4):

The home task deals with constructing a formal model and verifying it. Given a concrete distributed system (a communication protocol, a system of interacting agents, a resource producing/consuming system etc.) students should complete the following tasks:

- develop a model for a given distributed system;
- describe basic behavioral properties of the constructed model;
- classify the behavioral properties and chose appropriate methods and/or tools for specifying and verifying these properties;
- verify the behavior of the constructed system;
- prepare a concise written report.

Students should demonstrate:

- skills of modeling complex distributed systems in PROMELA modeling language;
- skills of using SPIN verification system to debug and conduct model checking of models constructed in PROMELA;
- ability to argument the suggested solution;
- ability to analyze advantages and disadvantages of the proposed solution;
- ability to propose further improvements of the solution;
- ability to find alternative solutions of the given assignment.

Evaluation criteria are as follows:

- correctness of the suggested solution;
- completeness of the solution (whether all of the possible problems concerning correct-ness and performance are taken into account);
- analysis of the suggested solution (recognition of shortages and benefits of the suggest-ed solution; diagnosis of the solution performance bottlenecks, or explanation, why the solution is free of them);
- argumentation of the suggested solution correctness;
- suggested alternative solutions and comparison of them with the submitted solution;
- accuracy of the presented report.

When submitting an assignment, students are expected to answer questions in class to demonstrate understanding of the content of the assignment and course material, to present and explain their own solutions, to answer questions. If a student is not able to answer or argue the question, then the grade may be reduced. After an assignment is graded, the topic is discussed in class. The assignment explanation takes 10-15 minutes of the class time.

Exam, final control (module 4):

A 120-minute computer-based test covering topics of the course:

1. Operational, denotational and axiomatic semantics of sequential program.
2. The least fixpoint semantics of loop statement.
3. Verification of sequential programs with partial and total correctness assertions.
4. Interleaving semantics of concurrent programs.
5. Labeled transition systems.
6. Formal models of concurrent and distributed systems.
7. Theory of process algebras.
8. Branching time semantics of concurrent processes.
9. Trace and bisimulation equivalence of concurrent programs. Strong and weak bisimulation.
10. Hennessy-Milner logic for process algebra CCS.
11. Process algebra CSP.
12. Petri net theory.
13. Interleaving and concurrent semantics for Petri nets.
14. Structural properties of Petri nets.
15. Classification of Petri nets.
16. Expressibility of Petri nets.
17. Proving Petri nets properties with reachability and coverability trees.
18. Temporal logics for specification of concurrent systems behavior.
19. Syntax, semantics and equational laws of Linear Temporal Logic LTL.
20. Syntax, semantics and equational laws of Computational Tree Logic CTL.
21. Comparing LTL and CTL expressibility.
22. Automata on infinite words and ω-regular languages.
23. Model checking of LTL and CTL formulae.

8. Methods of Instruction

The following educational technologies are used:

- *lectures*: presentation of the course material and demonstration of the practical value;
- *problem solving* and *case studies*: students do practical assignments in a classroom.

- *classroom discussion*: all students may interact with instructors by asking questions concerning the material of this course.

An official mean of communicating with students is e-mail. Students can ask their questions about assignments and theoretical issues in classrooms, as well as by e-mail.

9. Special Equipment and Software Support (if required)

The following hardware is necessary for the course:

- projector for demonstrating lecture slides;
- personal computer for instructors;
- personal computers for students with installed Java Runtime Environment (free software, accessible via: https://java.com/en/download/).

The following free software is used during the educational process:

- **The Edinburgh Concurrency Workbench** (CWB) – a tool for describing, exploring and automatically verifying systems. Accessible via: http://homepages.inf.ed.ac.uk/perdita/cwb/ .
- **CPN Tools** – a tool for editing, simulating, and analyzing Colored Petri Nets. Accessible via: http://cpntools.org/category/downloads/ .
- **Spin** – a tool for the formal verification of multi-threaded software applications. Accessible via: http://spinroot.com/spin/Man/README.html .