Formalna analiza koncepata i kombinatorno testiranje za automatiziranu provjeru znanja u sustavima za e-učenje

Škopljanac-Mačina, Frano

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Frano Škopljanac-Mačina

FORMALNA ANALIZA KONCEPATA I KOMBINATORNO TESTIRANJE ZA AUTOMATIZIRANU PROVJERU ZNANJA U SUSTAVIMA ZA E-UČENJE

DOKTORSKI RAD

Zagreb, 2023.

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Frano Škopljanac-Mačina

FORMALNA ANALIZA KONCEPATA I KOMBINATORNO TESTIRANJE ZA AUTOMATIZIRANU PROVJERU ZNANJA U SUSTAVIMA ZA E-UČENJE

DOKTORSKI RAD

Mentor: Izv. prof. dr. sc. Bruno Blašković

Zagreb, 2023.

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Frano Škopljanac-Mačina

FORMAL CONCEPT ANALYSIS AND COMBINATORIAL TESTING FOR AUTOMATED ASSESSMENT IN E-LEARNING SYSTEMS

DOCTORAL THESIS

Supervisor: Associate Professor Bruno Blašković, PhD

Zagreb, 2023

Doktorski rad izrađen je na Sveučilištu u Zagrebu Fakultetu elektrotehnike i računarstva, na Zavodu za osnove elektrotehnike i električka mjerenja.

Mentor: izv. prof. dr. sc. Bruno Blašković

Doktorski rad ima: 206 stranica

Doktorski rad br.: _____

O mentoru

Bruno Blašković je diplomirao (1982.), magistrirao (1985.) i doktorirao (1996.) na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu.
Od prosinca 1986. zaposlen je na Zavodu za osnove elektrotehnike i električka mjerenja. U prosincu 2003. izabran je u zvanje izvanrednog profesora. Objavio je preko 60 radova u časopisima i na konferencijama. Područje rada obuhvaća sintezu protokola, transformacije modela, formalne metode, testiranje programske potpore, provjeru modela, SMT i pouzdanost mreže. Član je više programskih odbora konferencija kao i urednik područja u časopisu.

About the Supervisor

Bruno Blašković received B.Sc., M.Sc. and Ph.D. degrees in electrical engineering from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER), Zagreb, Croatia, in 1982, 1985 and 1996, respectively.

From December 1986 he is working at the Department of Electrical Engineering Fundamentals and Measurements at FER. In December 2003 he was promoted to associate professor. He published over 60 papers in journals and conference proceedings. His current research interests are in the field of protocol synthesis, model transformations, business process modeling, formal methods, software testing, model checking, SMT and network reliability. Professor Blašković is also a member of several international professional associations. He participated in conference international programs committees, and he serves as an associate editor and technical reviewer for international journals.

Zahvale

Zahvaljujem svojem mentoru izv. prof. dr. sc. Bruni Blaškoviću na tome što me je uveo u područje formalnih metoda, kao i na svom prenesenom znanju i izdvojenom vremenu te na mnogim vrijednim sugestijama kroz doktorski studij.

Zahvaljujem svim kolegama sa Zavoda za osnove elektrotehnike i električka mjerenja na FER-u na poticajnoj i motivirajućoj radnoj atmosferi. Isto tako, hvala doc. dr. sc. Ivoni Zakariji sa Sveučilišta u Dubrovniku na odličnoj suradnji u znanstveno-istraživačkom radu. Posebnu zahvalu upućujem i prof. dr. sc. Zoranu Skočiru na svim savjetima i na iskazanoj podršci za moj znanstveni i stručni rad.

Hvala mojim roditeljima i cijeloj obitelji koji su me uvijek podupirali tijekom ovoga studija.

I od srca hvala mojoj supruzi Mirti na velikom strpljenju, razumijevanju i ljubavi te na stalnoj inspiraciji i motivaciji!

Sažetak

Tema ove disertacije je istraživanje primjene metode formalne analize koncepata i tehnike kombinatornog testiranja za automatiziranje pripreme i odabira pitanja za provjere znanja u sustavima za e-učenje. Nakon teorijskog pregleda svih korištenih metoda i tehnika predlaže se novi model za automatiziranu provjeru znanja. Na temelju skupa ispitnih pitanja označenih s definiranim atributima gradi se primjenom metode za formalnu analizu koncepata formalni kontekst, binarna matrica u kojoj je zapisano koje atribute ima svako pitanje u odabranom skupu. Potom se automatiziranom metodom kombinatornog testiranja generira gotovo minimalan broj testnih slučajeva opisanih s definiranim atributima iz formalnog konteksta tako da je svaka n-torka atributa zadane veličine n pokrivena s barem jednim testnim slučajem odnosno opisom pitanja. Nakon toga se za svaki generirani testni slučaj automatski traže odgovarajuća ispitna pitanja. Kada se pronađe sažeti skup pitanja koji pokriva sve generirane testne slučajeve prelazi se na izgradnju njegove konceptualne rešetke primjenom metode formalne analize koncepata. Konceptualna rešetka predstavlja ontologiju dijela nastavnog gradiva opisanog s odabranim sažetim skupom ispitnih pitanja. Struktura konceptualne rešetke je parcijalno uređeni skup formalnih koncepata sa zadanom relacijom poretka *natkoncept-potkoncept*, a svaki formalni koncept je par povezanih skupova pitanja i atributa. Potom se konceptualna rešetka automatski topološki sortira kako bi se iz nje dobio odgovarajući potpuno uređeni skup formalnih koncepata u kojem su sačuvani svi odnosi *natkoncept-potkoncept*. Iz generiranih linearnih poredaka formalnih koncepata se potom automatski izdvajaju prikladni nizovi pitanja za formativnu provjeru znanja u sustavu za e-učenje. U vlastitom sustavu za e-učenje je implementiran automatizirani modul za pripremu i vođenje formativne provjere znanja koji studente vodi po pripremljenim putevima rješavanja provjere i po potrebi nudi pomoć u obliku interaktivnih nastavnih materijala. Na kraju se provodi verifikacija predložene metode automatizirane provjere znanja formalnom metodom provjere modela. Za tu svrhu je predložen i implementiran prototipni sustav koji s nadograđenim L^* algoritmom za učenje deterministčkih konačnih automata i uz pomoć alata za provjeru modela *Spin* automatski otkriva model formativne provjere znanja. Potom se otkriveni model može formalno verificirati i simulirati s alatom *Spin*. U radu su prikazani rezultati svih predloženih metoda nad studijskim primjerom od 473 ispitna pitanja označena s 50 atributa korištenih na predmetu *Osnove elektrotehnike*. Na kraju su prikazani i rezultati verifikacije i simulacije predložene metode za automatiziranu provjeru znanja.

Ključne riječi: formalna analiza koncepata (FCA), kombinatorno testiranje, topološko sortiranje, formalna metoda provjere modela, *Spin*/*Promela*, L^* algoritam, automatizirana provjera znanja, oblikovanje sustava za e-učenje

Formal concept analysis and combinatorial testing for automated assessment in e-learning systems

Introduction

The topic of this dissertation is the research on the application of the Formal concept analysis method and the combinatorial testing technique for automating preparation and selection of questions for assessments in e-learning systems. After a theoretical review of all methods and techniques used in this dissertation, a new model for automated formative assessment is proposed. Based on a set of test questions annotated with defined attributes, a formal context is built using the formal concept analysis method. A formal context holds data about the list of attributes for each of the test questions. Then, the automated method of combinatorial testing will generate an almost minimal number of test cases (i.e. test question descriptions) described with defined attributes from the formal context. In order to achieve that first we need to group defined attributes into disjoint sets, and after the combinatorial testing process each n-tuple of attributes of the given size n from disjoint sets will be covered by at least one generated test case or a test question description. In the next step an automated search is performed that finds all the test questions from the e-learning system database that correspond to the generated test cases. When a concise set of test questions is found that covers all the generated test cases, its concept lattice can be built using the method of formal concept analysis. The concept lattice represents the ontology of a part of the teaching material described with a selected concise set of test questions. Then the concept lattice is automatically topologically sorted, so that the corresponding totally ordered set of its nodes (formal concepts) is obtained. From the generated linear ordering of formal concepts, appropriate sets of questions for formative assessments in e-learning systems are then automatically extracted. An automated module for the preparation and management of formative assessments is implemented in our in-house e-learning system, which guides students along the prepared assessment solving paths and, if necessary, offers them help in a form of various interactive teaching materials. Furthermore, the verification of the proposed automated assessment method is carried out using the formal method of model checking. For this purpose, a prototype system was proposed and implemented which, with the upgraded L^* algorithm for learning deterministic finite automata and with the help of the *Spin* model checking tool, automatically builds a model of the formative assessment process. Then the discovered model can be formally verified and simulated with the *Spin* tool. The paper presents the results of all proposed methods on a study example of 473 exam questions annotated with 50 attributes used in the course *Fundamentals of Electrical Engineering* at University of Zagreb, Faculty of Electrical Engineering and Computing. Finally, the dissertation presents verification and simulation results of the proposed method for automated assessment in e-learning

systems.

Thesis outline

The work is divided into eight chapters, and first of them is the Introduction that states the research problem, proposed solution and the scientific contribution of the dissertation. Second chapter provides an overview of relevant works related to the topics and areas of this research. Also, this chapter gives a brief review of selected own previous research papers written together with different co-authors.

In the third chapter, a detailed overview of the theoretical foundations of this dissertation is presented. First of all, we discuss the method of formal concept analysis (FCA) and all of its features, such as the formal context, the formal concept, the concept lattice, and attribute implications and association rules. Formal context is a binary matrix of objects from selected domain and their defined attributes. Formal concept is a pair of sets of objects and their shared attributes. Concept lattice represents an ontology of the data from the formal context. It is a partially ordered set of all formal concepts identified from the formal context. Concept lattice is ordered using *superconcept-subconcept* relation between formal concepts and it can be visualized as a directed acyclic graph or a line diagram where each node is a formal concept and each directed edge represents a *superconcept-subconcept* relation. Also, based on the formal context we can automatically infer all the association rules between attributes that hold for the objects in the formal context, as well as their more strict variant of attribute implications. Various own examples are given that illustrate the most important features of the FCA method using FCA tools such as *Concept Explorer 1.3* and *Lattice Miner 2.0*. Then, the basic mathematical foundations of the FCA method are considered, such as partial ordered sets, lattices, complete lattices, Galois connections and closure operators. Also, we give an overview of techniques for compacting very large formal contexts or dense concept lattices and describe several selected extensions of the FCA method. Furthermore, the third chapter formally describes the combinatorial testing procedure based on orthogonal fields and covering fields. Combinatorial testing is a software testing method for minimizing the number of test cases while ensuring that almost all faults are detected. This is achieved with covering each pair or triple of values of different input parameters with at least one generated test case. The process of combinatorial testing is illustrated with various examples, and at the end, software tools for combinatorial testing are discussed, primarily the *Jenny* tool that was used later in the research. Next, third chapter deals with the topic of topological sorting of partially ordered sets and shows how this procedure was applied to topologically sort concept lattices. The next topic dealt with in the third chapter is the learning of finite automata. After the introductory part with formal definitions of transition systems, labeled transition systems, deterministic and non-deterministic finite automata and a brief overview of Büchi automata, we move on to a detailed review of Dana Angluin's L^* algo-

rithm for learning deterministic finite automata (DFA), illustrated with our own example. At the end, some more important upgrades of the L^* algorithm from the literature are listed. The third chapter concludes with the presentation of the last topic in this research, which is the formal method of model checking. This method is primarily used for formal verification of software systems, and after reviewing the topic of software verification, a more detailed review of the model checking method is given, including linear temporal logic which is used for specifying the properties that the model of the system must satisfy. In doing so, some common formulas of linear temporal logic are presented through various examples. After that, we describe the model checking tool *Spin* and its *Promela* language. Various examples are given that show how model checking tool *Spin* is used to automatically simulate and verify process models written in *Promela*.

In the fourth chapter, the proposed system model for automated assessment is presented. After annotating the exam questions with attributes and preparing the initial formal context with the FCA method, a combinatorial testing procedure is carried out on the data in a formal context. The goal of combinatorial testing is to generate an almost minimal set of questions for formative testing, while covering all allowed pairs or triples of defined attributes. After the generation of a corresponding concise set of questions we automatically build its final formal context and the FCA method generates its concept lattice. This is as an ontology of the given teaching material from this concise annotated set of questions. Then the concept lattice is automatically topologically sorted to obtain a linear order of formal concepts, ordered from the most general to the most specific formal concept. From such a list of formal concepts, system automatically extracts question sequences also ordered from those more general to those more specific. The generated question sequences will be used to automatically create formative assesments in our own e-learning system.

And in the next, fifth chapter, the topic of combinatorial testing used for preparing concise sets of questions for formative assessment is covered in greater detail. Different strategies for selecting the number and values (attributes) of each input parameter for the combinatorial testing procedure are discussed and the proposed combinatorial testing procedure is demonstrated on a study example of a large number of 473 annotated questions from the database of our own e-learning system. After presenting results of initial combinatorial testing runs using various strategies, a complete combinatorial testing process is performed using most appropriate strategy that uses four parameters (disjoint sets of attribute values), forbidden attribute combinations and introduces the concept of feature groups that group similar attributes together. After this, procedure generated 72 test cases that cover all allowed pairs of different parameter values. We needed to add 49 new questions to cover each generated test cases (questions descriptions) with at least one question. Finally, a concise set of 139 questions is found that covers all the 72 generated test cases. All the test cases are presented, as well as an example of a new ques-

tion that implements one previously uncovered test case. Final formal context is then built that includes all identified 139 questions.

The sixth chapter continues with the preparation of questions for formative assessments. At the beginning, the concept lattice of the expanded initial set of 522 questions and the concise set of 139 questions are analyzed in detail. Then a method is proposed for automated topological sorting of the concept lattice of the final formal context and generation of all complete and partial sequences of questions for formative assessments. After that, we present results of different sequences of questions from the concise set of questions obtained by combinatorial testing. At the end, various examples of formative assessment runs in our e-learning system are presented, as well as examples of interactive teaching materials that are used to help students learn and revise topics from the *Fundamentals of Electrical Engineering* course.

In the seventh chapter, the verification procedure of the proposed method for automated formative assessment is carried out. To achieve this a upgraded L^* algorithm with the new *Teacher* algorithm is proposed and implemented in *Python*, which will automate the discovery of a deterministic finite automaton with the help of the *Spin* model checking tool. After the presentation of the algorithm and its implementation, the results of the verification and simulation of the automated formative assessment model are presented. After that, the results of the verification of the shorter assessment with a smaller number of questions are presented first, and finally the results of the verification of the larger formative assessment obtained in the sixth chapter are presented and discussed. In this case a set of 888 partial and complete question sequences is used to automatically discover a deterministic finite automaton that accepts all words that correspond to the 888 question sequences. The discovered DFA is automatically transformed form textual *.dot* format to the *Promela* process model and various specifications are automatically added to the model using `never` claims or `assert` statements. Additional *Python* script automates the simulation, verification and visualisation of the discovered *Promela* models. Using verification it is shown that the models accept only those questions sequences which are generated in sixth chapter. Also, by simulation of automatically expanded *Promela* simulation models that include jumps to previous questions in case of wrong answers we can obtain different scenarios of formative assessment process.

And the final, eight chapter provides conclusions about the conducted research and scientific contributions, as well as guidelines for future research work.

Contribution

In this dissertation, a new method was developed for combinatorial testing of an annotated set of exam questions, which generates a concise set of questions that cover all allowed pairs or triplets of defined attributes that describe teaching materials. The dissertation shows how an ontology of domain knowledge in the form of a concept lattice can be formally built from a set of annotated

exam questions using the method of formal concept analysis. A method for automatic selection of question sequences based on topological sorting of formal concepts in a concept lattice is proposed and a procedure for automated formative assessment is implemented that uses those generated question series. In the end, a system prototype was built for the verification of the proposed method for automated assessment based on the upgraded L^* algorithm for learning deterministic finite automata and on the formal method of model checking. The goal of the research was achieved and the following main original contributions were made:

- Formal description of a machine learning method for building an ontology of domain knowledge in a form of a concept lattice based on a concise set of semantically annotated exam questions.
- A method for automated assessment based on the automatic selection of test questions sequences from the concept lattice.
- Prototype system for the verification of the automated assessment method based on the formal method of model checking.

In addition to these main original contributions, an upgraded L^* algorithm and a new *Teacher* algorithm for learning deterministic finite automata with the help of the *Spin* model checking tool is proposed and implemented. The discovered finite automata are then automatically transformed into process models in *Promeli* which can be then simulated and verified with the *Spin* tool.

Conclusion and future research directions

In this dissertation, a method for the automated preparation and management of formative assessments in e-learning systems is proposed. The main prerequisite for the application of the proposed method is the definition of a set of attributes that describes the given teaching material and the preparation of a selected set of test questions annotated with such attributes. Then, an initial formal context is built, i.e. a binary matrix in which the rows indicate the selected test questions, and the columns indicate the defined attributes, and the attributes of each test question can be read from the contents of that binary matrix. After that, the proposed method of combinatorial testing is carried out, which automatically identifies a concise subset of all selected test questions that, among their attributes, cover all allowed pairs or triples of attributes from the formal context. An appropriate final formal context is built from the found concise subset of exam questions, and by applying the method of formal concepts analysis, its concept lattice is automatically built. The concept lattice represents the ontology of the given teaching material described with annotated questions. By applying topological sorting on the concept lattice, a totally ordered set of formal concepts is obtained in which all relations *superconcept-subconcept* are preserved. Furthermore, from the totally ordered set we can automatically extract questions sequences that start with most general questions and end with most specific questions,

which will be used for formative assessments in our in-house e-learning system. A procedure for performing formative assessment that guides students through selected questions sequences by selecting the next question based on the answer to the previous question is also proposed. The use of interactive teaching materials was suggested as an useful aid in solving the formative assessment. In the end, the verification of the proposed method for automated assessment was carried out using a formal method of model checking. Formative assessment model in the form of a deterministic finite automaton are automatically discovered using the upgraded Dana Angluin's *L** algorithm and the *Spin* model checking tool. The generated models of the assessment process are automatically transformed into corresponding process models in the *Promela* language, and then they were simulated and verified with the *Spin* tool.

Proposed method for automated preparation and management of formative assessments can help teachers and other domain experts in selection of questions and composing new questions, and such a system can provide students with additional help in the process of learning and revising. And in the future work, the results of the formative assessment simulations will be compared using process mining techniques with the real data from the log records of formative assessments attempts in the e-learning system.

Key words: formal concept analysis (FCA), combinatorial testing, topological sorting, formal method of model checking, *Spin*/*Promela*, *L** algorithm, automated assessment, e-learning system design

Sadržaj

Poglavlje 1

Uvod

U ovom doktorskom istraživanju zadatak je pronaći i primijeniti nove metode za automatiziranje provjera znanja u sustavima za e-učenje. Naglasak je na formativnim provjerama znanja koje bi pomogle studentima u usvajanju i vježbanju novog gradiva, poglavito u inženjerskoj edukaciji. Kako bi se ostvario ovaj cilj kombinirat će se različiti pristupi, od metoda strojnog učenja poput formalne analize koncepata, preko metode kombinatornog testiranja koja se koristi u testiranju programske potpore, do formalne metode provjere modela koja se koristi za formalnu verifikaciju komunikacijskih protokola i programske potpore te L^* algoritma za otkrivanje deterministčkih konačnih automata.

S primjenom kombinatornog testiranja osigurat će se da skup pitanja odabran za formativnu provjeru znanja pokriva sve dozvoljene parove ili trojke definiranih atributa koji opisuju nastavne teme ili sadržaje. Dakle, naglasak će biti na pronalasku pitanja koje povezuju različite dijelove gradiva. Time se nastoji pomoći studentima u lakšem povezivanju i usvajanju gradiva.

Metoda formalne analize koncepata će se iskoristiti za stvaranje ontologije domenskoga znanja – dijela gradiva opisanog s ispitnim pitanjima. Iz ontologije u obliku konceptualne rešetke – parcijalno uređenog skupa formalnih koncepata s relacijom poretka *natkoncept-potkoncept* se primjenom topološkog sortiranja dobije linearni poredak formalnih koncepata, odnosno parova povezanih skupova pitanja i atributa. Iz tog linearnog poretka formalnih koncepata od onih općenitijih do onih specifičnijih automatski će se izdvajati odgovarajući nizovi pitanja za provjeru. S tim generiranim nizovima pitanja će se automatski provoditi formativna provjera znanja u sustavu za e-učenje. Na temelju ocjene svakog odgovora sustav će studenta voditi po odabranom putu rješavanja provjere. U slučaju krivog odgovora sustav će studenta vratiti na prethodno pitanje u nizu, a može mu ponuditi i pomoć u obliku različitih interaktivnih nastavnih materijala.

Na kraju se provodi formalna verifikacija predložene metode za automatiziranu provjeru znanja. U radu se predložio i implementirao prototipni sustav kojim se može automatski otkriti model procesa provjere znanja korištenjem nadograđenog L^* algoritma i alata za provjeru

modela *Spin*. A nakon što se pronađe odgovarajući model procesa provjere znanja on će se simulirati i formalno verificirati s alatom *Spin*.

1.1 Cilj i hipoteze istraživanja

Cilj ovoga istraživanja je opisati postupak za izgradnju ontologije domenskog znanja, predložiti i realizirati novu metodu za automatiziranu provjeru znanja, te izraditi model za verifikaciju i primijeniti ga za vrednovanje predložene metode.

Ovo su hipoteze koje su bile postavljene prije početka rada na disertaciji:

- Mogu će je izgraditi ontologiju domenskog znanja kroz semantičko označavanje ispitnih pitanja, primjenom metode kombinatornog testiranja za stvaranje sažetog skupa ispitnih pitanja, te primjenom metode strojnog učenja, formalne analize koncepata.
- Na temelju izra đene ontologije domenskog znanja moguće je razviti novu metodu za automatiziranu provjeru znanja kroz automatsko stvaranje nizova ispitnih pitanja.
- Mogu će je izgraditi izvorni model za verifikaciju rezultata predloženog istraživanja zasnovan na formalnoj metodi provjere modela.

1.2 Doprinosi

U ovoj disertaciji razvijena je nova metoda za kombinatorno testiranje označenog skupa ispitnih pitanja kojom se generira sažeti skup pitanja koja pokrivaju sve dozvoljene parove ili trojke definiranih atributa kojima se opisuju nastavne teme i sadržaji. U disertaciji je prikazano kako se može formalan način iz skupa označenih ispitnih pitanja izgraditi ontologija domenskog znanja u obliku konceptualne rešetke. Predložena je metoda za automatski odabir nizova pitanja temeljen na topološkom sortiranju formalnih koncepata u konceptualnoj rešetci te je implementiran postupak za automatiziranu formativnu provjeru znanja koja će se odvijati po tim generiranim nizovima pitanja. Na kraju je izgrađen prototip sustava za verifikaciju predložene metode za automatiziranu provjeru znanja utemeljen na nadograđenom L^* algoritmu za učenje determinističkih konačnih automata te na formalnoj metodi provjere modela.

U skladu s postavljenim hipotezama postignut je cilj istraživanja te su ostvareni sljedeći glavni izvorni doprinosi:

- Formalni opis metode strojnog u čenja za izgradnju ontologije domenskog znanja u obliku konceptualne rešetke na temelju sažetog skupa semantički označenih ispitnih pitanja.
- Metoda za automatiziranu provjeru znanja utemeljena na automatskom odabiru nizova ispitnih pitanja iz konceptualne rešetke.
- Prototip sustava za verifikaciju metode za automatiziranu provjeru znanja utemeljen na formalnoj metodi provjere modela.

Uz ove glavne izvorne doprinose predložen je i implementiran nadograđeni L^* algoritam kao i novi algoritam *Učitelja* za učenje deterministčkih konačnih automata uz pomoć alata za provjeru modela *Spin*. Pronađeni konačni automati su potom automatski prevedeni u procesne modele u *Promeli* koje se potom može simulirati i verificirati s alatom *Spin*.

1.3 Struktura disertacije

Rad je podijeljen na osam poglavlja: nakon Uvoda u drugom poglavlju se donosi pregled područja i povezanih istraživanja. U tom poglavlju daje se pregled relevantnih radova vezanih uz teme i područja ovoga istraživanja. Isto tako u drugom poglavlju se daje osvrt i na prethodna istraživanja provedena sa suradnicima.

U trećem poglavlju izložen je detaljni pregled teorijskih osnova ovoga istraživanja. Na početku se obrađuje metoda formalne analize koncepata (FCA) i svih njenih značajki poput formalnog konteksta, formalnog koncepta, konceptualne rešetke te implikacija atributa i asocijacijskih pravila. Pritom se daju vlastiti primjeri kojima se ilustriraju najvažnije značajke ove metode. Potom se razmatraju osnovni matematički temelji metode formalne analize koncepata, a na kraju se opisuju postupci za sažimanje podataka u formalnim kontekstima i konceptualnim rešetkama te nekoliko izabranih proširenja ove metode.

Nadalje, u trećem poglavlju se formalno opisuje postupak kombinatornog testiranja utemeljen na ortogonalnim poljima i prekrivajućim poljima. Postupak kombinatornog testiranja ilustrira se s različitim primjerima, a na kraju se razmatraju programski alati za kombinatorno testiranje, prije svega alat *Jenny* koji se koristio kasnije u istraživanju.

U nastavku trećeg poglavlja obrađuje se tema topološkog sortiranja parcijalno uređenih skupova i pokazuje se kako se ovaj postupak primjenjivao u ovom istraživanju.

Sljedeća tema kojom se bavi treće poglavlje je učenje konačnih automata. Nakon uvodnog dijela s formalnim definicijama sustava s prijelazima, labeliranog sustava s prijelazima, deterministčkih i nedeterministčkih konačnih automata te kratkog osvrta na Büchijeve automate prelazi se na detaljan pregled L^* algoritma Dane Angluin za učenje deterministčkih konačnih automata ilustriran s vlastitim primjerom. Na kraju se navode neke važnije nadogradnje L^* algoritma iz literature.

Treće poglavlje se zaključuje s izlaganjem zadnje teme u ovom istraživanju, a to je formalna metoda provjere modela. Ova metoda se koristi za formalnu verifikaciju prvenstveno softverskih sustava, a nakon pregleda teme verifikacije programske potpore slijedi podrobniji pregled metode provjere modela i linearne temporalne logike koja se koristi za zadavanje svojstava koje model sustava mora zadovoljiti. Pritom se kroz različite primjere prikazuju neke česte formule linearne temporalne logike. Nakon toga se uz više vlastitih primjera detaljnije opisuje alat za provjeru modela *Spin* te jezik *Promela* za formalni zapis procesnih modela koji se mogu simu-

lirati i verificirati sa *Spinom*.

U četvrtom poglavlju prikazuje se predloženi model sustava za automatiziranu provjeru znanja. Nakon označavanja ispitnih pitanja i pripreme inicijalnog formalnog konteksta s metodom FCA provodi se nad podacima u formalnom kontekstu postupak kombinatornog testiranja. Cilj kombinatornog testiranja je generiranje što manjeg skupa pitanja za formativnu provjeru, a da su pritom pokriveni svi dozvoljeni parovi ili trojke atributa. Nakon generiranja odgovarajućeg sažetog skupa pitanja provodi se nastavak metode FCA koja će iz tog sažetog označenog skupa pitanja generirati konceptualnu rešetku kao ontologiju zadanog nastavnog gradiva. Potom će se konceptualna rešetka topološki sortirati kako bi se dobio linearni poredak formalnih koncepata od onih najopćenitijih do onih najspecifičnijih. Iz takve liste formalnih koncepata izdvojit će se pitanja poredana na isti način kao i formalni koncepti. Generirani nizovi pitanja koristit će se za automatsko stvaranje formativne provjere znanja u vlastitom sustavu za e-učenje.

A u sljedećem, petom poglavlju je detaljno obrađena tema kombinatornog testiranja u kontekstu pripreme pitanja za formativnu provjeru znanja. Razmatraju se različite strategije za odabir ulaznih parametara za postupak kombinatornog testiranja te se demonstrira predloženi postupak na studijskom primjeru većeg broja označenih pitanja iz baze podataka vlastitog sustava za e-učenje.

Šesto poglavlje opisuje nastavak pripreme pitanja za formativne provjere znanja. Na početku se analiziraju konceptualne rešetke inicijalnog skupa pitanja i sažetoga skupa pitanja. Potom se predlaže metoda za topološko sortiranje konceptualne rešetke te generiranje svih potpunih i djelomičnih nizova pitanja za formativnu provjeru znanja. Nakon toga se predstavljaju rezultati generiranja različitih nizova pitanja iz sažetog skupa pitanja dobivenog kombinatornim testiranjem. Na kraju se prikazuje nekoliko primjera izvođenja formativne provjere znanja u vlastitom sustavu za e-učenje, a pritom se prikazuju i različiti interaktivni nastavni materijali za pomoć studentima pri učenju koji su razvijeni tijekom ovoga istraživanja.

U sedmom poglavlju se provodi postupak verifikacije predložene metode za automatiziranu formativnu provjeru znanja. U tu svrhu predlaže se nadogradnja *L** algoritma s algoritmom *Učitelja* kojim će se automatizirati proces učenja determinističkog konačnog automata uz pomoć alata za provjeru modela *Spin*. Nakon prikaza algoritma i njegove implementacije predstavljaju se rezultati verifikacije i simulacije modela automatizirane formativne provjere znanja. Potom se prvo prikazuju rezultati verifikacije kraće provjere znanja s manjim brojem pitanja, a na kraju se predstavljaju rezultati verifikacije veće formativne provjere znanja dobivene u šestom poglavlju.

Na kraju se u osmom poglavlju donose zaključci o provedenom istraživanju i ostvarenim doprinosima te daju smjernice za budući istraživački rad.

Poglavlje 2

Pregled područja i povezanih istraživanja

Sustavi za e-učenje su u današnjem svijetu vrlo rašireni i koriste se za formalno i neformalno obrazovanje, kao i za cjeloživotno učenje. U formalnom obrazovanju sustavi za e-učenje se koriste kao računalna potpora nastavnim procesima u školama i na fakultetima. Posljednjih nekoliko godina intenzivno su se koristili različiti sustavi za e-učenje jer se u mnogim zemljama nastava u školama i na fakultetima odvijala udaljeno zbog pandemije COVID-19. Sustavi za e-učenje se koriste i za samoobrazovanje, najviše putem golemih otvorenih mrežnih tečajeva (engl. *Massive open online course* ili *MOOC*). Sustavi za e-učenje su našli primjenu i u cjeloživotnom učenju kao efikasan i isplativ način podučavanja i testiranja zaposlenika u velikim tvrtkama i organizacijama.

Autori u [1] su proveli sustavnu analizu glavnih tema znanstvenih članaka vezanih uz polje e-učenja u razdoblju od 2010. do 2018. Zaključili su kako je jedan od glavnih izazova u polju e-učenja stvaranje efikasnog i inovativnog okruženja za učenje koje će se prilagoditi potrebama učenika i njihovim stilovima učenja. Autori su predložili i definiciju e-učenja koja glasi: "*E-učenje je inovativni web sustav utemeljen na digitalnim tehnologijama i drugim oblicima edukacijskih materijala čiji je glavni cilj učenicima osigurati personalizirano, otvoreno, ugodno i interaktivno okruženje za učenje koji podupire i poboljšava nastavni proces*". Kao što je napomenuto u uvodu i u ovom radu je cilj pronalazak i primjena postupaka kojima će se nastojati podupirati nastavni proces i olakšati proces učenja i utvrđivanja novog gradiva.

Kroz dostupnu literaturu autori predlažu različite pristupe u prilagodbi sustava za e-učenje učenicima i studentima. Autori u radu [2] su dali pregled različitih tehnologija za testiranje u sustavima za e-učenje i predložili su svoj pristup adaptivnom testiranju kroz rješenje u oblaku. Detaljni komparativni pregled istraživanja u polju adaptivnih sustava za e-učenje je dan u radu [3], s naglaskom na inteligentne tutorske sustave (engl. *Intelligent tutoring system* ili *ITS*). Inteligentni tutorski sustavi su adaptivni sustavi za e-učenje utemeljeni na znanju (engl. *Knowledge-based system* ili *KBS*) jer obuhvaćaju i domensko znanje, pedagoško znanje te znanje o studentu. Autori tvrde da je prilagodljivo podučavanje ključno za uspješno usvajanje novog gradiva. Nas-

tavni sadržaji se prilagođuju kroz odabire puteva učenja kroz graf aktivnosti učenja, konceptne mape i ontologije. Na kraju autori zaključuju kako u području provjere u sustavima za e-učenje postoje različiti smjerovi istraživanja, ali još nije ostvareno potpuno automatsko generiranje pitanja iz zadanog domenskog znanja.

U zanimljivom ranijem radu [4] ista grupa autora predstavlja sustav za automatizirano generiranje testova kojima se ispituje znanje sadržano u *OWL* ontologiji (engl. *Ontology Web Language*). U radovima [5, 6, 7] Gulwani i sur. istražuju mogućnosti dinamičkog generiranja pitanja u edukaciji u STEM području poput sintetiziranja problemskih zadataka iz geometrije i algebre. Isto tako, autori u radu [8] predstavljaju metodu za automatsko generiranje novih varijanti zadataka iz matematičke analize na temelju početno zadanog zadatka.

Jedna od metoda strojnog učenja za prikaz i obradu domenskog znanja je formalna analiza koncepata (engl. *Formal concept analysis* ili FCA) [9, 10, 11, 12, 13, 14]. U istraživanju će se koristiti za pripremu domenskog znanja te za izgradnju ontologije domenskog znanja iz koje će se pronalaziti nizovi pitanja za provjeru znanja. Metoda FCA iz pripremljenih ulaznih podataka pronalazi i vizualizira sve koncepte iz domenskog znanja kao i njihove međuovisnosti. Pritom se automatski generira ontologija domenskog znanja pod nazivom konceptualna rešetka, a uz to se otkriva i skup svih asocijacijskih pravila među definiranim atributima. Trenutno je FCA predmet različitih istraživanja, a neka od njih bave se primjenom FCA u sustavima za e-učenje [15, 16, 17, 18]. U radovima [19] i [20] autori navode pregled aktualnih trendova i smjerova istraživanja primjene metode FCA. Autori u zanimljivom radu [21] navode mogućnost korištenja metode FCA za automatsko generiranje domenskog znanja iz teksta.

U ovom istraživanju će se primjenjivati i metode iz područja testiranja programske potpore. Jedno od njih je kombinatorno testiranja, metoda testiranja programske potpore kojom se želi smanjiti broj testova koje treba provesti, a da se pritom osigura otkrivanje velike većine kvarova [22, 23, 24, 25]. Kuhn i sur. su u radu [26] analizirali greške u programskim sustavima i primijetili su da se gotovo sve greške događaju prilikom interakcije najviše 4 do 6 različitih parametara. Također su zaključili da je do 90% svih grešaka u programskim sustavima uzrokovano jednim parametrom ili interakcijom dva različita parametra. Stoga je metoda kombinatornog testiranja usredotočena na interakcije između ulaznih parametara programskog sustava, a cilj metode je generirati gotovo minimalan broj testnih slučajeva, a da je pritom svaka n-torka vrijednosti različitih parametara pokrivena s barem jednim testom. Najčešći oblik kombinatornog testiranja je testiranje parova (engl. *Pairwise testing*) s kojim se želi testovima pokriti sve parove vrijednosti različitih parametara. U području e-učenja autori su koristili kombinatorno testiranje za procjenu uspješnosti podučavanja uz pomoć novih tehnologija poput računarstva u oblaku kao i kod generiranja testova različite kompleksnosti [27, 28]

Još jedna zanimljiva metoda testiranja programske potpore je mutacijsko testiranje s kojim se namjerno unose sitne promjene u originalni programski kod, a koje pritom oponašaju tipične

pogreške programera. Mutacijskim testiranjem treba eliminirati ili minimizirati pojavu da i originalni program i njegov mutant uspješno prođu testiranje s istim testnim podacima [29]. U prethodno provedenim istraživanjima koristili su se principi mutacijskog testiranja kod pripreme računskih zadataka na koje se predaje numerički odgovor kako bi se minimizirala mogućnost da student i krivim ("*mutiranim*") postupkom dođe dovoljno blizu točnoga rješenja pa da mu se prizna odgovor kao točan [30].

Provjera modela (engl. *Model checking*) je jedna od formalnih metoda verifikacije komunikacijskih protokola kao i programske potpore [31]. Provjera modela ispituje zadovoljava li model sustava zadanu specifikaciju. Pritom se koristi linearna temporalna logika (engl. *Linear temporal logic* ili LTL) za provjeru temporalnih svojstava. Programi ili automati mogu se opisati kao modeli procesa u jeziku *Promela*. Takve procesne modele u *Promeli* moguće je potom simulirati i verificirati s alatom za provjeru modela *Spin* [32]. Za vrjednovanje predložene metode automatizirane formativne provjere znanja koristit će se upravo metoda provjere modela i navedeni alat *Spin*.

Sustavi za e-učenje su distribuirani mrežni sustavi i stoga je vrlo zahtjevno temeljito testiranje takvih aplikacija. Autori u radu [33] predstavljaju sveobuhvatan pregled pristupa za analizu i testiranje web aplikacija. Predložili su izgradnju UML modela čitave web aplikacije ili nekog njenog kritičnog modula. Potom se izgrađeni UML model može iskoristiti za poluautomatsko generiranje testnih slučajeva.

U radu [34] se po uzoru na mrežne agente koji automatski istražuju web stranice (engl. *Web crawlers*) predlaže primjena procesnih robota (engl. *Process crawlers*) koji mogu pratiti različite puteve izvođenja web aplikacija. Iz istraženih puteva izvođenja automatski se gradi njen model tijeka rada (engl. *Workflow model*) te se generirani model može koristiti za stvaranje testnih slučajeva.

Autori u radu [35] se zalažu za izgradnju web aplikacija utemeljenu na modelima (eng. *Model based design*). Zato bi prvo trebalo ručno u jeziku *Promela* opisati procesni model buduće web aplikacije, a aplikaciju implementirati tek nakon što se njen procesni model uspješno verificira alatom za provjeru modela *Spin*. U prethodno provedenim istraživanjima se na sličan način verificiralo s alatom *Spin* ranije izrađene interaktivne web aplikacije za e-učenje [36].

U sustavima za e-učenje generira se i pohranjuje veliki broj podataka o korisnicima i njihovim aktivnostima. Već danas obrada takve glomazne količine podataka (eng. *Big Data*) više nije efikasna korištenjem uobičajenih alata te se pronalaze drugi pristupi za obradu takvih podatkovnih skupova [37, 38, 39]. Jedna od tehnika za obradu glomazne količine podataka je dubinska analiza procesa (engl. *Process mining*) [40, 41, 42, 43]. To je relativno nova disciplina u području poslovne inteligencije, a omogućava otkrivanje i analizu poslovnih procesa temeljem obrade zapisa u dnevnicima događaja velikih poslovnih informacijskih sustava. Tehnikama dubinske analize procesa mogu se identificirati uzorci ponašanja poslovnog sustava čak

i iz golemog skupa dnevničkih zapisa. Alati za dubinsku analizu procesa mogu iz ulaznih dnevničkih podataka automatski identificirati i vizualizirati modele procesa, odnosno uzorke ponašanja poslovnog sustava. Eksperti za dubinsku analizu procesa potom dobivene modele procesa analiziraju, provjeravaju njihovu usklađenost sa stvarnim poslovnim procesima, te prema tome daju prijedloge za popravak i unaprjeđenje poslovnih procesa. Dakako, dubinska analiza procesa se koristi i u drugim područjima pa tako i u polju e-učenja [44]. U radu [45] autori su primjenom dubinske analize procesa istraživali učinke prezentacije i personalizacije udaljenih provjera znanja koje se sastoje od različitih stilova pitanja s ponuđenim odgovorima (strogi redoslijed pitanja i dobivanje povratnih informacija o odgovoru ili fleksibilni redoslijed pitanja, ali bez povratnih informacija o odgovoru).

Također, i u radu [46] autori analiziraju uspjeh studenata na udaljenim provjerama znanja korištenjem metode dubinske analize procesa. U skladu s očekivanjima, studenti koji su pali predmet puno su manje pratili snimke predavanja prije završnih ispita. Autori su ustanovili pozitivnu korelaciju između aktivnog praćenja predavanja preko mreže i konačnih ocjena na predmetu.

Poglavlje 3

Teorijske osnove

U ovom poglavlju daje se podrobniji teorijski pregled svih metoda i tehnika korištenih u ovome radu, a to su metoda formalne analize koncepata, postupak kombinatornog testiranja, topološko sortiranje grafova, učenje konačnih automata i algoritam L^* Dane Angluin te metoda provjere modela, temporalna logika LTL i alat za provjeru modela *Spin*.

Na početku treba naglasiti da je ishodišna točka ovog doktorskog istraživanja formalni opis domenskoga znanja koji se može dobiti izgradnjom ontologije. Jednu od poznatijih definicija pojma ontologije u području računarske znanosti predložili su Studer i sur. u [47]: *"Ontologija je formalna, eksplicitna specifikacija dijeljene konceptualizacije."* U ovom radu prvenstveno smo se vodili sljedećom širokom definicijom pojma ontologije koju je predložio Sowa u [48, 49]: *"Ontologija definira vrste stvari koje postoje u izabranoj domeni."* Vrste stvari u zadanoj domeni možemo opisati skupom objekata gdje svaki objekt ima svoja obilježja ili atribute. Slični objekti su oni koji dijele iste atribute, a takvi objekti mogu se zajedno povezati u zajednički pojam ili koncept. Naposljetku se hijerarhija svih ustanovljenih koncepata, od općenitijih prema onim specifičnijim, može smatrati ontologijom. Za izgradnju takve ontologije domenskog znanja u ovom radu koristi se metoda formalne analize koncepata.

3.1 Formalna analiza koncepata

Metoda formalne analize koncepata (izvorno njem. *Formale Begriffsanalyse* – FBA, engl. *Formal Concept Analysis* – FCA) je tehnika nenadziranog strojnog učenja za otkrivanje i predstavljanje znanja te za analizu podataka [11, 50]. Metoda FCA iz ulaznih podataka identificira formalne koncepte (skupove objekata sa zajedničkim atributima), pronalazi njihove međuovisnosti te ih vizualizira u obliku usmjerenog acikličkog grafa koji se naziva *konceptualna rešetka* ili *konceptna rešetka* (engl. *Concept Lattice*). Metodu FCA je razvio ranih 1980-ih godina njemački matematičar Rudolf Wille na temeljima matematičke teorije rešetki (engl. *Lattice theory*) i teorije skupova [9, 10]. Wille je u svom radu preuzeo iz područja filozofije tumačenje kon-

cepta (*pojma*) kao jedinice misli oblikovane generalizacijom koja obuhvaća neki skup objekata te skup njihovih zajedničkih atributa.

U nastavku će se prvo dati uvodni pregled glavnih značajki metode FCA, a potom prikaz njenih matematičkih temelja. Na kraju će se izložiti postupci za sažimanje ulaznih i izlaznih podataka te kratki pregled proširenja metode FCA.

3.1.1 Uvod u metodu FCA

Za primjenu metode formalne analize koncepata prvo treba odabrati domenu ulaznih podataka kroz odabir nekog skupa objekata te definiranje skupa atributa kojima se objekti mogu opisati. Potom treba pripremiti ulazne podatke tako da tvore binarnu dvodimenzionalnu matricu objekata (redci matrice) i atributa (stupci matrice). Za svaki objekt u matrici treba naznačiti sve atribute koje posjeduje. Ako neki objekt posjeduje neki atribut u matricu se upisuje oznaka "*X*" na sjecištu retka tog objekta i stupca tog atributa. Ovako pripremljena matrica ulaznih podataka naziva se formalni kontekst. Provođenjem metodom FCA nad formalnim kontekstom otkrivaju se dvije vrste podataka. Prva vrsta je konceptualna rešetka, struktura u obliku usmjerenog acikličkog grafa sa svim identificiranim formalnim konceptima. Uz konceptualnu rešetku metoda FCA rezultira i s listom asocijacijskih pravila, odnosno svim identificiranim relacijama među atributima [51].

Definicija 3.1. Formalni kontekst je trojka $\langle X, Y, I \rangle$ gdje je X neprazni skup objekata, Y neprazni skup atributa te I binarna relacija između elemenata skupova X i Y: $I \subseteq X \times Y$.

Dakle, za ulaznu matricu s n redaka i m stupaca formalni kontekst $\langle X, Y, I \rangle$ se sastoji od skupa objekata $X = \{x_1, \ldots, x_n\}$ i skupa atributa $Y = \{y_1, \ldots, y_m\}$ te relacije I definirane kao $(x_i, y_j) \in I$ ako i samo ako sjecište i-tog retka i j-stupca ulazne matrice nije prazno, odnosno ako i samo ako objekt x_i ima atribut y_j.

Definicija 3.2. Za formalni kontekst $\langle X, Y, I \rangle$ definirani su operatori formiranja koncepata $^{\uparrow}$: $2^X \to 2^Y$ i $^{\downarrow} : 2^Y \to 2^X$ za svaki $A \subseteq X$ i $B \subseteq Y$ kao $A^{\uparrow} = \{y \in Y | \forall x \in A : (x, y) \in I\}$, odnosno $B^{\downarrow} = \{x \in X | \forall y \in B : (x, y) \in I\}$.

Primjenom operatora formiranja koncepata $^{\uparrow}$ na skup objekata A, tj. $A^{\uparrow}$ dobiva se skup svih atributa koji imaju svi objekti iz skupa A. Analogno tome, primjenom operatora formiranja koncepata $^{\downarrow}$ na skup atributa B, odnosno $B^{\downarrow}$ generira se skup svih objekata koji dijele sve atribute iz skupa B. Operatori formiranja koncepata su alat za pronalaženje segmenata formalnog konteksta u kojem različiti objekti dijele iste atribute, a takvi segmenti formalnog konteksta nazivaju se formalni koncepti.

Definicija 3.3. Formalni koncept u formalnom kontekstu $\langle X, Y, I \rangle$ je par $\langle A, B \rangle$ gdje je $A \subseteq X$ i $B \subseteq Y$ za koji vrijedi $A^{\uparrow} = B$ i $B^{\downarrow} = A$.

Dakle, $\langle A,B\rangle$ je formalni koncept ako i samo ako se A sastoji samo od objekata koji imaju sve atribute iz B, a B se sastoji samo od atributa koje dijele svi objekti iz A. Skup objekata A naziva se ekstenzija (engl. *extent*) formalnog koncepta $\langle A,B\rangle$, a skup atributa B naziva se intencija (engl. *intent*) formalnog koncepta $\langle A,B\rangle$. Ekstenzija formalnog koncepta može se definirati kao $Ext(X,Y,I) = \left\{B^{\downarrow} | B \subseteq Y\right\}$, a intencija kao $Int(X,Y,I) = \left\{A^{\uparrow} | A \subseteq X\right\}$.

Treba naglasiti da se formalni koncepti međusobno nalaze u hijerarhijskom odnosu potkoncept (engl. *subconcept*) i natkoncept (engl. *superconcept*). Takva hijerarhijska relacija među formalnim konceptima može se iskazati operatorom $\leq$.

Definicija 3.4. Za formalne koncepte $\langle A_1,B_1\rangle$ i $\langle A_2,B_2\rangle$ iz formalnog konteksta $\langle X,Y,I\rangle$ vrijedi $\langle A_1,B_1\rangle \leq \langle A_2,B_2\rangle$ ako i samo ako vrijedi $A_1 \subseteq A_2$ i $B_2 \subseteq B_1$.

Ovdje je $\langle A_2,B_2\rangle$ natkoncept formalnog koncepta $\langle A_1,B_1\rangle$ jer su svi objekti iz A_1 sadržani u A_2, a svi atributi iz B_2 su sadržani u B_1. Sve pronađene formalne koncepte i njihove međusobne hijerarhijske odnose može se prikazati u obliku konceptualne rešetke, usmjerenog acikličkog grafa (Hasseovog ili linijskog dijagrama [52]).

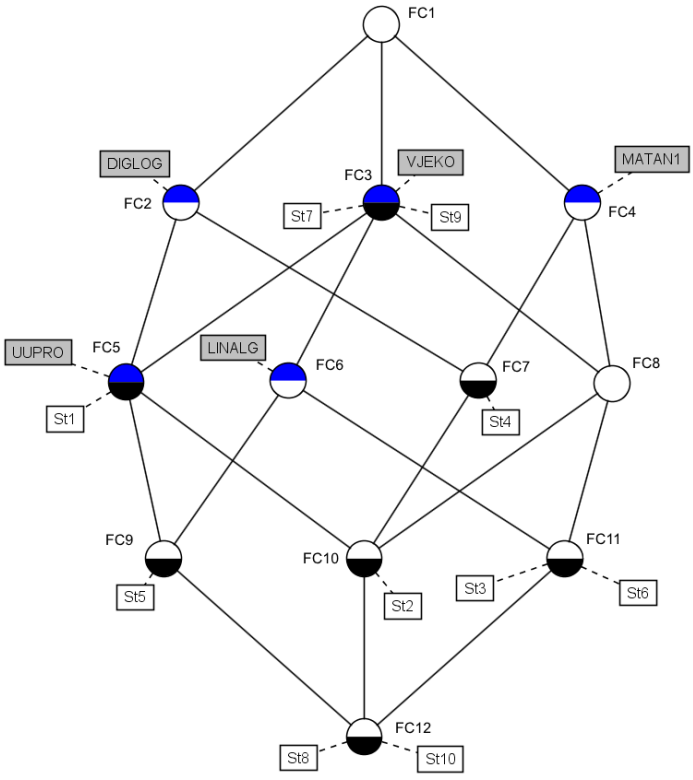
Definicija 3.5. Skup svih formalnih koncepata iz formalnog konteksta $\langle X,Y,I\rangle$ je $\mathfrak{B}(X,Y,I) = \left\{\langle A,B\rangle \in 2^X \times 2^Y | A^{\uparrow} = B, B^{\downarrow} = A\right\}$, a zajedno s hijerarhijskom relacijom $\leq$ čini konceptualnu rešetku $\underline{\mathfrak{B}}(X,Y,I) = (\mathfrak{B}(X,Y,I),\leq)$.

Konceptualna rešetka je parcijalno uređeni skup u kojem bilo koja dva formalna koncepta imaju zajednički *supremum* (najmanji zajednički natkoncept) i zajednički *infimum* (najveći zajednički potkoncept). Grafički se konceptualna rešetka prikazuje kao usmjereni aciklički graf u kojem su formalni koncepti čvorovi, a usmjereni bridovi sa strelicom od vrha prema dolje određuju hijerarhiju među čvorovima odnosno formalnim konceptima. Čvor na vrhu grafa predstavlja najopćenitiji formalni koncept koji uključuje skup svih objekata i (najčešće prazni) skup atributa kojeg dijele svi objekti. S druge strane, čvor na dnu grafa predstavlja najspecifičniji formalni koncept koji uključuje skup svih atributa i (najčešće prazni) skup objekata koji imaju sve atribute. Kroz strukturu konceptualne rešetke vizualiziraju se svi odnosi između objekata i atributa iz formalnog konteksta. Zbog jednostavnosti prikaza često se ispušta strelica kod bridova, ali oni su i dalje implicitno usmjereni od vrha prema dolje, odnosno od natkoncepta prema potkonceptu/ima. Isto tako, radi preglednosti obično se ne navodi ime objekta u svakom formalnom konceptu u kojem se nalazi nego samo u onom najnižem (najspecifičnijem) u grafu. Implicitno se zna da se taj objekt nalazi i u svim natkonceptima do kojih se može doći prateći bridove do tog najnižeg formalnog koncepta do vrha grafa. Analogno tome, i ime atributa se navodi samo u najvišem (najopćenitijem) formalnom konceptu, a pretpostavlja se da je taj atribut sadržan i u svim njegovim potkonceptima prateći bridove sve do dna grafa. Kao ilustraciju rezultata metode FCA navest će se sljedeći primjer.

Primjer 3.1. U tablici 3.1 zapisani su umjetno generirani podaci o uspješnosti male grupe studenata na prvoj godini preddiplomskog studija na FER-u. Ulazni podaci čine formalni konteksta s 10 objekata (identifikacijske oznake studenata, npr. St1 je oznaka za prvog studenta) i 5 atributa (oznake položenih predmeta prve godine preddiplomskog studija FER-a[1]). Uz svakog studenta naznačeno je koje je predmete uspješno položio (oznake "*X*" na presjeku odgovarajućeg retka i stupca).

Tablica 3.1: Formalni kontekst (10 objekata – studenti, 5 atributa – položeni predmeti)

	DIGLOG	LINALG	MATAN1	UUPRO	VJEKO
St1	*X*			*X*	*X*
St2	*X*		*X*	*X*	*X*
St3		*X*	*X*		*X*
St4	*X*		*X*		
St5	*X*	*X*		*X*	*X*
St6		*X*	*X*		*X*
St7					*X*
St8	*X*	*X*	*X*	*X*	*X*
St9					*X*
St10	*X*	*X*	*X*	*X*	*X*



Slika 3.1: Konceptualna rešetka za formalni kontekst iz tablice 3.1

[1]Objašnjenje imena atributa: DIGLOG – *Digitalna logika*, LINALG – *Linearna algebra*, MATAN1 – *Matematička analiza 1*, UUPRO – *Uvod u programiranje* i VJEKO – *Vještine komuniciranja*.

Na slici 3.1 prikazana je automatski generirana konceptualna rešetka dobivena s programom otvorenog koda *ConExp - Concept Explorer 1.3* [53]. U njoj je pronađeno 12 formalnih koncepata koji su povezani s 20 bridova implicitno usmjerenih od vrha prema dnu, od natkoncepta prema potkonceptu. Atributi su upisani u sivo osjenčane pravokutnike iznad formalnih koncepata, a objekti su upisani u bijelo osjenčane pravokutnike ispod formalnih koncepata. Kao što je ranije spomenuto, radi preglednosti atributi se ispisuju samo u najopćenitijem formalnom konceptu u kojem se pojavljuju (prisutni su i u svim njegovim potkonceptima), a objekti se ispisuju samo u najspecifičnijem formalnom konceptu u kojem se pojavljuju (sastavni su dio i svih njegovih natkoncepata). Dodatno, ručno su dodane oznake svakog formalnog koncepta (FC1 – FC12). Npr. formalni koncept FC7 obuhvaća sve studente koji su položili i *Digitalnu logiku* i *Matematičku analizu 1* što proizlazi iz njegovih natkoncepata (FC2, FC4 i FC1). To je student s oznakom St4 te studenti s oznakama St2, St8 i St10 što se vidi iz njegovih potkoncepata (FC10 i FC12). Treba primijetiti kako se iz formalnog koncepta na vrhu (FC1) može iščitati kako nema predmeta kojeg je položilo baš svih 10 analiziranih studenata (uključeno svih 10 objekata – studenata, ali je skup atributa – položenih predmeta prazan). Isto tako iz formalnog koncepta na dnu (FC12) vidi se kako je dvoje promatranih studenata položilo svih pet predmeta – to su studenti s oznakama St8 i St10.

Razvijeno je više algoritama za izračun konceptualne rešetke, a zajedničko im je da zahtijevaju kao ulaz formalni kontekst $\langle X, Y, I \rangle$. Jednostavniji algoritmi daju kao izlaz skup svih formalnih koncepata $\mathfrak{B}(X, Y, I)$ iz koje se naknadno mogu izračunati svi hijerarhijski odnosi i dobiti konceptualna rešetka $\underline{\mathfrak{B}}(X, Y, I)$. Složeniji algoritmi su efikasniji i direktno kao izlaz daju strukturu konceptualne rešetke. Ganter je 1987. u [54] razvio algoritam *NextClosure*, osnovni algoritam za generiranje konceptualne rešetke. Kao ulazni podatak algoritam uzima formalni kontekst $\langle X, Y, I \rangle$, a kao izlaz daje $Int(X, Y, I)$ – leksikografski sortiranu listu svih intencija formalnih koncepata iz ulaznog formalnog konteksta. Iz te liste rekonstruira se skup $\mathfrak{B}(X, Y, I)$ jer vrijedi $\mathfrak{B}(X, Y, I) = \left\{ \langle B^{\downarrow}, B \rangle | B \in Int(X, Y, I) \right\}$. Na kraju se pronađu hijerarhijski odnosi među formalnim konceptima u skupu $\mathfrak{B}(X, Y, I)$ korištenjem definicije 3.4 kako bi se dobila konceptualna rešetka $\underline{\mathfrak{B}}(X, Y, I)$.

Ovdje će se proces generiranja svih formalnih koncepata iz primjera 3.1 ilustrirati primjenom jednostavnog algoritma iz [50]:

Algoritam 1 Generiranje formalnih koncepata

1. izračunaj ekstenziju čitavog formalnog konteksta $\langle X, Y, I \rangle$: $Ext(X, Y, I) = \left\{ B^{\downarrow} | B \subseteq Y \right\}$
2. za $\forall A \in Ext(X, Y, I)$ pronađi $\langle A, A^{\uparrow} \rangle$

Prvo se pronalaze podskupovi atributa iz formalnog konteksta u tablici 3.1 – kreće se od praznog

skupa, potom kombinacija atributa navedenih u tablici te na kraju skupa svih atributa Y:

$$B_0 = \emptyset, B_1 = \{\text{DIGLOG}\}, B_2 = \{\text{LINALG}\}, B_3 = \{\text{MATAN1}\}, B_4 = \{\text{UUPRO}\},$$
$$B_5 = \{\text{VJEKO}\}, B_6 = \{\text{DIGLOG}, \text{LINALG}\}, B_7 = \{\text{DIGLOG}, \text{MATAN1}\},$$
$$B_8 = \{\text{DIGLOG}, \text{UUPRO}\}, B_9 = \{\text{DIGLOG}, \text{VJEKO}\}, B_{10} = \{\text{LINALG}, \text{MATAN1}\},$$
$$B_{11} = \{\text{LINALG}, \text{UUPRO}\}, B_{12} = \{\text{LINALG}, \text{VJEKO}\}, B_{13} = \{\text{MATAN1}, \text{UUPRO}\},$$
$$B_{14} = \{\text{MATAN1}, \text{VJEKO}\}, B_{15} = \{\text{UUPRO}, \text{VJEKO}\}, B_{16} = \{\text{DIGLOG}, \text{UUPRO}, \text{VJEKO}\},$$
$$B_{17} = \{\text{DIGLOG}, \text{MATAN1}, \text{UUPRO}\}, B_{18} = \{\text{DIGLOG}, \text{MATAN1}, \text{VJEKO}\},$$
$$B_{19} = \{\text{MATAN1}, \text{UUPRO}, \text{VJEKO}\}, B_{20} = \{\text{LINALG}, \text{MATAN1}, \text{VJEKO}\},$$
$$B_{21} = \{\text{DIGLOG}, \text{LINALG}, \text{UUPRO}\}, B_{22} = \{\text{DIGLOG}, \text{LINALG}, \text{VJEKO}\},$$
$$B_{23} = \{\text{LINALG}, \text{UUPRO}, \text{VJEKO}\}, B_{24} = \{\text{DIGLOG}, \text{MATAN1}, \text{UUPRO}, \text{VJEKO}\},$$
$$B_{25} = \{\text{DIGLOG}, \text{LINALG}, \text{UUPRO}, \text{VJEKO}\},$$
$$B_{26} = Y = \{\text{DIGLOG}, \text{LINALG}, \text{MATAN1}, \text{UUPRO}, \text{VJEKO}\}$$

Potom se za svaki navedeni podskup atributa B_i traži odgovarajući podskup objekata $A_i = B_i{}^{\downarrow}$. Nadalje, ako vrijedi i $A_i{}^{\uparrow} = B_i$ onda A_i pripada ekstenziji čitavog formalnog konteksta $Ext(X,Y,I)$ te je time pronađen novi formalni koncept $\langle A_i, A_i{}^{\uparrow}\rangle$, odnosno $\langle A_i, B_i\rangle$:

$$A_0 = B_0{}^{\downarrow} = X \text{ i } A_0{}^{\uparrow} = B_0 \text{ pa je nađen } \langle A_0, B_0\rangle$$
$$A_1 = B_1{}^{\downarrow} = \{\text{St1}, \text{St2}, \text{St4}, \text{St5}, \text{St8}, \text{St10}\} \text{ i } A_1{}^{\uparrow} = B_1 \text{ pa je nađen } \langle A_1, B_1\rangle$$
$$A_2 = B_2{}^{\downarrow} = \{\text{St3}, \text{St5}, \text{St6}, \text{St8}, \text{St10}\}, \text{ ali } A_2{}^{\uparrow} = \{\text{LINALG}, \text{VJEKO}\} \neq B_2$$
$$A_3 = B_3{}^{\downarrow} = \{\text{St2}, \text{St3}, \text{St4}, \text{St6}, \text{St8}, \text{St10}\} \text{ i } A_3{}^{\uparrow} = B_3 \text{ pa je nađen } \langle A_3, B_3\rangle$$
$$A_4 = B_4{}^{\downarrow} = \{\text{St1}, \text{St2}, \text{St5}, \text{St8}, \text{St10}\}, \text{ ali } A_4{}^{\uparrow} = \{\text{DIGLOG}, \text{UUPRO}, \text{VJEKO}\} \neq B_4$$
$$A_5 = B_5{}^{\downarrow} = \{\text{St1}, \text{St2}, \text{St3}, \text{St5}, \text{St6}, \text{St7}, \text{St8}, \text{St9}, \text{St10}\} \text{ i } A_5{}^{\uparrow} = B_5 \text{ pa je nađen } \langle A_5, B_5\rangle$$
$$A_6 = B_6{}^{\downarrow} = \{\text{St5}, \text{St8}, \text{St10}\}, \text{ ali } A_6{}^{\uparrow} = \{\text{DIGLOG}, \text{LINALG}, \text{UUPRO}, \text{VJEKO}\} \neq B_6$$
$$A_7 = B_7{}^{\downarrow} = \{\text{St2}, \text{St4}, \text{St8}, \text{St10}\} \text{ i } A_7{}^{\uparrow} = B_7 \text{ pa je nađen } \langle A_7, B_7\rangle$$
$$A_8 = B_8{}^{\downarrow} = \{\text{St1}, \text{St2}, \text{St5}, \text{St8}, \text{St10}\}, \text{ ali } A_8{}^{\uparrow} = \{\text{DIGLOG}, \text{UUPRO}, \text{VJEKO}\} \neq B_8$$
$$A_9 = B_9{}^{\downarrow} = \{\text{St1}, \text{St2}, \text{St5}, \text{St8}, \text{St10}\}, \text{ ali } A_9{}^{\uparrow} = \{\text{DIGLOG}, \text{UUPRO}, \text{VJEKO}\} \neq B_9$$
$$A_{10} = B_{10}{}^{\downarrow} = \{\text{St3}, \text{St6}, \text{St8}, \text{St10}\}, \text{ ali } A_{10}{}^{\uparrow} = \{\text{LINALG}, \text{MATAN1}, \text{VJEKO}\} \neq B_{10}$$
$$A_{11} = B_{11}{}^{\downarrow} = \{\text{St5}, \text{St8}, \text{St10}\}, \text{ ali } A_{11}{}^{\uparrow} = \{\text{DIGLOG}, \text{LINALG}, \text{UUPRO}, \text{VJEKO}\} \neq B_{11}$$
$$A_{12} = B_{12}{}^{\downarrow} = \{\text{St3}, \text{St5}, \text{St6}, \text{St8}, \text{St10}\} \text{ i } A_{12}{}^{\uparrow} = B_{12} \text{ pa je nađen } \langle A_{12}, B_{12}\rangle$$
$$A_{13} = B_{13}{}^{\downarrow} = \{\text{St2}, \text{St8}, \text{St10}\}, \text{ ali } A_{13}{}^{\uparrow} = \{\text{DIGLOG}, \text{MATAN1}, \text{UUPRO}, \text{VJEKO}\} \neq B_{13}$$
$$A_{14} = B_{14}{}^{\downarrow} = \{\text{St2}, \text{St3}, \text{St6}, \text{St8}, \text{St10}\} \text{ i } A_{14}{}^{\uparrow} = B_{14} \text{ pa je nađen } \langle A_{14}, B_{14}\rangle$$
$$A_{15} = B_{15}{}^{\downarrow} = \{\text{St1}, \text{St2}, \text{St5}, \text{St8}, \text{St10}\}, \text{ ali } A_{15}{}^{\uparrow} = \{\text{DIGLOG}, \text{UUPRO}, \text{VJEKO}\} \neq B_{15}$$

$$A_{16} = B_{16}{}^{\downarrow} = \{\text{St1}, \text{St2}, \text{St5}, \text{St8}, \text{St10}\} \text{ i } A_{16}{}^{\uparrow} = B_{16} \text{ pa je nađen } \langle A_{16}, B_{16} \rangle$$

$$A_{17} = B_{17}{}^{\downarrow} = \{\text{St2}, \text{St8}, \text{St10}\}, \text{ ali } A_{17}{}^{\uparrow} = \{\text{DIGLOG}, \text{MATAN1}, \text{UUPRO}, \text{VJEKO}\} \neq B_{17}$$

$$A_{18} = B_{18}{}^{\downarrow} = \{\text{St2}, \text{St8}, \text{St10}\}, \text{ ali } A_{18}{}^{\uparrow} = \{\text{DIGLOG}, \text{MATAN1}, \text{UUPRO}, \text{VJEKO}\} \neq B_{18}$$

$$A_{19} = B_{19}{}^{\downarrow} = \{\text{St2}, \text{St8}, \text{St10}\}, \text{ ali } A_{19}{}^{\uparrow} = \{\text{DIGLOG}, \text{MATAN1}, \text{UUPRO}, \text{VJEKO}\} \neq B_{19}$$

$$A_{20} = B_{20}{}^{\downarrow} = \{\text{St3}, \text{St6}, \text{St8}, \text{St10}\} \text{ i } A_{20}{}^{\uparrow} = B_{20} \text{ pa je nađen } \langle A_{20}, B_{20} \rangle$$

$$A_{21} = B_{21}{}^{\downarrow} = \{\text{St5}, \text{St8}, \text{St10}\}, \text{ ali } A_{21}{}^{\uparrow} = \{\text{DIGLOG}, \text{LINALG}, \text{UUPRO}, \text{VJEKO}\} \neq B_{21}$$

$$A_{22} = B_{22}{}^{\downarrow} = \{\text{St5}, \text{St8}, \text{St10}\}, \text{ ali } A_{22}{}^{\uparrow} = \{\text{DIGLOG}, \text{LINALG}, \text{UUPRO}, \text{VJEKO}\} \neq B_{22}$$

$$A_{23} = B_{23}{}^{\downarrow} = \{\text{St5}, \text{St8}, \text{St10}\}, \text{ ali } A_{23}{}^{\uparrow} = \{\text{DIGLOG}, \text{LINALG}, \text{UUPRO}, \text{VJEKO}\} \neq B_{23}$$

$$A_{24} = B_{24}{}^{\downarrow} = \{\text{St2}, \text{St8}, \text{St10}\} \text{ i } A_{24}{}^{\uparrow} = B_{24} \text{ pa je nađen } \langle A_{24}, B_{24} \rangle$$

$$A_{25} = B_{25}{}^{\downarrow} = \{\text{St5}, \text{St8}, \text{St10}\} \text{ i } A_{25}{}^{\uparrow} = B_{25} \text{ pa je nađen } \langle A_{25}, B_{25} \rangle$$

$$A_{26} = B_{26}{}^{\downarrow} = \{\text{St8}, \text{St10}\} \text{ i } A_{26}{}^{\uparrow} = Y = B_{26} \text{ pa je nađen } \langle A_{26}, B_{26} \rangle$$

Ukupno je ovim ručnim postupkom pronađeno 12 formalnih koncepata, što odgovara i broju čvorova odnosno formalnih koncepata u automatski generiranoj konceptualnoj rešetci na slici 3.1. Sada se može i detaljnije opisati svaki formalni koncept sa slike 3.1 redom kako su na njoj označeni – FC1: $\langle A_0, B_0 \rangle$, FC2: $\langle A_1, B_1 \rangle$, FC3: $\langle A_5, B_5 \rangle$, FC4: $\langle A_3, B_3 \rangle$, FC5: $\langle A_{16}, B_{16} \rangle$, FC6: $\langle A_{12}, B_{12} \rangle$, FC7: $\langle A_7, B_7 \rangle$, FC8: $\langle A_{14}, B_{14} \rangle$, FC9: $\langle A_{25}, B_{25} \rangle$, FC10: $\langle A_{24}, B_{24} \rangle$, FC11: $\langle A_{20}, B_{20} \rangle$ i FC12: $\langle A_{26}, B_{26} \rangle$. Primjenom definicije 3.4 može se konstruirati konceptualna rešetka sa slike 3.1 – vidi se da vrijede sljedeći odnosi potkoncept-natkoncept: FC2 $\leq$ FC1, FC3 $\leq$ FC1, FC4 $\leq$ FC1, FC5 $\leq$ FC2, FC5 $\leq$ FC3, FC6 $\leq$ FC3, FC7 $\leq$ FC2, FC7 $\leq$ FC4, FC8 $\leq$ FC3, FC8 $\leq$ FC4, FC9 $\leq$ FC5, FC9 $\leq$ FC6, FC10 $\leq$ FC5, FC10 $\leq$ FC7, FC10 $\leq$ FC8, FC11 $\leq$ FC6, FC11 $\leq$ FC8, FC12 $\leq$ FC9, FC12 $\leq$ FC10 i FC12 $\leq$ FC11. Nadalje treba primijetiti kako se ne može svaki par formalnih koncepata dovesti u odnos potkoncept-natkoncept, npr. to su formalni koncepti FC5: $\langle A_{16}, B_{16} \rangle = \langle \{\text{St1}, \text{St2}, \text{St5}, \text{St8}, \text{St10}\}, \{\text{DIGLOG}, \text{UUPRO}, \text{VJEKO}\} \rangle$ i FC6: $\langle A_{12}, B_{12} \rangle = \langle \{\text{St3}, \text{St5}, \text{St6}, \text{St8}, \text{St10}\}, \{\text{LINALG}, \text{VJEKO}\} \rangle$ jer ne vrijedi $A_{12} \subseteq A_{16}$ i $B_{16} \subseteq B_{12}$ niti $A_{16} \subseteq A_{12}$ i $B_{12} \subseteq B_{16}$. Vizualno se takvi formalni koncepti prikazuju paralelno na istoj razini konceptualne rešetke – npr. ovdje su na trećem nivou konceptualne rešetke formalni koncepti FC5, FC6, FC7 i FC8.

Osim korisnog vizualnog rezultata u obliku konceptualne rešetke metodom FCA se stvara i drugi skup rezultata, a to je sažeti popis međuovisnosti atributa, odnosno asocijacijska pravila [55] i implikacije atributa iz formalnog konteksta [50]. Ovo pogotovo može biti interesantno kod analize velikih formalnih konteksta, odnosno golemih konceptualnih rešetki gdje je teško vizualno iščitati identificirane međuovisnosti i pravilnosti u ulaznim podacima.

Definicija 3.6. Asocijacijsko pravilo je par $A \rightarrow B$ gdje su A i B podskupovi skupa atributa Y u

formalnom kontekstu $\langle X, Y, I \rangle$. Podrška asocijacijskog pravila (engl. *Support*) se definira kao:

$$supp(A \to B) = \frac{\left|(A \cup B)^{\downarrow}\right|}{|X|} \tag{3.1}$$

Pouzdanost asocijacijskog pravila (engl. *Confidence*) se definira kao:

$$conf(A \to B) = \frac{\left|(A \cup B)^{\downarrow}\right|}{\left|A^{\downarrow}\right|} \tag{3.2}$$

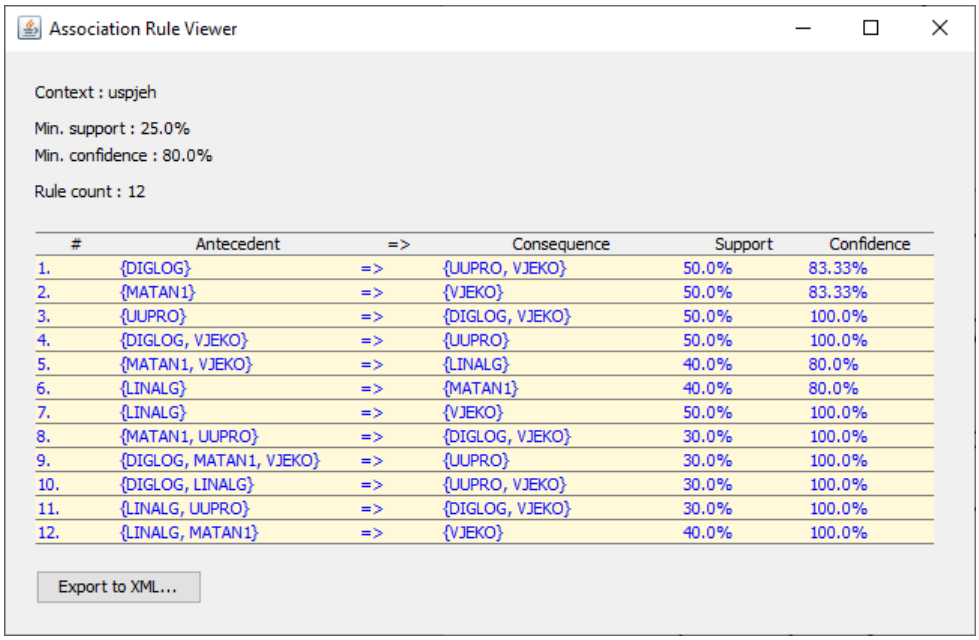
Npr. iz formalnog konteksta iz primjera 3.1 za prvi objekt (student St1) vrijedi asocijacijsko pravilo $\{\text{DIGLOG}\} \to \{\text{VJEKO}\}$ (student koji je položio *Digitalnu logiku* položio je i *Vještine komuniciranja*). Može se izračunati podrška ovog asocijacijskog pravila u čitavom formalnom kontekstu kao omjer broja objekata u kojem pravilo vrijedi (za pet studenata: St1, St2, St5, St8 i St10) i ukupnog broja objekata (10 studenata), dakle $supp(\{\text{DIGLOG}\} \to \{\text{VJEKO}\}) = 50\%$. Nadalje, pouzdanost ovog asocijacijskog pravila u formalnom kontekstu računa se kao omjer broja objekata u kojem pravilo vrijedi (5) i broja svih objekata koji sadrže antecedent (6 studenata ima atribut DIGLOG): $conf(\{\text{DIGLOG}\} \to \{\text{VJEKO}\}) = 83{,}33\%$.

Definicija 3.7. Implikacija atributa je par $A \Rightarrow B$ gdje su A i B podskupovi skupa atributa Y u formalnom kontekstu $\langle X, Y, I \rangle$. Implikacija atributa $A \Rightarrow B$ je istinita u formalnom kontekstu $\langle X, Y, I \rangle$ ako svaki njegov objekt koji ima sve atribute iz skupa A ima i sve atribute iz skupa B.

Implikacija atributa $A \Rightarrow B$ je stroži oblik asocijacijskog pravila jer mora vrijediti za čitavi formalni kontekst, odnosno pouzdanost implikacije atributa mora biti $conf(A \Rightarrow B) = 100\%$. Primjerice implikacija atributa $\{\text{LINALG}, \text{MATAN1}\} \Rightarrow \{\text{VJEKO}\}$ je istinita u formalnom kontekstu iz primjera 3.1 (jer vrijedi da su svi studenti koji su položili *Linearnu algebru* i *Matematičku analizu 1* ujedno položili i *Vještine komuniciranja*).

Kako bi se izbjeglo pronalaženje suvišnih implikacija atributa mogu se odrediti teorija i model formalnog konteksta. Teorija T obuhvaća neki osnovni skup implikacija atributa nad formalnim kontekstom. Može se osigurati da je teorija neredundantna, odnosno da ne sadrži suvišne implikacije atributa koje se mogu izvesti iz ostalih korištenjem Armstrongovih pravila [56, 57]. Model teorije odgovara nekom podskupu atributa M formalnog konteksta u kojem vrijedi svaka implikacija atributa iz zadane teorije T. Teorija T je neredundantna ako je kompletna i ako nakon uklanjanja bilo koje implikacije iz teorije T ta implikacija više semantički ne slijedi iz teorije T. Teorija T nekog formalnog konteksta $\langle X, Y, I \rangle$ je kompletna ako i samo ako svaka njegova istinita implikacija atributa slijedi iz teorije T. Jedan od značajnih algoritama koji računaju neredundantnu teoriju formalnog konteksta, odnosno minimalni skup implikacija atributa koje su istinite u zadanom formalnom kontekstu je algoritam autora Duquennea i Guiguesa [58, 59].

Asocijacijska pravila i implikacije atributa mogu se u jednostavnijim slučajevima pronalaziti i ručno, ali to je dugotrajan i zamoran proces. Napredniji računalni programi za izvođenje metode FCA taj posao odrađuju automatski. Npr. *Lattice Miner Platform 2.0* pronalazi asocijacijska pravila uz zadane mjere podrške i pouzdanosti kao i minimalni skup svih implikacija atributa koje su istinite u zadanom formalnom kontekstu (baza implikacija po Duquenneu i Guiguesu) [60]. Primjerice, za formalni kontekst iz primjera 3.1 i zadane minimalne vrijednosti podrške od 25% i pouzdanosti od 80% navedeni alat je identificirao 12 asocijacijskih pravila prikazanih na slici 3.2.



#	Antecedent	=>	Consequence	Support	Confidence
1.	{DIGLOG}	=>	{UUPRO, VJEKO}	50.0%	83.33%
2.	{MATAN1}	=>	{VJEKO}	50.0%	83.33%
3.	{UUPRO}	=>	{DIGLOG, VJEKO}	50.0%	100.0%
4.	{DIGLOG, VJEKO}	=>	{UUPRO}	50.0%	100.0%
5.	{MATAN1, VJEKO}	=>	{LINALG}	40.0%	80.0%
6.	{LINALG}	=>	{MATAN1}	40.0%	80.0%
7.	{LINALG}	=>	{VJEKO}	50.0%	100.0%
8.	{MATAN1, UUPRO}	=>	{DIGLOG, VJEKO}	30.0%	100.0%
9.	{DIGLOG, MATAN1, VJEKO}	=>	{UUPRO}	30.0%	100.0%
10.	{DIGLOG, LINALG}	=>	{UUPRO, VJEKO}	30.0%	100.0%
11.	{LINALG, UUPRO}	=>	{DIGLOG, VJEKO}	30.0%	100.0%
12.	{LINALG, MATAN1}	=>	{VJEKO}	40.0%	100.0%

Slika 3.2: Asocijacijska pravila za formalni kontekst u tablici 3.1 uz podršku min. 25% i pouzdanost min. 80%

Treba primijetiti kako na slici 3.2 među pronađenih 12 asocijacijskih pravila ima 8 implikacija atributa jer im je pouzdanost 100% (pravila pod rednim brojevima 3. i 4. te 7. do 12.). Dodatno, alat *Lattice Miner Platform 2.0* je pronašao minimalni skup od tri implikacije atributa za formalni kontekst iz primjera 3.1. Njihova pouzdanost je prema definiciji 3.7 $conf(A \Rightarrow B) = 100\%$, a za sve tri implikacije atributa izračunata je podrška od 50%. To su redom:

- $\{\text{UUPRO}\} \Rightarrow \{\text{DIGLOG}, \text{VJEKO}\}$, odnosno svi studenti koji su položili *Uvod u programiranje* položili su i *Digitalnu logiku* i *Vještine komuniciranja*
- $\{\text{LINALG}\} \Rightarrow \{\text{VJEKO}\}$, tj. svi studenti koji su položili *Linearnu algebru* položili su i *Vještine komuniciranja*
- $\{\text{DIGLOG}, \text{VJEKO}\} \Rightarrow \{\text{UUPRO}\}$, dakle ovdje vrijedi i obrat prve implikacije – svi studenti koji su položili *Digitalnu logiku* i *Vještine komuniciranja* položili su i *Uvod u programiranje*

Primjenom Armstrongovih aksioma i pravila nad pronađenom minimalnom bazom implikacija mogu se dobiti i ostale implikacije atributa sa slike 3.2. Npr., 12. implikacija atributa $\{\text{LINALG}, \text{MATAN1}\} \Rightarrow \{\text{VJEKO}\}$ izvodi se iz implikacije atributa $\{\text{LINALG}\} \Rightarrow \{\text{VJEKO}\}$ primjenom aksioma refleksivnosti te aksioma tranzitivnosti:

1. vrijedi $\text{LINALG} \subseteq \text{LINALG} \cup \text{MATAN1}$ pa prema aksiomu refleksivnosti (ako $Y \subseteq X$ onda $X \to Y$) izvodi se sljedeća implikacija atributa: $\{\text{LINALG}, \text{MATAN1}\} \Rightarrow \{\text{LINALG}\}$
2. iz izvedene implikacije atributa $\{\text{LINALG}, \text{MATAN1}\} \Rightarrow \{\text{LINALG}\}$ i zadane implikacije atributa $\{\text{LINALG}\} \Rightarrow \{\text{VJEKO}\}$ primjenom aksioma tranzitivnosti (ako $X \to Y$ i $Y \to Z$ onda $X \to Z$) izvodi se tražena implikacija atributa: $\{\text{LINALG}, \text{MATAN1}\} \Rightarrow \{\text{VJEKO}\}$

3.1.2 Matematički temelji metode FCA

Nakon izloženog uvoda u metodu FCA u ovom potpoglavlju će se dati sažeti pregled njenih matematičkih temelja prema izvorima u [9, 10, 11, 50, 61, 62]. Metoda FCA je utemeljena na matematičkoj teoriji poretka (engl. *Order theory*) koja se bavi s uređenim skupovima, a s posebnim naglaskom na potpune rešetke (engl. *Complete lattice*). Ključni mehanizmi metode FCA putem kojih se stvaraju formalni koncepti su Galoisove[2] veze (engl. *Galois connections*) i operatori zatvaranja (engl. *Closure operators*).

Definicija 3.8. Binarna relacija R između skupova A i B je skup parova (a,b) gdje je $a \in A$ i $b \in B$. Uz zapis binarne relacije $(a,b) \in R$ može se koristiti i kraći zapis u obliku aRb. Za inverznu binarnu relaciju R^{-1} vrijedi $bR^{-1}a :\Leftrightarrow aRb$. Ako vrijedi $B = A$, onda je riječ o binarnoj relaciji R nad skupom A.

Definicija 3.9. Binarna relacija R nad skupom A naziva se relacija uređaja (poretka) ako su zadovoljeni sljedeći uvjeti za sve elemente $x, y, z \in A$:

- xRx (refleksivnost)
- ako je xRy i $x \neq y$ onda ne vrijedi yRx (antisimetričnost)
- ako su xRy i yRz onda je i xRz (tranzitivnost)

Binarna relacija R će se označavati sa simbolom $\leq$ (manje ili jednako). Inverzna relacija R^{-1} označava se sa simbolom $\geq$ (veće ili jednako).

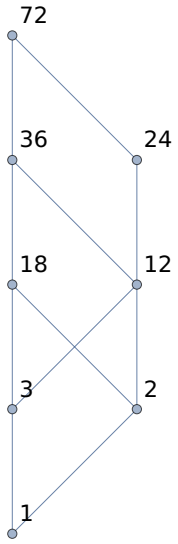
Definicija 3.10. Parcijalno uređeni skup je par $(A, \leq)$, gdje je A skup, a $\leq$ relacija uređaja ili poretka nad skupom A.

Obično se koristi izraz parcijalno uređeni skup (engl. *poset*) jer u općem slučaju ne moraju svi elementi skupa A biti međusobno usporedivi s relacijom $\leq$. U slučaju kada su svi elementi

[2] Évariste Galois (1811 – 1832), francuski matematičar

skupa A međusobno usporedivi s relacijom $\leq$ onda se radi o potpuno uređenom skupu ili linearno uređenom skupu.

Uređeni skupovi mogu se prikazati u obliku linijskog dijagrama ili Hasseovog dijagrama[3]. Linijski dijagram je graf u kojem čvorovi predstavljaju elemente skupa A. Čvorovi se ucrtavaju tako da se sačuva relacija uređaja između elemenata skupa. Čvor $y \in A$ se ucrta iznad čvora $x \in A$ te se ta dva čvora povežu s bridom ako vrijedi $x \leq y$ i $x \neq y$, a nema niti jednog elementa $z \in A$ za koji bi vrijedilo $x \leq z \leq y$. Kao primjer se na slici 3.3 prikazuje linijski dijagram parcijalno uređenog skupa $(\{72,36,24,18,12,3,2,1\},\leq)$ gdje $x \leq y$ znači da je y djeljivo s x bez ostatka[4]. Iz ovog linijskog grafa se vizualno može iščitati kako su svi elementi skupa $\{72,36,24,18,12,3,2,1\}$ djeljivi s 1, a npr. 24 je djeljivo još i s 12, 3 i 2.



Slika 3.3: Primjer linijskog dijagrama

Definicija 3.11. Neka je $(X,\leq)$ parcijalno uređeni skup, a skup Y je $Y \subseteq X$. Donja granica skupa Y je element $s \in X$ za koji vrijedi $s \leq y$ za sve $y \in Y$. Ako postoji najveći element skupa svih donjih granica od Y onda se on naziva infimum od Y i označava se s $\inf Y$ ili $\bigwedge Y$. Gornja granica skupa Y je element $s \in X$ za koji vrijedi $s \geq y$ za sve $y \in Y$. Ako postoji najmanji element skupa svih gornjih granica od Y onda se on naziva supremum od Y i označava se sa $\sup Y$ ili $\bigvee Y$.

Može se vidjeti iz slike 3.3 kako podskup $\{36,24\}$ ima supremum 72 te infimum 12. Treba primijetiti kako podskup $\{3,2\}$ ima infimum 1, ali nema supremum jer nema najmanjeg elementa u skupu gornjih granica $\{18,12\}$. Naime, u ovom slučaju 18 i 12 su neusporedivi jer nisu međusobno djeljivi bez ostatka. Isto tako podskup $\{18,12\}$ ima supremum 36, ali nema infi-

[3] Helmut Hasse (1898 – 1979), njemački matematičar

[4] Linijski dijagram na slici 3.3 dobiven je u programu *Mathematica* [63] sa sljedećom naredbom:
`ResourceFunction["HasseDiagram"][Mod[#1, #2] == 0 & , {72,36,24,18,12,3,2,1}, VertexLabels -> "Name", EdgeShapeFunction -> "Line", VertexLabelStyle -> 18]`

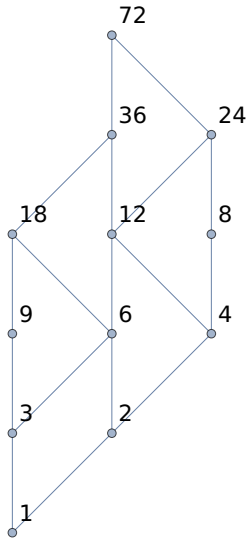
mum jer nema najvećeg elementa u skupu donjih granica $\{3,2\}$ (opet se radi o neusporedivim elementima jer nisu međusobno djeljivi bez ostatka).

Definicija 3.12. Parcijalno uređeni skup $\mathbf{L} := (L, \leq)$ je rešetka, ako za bilo koji par elemenata $x \in L$ i $y \in L$ postoji supremum i infimum.

Primjer rešetke je $(\mathbb{N}, \leq)$, parcijalno uređeni skup prirodnih brojeva i relacije uređaja $\leq$ s uobičajenim značenjem manje ili jednako. S druge strane parcijalno uređeni skup prikazan na slici 3.3 nije rešetka jer kao što je već spomenuto postoje parovi elemenata koji nemaju supremum ili infimum.

Definicija 3.13. Parcijalno uređeni skup $\mathbf{L} := (L, \leq)$ je potpuna rešetka, ako za bilo koji podskup elemenata iz L postoji supremum i infimum. Svaka potpuna rešetka $\mathbf{L}$ ima najveći element $\bigvee L$ te najmanji element $\bigwedge L$.

Kako svaki podskup elemenata X iz L ima supremum i infimum to mora vrijediti i za $X = \emptyset$ – infimum praznog skupa je najveći element u potpunoj rešetci $\mathbf{L}$: $\bigwedge \emptyset = \bigvee L$, a supremum praznog skupa je najmanji element u potpunoj rešetci $\mathbf{L}$: $\bigvee \emptyset = \bigwedge L$. Ranije spomenuta rešetka $(\mathbb{N}, \leq)$ nije i potpuna rešetka jer nema gornju granicu, odnosno najveći element. S druge strane, primjer potpune rešetke je parcijalni uređeni skup zatvorenog intervala prirodnih brojeva $([x,y], \leq)$ i relacije uređaja $\leq$ s uobičajenim značenjem manje ili jednako. I svaki partitivni skup[5] s relacijom inkluzije $\subseteq$ predstavlja potpunu rešetku. Na slici 3.4 prikazuje se još jedan primjer potpune rešetke – to je parcijalni uređeni skup $(\{72,36,24,18,12,9,8,6,4,3,2,1\}, \leq)$ gdje $x \leq y$ znači da je y djeljivo s x bez ostatka[6].



Slika 3.4: Primjer linijskog dijagrama potpune rešetke

[5]Partitivni skup (engl. *power set*) $\mathbf{P}(S)$ sadrži sve podskupove zadanog skupa S uz prazni skup $\emptyset$ te sami zadani skup S.

[6]Linijski dijagram na slici 3.4 dobiven je u programu *Mathematica* sa sljedećom naredbom:
`ResourceFunction["HasseDiagram"][Mod[#1, #2] == 0 & , {72,36,24,18,12,9,8,6,4,3,2,1}, VertexLabels -> "Name", EdgeShapeFunction -> "Line", VertexLabelStyle -> 18]`

Za razliku od parcijalno uređenog skupa prikazanog linijskim dijagramom na slici 3.3, ovdje su na slici 3.4 uključeni svi djelitelji broja 72 pa je time moguće naći za svaki podskup elemenata $\{72,36,24,18,12,9,8,6,4,3,2,1\}$ supremum i infimum. Dodatno, u ovom linijskom dijagram postoji najveći i najmanji element pa se radi o potpunoj rešetci.

Definicija 3.14. Neka je $\mathbf{L} := (L, \leq)$ potpuna rešetka. Skup $X \subseteq L$ je supremum-gust (engl. *supremum-dense*) u L, ako svaki element iz L predstavlja supremum nekog podskupa od X, $\forall l \in L : l = \bigvee\{x \in X | x \leq l\}$. Analogno tome, skup $X \subseteq L$ je infimum-gust (engl. *infimum-dense*) u L, ako svaki element iz L predstavlja infimum nekog podskupa od X, $\forall l \in L : l = \bigwedge\{x \in X | l \leq x\}$.

Za potpunu rešetku $(L, \leq)$ na slici 3.4, gdje je $L = \{72,36,24,18,12,9,8,6,4,3,2,1\}$ može se pronaći skup $X = \{9,8,4,3,2,1\}$ koji je supremum-gust u L. Dakle, svaki element iz L je supremum nekog podskupa od X, za što je dovoljno pokazati sljedeće: $\bigvee\{\emptyset\} = 1$, $\bigvee\{1,2\} = 2$, $\bigvee\{1,3\} = 3$, $\bigvee\{2,4\} = 4$, $\bigvee\{2,3\} = 6$, $\bigvee\{4,8\} = 8$, $\bigvee\{3,9\} = 9$, $\bigvee\{3,4\} = 12$, $\bigvee\{2,9\} = 18$, $\bigvee\{3,8\} = 24$, $\bigvee\{4,9\} = 36$ i $\bigvee\{8,9\} = 72$.

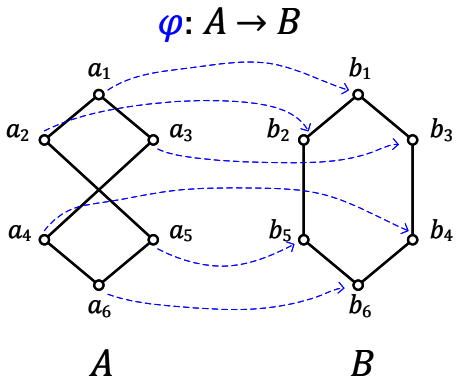
Definicija 3.15. Operator zatvaranja φ nad skupom X je preslikavanje kojim se pridaje zatvarač (engl. *closure*) $\varphi A \subseteq X$ za svaki podskup $A \subseteq X$ uz zadovoljene sljedeće uvjete:

1. $A \subseteq B \Rightarrow \varphi A \subseteq \varphi B$ (monotonost)
2. $A \subseteq \varphi A$ (ekstenzivnost)
3. $\varphi\varphi A = \varphi A$ (idempotentnost)

Skup X je zatvoren s obzirom na operaciju zatvaranja ako se njom iz nekog podskupa od X opet dobije neki podskup elemenata od X. Npr. skup prirodnih brojeva $\mathbb{N}$ je zatvoren s obzirom na operaciju zbrajanja, ali nije zatvoren s obzirom na operaciju oduzimanja jer se s njom može dobiti kao rezultat cijeli broj iz skupa $\mathbb{Z}$.

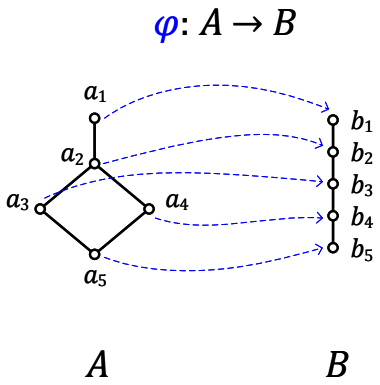
Definicija 3.16. Neka je $\varphi : P \to Q$ preslikavanje između parcijalno uređenih skupova $(P, \leq)$ i $(Q, \leq)$. Ako za sve elemente $x, y \in P$ vrijedi $x \leq y \Rightarrow \varphi x \leq \varphi y$ onda je φ preslikavanje koje čuva poredak. Preslikavanje koje čuva poredak $\varphi : P \to Q$ je izomorfizam ako postoji i njeno inverzno preslikavanje $\varphi^{-1} : Q \to P$ koje također čuva poredak. U tom slučaju su parcijalno uređeni skupovi $(P, \leq)$ i $(Q, \leq)$ izomorfni $(P \cong Q)$.

Na slici 3.5 prikazan je primjer izomorfnog preslikavanja $\varphi : A \to B$ između parcijalno uređenih skupova $(A, \leq)$ i $(B, \leq)$. Može se primijetiti kako je izomorfno preslikavanje bijektivno jer se svaki element iz A preslika u točno jedan element iz B i obratno, a povrh toga izomorfno preslikavanje u oba smjera čuva i poredak. Npr. ovdje vrijedi $a_5 \leq a_2 \Rightarrow \varphi a_5 \leq \varphi a_2$ odnosno $a_5 \leq a_2 \Rightarrow b_5 \leq b_2$, a vrijedi i $b_5 \leq b_2 \Rightarrow \varphi^{-1} b_5 \leq \varphi^{-1} b_2$ odnosno $b_5 \leq b_2 \Rightarrow a_5 \leq a_2$. Dakle, izomorfni parcijalno uređeni skupovi poput potpunih rešetki mogu se prikazati na jednak način jer im je struktura ista, a isti im je i poredak među elementima.



Slika 3.5: Primjer izomorfnog preslikavanja

Primjer preslikavanja koje nije izomorfno prikazan je na slici 3.6. Ovo preslikavanje $\varphi : A \to B$ između parcijalno uređenih skupova $(A, \leq)$ i $(B, \leq)$ čuva poredak i bijektivno je, ali nije izomorfizam. Svaki element iz A se preslika u jedan element iz B i obratno, ali kod preslikavanja nije sačuvan poredak u oba smjera. Naime, može se pokazati kako ne postoji inverzno preslikavanje koje čuva poredak $\varphi^{-1} : B \to A$ jer vrijedi $b_4 \leq b_3$, ali ne vrijedi $\varphi^{-1} b_4 \leq \varphi^{-1} b_3$ jer a_4 i a_3 nisu međusobno usporedivi.



Slika 3.6: Primjer neizofmorfnog preslikavanja

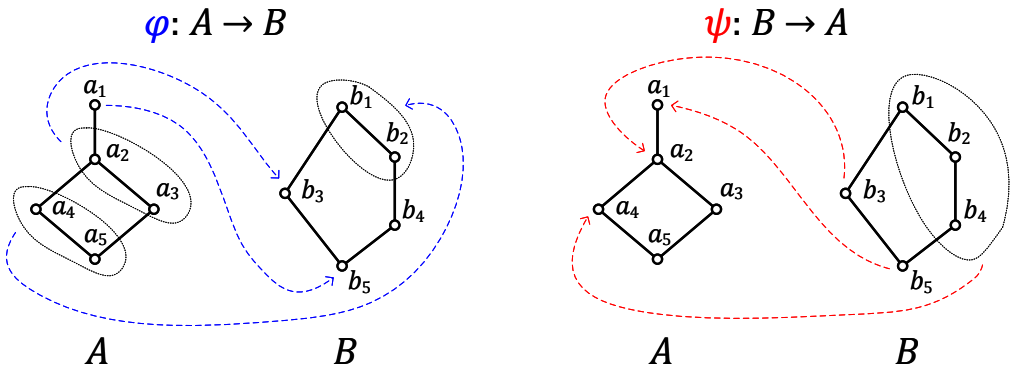
Definicija 3.17. Neka su $(P, \leq)$ i $(Q, \leq)$ parcijalno uređeni skupovi, a $\varphi : P \to Q$ i $\psi : Q \to P$ preslikavanja između njih. Par preslikavanja (φ, ψ) se naziva Galoisova veza između parcijalno uređenih skupova ako za sve elemente $p \in P$ i $q \in Q$ vrijedi sljedeće:

1. $p_1 \leq p_2 \Rightarrow \varphi p_1 \geq \varphi p_2$
2. $q_1 \leq q_2 \Rightarrow \psi q_1 \geq \psi q_2$
3. $p \leq \psi\varphi p$ i $q \leq \varphi\psi q$

Iz definicije 3.17 se izvodi i sljedeća tvrdnja o postojanju Galoisove veze. Par preslikavanja (φ, ψ) je Galoisova veza ako i samo ako vrijedi za sve elemente $p \in P$ i $q \in Q$:

$$p \leq \psi q \Leftrightarrow q \leq \varphi p \tag{3.3}$$

Primjer Galoisove veze (φ, ψ) kao preslikavanja između dva parcijalno uređena skupa A i B (dvije potpune rešetke u obliku linijskih dijagrama) prikazan je na slici 3.7.



Slika 3.7: Primjer Galoisove veze

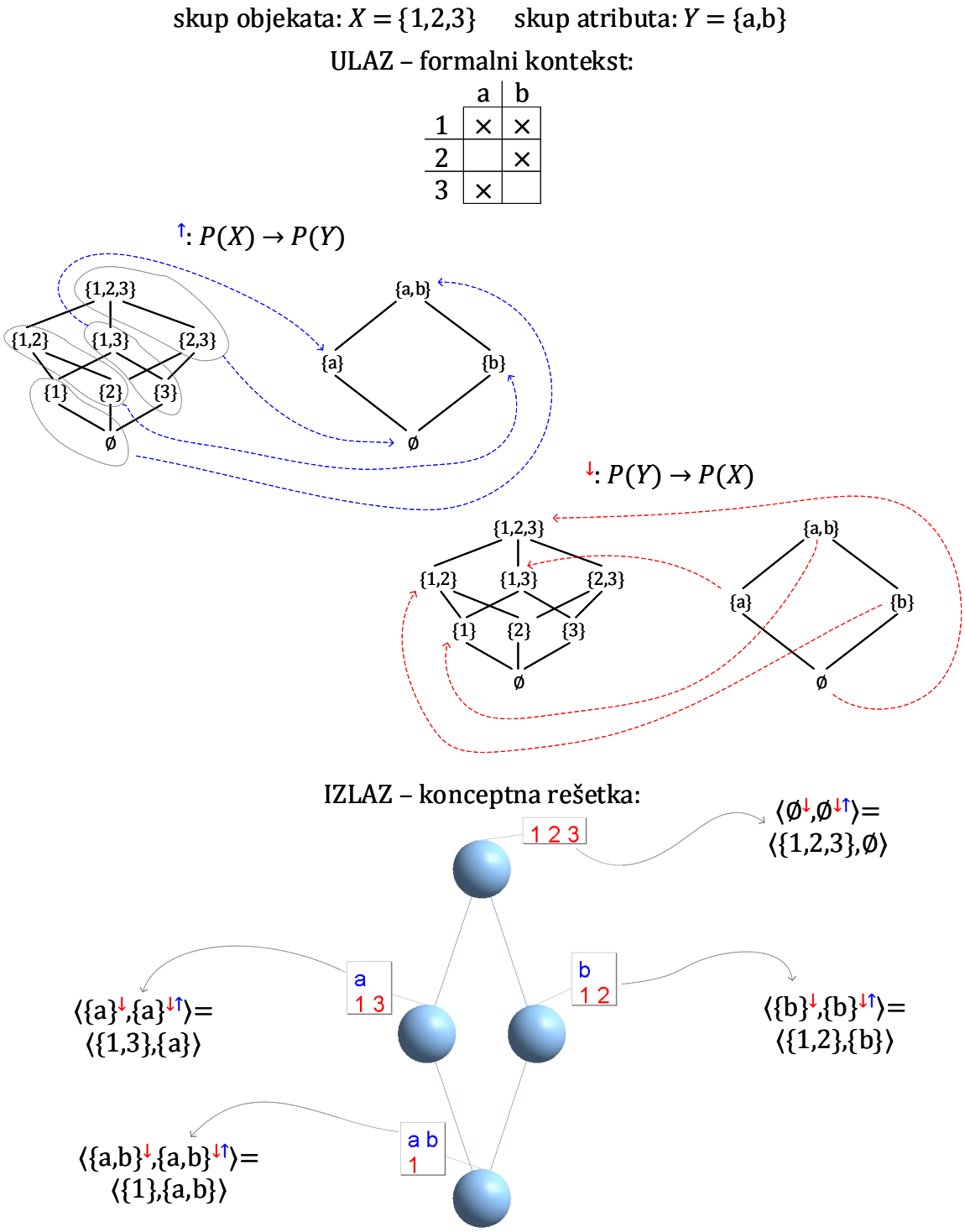
Kao primjer provjerit će se svaka tvrdnja iz definicije 3.17 za proizvoljne elemente iz A i B sa slike 3.7, ali naravno za potpunu provjeru treba ispitati sve elemente $a \in A$ i $b \in B$. Npr. ako je $a_5 \leq a_2$, onda vrijedi $\varphi a_5 \geq \varphi a_2$, odnosno $\{b_1, b_2\} \geq b_3$. Nadalje, ako je $b_3 \leq b_1$, onda vrijedi i $\psi b_3 \geq \psi b_1$, tj. $a_1 \geq a_4$. Za treću tvrdnju vidimo da vrijedi npr. $a_4 \leq \psi\varphi a_4 = a_4 \leq \psi\{b_1, b_2\} = a_4 \leq a_4$ te isto tako npr. $b_2 \leq \varphi\psi b_2 = b_2 \leq \varphi a_4 = b_2 \leq \{b_1, b_2\}$. Na drugi način može se provjeriti radi li se doista o Galoisovoj vezi provjerom ekvivalencije $a \leq \psi b \Leftrightarrow b \leq \varphi a$. Ovdje će se kao primjer proizvoljno odabrati elementi a_2 i b_3. Prvo se traži $\psi b_3 = \{a_2, a_3\}$, dakle vrijedi $a_2 \leq \{a_2, a_3\}$. Potom se traži $\varphi a_2 = b_3$ i vidi se da onda isto vrijedi $b_3 \leq b_3$.

Treba se prisjetiti kako su ulazni podaci za metodu FCA oblikovani kao formalni kontekst koji objedinjuje skup objekata X i skup atributa Y kao i binarnu relaciju između njih. Između partitivnog skupa objekata $\mathbf{P}(X)$ te partitivnog skupa atributa $\mathbf{P}(Y)$ mogu se zadati preslikavanja φ kojim se svaki podskup objekata iz X preslikati u neki podskup atributa iz Y te ψ kojim se svaki podskup atributa iz Y preslika u neki podskup objekata iz X. Ako za par preslikavanja (φ, ψ) vrijede tvrdnje iz definicije 3.17 (uz korištenje relacije inkluzije $\subseteq$ umjesto relacije uređaja $\leq$) onda se radi o Galoisovoj vezi između skupova X i Y. Dakle, skupovi objekata X i atributa Y iz formalnog konteksta $\langle X, Y, I \rangle$ povezani su Galoisovom vezom, koju predstavlja par operatora formiranja koncepata $(^{\uparrow}, ^{\downarrow})$: $^{\uparrow} : 2^X \to 2^Y$ i $^{\downarrow} : 2^Y \to 2^X$. Operatori zatvaranja su definirani kao kompozicije operatora za formiranje koncepata, $^{\uparrow\downarrow} : 2^X \to 2^X$ (podskup objekata se prvo preslikava u odgovarajući podskup zajedničkih atributa, a potom se taj podskup atributa preslikava natrag u podskup objekata koji dijele te atribute) i analogno tome $^{\downarrow\uparrow} : 2^Y \to 2^Y$ (iz podskupa atributa preslikava se u odgovarajući podskup objekata koji dijele te atribute i zatim natrag u odgovarajući podskup njihovih zajedničkih atributa). Dakle, pomoću Galoisove veze – operatora formiranja koncepata $^{\uparrow}$ i $^{\downarrow}$ te operatora zatvaranja $^{\uparrow\downarrow}$ i $^{\downarrow\uparrow}$ generiraju se formalni koncepti na sljedeći način:

•iz svakog podskupa objekata $A \subseteq X$ može se dobiti neki formalni koncept $\langle A^{\uparrow\downarrow}, A^{\uparrow} \rangle$

•te iz svakog podskupa atributa $B \subseteq X$ može se dobiti neki formalni koncept $\langle B^{\downarrow}, B^{\downarrow\uparrow} \rangle$

Prema definiciji 3.4 se pronađeni formalni koncepti mogu dovesti u hijerarhijski odnos potkoncept – natkoncept. Potom, prema definiciji 3.5 skup svih pronađenih formalnih koncepata uz uključenu hijerarhijsku relaciju $\leq$ čini konceptualnu rešetku. Sve prethodno rečeno sumirano je kroz primjer na slici 3.8 kojim se ilustrira provođenje metode FCA.



Slika 3.8: Prikaz provođenja metode FCA

Na slici 3.8 se vidi kako se prema podacima u formalnom kontekstu $\langle X, Y, I \rangle$ uspostavlja Galoisova veza u obliku preslikavanjâ između partitivnih skupova objekata i atributa. Npr. podskupovi objekata $\{1,3\}$ i $\{3\}$ preslikavaju se u podskup atributa $\{\mathrm{a}\}$, a podskup objekata $\{1\}$ kao i prazni skup objekata $\emptyset$ preslikava se u podskup svih atributa $\{\mathrm{a},\mathrm{b}\}$. S druge strane se podskup atributa $\{\mathrm{a}\}$ preslikava u podskup objekata $\{1,3\}$, a prazni skup atributa $\emptyset$ u podskup svih objekata $\{1,2,3\}$. Primjenom operatora zatvaranja $^{\downarrow\uparrow}$ na svaki element skupa atributa $y \in Y$ atributa generiraju se ukupno četiri formalna koncepta u obliku $\langle y^{\downarrow}, y^{\downarrow\uparrow} \rangle$, a prikazuju se u obliku konceptualne rešetke.

Nakon ovoga sažetog prikaza matematičkih temelja metode FCA može se dati i iskaz njenog glavnog teorema. Teorem je objavio R. Wille u [9], a njegov dokaz se može naći u [9, 10, 50].

Teorem 3.1. *Osnovni teorem o konceptualnim rešetkama*
Konceptualna rešetka $\underline{\mathfrak{B}}(X,Y,I)$ je potpuna rešetka u kojoj je svaki podskup formalnih koncepata $\langle A_t, B_t \rangle$ supremum zadan s $\vee_{t\in T}\langle A_t, B_t\rangle = \langle (\cup_{t\in T} A_t)^{\uparrow\downarrow}, \cap_{t\in T} B_t \rangle$, a infimum s $\wedge_{t\in T}\langle A_t, B_t\rangle = \langle \cap_{t\in T} A_t, (\cup_{t\in T} B_t)^{\downarrow\uparrow} \rangle$. Potpuna rešetka $\mathbf{V}$ je izomorfna konceptualnoj rešetci $\underline{\mathfrak{B}}(X,Y,I)$ ako i samo ako postoje preslikavanja $\gamma : X \to V$ i $\mu : Y \to V$ takva da je $\gamma(X)$ supremum-gusto u $\mathbf{V}$ i $\mu(Y)$ infimum-gusto u $\mathbf{V}$ te $(x,y) \in I \Leftrightarrow \gamma x \leq \mu y$ za sve $x \in X$ i sve $y \in Y$.

Teorem će se razjasniti na primjeru jednostavne konceptualne rešetke na dnu slike 3.8. Neka se traži se supremum i infimum formalnih koncepata $\langle \{1,3\}, \{\mathrm{a}\} \rangle$, $\langle \{1,2\}, \{\mathrm{b}\} \rangle$ i $\langle \{1\}, \{\mathrm{a},\mathrm{b}\} \rangle$. Njihov supremum je $\langle (\{1,2\} \cup \{1,3\} \cup \{1\})^{\uparrow\downarrow}, \{\mathrm{a}\} \cap \{\mathrm{b} \cap \{\mathrm{a},\mathrm{b}\} \rangle = \langle \{1,2,3\}^{\uparrow\downarrow}, \emptyset \rangle = \langle \emptyset^{\downarrow}, \emptyset \rangle = \langle \{1,2,3\}, \emptyset \rangle$. S druge strane infimum im je $\langle \{1,2\} \cap \{1,3\} \cap \{1\}, (\{\mathrm{a}\} \cup \{\mathrm{b}\} \cup \{\mathrm{a},\mathrm{b}\})^{\downarrow\uparrow} \rangle = \langle \{1\}, \{\mathrm{a},\mathrm{b}\}^{\downarrow\uparrow} \rangle = \langle \{1\}, \{1\}^{\uparrow} \rangle = \langle \{1\}, \{\mathrm{a},\mathrm{b}\} \rangle$. Ovim se potvrđuje automatski generirana konceptualna rešetka sa slike 3.8 jer tri odabrana formalna koncepta imaju upravo supremum $\langle \{1,2,3\}, \emptyset \rangle$ te infimum $\langle \{1\}, \{\mathrm{a},\mathrm{b}\} \rangle$. Kao što je ranije spomenuto kod definicije potpune rešetke i prazan podskup formalnih koncepata iz konceptualne rešetke $\underline{\mathfrak{B}}(X,Y,I)$ ima svoj infimum koji odgovara formalnom konceptu na vrhu te supremum koji odgovara formalnom konceptu na dnu.

Za komentar drugog dijela teorema razmotrit će se posebni slučaj kada je $\mathbf{V} = \underline{\mathfrak{B}}(X,Y,I)$. Tada se definiraju preslikavanja iz objekta u formalni koncept, $\gamma x : \langle \{x\}^{\uparrow\downarrow}, \{x\}^{\uparrow} \rangle$ za svaki $x \in X$ te preslikavanje iz atributa u formalni koncept, $\mu y : \langle \{y\}^{\downarrow}, \{y\}^{\downarrow\uparrow} \rangle$ za svaki $y \in Y$. Dakle, svi dobiveni elementi γx (odnosno μy) čine podskup svih formalnih koncepata iz $\underline{\mathfrak{B}}(X,Y,I)$. Onda svaki formalni koncept iz $\underline{\mathfrak{B}}(X,Y,I)$ predstavlja supremum nekog podskupa elemenata γx pa je γx supremum-gust u $\underline{\mathfrak{B}}(X,Y,I)$. Analogno tome svaki formalni koncept iz $\underline{\mathfrak{B}}(X,Y,I)$ predstavlja infimum nekog podskupa elemenata μy pa je μy infimum-gust u $\underline{\mathfrak{B}}(X,Y,I)$. Na kraju se može pokazati da vrijedi ekvivalencija $(x,y) \in I \Leftrightarrow \gamma x \leq \mu y$. Primjerice ako se iz formalnog konteksta na slici 3.8 uzmu elementi $x = 2$ i $y = \mathrm{b}$ onda je $(2,\mathrm{b}) \in I$, $\gamma x = \langle \{2\}^{\uparrow\downarrow}, \{2\}^{\uparrow} \rangle = \langle \{\mathrm{b}\}^{\downarrow}, \{\mathrm{b}\} \rangle = \langle \{1,2\}, \{\mathrm{b}\} \rangle$, a $\mu y = \langle \{\mathrm{b}\}^{\downarrow}, \{\mathrm{b}\}^{\downarrow\uparrow} \rangle = \langle \{1,2\}, \{1,2\}^{\uparrow} \rangle = \langle \{1,2\}, \{\mathrm{b}\} \rangle$ pa vrijedi

$(x,y) \in I \Leftrightarrow \gamma x \leq \mu y$. U slučaju da ne vrijedi $(x,y) \in I$ onda ne vrijedi ni $\gamma x \leq \mu y$ što može pokazati odabirom $x = 2$ i $y =$ a u primjeru na slici 3.8.

3.1.3 Sažimanje formalnog konteksta

Obično nisu svi ulazni podaci za metodu FCA jedinstveni, odnosno različiti redci (ili stupci) u formalnom kontekstu često sadrže iste podatke. Isto tako može se dogoditi da se neki redak (ili stupac) može prikazati kao kombinacija drugih redaka (ili stupaca). Takvi suvišni podaci ne utječu na krajnji oblik konceptualne rešetke pa se mogu slobodno ukloniti iz formalnog konteksta. Tako se smanjuje volumen ulaznih podataka i potencijalno ubrzava proces izgradnje konceptualne rešetke. U nastavku će se opisati postupci sažimanja formalnog konteksta – raščišćavanje i reduciranje [10, 11, 50].

Raščišćavanje formalnog konteksta

Formalni kontekst se može raščistiti (engl. *Context clarification*) tako da se redci s identičnim sadržajem spoje u jedan redak, odnosno da se svi objekti koji dijele iste atribute spoje u jedan atribut. Analogno tome i stupci s identičnim sadržajem se mogu spojiti u jedan stupac, odnosno atributi koje dijele isti objekti spoje se u zajednički atribut. Ovime će u formalnom kontekstu svaki objekt imati svoj jedinstveni skup atributa (jedinstvenu intenciju), a svaki atribut dijeliti jedinstveni skup objekata (jedinstvenu ekstenziju).

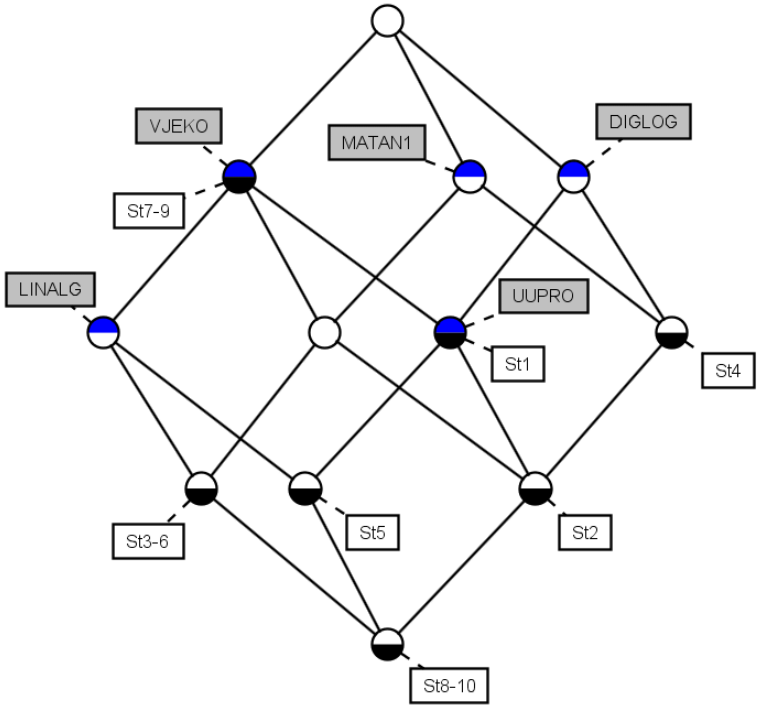
Treba naglasiti da će se nakon postupka raščišćavanja formalnog konteksta dobiti konceptualna rešetka koja je izomorfna konceptualnoj rešetci izvornog formalnog konteksta, dakle strukturno su istovjetne i u obije su sačuvani svi hijerarhijski odnosi. Za primjer raščistit će se formalni kontekst $\langle X, Y, I \rangle$ iz tablice 3.1 gdje je $X = \{\text{St1}, \text{St2}, \text{St3}, \text{St4}, \text{St5}, \text{St6}, \text{St7}, \text{St8}, \text{St9}, \text{St10}\}$ i $Y = \{\text{DIGLOG}, \text{LINALG}, \text{MATAN1}, \text{UUPRO}, \text{VJEKO}\}$:

- •studenti St3 i St6 koji imaju atribute $\{\text{LINALG}, \text{MATAN1}, \text{VJEKO}\}$ spoje se u St3-6
- •studenti St7 i St9 koji imaju samo atribut $\{\text{VJEKO}\}$ spoje se u St7-9
- •studenti St8 i St10 koji imaju sve atribute Y spoje se u St8-10

Tablica 3.2: Raščišćeni formalni kontekst iz tablice 3.1

	DIGLOG	LINALG	MATAN1	UUPRO	VJEKO
St1	X			X	X
St2	X		X	X	X
St3-6		X	X		X
St4	X		X		
St5	X	X		X	X
St7-9					X
St8-10	X	X	X	X	X

Ovako raščišćeni formalni kontekst je prikazan u tablici 3.2, a vidi se da broj objekata smanjen s 10 na 7, dok i dalje ima 5 atributa. Na slici 3.9 prikazana je i njegova konceptualna rešetka koja je izomorfna konceptualnoj rešetci sa slike 3.1 izvornog formalnog konteksta iz tablice 3.1.



Slika 3.9: Konceptualna rešetka za raščišćeni formalni kontekst iz tablice 3.2

Reduciranje formalnog konteksta

Reduciranje formalnog konteksta $\langle X, Y, I \rangle$ je složeniji postupak za sažimanje ulaznih podataka u metodi FCA. Provodi se nakon prethodnog raščišćavanja formalnog konteksta. Reduciranjem formalnog konteksta se iz njega brišu svi objekti (atributi) koji se mogu dobiti kao kombinacija ostalih objekata (atributa). Preciznije rečeno svaki stupac koji se može dobiti presjekom nekih drugih stupaca se uklanja iz formalnog konteksta, a analogno tome se iz formalnog konteksta briše i svaki redak koji je jednak presjeku dva ili više preostala redaka. Dodatno se iz formalnog konteksta mogu ukloniti svi popunjeni redci ili stupci. Naime, svaki popunjeni redak predstavlja objekt koji ima sve atribute iz Y, a već znamo da će pripadati formalnom konceptu na dnu konceptualne rešetke $\langle Y^{\downarrow}, Y \rangle$ jer on obuhvaća objekte koji imaju sve atribute. Nadalje, svaki popunjeni stupac predstavlja atribut kojega dijele svi objekti iz X te za njega znamo da će pripadati formalnom konceptu na vrhu konceptualne rešetke $\langle X, X^{\uparrow} \rangle$ koji obuhvaća sve objekte i njihove zajedničke atribute.

Iz reduciranog formalnog konteksta dobit će se konceptualna rešetka koja je opet izomorfna izvornoj konceptualnoj rešetci kao i konceptualnoj rešetci raščišćenog formalnog konteksta. S obzirom na to da je postupak reduciranja formalnog konteksta složeniji svakako se preporučuje korištenje programskih alata koji automatski provode taj postupak (npr. *ConExp - Concept*

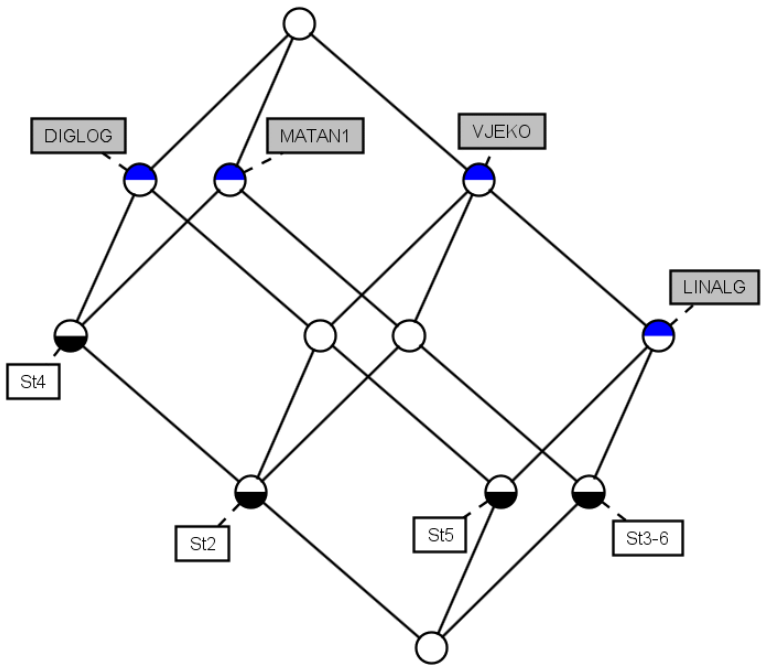
Explorer 1.3). Kao primjer prikazat će se nastavak sažimanja formalnog konteksta iz tablice 3.1. Sada će se reducirati već raščišćeni formalni kontekst iz tablice 3.2:

- briše se atribut UUPRO jer se taj stupac može dobiti kao presjek stupaca atributa DIGLOG i VJEKO: $\text{UUPRO}^{\downarrow} = \text{DIGLOG}^{\downarrow} \cap \text{VJEKO}^{\downarrow} = \{\text{St1}, \text{St2}, \text{St5}, \text{St8-10}\}$
- briše se objekt St7-9 jer se taj redak može dobiti presjekom redaka objekta St1 i St3-6: $\text{St7-9}^{\uparrow} = \text{St1}^{\uparrow} \cap \text{St3-6}^{\uparrow} = \{\text{VJEKO}\}$
- briše se objekt St1 jer se taj redak može dobiti presjekom redaka objekta St2 i St5: $\text{St1}^{\uparrow} = \text{St2}^{\uparrow} \cap \text{St5}^{\uparrow} = \{\text{DIGLOG}, \text{UUPRO}, \text{VJEKO}\}$
- uklanja se objekt St8-10 jer je taj redak popunjen odnosno ima sve atribute iz Y

Reducirani formalni kontekst je prikazan u tablici 3.3, a dodatno je sažet u odnosu na raščišćeni formalni kontekst iz tablice 3.2. Sada ima samo 4 objekta te 4 atributa u odnosu na izvornih 10 objekata i 5 atributa. Na kraju je na slici 3.10 prikazana konceptualna rešetka za ovako reducirani formalni kontekst. Može se provjeriti kako i ona ima opet 12 formalnih koncepata i kako je izomorfna izvornoj konceptualnoj rešetci (slika 3.1), baš kao što je izomorfna i konceptualnoj rešetci dobivenoj nakon raščišćavanja formalnog konteksta (slika 3.9).

Tablica 3.3: Reducirani formalni kontekst iz tablice 3.2

	DIGLOG	LINALG	MATAN1	VJEKO
St2	*X*		*X*	*X*
St3-6		*X*	*X*	*X*
St4	*X*		*X*	
St5	*X*	*X*		*X*



Slika 3.10: Konceptualna rešetka za reducirani formalni kontekst iz tablice 3.3

3.1.4 Dodatni načini prikaza konceptualnih rešetki

Treba naglasiti kako pri izgradnji formalnog konteksta (dodavanjem novih objekata i/ili atributa) njegova konceptualna rešetka može brzo rasti – u najgorem slučaju veličina konceptualne rešetke je eksponencijalno veća u odnosu na veličinu formalnog konteksta [64]. Veliki formalni konteksti obično rezultiraju s teško preglednim i nečitkim gustim konceptualnim rešetkama pa čak i kad imaju tek 50 – 100 formalnih koncepata. Jedan način za rješavanje ovoga problema je grupiranje atributa formalnog konteksta u manje podskupove. Ovako se formalni kontekst razdvaja na više manjih dijelova te se za svaki dio može izračunati zasebna manja konceptualna rešetka. Izračunom produkta svih manjih konceptualnih rešetki može se opet dobiti kompletna konceptualna rešetka čitavog formalnog konteksta. Na primjer, neka se raščišćeni formalni kontekst u tablici 3.2 podijeli na dva manja dijela. Prvi dio obuhvaća podskup atributa *Mat* matematičkih predmeta $\{\text{MATAN1}, \text{LINALG}\}$, a drugi dio podskup atributa *Ing* inženjerskih predmeta $\{\text{DIGLOG}, \text{UUPRO}, \text{VJEKO}\}$ kao što je prikazano u tablicama 3.4 i 3.5.

Tablica 3.4: *Mat* dio formalnog konteksta iz tablice 3.2 (matematički predmeti)

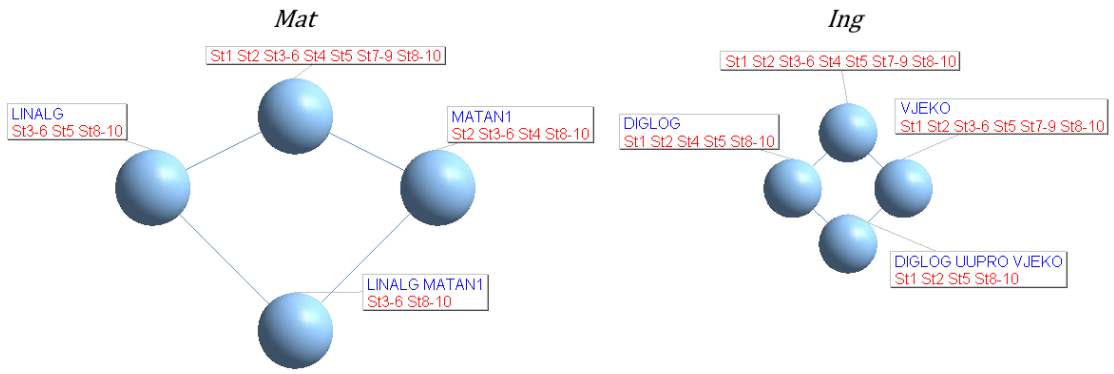
	LINALG	MATAN1
St1		
St2		*X*
St3-6	*X*	*X*
St4		*X*
St5	*X*	
St7-9		
St8-10	*X*	*X*

Tablica 3.5: *Ing* dio formalnog konteksta iz tablice 3.2 (inženjerski predmeti)

	DIGLOG	UUPRO	VJEKO
St1	*X*	*X*	*X*
St2	*X*	*X*	*X*
St3-6			*X*
St4	*X*		
St5	*X*	*X*	*X*
St7-9			*X*
St8-10	*X*	*X*	*X*

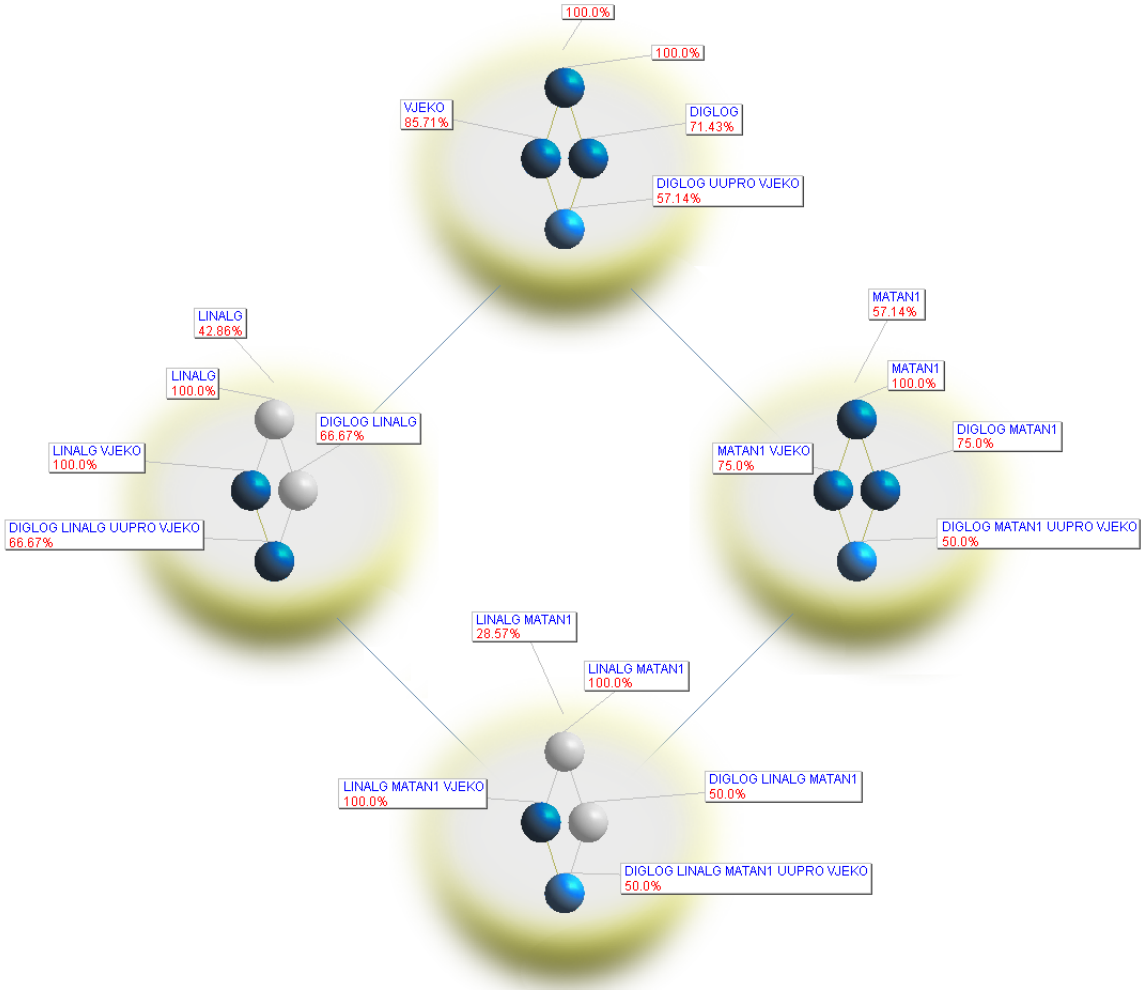
Pomoću alata *Lattice Miner Platform 2.0* generiraju se dvije jednostavnije konceptualne rešetke koje odgovaraju formalnom kontekstu *Mat* odnosno *Ing*, a prikazane su na slici 3.11. Vidi se kako obije konceptualne rešetke imaju po 4 formalna koncepta i izomorfne su iako su im formalni konteksti različiti. Naime, oba se mogu reducirati na istovjetni formalni kontekst s

dva objekta i dva atributa u kojem svaki objekt ima samo po jedan od atributa (oznake "*X*" na dijagonali).



Slika 3.11: Konceptualne rešetke za formalne kontekste iz tablica 3.4 i 3.5

Slika 3.12 prikazuje produkt pojedinačnih konceptualnih rešetki $Mat \times Ing$ tako što je u svakom čvoru konceptualne rešetke Mat ugniježđena je konceptualna rešetka Ing.



Slika 3.12: Produkt konceptualnih rešetki $Mat \times Ing$ sa slike 3.11

U odabranom načinu prikaza se uz svaki formalni koncept ispisuje postotak objekata koji

mu pripadaju. Svi podaci sadržani u ovom produktu konceptualnih rešetki mogu se naći u čitavoj konceptualnoj rešetci sa slike 3.9. Npr. vanjski formalni koncept na dnu (osjenčan zlatnom bojom) obuhvaća objekte koji imaju i atribut LINALG i MATAN1 – to je 28,57% objekata (ili 2 od 7 studenata), a unutar njega je ugniježđena konceptualna rešetka *Ing*. Od ta dva objekta iz vanjskog formalnog koncepta samo jedan objekt pripada formalnom konceptu na dnu ugniježđene konceptualne rešetke – to je student sa zajedničkom oznakom St8-10 koji je položio svih pet predmeta.

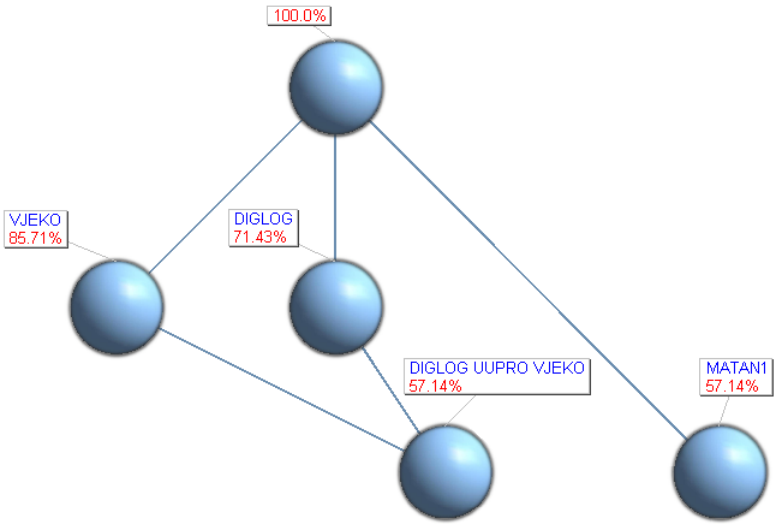
Složene konceptualne rešetke se mogu pojednostavniti i primjenom grupiranja odnosno *klasteriranja* formalnih koncepata. Tako će se dobiti konceptualne rešetke u obliku sante leda (engl. *Iceberg concept lattice*) jer one prikazuju samo vrh čitave konceptualne rešetke, dok je ostatak skriven [64]. U takvim konceptualnim rešetkama prikazuju se samo oni formalni koncepti čiji atributi imaju podršku veću od zadane granične vrijednosti. Svakom promjenom te granične vrijednosti podrške konceptualna rešetka u obliku sante leda se dinamički mijenja – smanjuje se (*sve više tone*) povećanjem granične vrijednosti, a povećava se (*sve više izranja*) smanjivanjem granične vrijednosti.

Podrška nekog skupa atributa $B \subseteq Y$ računa se kao količnik broja objekata koji imaju te atribute i ukupnog broja objekata u zadanom formalnom kontekstu $\langle X, Y, I \rangle$:

$$supp(B) = \frac{|B^{\downarrow}|}{|X|} \tag{3.4}$$

Definicija 3.18. Konceptualna rešetka oblika sante leda za formalni kontekst $\langle X, Y, I \rangle$ i minimalnu vrijednost podrške *minsupp* uključuje skup formalnih koncepata čija je podrška atributa veća ili jednaka od *minsupp*: $\{\langle A, B \rangle \in \underline{\mathfrak{B}}(X, Y, I) | supp(B) \geq minsupp\}$.

Kao primjer na slici 3.13 prikazana je konceptualna rešetka oblika sante leda za već raščišćeni formalni kontekst u tablici 3.2 uz zadanu graničnu vrijednost podrške od $minsupp = 0{,}55$.



Slika 3.13: Konceptualna rešetka oblika sante leda za formalni kontekst u tablici 3.2 (uz $minsupp = 0{,}55$)

Može se primijetiti kako nema niti jednog formalnog koncepta s atributom LINALG jer je taj predmet položilo 3 od 7 studenata (42,86%). Uz $minnsupp = 0{,}4$ pojavila bi se još tri formalna koncepta, a među njima jedan koji uključuje atribut LINALG.

U slučaju da se minimalna vrijednost podrške smanji na nulu ($minsupp = 0$) dobije se opet čitava konceptualna rešetka. Treba primijetiti da konceptualna rešetka oblika sante leda za $minsupp > 0$ općenito nije potpuna rešetka jer joj može nedostajati donja granica kao što se vidi na primjeru na slici 3.13. Preciznije rečeno, konceptualna rešetka oblika sante leda je polurešetka u kojoj svaki neprazni podskup formalnih koncepata ima supremum, ali ne nužno i infimum.

G. Stumme i sur. razvili su algoritam *Titanic* za izgradnju konceptualnih rešetki u obliku sante leda koji se temelji na funkciji izračuna podrške intencije formalnog koncepta [65]. Uz $minsupp = 0$ algoritam može izgraditi čitavu konceptualnu rešetku. Iako ima puno veće memorijske zahtjeve od algoritma *NextClosure* pogodniji je za rad s velikim formalnim kontekstima.

3.1.5 Proširenja metode FCA

Ulazni podaci koji ne mogu biti složeni u binarni formalni kontekst ipak se mogu nakon ručne prilagodbe obraditi metodom FCA. Primjerice, uspjeh studenata na predmetu ne mora biti opisan samo binarnim atributom (npr. položio – da/ne) nego i atributima koji mogu imati kontinuirane vrijednosti (npr. ukupni broj bodova) ili atributima s N diskretnih vrijednosti (npr. konačna ocjena). Takvi ulazni podaci tvore viševrijednosni kontekst koji se prije primjene metode FCA treba transformirati u binarni formalni kontekst [10]. Na primjer, viševrijednosni kontekst u tablici 3.6 opisuje uspjeh studenata na predmetu *Osnove elektrotehnike*, a jedan od mogućih rezultata transformacije u odgovarajući binarni formalni kontekst prikazan je tablicom 3.7. Treba primijetiti kako nakon transformacije može doći do gubitka dijela detaljnih informacija.

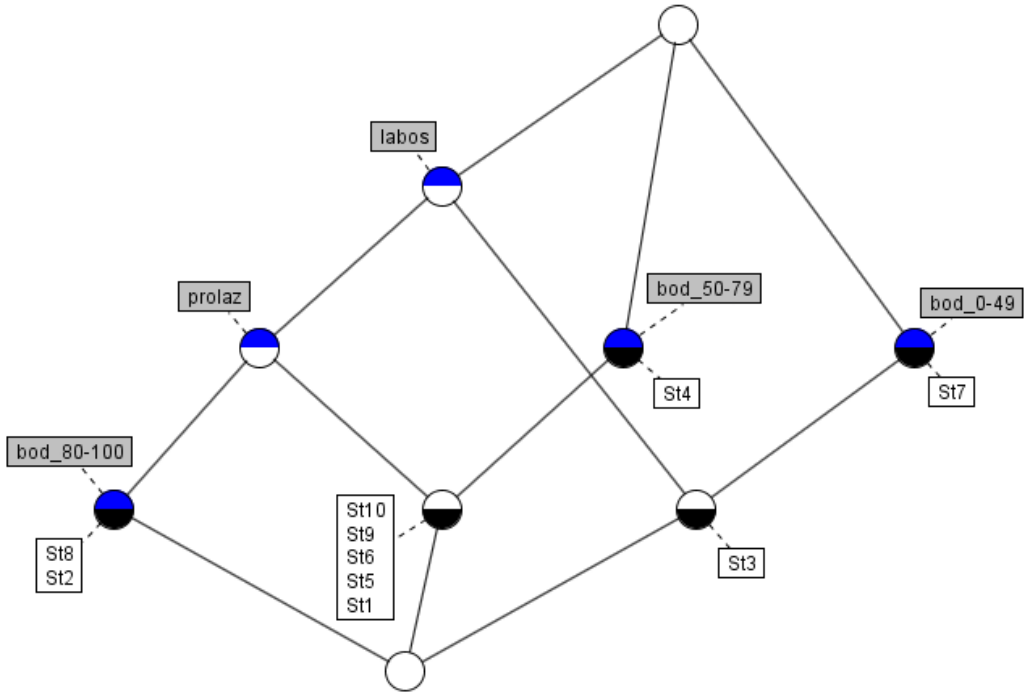
Tablica 3.6: Viševrijednosni kontekst – uspjeh studenata na predmetu *Osnove elektrotehnike*

	bodovi	odrađen_labos	ocjena
St1	77,0	da	4
St2	92,5	da	5
St3	44,5	da	1
St4	50,0	ne	1
St5	65,0	da	3
St6	50,0	da	2
St7	36,0	ne	1
St8	80,5	da	4
St9	71,0	da	3
St10	61,0	da	2

Tablica 3.7: Transformacija viševrijednosnog konteksta iz tablice 3.6 u formalni kontekst

	bod_0-49	bod_50-79	bod_80-100	labos	prolaz
St1		*X*		*X*	*X*
St2			*X*	*X*	*X*
St3	*X*			*X*	
St4		*X*			
St5		*X*		*X*	*X*
St6		*X*		*X*	*X*
St7	*X*				
St8			*X*	*X*	*X*
St9		*X*		*X*	*X*
St10		*X*		*X*	*X*

Nakon ovako provedene transformacije dobije se konceptualna rešetka sa slike 3.14 koja odgovara formalnom kontekstu iz tablice 3.7.



Slika 3.14: Konceptualna rešetka za formalni kontekst iz tablice 3.7

Vidi se kako su studenti grupirani u odgovarajuće formalne koncepte prema odabranim atributima iz formalnog konteksta. Primjerice, može se primijetiti kako je podskup od 5 studenata $\{\text{St1}, \text{St5}, \text{St6}, \text{St9}, \text{St10}\}$ u kojem su svi studenti koji su odradili laboratorijske vježbe te položili predmet s 50 do 79 ostvarenih bodova ili kako je izoliran jedan student (St4) koji je ostvario 50 bodova, ali nije položio predmet jer nije odradio sve laboratorijske vježbe. Kako je već napomenuto kod transformacije viševrijednosnih konteksta obično dolazi do gubitka dijela informacija

i zbog toga se iz transformiranog formalnog konteksta ne može uvijek vratiti u početni viševrijednosni kontekst. U ovom slučaju se iz formalnog konteksta u tablici 3.7 ne može znati npr. koji od studenata $\{\mathrm{St2}, \mathrm{St8}\}$ s više od 80 bodova ima ocjenu 5 jer se taj podatak u odabranom postupku transformacije ispustio.

Na kraju će se ukratko opisati i naprednija proširenja metode FCA kojima se mogu obraditi i ulazni podaci sa složenijim odnosima između objekata i atributa, kao što su trijadička FCA, neizrazita FCA i temporalna FCA.

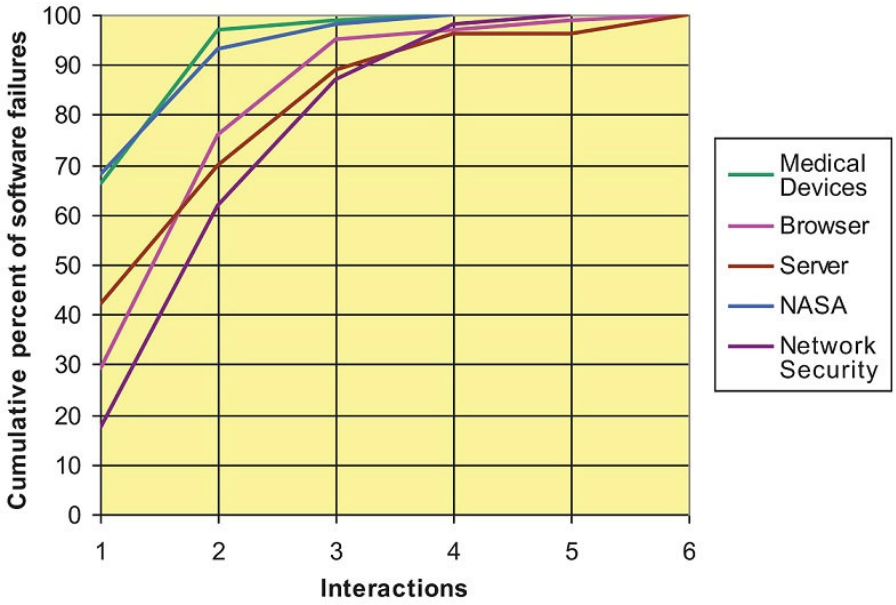
Metoda trijadičke FCA uvodi uz skup objekata X i skup atributa Y još i skup uvjeta Z kao treću dimenziju u formalni kontekst. U trijadičkom formalnom kontekstu definira se ternarna relacija koja određuje ima li objekt x atribut y uz zadani uvjet z. Radi jednostavnijeg prikaza trodimenzionalni trijadički formalni kontekst može se rastaviti na uobičajene dvodimenzionalne (dijadičke) formalne kontekste po svakom zadanom uvjetu [66, 67].

Neizrazita FCA uvodi mogućnost obrade ulaznih podataka kod kojih je odnos objekta $x \in X$ i atributa $y \in Y$ definiran funkcijom $f(x,y)$ s kodomenom realnih brojeva između 0 i 1, što su vrijednosti neizrazite logike (engl. *fuzzy logic*). Takvi neizraziti formalni konteksti mogu se analizirati kao uobičajeni binarni formalni konteksti uz zadanu najmanju graničnu vrijednost T. Pritom samo za vrijednosti $f(x,y) \geq T$ vrijedi $(x,y) \in I$, odnosno samo onda se stavlja oznaka "X" na presjek retka objekta x i stupca atributa y. Promjenom granične vrijednosti T mogu se dobiti različite konceptualne rešetke i skupovi asocijacijskih pravila [68].

Metoda temporalne FCA uvodi pojam vremena u formalni kontekst. U različitim vremenskim trenucima mijenjaju se odnosi između objekata i atributa i za svaki promatrani trenutak gradi se posebna konceptualna rešetka. Praćenjem vremenskih trenutaka mogu se pratiti promjene u konceptualnoj rešetci, odnosno analizirati procesi poput kemijskih reakcija ili meteoroloških prognoza [69].

3.2 Kombinatorno testiranje

Kombinatorno testiranje je tehnika za testiranje programske potpore kojim se može smanjiti skup testova koje treba provesti, a da se pritom i dalje otkrije većina grešaka u softverskom sustavu. Ova tehnika je vrsta funkcijskog testiranja (engl. *functional testing* ili *black-box testing*) jer se ispituje ponašanje sustava bez ulaženja u programski kod, a testni slučajevi se stvaraju prema specifikaciji softverskog sustava [70, 71]. Kuhn i sur. su u radu [26] analizirali greške u softverskim sustavima i opazili su kako su gotovo sve greške uzrokovane međusobnom interakcijom najviše 4 do 6 različitih parametara sustava. Na slici 3.15 iz [72] prikazana ovisnost kumulativnog postotka softverskih grešaka o broju parametara u međusobnoj interakciji prema više studija iz različitih domena.



Slika 3.15: Kumulativni postotak softverskih grešaka i broj parametara u međusobnoj interakciji [72]

Prema podacima iz grafa na slici 3.15 može se vidjeti kako je u nekim domenama i do 90% grešaka u softverskim sustavima uzrokovano jednim parametrom sustava ili međusobnom interakcijom dva različita parametra. Ako se uključi i međusobna interakcija tri parametra onda u svim obuhvaćenim studijama kumulativni postotak softverskih grešaka prelazi 90%, a u nekima se približava i vrijednosti 100%.

Iz ovih razloga se kombinatorno testiranje usredotočuje na interakcije između različitih parametara softverskog sustava. Kombinatornim testiranjem može se učinkovito ispitati kako odabir vrijednosti ulaznih podataka ili odabir konfiguracije utječe na ponašanje softverskog sustava. Glavni cilj ove tehnike testiranja je automatsko generiranje gotovo minimalnog broja testnih slučajeva, tako da je svaka moguća n-torka vrijednosti različitih parametara u barem jednom od svih generiranih testnih slučajeva. Ako je veličina n-torki vrijednosti parametara

postavljena na $n = 2$ onda će postupak kombinatornog testiranja automatski generirati skup testnih slučajeva koji će pokriti svaki mogući par vrijednosti različitih parametara barem jednom. Upravo ova varijanta kombinatornog testiranja je najčešća u praksi i u literaturi, a naziva se i testiranje *svih parova* (engl. *all pairs testing* ili *pairwise testing*).

U ovom istraživanju koristit će se tehnika kombinatornog testiranja u fazi pripreme podataka za metodu FCA kojom se gradi ontologija domenskog znanja u obliku konceptualne rešetke. Ovdje se domensko znanje nekog područja definira skupom ispitnih zadataka iz tog područja kao i sa skupom atributa koje mogu imati ti zadaci. Izgradnjom formalnog konteksta svaki zadatak će biti opisan s nekim podskupom atributa. Tehnika kombinatornog testiranja primijenit će se upravo u fazi izgradnje formalnoga konteksta kako bi se osiguralo da je svaka dozvoljena n-torka atributa pokrivena s barem jednim ispitnim zadatkom.

Ortogonalna polja i pokrivajuća polja

Temelji na kojima počiva postupak kombinatornog testiranja nastali su u polju statistike pod nazivom *dizajn eksperimenata* (engl. *Design of Experiments* ili kraće DoE), a koje se bavi mjerenjem i određivanjem kako različite kombinacije ulaznih parametara utječu na ishod eksperimenta, odnosno na varijablu odgovora (engl. *response variable*). Dizajn eksperimenata koristi se npr. u medicini za unaprjeđenje ishoda liječenja te u industriji kao i poljoprivredi za povećanje obujma proizvodnje. Postupak kombinatornog testiranja matematički je utemeljen na pojmovima ortogonalnih polja (engl. *orthogonal arrays*) i prekrivajućih polja (engl. *covering arrays*) [37, 72], a koji će se detaljnije opisati u nastavku.

Definicija 3.19. Ortogonalno polje, $OA_\lambda(N,t,k,v)$ je tablica veličine $N \times k$, gdje je N broj redaka tablice (broj testova ili eksperimenata), k je broj stupaca tablice (broj parametara), v je broj vrijednosti svakoga parametra i t je snaga ili stupanj međusobne pokrivenosti. U svakom potpolju veličine $N \times t$ ortogonalnog polja $OA_\lambda(N,t,k,v)$ svaki redak pojavljuje se točno λ puta.

Ako se postave vrijednosti $t = 2$ te $\lambda = 1$, onda će rezultirajuće ortogonalno polje sadržavati točno jedan od svakog para vrijednosti različitih parametara. Tablica 3.8 prikazuje primjer takvoga jednostavnog ortogonalnog polja ($t = 2$, $\lambda = 1$), s tri parametra ($k = 3$) i dvije vrijednosti parametara ($v = 2$, vrijednosti 0 i 1).

Tablica 3.8: Primjer trivijalnog ortogonalnog polja $OA_1(4,2,3,2)$

0	0	0
1	1	0
1	0	1
0	1	1

Prikazano ortogonalno polje $OA_1(4,2,3,2)$ ima četiri retka, i može se lako ručno provjeriti odabirom bilo koja dva stupca ($1-2$, $2-3$, $1-3$) da se sva četiri moguća para vrijednosti parametara (00, 01, 10, 11) pojavljuju točno jednom. Ovo ortogonalno polje je generirano korištenjem alata za kombinatorno testiranje *AllPairs* [73].

Izgradnja ortogonalnih polja proizvoljnih karakteristika u pravilu je vrlo zahtjevan matematički problem. U nekim konfiguracijama broja parametara i broja vrijednosti svakog parametra može biti i nemoguće ispuniti vrlo strog uvjet da se svaki redak u bilo kojem podskupu od t stupaca ortogonalnog polja pojavljuje točno λ puta, npr. točno jednom kada je $\lambda = 1$. Zato su za mnoge konfiguracije javno dostupna već izračunata ortogonalna polja koje i dalje kontinuirano pronalaze i objavljuju različite grupe istraživača. Ako se u dostupnim kolekcijama pronađe ortogonalno polje s traženim karakteristikama onda se ono može iskoristiti kao predložak za dizajn vlastitih eksperimenata ili testova. Jedna od takvih zbirki dostupna je na stranici [74]. Npr. u tablici 3.9 prikazan je primjer ortogonalnog polja $OA_1(8,3,4,2)$ iz te zbirke. Ovo ortogonalno polje ima 8 redaka ili eksperimenata ($N = 8$), 4 stupca ili parametra ($k = 4$), a svaki parametar ima 2 vrijednosti ($v = 2$) te snagu međusobne pokrivenosti od $t = 3$. Pritom se svaki redak u svakoj trojci stupaca pojavljuje točno jednom ($\lambda = 1$). Treba spomenuti kako bi za testiranje svih mogućih kombinacija vrijednosti različitih parametara u ovom slučaju trebalo dvostruko više testova ($2^4 = 16$).

Tablica 3.9: Ortogonalno polje $OA_1(8,3,4,2)$

0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

U ortogonalnom polju iz tablice 3.9 svaki parametar ima dvije vrijednosti 0 i 1 koje se kod testiranja mogu zamijeniti sa stvarnim vrijednostima odabranih parametara. Npr. za testiranje ponašanja aplikacija na različitim operacijskim sustavima mogu se zadati sljedeći parametri: prvi parametar je operacijski sustav (vrijednosti: *Windows 11* i *macOS 13*), drugi parametar je web preglednik (vrijednosti: *Safari* i *Firefox*), treći parametar je uredska aplikacija (vrijednosti: *Excel* i *Word*) te četvrti parametar je program za arhiviranje datoteka (vrijednosti: *WinRar* i *WinZip*). Tako se dobiju kombinatorni testni slučajevi prikazani u tablici 3.10.

Tablica 3.10: Kombinatorni testni slučajevi prema ortogonalnom polju u tablici 3.9

Windows 11	*Safari*	*Excel*	*WinRar*
Windows 11	*Safari*	*Word*	*WinZip*
Windows 11	*Firefox*	*Excel*	*WinZip*
Windows 11	*Firefox*	*Word*	*WinRar*
macOS 13	*Safari*	*Excel*	*WinZip*
macOS 13	*Safari*	*Word*	*WinRar*
macOS 13	*Firefox*	*Excel*	*WinRar*
macOS 13	*Firefox*	*Word*	*WinZip*

Svaki testni slučaj naveden u tablici 3.10 predstavlja konfiguraciju sustava koju treba testirati. Pažljivijim iščitavanjem vidi se kako neki od generiranih testnih slučajeva nisu izvedivi jer web preglednik *Safari* nije više dostupan na operacijskim sustavima *Windows*, dok program *WinRar* nije dostupan na operacijskim sustavima *macOS*. Ipak, ako se zanemari prvi testni slučaj kao neizvediv (zbog pojavljivanja para *Windows 11* i *Safari*) onda će se izgubiti npr. trojka (*Windows 11*, *Excel*, *WinRar*) koja se ne pojavljuje niti u jednom drugom testnom slučaju. U ovakvim situacijama može biti korisnije koristiti prekrivajuće polje. Naime, to je proširenje ortogonalnog polja u kojemu je dopuštena nejednolika redundancija, a dodatno je moguće i nametanje ograničenja o zabranjenim kombinacijama vrijednosti parametara.

Definicija 3.20. Prekrivajuće polje, $CA(N,t,k,v)$ je tablica veličine $N \times k$, gdje je N broj redaka tablice (broj testova ili eksperimenata), k je broj stupaca tablice (broj parametara), v je broj vrijednosti svakoga parametra i t je snaga ili stupanj međusobne pokrivenosti. U svakom potpolju veličine $N \times t$ prekrivajućeg polja $VA(N,t,k,v)$ svaki redak pojavljuje se najmanje jednom.

Može se vidjeti da je testiranje svih parova ($t = 2$) u skladu s definicijom prekrivajućeg polja. Štoviše, treba primijetiti kako je svako ortogonalno polje ujedno i prekrivajuće polje, ali općenito ne vrijedi i obrat. Razlog tome je nejednaka redundancija kod prekrivajućih polja zbog čega najčešće neće ispunjavati strogi uvjet ortogonalnog polja kako se svaka n-torka veličine t pojavljuje točno λ puta. Treba naglasiti kako se proširenjem osnovnih definicija ortogonalnih polja i prekrivajućih polja mogu dopustiti i miješane vrijednosti parametara, odnosno da svaki od parametara može imati različiti broj vrijednosti. U tom slučaju se mora specificirati struktura parametara u sljedećem obliku: ${v_1}^{k_1}\,{v_2}^{k_2}\,{v_3}^{k_3}\ldots$. Npr. struktura $2^4\,3^2$ se sastoji od šest parametara, i to četiri parametra koji imaju po dvije vrijednosti te još dva parametra od kojih svaki ima po tri vrijednosti. Kao što je ranije spomenuto daljnje proširenje prekrivajućih polja omogućuje i specificiranje ograničenja, odnosno zabranjenih kombinacija vrijednosti različitih parametara. Takve kombinacije se neće pojaviti niti u jednom generiranom testnom slučaju, a

pritom će svaka od preostalih n-torki zadane veličine t i dalje biti pokrivena barem s jednim testnim slučajem. Kasnije će se predložiti metoda kombinatornog testiranja koja koristi upravo takva proširenja prekrivajućih polja s kojima su dopuštene miješane vrijednosti parametara te zadavanje ograničenja kod generiranja testnih slučajeva.

Alati za kombinatorno testiranje

Tijekom istraživanja tražio se najprikladniji alat za kombinatorno testiranje, a stoga su se analizirali različiti dostupne aplikacije[7]. Tražio se besplatan alat koji je javno dostupan, a da se može pokrenuti iz komandne linije te da podržava zadavanje ograničenja na vrijednosti parametara. Međutim, većina dostupnih alata je komercijalna te je potrebno platiti licencu za njihovo korištenje, a mnogi od alata su dostupni samo kao web aplikacije. Ipak, pronađeno je više alata sa sučeljem u naredbenom retku kao što su prije spomenuti program *AllPairs* te alati *Tcases* [75] i *Jenny* [76], kao i *Microsoftov* alat *Pict* [77], te *NIST-ov* alat *ACTS* [78] koji se može pokrenuti i s grafičkim sučeljem. Program *AllPairs* ne podržava jednostavno zadavanje ograničenja pa nije bio prikladan za predloženu metodu kombinatornog testiranja. S druge strane, alati *ACTS*, *Pict* i *Tcases* omogućuju zadavanje složenih ograničenja, ali to su sve napredniji alati kod kojih treba posebno pripremati ulazne podatke kao i obrađivati dobivene rezultate. Iz tih razloga na kraju je odabran alat za kombinatorno testiranje *Jenny* kao najprikladniji za korištenje u implementaciji predložene metode kombinatornog testiranja. To je alat koji ima intuitivno korisničko sučelje, podržava zadavanje jednostavnih ograničenja, a efikasnost generiranja testnih slučajeva je usporediva s drugim etabliranim alatima[8].

Alat za kombinatorno testiranje *Jenny* je besplatan, a njegov programski kod je u jeziku C je dostupan u javnoj domeni. Ovaj alat generira testne slučajeve na temelju m dimenzija značajki (ili parametara) koji će pokriti sve n-torke (do $n \leq m$) značajki (ili vrijednosti parametara) primjenom tehnike kombinatornog testiranja. Generirani testni slučajevi mogu se npr. koristiti za regresijsko testiranje kojim se provjerava ponašanje softverskog sustava nakon svake promjene u njegovom razvoju. Primjenom tehnike kombinatornog testiranja alat *Jenny* umjesto iscrpne pretrage svih mogućih kombinacija značajki generira gotovo minimalan broj testnih slučajeva koji pokrivaju sve n-torke značajki zadane veličine n (obično $n = 2$ za parove značajki, odnosno $n = 3$ za trojke značajki). Prije pokretanja programa treba mu zadati veličinu n-torki koje će se pokriti s testovima (zastavica *-n*, pretpostavljena veličina je $n = 2$) kao i listu značajki za svaku dimenziju. Dodatno, mogu se zadati zabranjene kombinacije značajki (zastavica *-w*) koje se ne smiju pojaviti u generiranim testnim slučajevima.

Jedan primjer generiranja kombinatornih testova s alatom *Jenny* prikazan je na slici 3.16. U ovom slučaju program je pokrenut s naredbom *jenny.exe -n3 2 2 2 2*, odnosno tražena veličina

[7] https://jaccz.github.io/pairwise/tools.html
[8] https://jaccz.github.io/pairwise/efficiency.html

n-torki značajki postavljena je na $n = 3$ (testovi trebaju pokriti svaku trojku značajki barem jednom) i zadane su ukupno četiri dimenzije, a svaka ima po dvije značajke. Kod ispisa generiranih testnih slučajeva alat koristi internu nomenklaturu – dimenzije se označavaju brojem (ukupno dozvoljeno do 65535 dimenzija), a značajke s malim i velikim slovima (redom *a-z* pa *A-Z*). Pritom svaka dimenzija mora imati barem 2 značajke, dok ih najviše može imati 52.

```
1a 2b 3b 4b
1b 2a 3a 4a
1a 2a 3a 4b
1b 2b 3b 4a
1a 2b 3a 4a
1b 2a 3b 4b
1a 2a 3b 4a
1b 2b 3a 4b
```

Slika 3.16: Generirani kombinatorni testovi s naredbom *jenny.exe -n3 2 2 2 2*

Generirani kombinatorni testni slučajevi sa slike 3.16 čine prekrivajuće polje $CA(8,3,4,2)$ gdje je broj redaka jednak $N = 8$, snaga ili stupanj međusobne pokrivenosti je $t = 3$, broj parametara je $k = 4$ i svaki parametar ima po dvije vrijednosti ($v = 2$). Dakle, konfiguracija ovoga automatski stvorenog prekrivajućeg polja $CA(8,3,4,2)$ jednaka je ortogonalnom polju $OA_1(8,3,4,2)$ iz tablice 3.9. Kako bi se ove dvije strukture mogle lakše usporediti prekrivajuće polje $CA(8,3,4,2)$ prikazano je u tablici 3.11 – oznake *1a*, *2a*, *3a* i *4a* iz nomenklature alata *Jenny* zamijenjene su s nulom, a oznake *1b*, *2b*, *3b* i *4b* s jedinicom.

Tablica 3.11: Prekrivajuće polje $CA(8,3,4,2)$ sa slike 3.16

0	1	1	1
1	0	0	0
0	0	0	1
1	1	1	0
0	1	0	0
1	0	1	1
0	0	1	0
1	1	0	1

Iako prekrivajuće polje $CA(8,3,4,2)$ iz tablice 3.11 i ortogonalno polje $OA_1(8,3,4,2)$ iz tablice 3.9 nemaju niti jedan zajednički redak, u ovom slučaju ipak i jedna i druga struktura pokriva svaku trojku vrijednosti različitih parametara točno s po jednim testnim slučajem. Može se ručno provjeriti kako u ovom prekrivajućem polju $CA(8,3,4,2)$ svaka trojka stupaca uvijek

sadrži osam različitih redaka: 000, 001, 010, 011, 100, 101, 110 i 111. Dakle, ovdje vrijedi kako je generirano prekrivajuće polje ujedno i ortogonalno polje.

Na slici 3.17 je prikazan drugi primjer izvođenja programa *Jenny*, ovoga puta s naredbom *jenny.exe -n2 2 3 4 -w1a2a* kod koje je zadana i zabranjena kombinacija značajki.

```
1a 2c 3a
1b 2b 3d
1b 2a 3c
1b 2c 3b
1a 2b 3c
1b 2a 3a
1a 2b 3b
1a 2c 3d
1b 2a 3d
1b 2a 3b
1a 2b 3a
1b 2c 3c
```

Slika 3.17: Generirani kombinatorni testovi s naredbom *jenny.exe -n2 2 3 4 -w1a2a*

U ovom slučaju je tražena veličina n-torki značajki postavljena na uobičajenih $n = 2$ (generirani testovi moraju pokriti svaki par značajki barem jednom) i zadane su tri dimenzije, redom s po dvije, tri i četiri značajke. Dodatno, u ovom primjeru zadana je i jedna zabranjena kombinacija značajki (*1a*, *2a*), odnosno prva značaka (*a*) iz prve dimenzije (*1*) ne smije se pojaviti u niti jednom generiranom testnom slučaju uz prvu značajku (*a*) iz druge dimenzije (*2*). Nakon pokretanja programa s opisanim parametrima generirano je 12 kombinatornih testnih slučajeva koji pokrivaju sve parove značajki osim para (*1a*, *2a*). Npr., može se jednostavno provjeriti da je par (*1a*, *3d*) pokriven samo s osmim testnim slučajem (*1a*, *2c*, *3d*), a da je par (*1a*, *2c*) pokriven s prvim i osmim testnim slučajem. Treba isto tako još jednom naglasiti kako je kombinatornim testiranjem smanjen broj testnih slučajeva, jer bi se inače trebalo izvesti $2 \cdot 3 \cdot 4 = 24$ testna slučaja kojima bi se pokrile sve kombinacije značajki. Naravno, u slučaju kada bi isključili jednu zabranjenu kombinaciju značajki (*1a*, *2a*) ukupni broj testova bi bio 20, što je i dalje puno više od 12 testnih slučajeva generiranih kombinatornim testiranjem. Treba primijetiti kako u ovom slučaju generirani testni slučajevi čine prekrivajuće polje, ali ne i ortogonalno polje. To se odmah može provjeriti po tome što niti jedan testni slučaj ne pokriva kombinaciju (*1a*, *2a*) koja je na početku testiranja isključena. Isto tako može se vidjeti kako postoji nejednolika redundancija kod pokrivenih parova značajki – ne pojavljuje se svaki par točno zadanih λ puta kao kod ortogonalnih polja. Ovdje je par značajki (*1b*, *2b*) pokriven samo s jednim testom, a s druge strane par (*1a*, *2c*) je pokriven s dva testa. Nadalje, neki parovi značajki su pokriveni i s više od dva testna slučaja – npr. par (*1a*, *2b*) je pokriven s tri testna slučaja, a par (*1b*, *2a*) s čak četiri testna slučaja.

Alat za kombinatorno testiranje *Jenny* ima i dodatnu opciju (zastavica *-s*) s kojom se može

zadati početna vrijednost (engl. *seed*) za generator pseudo-slučajnih brojeva. Uobičajena vrijednost je 0 pa će stoga bez korištenja ove dodatne opcije *Jenny* za zadanu konfiguraciju uvijek izgenerirati istu listu kombinatornih testnih slučajeva. Ako se ipak aktivira ta dodatna opcija mogu se generirati i drukčije liste kombinatornih testnih slučajeva. Kao primjer prikazat će se rezultat izvođenja kombinatornog testiranja s naredbom: *jenny.exe -n2 2 3 4 -w1a2a -s123* na slici 3.18.

```
1a 2c 3d
1b 2a 3c
1a 2b 3a
1b 2c 3b
1b 2b 3d
1b 2a 3a
1a 2b 3b
1a 2c 3c
1b 2a 3d
1b 2a 3b
1a 2b 3c
1a 2c 3a
```

Slika 3.18: Generirani kombinatorni testovi s naredbom *jenny.exe -n2 2 3 4 -w1a2a -s123*

Iako je zadana ista konfiguracija kombinatornog testiranja kao na slici 3.17 generirani su drukčiji testni slučajevi zbog zadavanja početne vrijednosti za generator pseudo-slučajnih brojeva (*-s123*). Opet se može provjeriti kako se zabranjeni par značajki (*1a*, *2a*) ne pojavljuje u niti jednom testnom slučaju, a kako su svi ostali dozvoljeni parovi značajki pokriveni. Ipak sada se mogu razlikovati frekvencije pojavljivanja dozvoljenih parova značajki, npr. par značajki (*1a*, *2c*) pokriven s tri testna slučaja sa slike 3.18 dok je na slici 3.17 pokriven s dva testna slučaja.

Sada će se još jednom razmotriti primjer ortogonalnog polja iz tablice 3.10 u kojem postoje testni slučajevi koji nisu izvedivi, a koji se ipak ne smiju ukloniti iz tablice jer pokrivaju neke druge dozvoljene trojke vrijednosti različitih parametara. Pokazat će se kako bi se s alatom *Jenny* mogao riješiti taj problem. Kod pokretanja kombinatornog testiranja treba zadati istu konfiguraciju ortogonalnog polja $OA_1(8,3,4,2)$, ali uz zadana dodatna ograničenja na zabranjene parove vrijednosti parametara. Zato treba pokrenuti alat *Jenny* sa sljedećom naredbom: *jenny.exe -n3 2 2 2 2 -w1a2a -w1b4a*, a potom se generiraju kombinatorni testovi sa slike 3.19.

```
1a 2b 3b 4b
1b 2a 3a 4b
1a 2b 3a 4a
1b 2a 3b 4b
1b 2b 3b 4b
1a 2b 3b 4a
1a 2b 3a 4b
1b 2b 3a 4b
```

Slika 3.19: Generirani kombinatorni testovi s naredbom *jenny.exe -n3 2 2 2 2 -w1a2a -w1b4a*

Dakle, s opcijom *-n3* traži se pokrivanje svih trojki vrijednosti različitih parametara, a uz to se s opcijom *-w1a2a* isključuje kombinacija vrijednosti *Windows 11* i *Safari* (prva vrijednost prvog parametra i prva vrijednost drugog parametra), a s opcijom *-w1b4a* zabranjuje se kombinacija vrijednosti *macOS 13* i *WinRar* (druga vrijednost prvog parametra i prva vrijednost četvrtog parametra). Osam generiranih kombinatornih testova mogu se prikazati i kao prekrivajuće polje sa stvarnim konfiguracijama sustava koje treba testirati u tablici 3.12.

Tablica 3.12: Kombinatorni testni slučajevi prema prekrivajućem polju sa slike 3.19

Windows 11	*Firefox*	*Word*	*WinZip*
macOS 13	*Safari*	*Excel*	*WinZip*
Windows 11	*Firefox*	*Excel*	*WinRar*
macOS 13	*Safari*	*Word*	*WinZip*
macOS 13	*Firefox*	*Word*	*WinZip*
Windows 11	*Firefox*	*Word*	*WinRar*
Windows 11	*Firefox*	*Excel*	*WinZip*
macOS 13	*Firefox*	*Excel*	*WinZip*

Treba naglasiti kako je program vratio i poruku kako par (*2a*, *4a*), odnosno (*Safari*, *WinRar*) nije mogao biti pokriven niti s jednim testnim slučajem, ali to nije problem jer takav testni slučaj ionako ne bi bio izvediv jer se ta dva programa ne mogu naći zajedno na operacijskom sustavu *Windows 11* ili *macOS 13*. Formalnije iskazano to je još jedno implicitno ograničenje koje je nastalo eksplicitnim nametanjem ograničenja (*Windows 11*, *Safari*) i (*macOS 13*, *WinRar*). Primjerice, kako svaki parametar ima po dvije vrijednosti ne može se generirati testni slučaj koji bi pokrio trojku (*Safari*, *Excel*, *WinRar*) ili (*Safari*, *Word*, *WinRar*) jer bi se tako prekršilo ograničenje (*Windows 11*, *Safari*) ili (*macOS 13*, *WinRar*). Pažljivijim pregledom prekrivajućeg polja u tablici 3.12 može se provjeriti kako je svaka dopuštena trojka vrijednosti različitih parametara pokrivena s nekim od 8 generiranih testnih slučajeva. Isto tako u niti jednom testnom slučaju se ne pojavljuju tri spomenute zabranjene kombinacije vrijednosti različitih parametara pa sada svaki od 8 generiranih testnih slučajeva opisuje izvedivu konfiguraciju sustava koju treba testirati.

S kombinatornim testiranjem će se u ovom istraživanju dobiti formalni kontekst u kojem je svaki dozvoljeni par ili trojka atributa pokriven s barem jednim objektom, odnosno zadatkom. Iz tako priređenoga formalnog konteksta se alatima za metodu FCA automatski generira odgovarajuća konceptualna rešetka kao ontologija zadanog domenskog znanja. A u nastavku će se pokazati kako se s topološkim sortiranjem konceptualne rešetke dobiva odgovarajući linearni poredak formalnih koncepata.

3.3 Topološko sortiranje

Topološko sortiranje (en. *Topological sort*) je postupak kojim se generira linearni poredak čvorova usmjerenog acikličkog grafa (en. *directed acyclic graph* ili kraće DAG). Za početak će se navesti definicija usmjerenih grafova ili *digrafova* iz [79, 80].

Definicija 3.21. Usmjereni graf D je uređeni par (V,E) koji se sastoji od skupa čvorova V i skupa bridova E, zajedno s funkcijom incidencije ψ_D između bridova iz V i uređenih parova čvorova iz E. Ako je a brid i vrijedi $\psi_D(a) = (u,v)$ tada je brid a usmjeren od čvora u prema čvoru v.

Nadalje, može se definirati i usmjereni aciklički graf kao usmjereni graf bez ciklusa, odnosno nepraznog usmjerenog puta koji bi započeo i završio s istim čvorom. Treba primijetiti kako se svaki parcijalno uređeni skup $(A,\leq)$ može prikazati kao usmjereni aciklički graf $D = (V,E)$. Pritom se skup elemenata uređenog parcijalnog skupa A uzima kao skup čvorova V od usmjerenog acikličkog grafa D, a ako vrijedi $u \leq v$ za elemente $u \in A$ i $v \in A$ onda će postojati i usmjereni brid (u,v) u D.

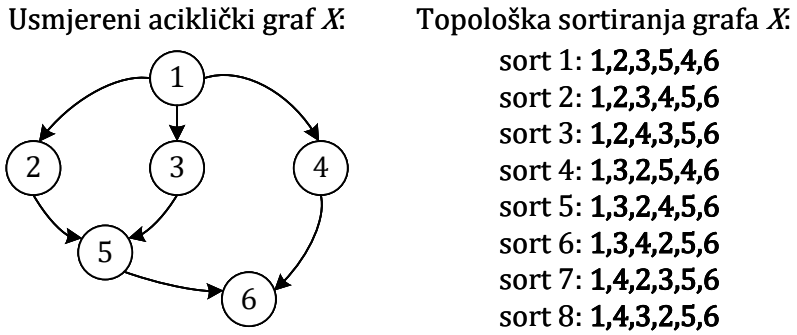
Dakle i konceptualna rešetka $\underline{\mathfrak{B}}(X,Y,I)$ se može predstaviti kao usmjereni aciklički graf (en. *directed acyclic graph*). U njemu su čvorovi formalni koncepti, a međusobno su povezani usmjerenim granama s orijentacijom od natkoncepta prema potkonceptu. Na ovaj način se na konceptualnu rešetku mogu primijeniti algoritmi za topološko sortiranje iz teorije grafova. U ovom radu je posebno važno odrediti iz konceptualne rešetke linearno poredanu listu formalnih koncepata u kojoj će biti sačuvani međusobni odnosi natkoncepta i potkoncepta.

Algoritmi za topološko sortiranje usmjerenih acikličkih grafova daju linearno poredanu listu čvorova tako da za svaku granu (u,v) iz usmjerenog acikličkog grafa D čvor u dolazi u linearni poredak prije čvora v [81, 82]. Dakle svaki čvor v ulazi u listu kada su u listi već svi njegovi čvorovi roditelji ili ako taj čvor v uopće nema čvorova roditelja. Ako se algoritmi za topološko sortiranje primjene nad parcijalno uređenim skupovima onda će dobiti odgovarajući potpuno uređeni skupom ili *lanac*.

Postoji više algoritama za topološko sortiranje, ali ih se može podijeliti u dvije skupine – prva skupina su algoritmi temeljeni na uklanjanju izvora (čvora bez ulaznih grana), a drugu skupinu čine algoritmi koji se temelje na pretraživanju grafa u dubinu (en. *DFS - depth first search*). Algoritam za uklanjanje izvora (*Kahnov algoritam*) na početku traži čvor(ove) bez ulaznih grana, briše ih (zajedno s njihovim izlaznim granama) iz grafa i ubacuje u na kraj (inicijalno prazne) liste. Ova dva koraka ponavlja sve dok graf ne bude potpuno prazan, a tada je generirana lista svih čvorova odnosno linearni poredak čvorova [83]. Algoritam temeljen na pretraživanju u dubinu vrši DFS prolazak kroz graf i bilježi poredak po kojem su čvorovi potpuno istraženi. Na kraju se obrtanjem zabilježenog poretka dobije traženi linearni poredak svih čvorova [82]. Treba naglasiti kako ovi algoritmi provjeravaju postoji li u grafu ciklus jer se

usmjereni graf s jednim ili više ciklusa ne može topološki sortirati. U tom slučaju se izvođenje algoritma odmah prekida.

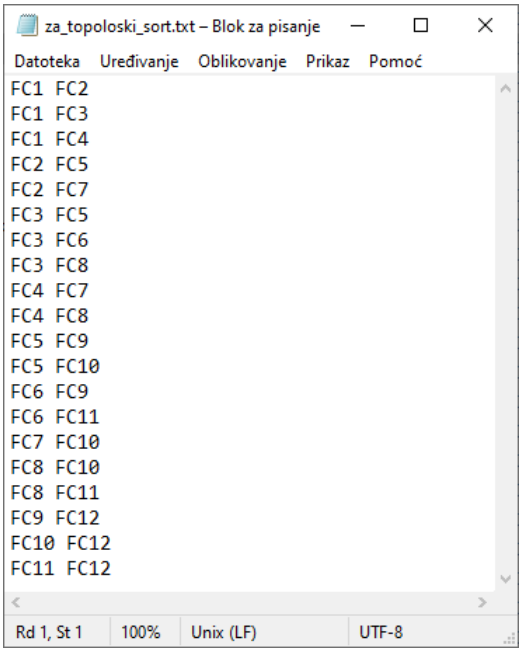
Topološkim sortiranjem nekog usmjerenog acikličkog grafa moguće je dobiti više od jednog ispravnog linearnog poretka čvorova, dakle rješenje nije jedinstveno. Rješenje nije uvijek jedinstveno, može postojati više ispravnih topoloških sortiranja istoga usmjerenog acikličkog grafa kao što je prikazano na slici 3.20.



Slika 3.20: Primjer usmjerenog acikličkog grafa i njegovih topoloških sortiranja

Ipak, ako u grafu postoji usmjereni *Hamiltonov put* (razapinjući put jer prolazi kroz svaki čvor grafa točno jedanput [79]) onda svaki algoritam za topološko sortiranje daje isto rješenje. Na primjer to će se dogoditi ako je usmjereni aciklički graf u obliku lanca – tada se linearno sortirana lista čvorova može iščitati direktno iz grafa.

Treba prikazati kako se može topološki sortirati konceptualna rešetka $\underline{\mathfrak{B}}(X,Y,I)$. Kao primjer uzet će se konceptualna rešetka sa slike 3.1 s 12 formalnih koncepata označenih s FC1, ..., FC12 i 20 grana (implicitno usmjerenih od vrha prema dnu).



Slika 3.21: Ulazni podaci za topološko sortiranje konceptualne rešetke sa slike 3.1

Za topološko sortiranje iskoristit će se naredba *tsort* koja je dostupna kroz naredbeni redak u operacijskim sustavima temeljenim na *Unixu*, ali isto tako je implementirana i u *PowerShell* nadograđenom naredbenom retku u operacijskom sustavu *Windows 10* [84]. Kao ulazni podatak potrebno je zadati sve parove (u,v) iz usmjerenog acikličkog grafa. Na slici 3.21 je prikazana pripremljena ulazna datoteka koja odgovara svim odnosima natkoncept – potkoncept iz konceptualne rešetke sa slike 3.1. Topološkim sortiranjem dobiva se sljedeći linearni poredak (potpuno uređeni skup) formalnih koncepata: FC1 $\leq$ FC4 $\leq$ FC3 $\leq$ FC2 $\leq$ FC8 $\leq$ FC6 $\leq$ FC7 $\leq$ FC5 $\leq$ FC11 $\leq$ FC10 $\leq$ FC9 $\leq$ FC12. Kao što je ranije rečeno ovo rješenje ne mora biti jedinstveno, a to se može provjeriti zamjenom mjesta redaka u ulaznoj datoteci za program *tsort*[9].

Zanimljivo je dodatno analizirati iz dobivenog topološkog sortiranja i moguće redoslijede atributa i objekata. Kod prikaza konceptualne rešetke odabrano je sažeto označavanje kod kojeg se svaki atribut zapisuje samo u najvišem formalnom konceptu kojem pripada (implicitno pripada i svim potkonceptima), a svaki objekt se zapisuje samo u najnižem formalnom konceptu kojem pripada (implicitno pripada i svim natkonceptima). Ovako će se dobiti odgovarajući redoslijed atributa: MATAN1, VJEKO, DIGLOG, LINALG, UUPRO. Nadalje, dobit će se i odgovarajući redoslijed objekata: $\{$St7, St9$\}$, St4, St1, $\{$St3, St6$\}$, St2, St5, $\{$St8, St10$\}$. Iz ovoga se mogu dobiti sljedeći redoslijedi objekata (po jedan iz svakog formalnog koncepta):

- St7, St4, St1, St3, St2, St5, St8
- St7, St4, St1, St6, St2, St5, St8
- St7, St4, St1, St3, St2, St5, St10
- St7, St4, St1, St6, St2, St5, St10
- St9, St4, St1, St3, St2, St5, St8
- St9, St4, St1, St6, St2, St5, St8
- St9, St4, St1, St3, St2, St5, St10
- St9, St4, St1, St6, St2, St5, St10

U ovome radu će se iskoristiti navedeni pristup topološkog sortiranja za dobivanje redoslijeda pitanja za formativnu provjeru znanja u sustavima za e-učenje. Naravno, prvo je potrebno generirati konceptualnu rešetku koja čini ontologiju zadanog gradiva na temelju opisa skupa pitanja iz sustava za e-učenje. Topološkim sortiranjem te konceptualne rešetke dobiva se linearni poredak formalnih koncepata, sortiranih od najopćenitijih do najspecifičnijih. Pretpostavimo da su općenitiji formalni koncepti lakši jer u pravilu obuhvaćaju manje atributa, a da su specifičniji formalni koncepti teži jer im u pravilu pripada više atributa. Tako se može automatski dobiti iz topološkog sortiranja formalnih koncepata jedan ili više redoslijeda objekata (pitanja) poredanih od lakših prema težim pitanjima. Na temelju tih sekvenci pitanja moguće je automatski otkriti konačni automat kojim bi se sintetizirao proces provjere znanja.

[9]Npr. ako se na slici 3.21 zamijeni prvi redak (FC1 FC2) i drugi redak (FC1 FC3) onda će se dobiti sljedeće ispravno topološko sortiranje: FC1 $\leq$ FC4 $\leq$ FC2 $\leq$ FC3 $\leq$ FC7 $\leq$ FC8 $\leq$ FC6 $\leq$ FC5 $\leq$ FC11 $\leq$ FC10 $\leq$ FC9 $\leq$ FC12.

3.4 Učenje konačnih automata

U ovom potpoglavlju objasnit će se pojmovi sustava s prijelazima i konačnih automata kako bi se potom mogao prikazati algoritam L^* Dane Angluin za učenje deterministčkih konačnih automata. Kroz primjenu ovoga algoritma izgradit će se konačni automati koji opisuju proces provjere znanja. Takvi automati mogu poslužiti kao predložak za implementaciju formativne provjere znanja u sustavima za e-učenje, a iskoristit će se i za automatiziranu simulaciju i formalnu verifikaciju otkrivenog procesa provjere znanja.

3.4.1 Uvod

Na početku će se definirati pojam sustava s prijelazima (engl. *transition system – TS*) kao matematičke strukture s kojom se može opisati ponašanje nekog sustava [85].

Definicija 3.22. Sustav s prijelazima je par $(Q, \rightarrow)$ gdje je Q skup stanja, a $\rightarrow$ je binarna relacija nad Q, koja predstavlja skup prijelaza.

Prijelaz iz stanja $p \in Q$ u stanje $q \in Q$ zapisuje se kao $p \rightarrow q$, a odgovara podskupu od $Q \times Q$. Svaki pojedini prijelaz označava nedjeljivi pomak iz jednoga stanja u drugo. Važno je naglasiti kako ni skup stanja Q niti skup prijelaza $\rightarrow$ ne moraju biti konačni, a nema zadanih inicijalnih i završnih stanja. U slučaju da se u sustav s prijelazima dodaju oznake, odnosno labele na prijelaze između stanja govori se o se labeliranom sustavu s prijelazima (engl. *labelled transition system – LTS* ili *named transition system*) [85].

Definicija 3.23. Labelirani sustav s prijelazima je trojka $(Q, \rightarrow, \Sigma)$ gdje je $(Q, \rightarrow)$ sustav s prijelazima i svakom prijelazu iz $\rightarrow$ pridijeljena je jedna ili više labela iz skupa Σ.

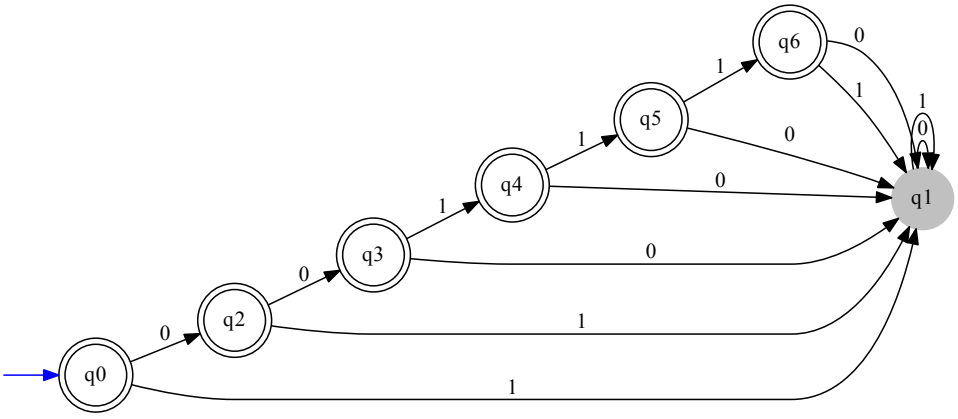
Prijelaz s labelom $a \in \Sigma$ iz stanja $p \in Q$ u stanje $q \in Q$ zapisuje se kao $p \xrightarrow{a} q$, a odgovara podskupu od $Q \times \Sigma \times Q$. Ako za neki par stanja (p,q) i labele a postoji samo jedan prijelaz $p \xrightarrow{a} q$, onda je labela a za stanje p deterministička. Dodatno, ako za neki par stanja (p,q) i labele a postoji barem jedan prijelaz $p \xrightarrow{a} q$, onda je labela a za stanje p izvršna.

Za razliku od (labeliranih) sustava s prijelazima strože matematičke strukture konačnih automata imaju konačni skup stanja, prijelaza i simbola (labela) te imaju definirano inicijalno stanje kao i skup završnih stanja. Konačne automati mogu se podijeliti na determinističke (engl. *deterministic finite automaton* ili skraćeno DFA) i nedeterminističke (engl. *nondeterministic finite automaton* ili skraćeno NFA) ovisno o funkciji prijelaza između stanja [86].

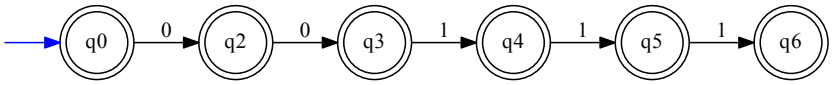
Definicija 3.24. Deterministički konačni automat je petorka $DFA = (S, \Sigma, \delta, s_0, F)$, gdje je S konačni skup stanja, Σ je alfabet, odnosno konačni skup ulaznih simbola, δ je funkcija prijelaza između stanja, odnosno $\delta : S \times \Sigma \rightarrow S$, s_0 je inicijalno stanje $(s_0 \in S)$ i F je skup završnih (prihvatljivih) stanja $(F \subseteq S)$.

Kada se crta graf determinističkog konačnog automata, prijelazna funkcija δ odgovara labeliranom usmjerenom bridu (na strelici je jedan simbol iz alfabeta Σ) između dva čvora (stanja iz S). Kod DFA se iz jednog stanja p za zadani simbol a može prijeći u samo jedno sljedeće stanje q, odnosno rezultat funkcije prijelaza δ iz stanja p i simbola a je uvijek samo jedno stanje $q \in S$. Deterministički konačni automati mogu prihvatiti ili odbaciti neku riječ w koja odgovara nizu simbola iz Σ. Ako se kroz DFA može proći od inicijalnog stanja s_0 do nekog završnog stanja kroz označene prijelaze koji redom odgovaraju nizu simbola koji čine riječ w onda se kaže da DFA prihvaća riječ w. S druge strane, ako se završi u stanju koje nije završno onda se kaže kako DFA ne prihvaća riječ w. Prazna riječ se označava s ε, a ako je DFA prihvaća onda mu inicijalno stanje ujedno pripada i skupu završnih stanja. Skup svih riječi (nizova simbola) koje DFA propušta od inicijalnog stanja s_0 do nekog od završnih ili prihvatljivih stanja naziva se regularni jezik deterministčkog konačnog automata [86].

Strogo gledajući, funkcija prijelaza $\delta : S \times \Sigma \to S$ je potpuna, tj. iz svakog stanja $s \in S$ funkcija δ treba odrediti iduće stanje $s' \in S$ za svaki mogući simbol iz Σ. Zbog ovoga grafički prikaz DFA može biti vrlo složen jer će DFA pokazivati put do završnog stanja za svaku riječ koje prihvaća, ali isto tako mora pokazati put i za svaku riječ koju ne prihvaća. Takvi putevi mogu voditi do stanja iz kojeg nema daljnjeg napretka, a nazivaju se neregularna završna stanja (engl. *sink state* ili *dead state*). Jednom kada se uđe u takvo stanje onda se za svaki sljedeći simbol iz Σ opet ostaje u tom istom stanju. Radi preglednosti grafičkog prikaza DFA mogu se iz njega ukloniti takva neregularna završna stanja, ali se i dalje implicitno pretpostavlja da ona postoje kako bi u njima mogle završiti riječi koje DFA ne prihvaća. Kao primjer razmotrit će se DFA koji ima alfabet $\Sigma = \{0, 1\}$, a prihvaća samo sljedeće riječi: ε, 0, 00, 001, 0011 i 00111. Na slici 3.22 je potpuni prikaz ovoga DFA s neregularnim završnim stanjem (osjenčano sivo), a na slici 3.23 je sažeti prikaz istoga DFA bez neregularnog završnog stanja – pretpostavlja se da sve druge riječi DFA odbacuje.



Slika 3.22: Potpuni prikaz DFA koji prihvaća riječi ε, 0, 00, 001, 0011 i 00111



Slika 3.23: Sažeti prikaz DFA koji prihvaća riječi ε, 0, 00, 001, 0011 i 00111

U grafičkom prikazu DFA inicijalno stanje je označeno s ulaznom plavom strelicom, završno ili prihvatljivo stanje s dva koncentrična kruga, a ostala stanja s običnim krugom. Treba primijetiti kako će se za neregularnu riječ (npr. 0111) u DFA na slici 3.22 završiti u sivo osjenčanom neregularnom završnom stanju. Kod sažetog prikaza na slici 3.23 vidi se da se riječ 0111 odbacuje jer već iz stanja q_2 nema prijelaza sa simbolom 1.

Definicija 3.25. Nedeterministički konačni automat je petorka $(S, \Sigma, \Delta, s_0, F)$, gdje je S konačni skupa stanja, Σ je alfabet, odnosno konačni skup ulaznih simbola, Δ je funkcija prijelaza između stanja, odnosno $\Delta : S \times \Sigma \to P(S)$, s_0 je inicijalno stanje $(s_0 \in S)$ i F je skup završnih (prihvatljivih) stanja $(F \subseteq S)$.

Za razliku od DFA kod nedeterminističkih konačnih automata kodomena prijelazne funkcije Δ je partitivni skup $\mathbf{P}(S)$, odnosno skup svih podskupova od S. Dakle, kod NFA je rezultat funkcije prijelaza iz stanja p za neki simbol a podskup stanja S, a ne nužno samo jedno stanje $q \in S$ kao kod DFA. Regularni jezik nedeterminističkog automata je skup svih riječi koje NFA propušta od inicijalnog stanja s_0 do nekog od završnih odnosno prihvatljivih stanja [86]. Svaki NFA se može pretvoriti u ekvivalentni DFA, ali pritom će deterministički konačni automat obično imati više stanja i prijelaza od početnog nedeterminističkog konačnog automata.

Deterministički i nedeterministički konačni automati prihvaćaju riječi konačne dužine, a za prihvaćanje beskonačno dugih riječi (ω-riječi) razvijeni su posebni ω-automati. Deterministički i nedeterministički Büchijevi automati su vrste ω-automata koji imaju istu sintaksu ako DFA odnosno NFA, ali prihvaćaju beskonačno duge ω-riječi[10] ako se prilikom izvođenja automata beskonačno često ulazi u barem jedno od završnih ili prihvatljivih stanja [87].

3.4.2 *L** algoritam Dane Angluin

*L** algoritam (ili *Lstar*) definira egzaktnu metodu učenja za otkrivanje inicijalno nepoznatog DFA na temelju upita i protuprimjera. Objavila ga je 1987. američka računarska znanstvenica Dana Angluin u radu [88].

Ovaj algoritam formalno definira ulogu *Učenika* koji u interakciji sa sveznajućim (engl. *oracle*) *Učiteljem* želi otkriti inicijalno nepoznati DFA, odnosno njegov regularni jezik (skup

[10] Jedan primjer beskonačno duge ω-riječi je $(ab)^{\omega} = abababababab\ldots$

svih riječi koje taj DFA prihvaća). Glavni preduvjet je da *Učenik* poznaje alfabet Σ nepoznatog regularnog jezika. Uz to *Učitelj* mora poznavati točan regularni jezik koji *Učenik* želi naučiti te nikada ne smije dati krivi odgovor *Učeniku*. Potom *Učenik* kroz interakciju s *Učiteljem* putem iterativnih upita i odgovora postupno otkriva nepoznati DFA i njegov regularni jezik. *Učenik* može postaviti *Učitelju* dvije vrste pitanja – upite o pripadnosti i upite ekvivalencije.

Kod upita o pripadnosti *Učenik* želi saznati pripada li neka riječ regularnom jeziku, a *Učitelj* odgovara s *da* u slučaju da mu pripada ili s *ne* ako mu ne pripada. Kada *Učenik* ima dovoljno prikupljenih podataka generira svoju pretpostavku o traženom DFA. Nadalje, svoju pretpostavku provjerava slanjem upita ekvivalencije *Učitelju*. On zatim mora ocijeniti je li *Učenik* pogodio točni DFA ili nije. Ako je *Učenikova* pretpostavka ispravna onda *Učitelj* odgovara s *da* i algoritam završava, a inače će odgovoriti s protuprimjerom. Pozitivni protuprimjer je riječ iz točnog regularnog jezika koju *Učenikov* predloženi DFA ne prihvaća. S druge strane, negativni protuprimjer može biti neka riječ iz *Učenikovog* predloženog DFA, ali koja ne pripada točnom regularnom jeziku. U oba slučaja *Učenik* mora uključiti dobiveni protuprimjer u svoju bazu znanja i krenuti potom s novim krugom upita o pripadnosti te upita o ekvivalenciji. Na kraju će u konačnom broju koraka *Učenik* uspjeti pogoditi točan DFA odnosno točni regularni jezik.

Treba naglasiti kako L^* algoritam detaljno definira samo ulogu i aktivnosti *Učenika*, ali za *Učitelja* nije predložen algoritam. Ipak, za ulogu *minimalno adekvatnog Učitelja* specificirano je kako treba znati točni DFA te kako treba odgovarati na upite, a uz to se zahtjeva te da uvijek mora davati točne i konzistentne, odnosno dosljedne odgovore.

Opis L^* algoritma

Prije podrobnijeg opisa L^* algoritma treba uvesti potrebne oznake i objasniti mehanizme kojma će se *Učenik* služiti pri učenju nekog nepoznatog regularnog jezika. Za nepoznati regularni jezik koristit će se oznaka U, a pretpostavlja se da U sadrži riječi sastavljene od simbola iz konačnog alfabeta A, a koji je poznat *Učeniku*. L^* algoritam gradi svoju bazu znanja u obliku opservacijske tablice (engl. *observation table*) (S,E,T) u kojoj su svi trenutno prikupljeni podaci o članstvu pojedine riječi u regularnom jeziku U. Opservacijska tablica se sastoji od tri dijela:

- od skupa niza znakova S, koji je neprazan, konačan i prefiksno-zatvoren[11]
- od skupa niza znakova E, koji je neprazan, konačan i sufiksno-zatvoren[12]
- i od funkcije T koja preslikava neki niz znakova[13] $((S \cup (S \cdot A) \cdot E)$ na elemente skupa $\{0,1\}$, a pritom za neki niz znakova u vrijedi $T(u) = 1$ ako i samo ako je u riječ nepoznatog regularnog jezika U

[11] Skup je prefiksno-zatvoren ako i samo ako je svaki prefiks od svakog elementa skupa također element skupa.
[12] Skup je sufiksno-zatvoren ako i samo ako je svaki sufiks od svakog elementa skup također element skupa.
[13] Operator $\cdot$ označava operaciju konkatenacije odnosno spajanja nizova znakova.

U opservacijskoj tablici inicijalno vrijedi $S = E = \{\varepsilon\}$, odnosno S i E sadrže samo prazan niz znakova ε. Ova tablica se može prikazati kao dvodimenzionalno polje u kojem su redci označeni s elementima $s \in S \cup (S \cdot A)$, a stupci su označeni s elementima $e \in E$, a na presjeku retka s i stupca e upisuje se vrijednost $T(s \cdot e)$. Definira se i pomoćna funkcija $redak(s)$ koja ispisuje sadržaj čitavog retka $s \in S \cup (S \cdot A)$. Redci opservacijske tablice se grupiraju na sljedeći način – gornji dio tablice sadrži retke $s \in S$, a donji dio retke $t \in S \cup A$. Pritom su redci iz gornjeg dijela tablice su kandidati za stanja DFA kojeg se treba naučiti, a redci iz donjeg dijela tablice se koriste za izgradnju funkcije prijelaza u tom DFA. Opservacijska tablica je zatvorena ako za svaki $t \in S \cdot A$ postoji $s \in S$ tako da vrijedi $redak(t) = redak(s)$. Drugim riječima iskazano, opservacijska tablica je zatvorena ako se za svaki redak iz donjeg dijela tablice može pronaći redak istog sadržaja u gornjem dijelu tablice. Nadalje, opservacijska tablica je konzistentna ako vrijedi sljedeće: kad god postoje elementi $s_1 \in S$ i $s_2 \in S$ takvi da $redak(s_1) = redak(s_2)$ onda za sve $a \in A$ treba vrijediti $redak(s_1 \cdot a) = redak(s_2 \cdot a)$.

Ako je opservacijska tablica $S(S,E,T)$ zatvorena i konzistentna *Učenik* može iz nje automatski napraviti pretpostavku M o deterističkom konačnom automatu koji bi odgovarao nepoznatom regularnom jeziku U. Dakle, zatvorenoj i konzistentnoj opservacijskoj tablici $S(S,E,T)$ odgovara DFA $M(S,E,T) = (Q,q_0,A,\delta,F)$ gdje je Q konačni skup stanja, q_0 inicijalno stanje, A konačni skup simbola poznatoga alfabeta, δ funkcija prijelaza, a F skup završnih ili prihvatljivih stanja. Kao što je ranije rečeno, elementi skupa stanja Q nalaze se među redcima iz gornjeg dijela opservacijske tablice (S,E,T). Svako stanje $q \in Q$ je kodirano s jedinstvenim sadržajem pojedinog retka $s \in S$, odnosno s vrijednošću $redak(s)$:

$$Q = \{redak(s) : s \in S\} \tag{3.5}$$

Inicijalno stanje q_0 je ono stanje koje odgovara retku označenom s $s = \varepsilon$, odnosno kodiranom sa sadržajem retka $redak(\varepsilon)$:

$$q_0 = redak(\varepsilon) \tag{3.6}$$

Funkcija prijelaza generira se tako da se iz retka s u gornjem dijelu opservacijske tablice, odnosno iz stanja kodiranog s $redak(s)$ prelazi s nekim simbolom $a \in A$ u redak $s \cdot a$ u donjem dijelu opservacijske tablice, odnosno u stanje kodirano s $redak(s \cdot a)$. Pritom treba primijetiti kako je opservacijska tablica zatvorena pa svaki redak iz donjeg dijela tablice ima redak identičnog sadržaja u gornjem dijelu tablice, dakle s funkcijom prijelaza će se sigurno iz nekog stanja $q \in Q$ prijeći u neko već definirano stanje $p \in Q$:

$$\delta(redak(s),a) = redak(s \cdot a) \tag{3.7}$$

A skup završnih stanja F odgovara redcima $s \in S$ gornjeg dijela tablice za koje vrijedi $T(s) = 1$,

odnosno kada je s riječ iz U:

$$F = \{redak(s) : s \in S \text{ i } T(s) = 1\} \tag{3.8}$$

Dakle za provjeru pripadnosti stanja $q \in Q$ skupu završnih stanja $F \subseteq Q$ treba ispitati nalazi li se broj 1 na presjecištu odgovarajućeg retka $s \in S$ i stupca ε, odnosno vrijedi li $T(s \cdot \varepsilon) = T(s) = 1$.

A nakon uvođenja oznaka, opservacijske tablice i svih funkcija koje se koriste prikazat će se svi koraci L^* algoritma koji podrobno opisuje ulogu i aktivnosti *Učenika*:

Algoritam 2 Opis uloge *Učenika* – L^* algoritam

Inicijaliziraj skupove $S = \{\varepsilon\}$ i $E\{\varepsilon\}$.
Izvedi upite o pripadnosti za ε te za svaki $a \in A$.
Izgradi inicijalnu opservacijsku tablicu (S, E, T).

Ponavljaj sve dok *Učitelj* ne odgovori *da* na pretpostavku M:

Ponavljaj dok (S, E, T) nije zatvorena ili nije konzistentna:

Ako (S, E, T) nije konzistentna,
nađi $s_1 \in S$ i $s_2 \in S$, $a \in A$, i $e \in E$ tako da je:
$redak(s_1) = redak(s_2)$ i $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$,
dodaj $a \cdot e$ u E,
i proširi T na $(S \cup (S \cdot A)) \cdot E$ korištenjem upita o pripadnosti.

Ako (S, E, T) nije zatvorena,
nađi $s_1 \in S$ i $a \in A$ tako da je:
$redak(s_1 \cdot a)$ različito od $redak(s)$ za sve $s \in S$,
dodaj $s_1 \cdot a$ u S,
i proširi T na $(S \cup (S \cdot A)) \cdot E$ korištenjem upita o pripadnosti.

Kad je (S, E, T) zatvorena i konzistentna neka joj je odgovarajući DFA $M = M(S, E, T)$.
Napravi pretpostavku M.
Ako *Učitelj* odgovori s protuprimjerom t, onda:
dodaj t i sve njegova prefikse u S,
i proširi T na $(S \cup (S \cdot A)) \cdot E$ korištenjem upita o pripadnosti.

Kada *Učitelj* odgovori s *da* na pretpostavku M zaustavi algoritam i ispiši M.

Može se primijetiti kako se algoritam 2 iterativno izvodi sve dok *Učitelj* ne prihvati *Učenikovu* pretpostavku M o traženom regularnom jeziku kao točnu. Pritom se unutarnja petlja izvodi sve dok trenutna opservacijska tablica ne postane zatvorena i konzistentna. U tom slučaju *Učenik* šalje svoju pretpostavku M na ocjenu *Učitelju*. Ako opservacijska tablica nije konzistentna *Učenik* će tražiti uzrok nekonzistentnosti. To je niz znakova $a \cdot e$ gdje je $a \in A$ i $e \in E$ tako da za neke $s_1 \in S$ i $s_2 \in S$ vrijedi da imaju isti sadržaj redaka $(redak(s_1) = redak(s_2))$, ali

vrijednosti polja na presjecištu retka $s_1 \cdot a$ i stupca e te retka $s_2 \cdot a$ i stupca e nisu iste ($T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$). Taj pronađeni niz znakova $a \cdot e$ se dodaje u skup E, a pritom skup E ostaje sufiksno-zatvoren. Nakon toga treba za sve već neispitane nizove znakova iz $(S \cup (S \cdot A)) \cdot E$ provjeriti pripadaju li nepoznatom regularnom jeziku U. A ako opservacijska tablica nije zatvorena opet treba tome naći razlog. Sada će to biti niz znakova $s_1 \cdot a$ gdje je $s_1 \in S$ i $a \in A$ za koji vrijedi da vrijednost $redak(s_1 \cdot a)$ nije jednaka niti jednom retku iz gornjeg dijela tablice ($redak(s_1 \cdot a) \neq redak(s), \forall s \in S$). Nadalje, utvrđeni niz znakova $s_1 \cdot a$ se dodaje u skup S koji tako ostaje prefiksno-zatvoren te se opet ispituje pripadnost regularnom jeziku U svakog novog niza znakova $(S \cup (S \cdot A)) \cdot E$.

Iz unutarnje petlje algoritma 2 će se naposljetku uvijek izaći jer će trenutna opservacijska tablica postati zatvorena i konzistentna. Zbog toga će *Učenik* naposljetku uvijek moći zadati svoju pretpostavku M i poslati je na ocjenu *Učitelju*. Isto tako, vanjska petlja u algoritmu 2 će se na kraju prekinuti kada *Učenik* postavi točnu pretpostavku M. D. Anlguin je u svom radu [88] dokazala kao će L^* algoritam uspjeti naučiti nepoznati regularni jezik od minimalno adekvatnog *Učitelja* u konačnom broju koraka i vremenu koje je ograničeno polinomnom funkcijom od m i n, gdje je m dužina najduljeg protuprimjera kojeg je dao *Učitelj*, a n je broj stanja minimalnog DFA za nepoznati regularni jezik U.

Primjer izvođenja L^* algoritma

Jednostavniji primjeri, kao npr. uvodni primjer[14] iz izvornog rada D. Angluin [88] mogu se u cijelosti provesti ručno što veoma pomaže u upoznavanju sa svim koracima algoritma. Naravno, ipak se preporučuje korištenje programske implementacija algoritma. U ranije provedenim vlastitim istraživanjima prvo je bila korištena implementacija L^* algoritma u *libalf* okruženju gdje ulogu *Učitelja* preuzima ekspert te ručno odgovara na upite *Učenika* [89, 90]. Kasnije je u sklopu ovog doktorskog istraživanjima napravljena vlastita implementacija L^* algoritma u jeziku *Python*, a potom i predložen i implementiran algoritam *Učitelja* koji automatski odgovara na upite *Učenika* koristeći alat za provjeru modela *Spin*. Ova implementacija će biti predstavljena i podrobnije opisana u poglavlju 7.

Za demonstraciju izvođenja L^* algoritma prikazat će se rezultati učenja regularnog jezika U koji se sastoji od prazne riječi ε te svih nizova znakova iz alfabeta $A = \{a, b\}$ koji imaju točno jedan simbol a te nula ili paran broj simbola b: $U = \{\varepsilon, a, abb, bab, bba, abbbb, babbb, bbabb, \ldots\}$. Prikazat će se i komentirati rezultati dobiveni automatiziranom implementacijom *Učenika* i *Učitelja* iz poglavlja 7.

Na početku *Učenik* samo zna da je jezik $A = \{a, b\}$ i postavlja $S = E = \{\varepsilon\}$. Nakon toga šalje upite o pripadnosti za ε ($T(\varepsilon) = 1$ pa ε pripada U), a ($T(a) = 1$ pa a pripada U) i b

[14] Učenje regularnog jezika U nad alfabetom $\{0, 1\}$ koji se sastoji od svih riječi s parnim brojem nula i parnim brojem jedinica.

($T(b) = 0$ pa b ne pripada U). Potom gradi prvu inačicu opservacijske tablice $T_1 = (S, E, T)$ koja je prikazana u tablici 3.13. U sivo osjenčanom gornjem dijelu tablice ($s \in S$) je zasad samo redak ε, a u donjem dijelu tablice ($s \cdot a \in S \cdot A$) su redci označeni s $\varepsilon \cdot a = a$ i $\varepsilon \cdot b = b$.

Tablica 3.13: Opservacijska tablica T_1

T_1	ε
ε	1
a	1
b	0

Zaključuje se da je T_1 konzistentna jer u gornjem dijelu tablice (*sivo osjenčan*) samo redak označen s ε pa se uopće ne mogu naći dva ista retka. S druge strane, T_1 nije zatvorena jer zadnji redak iz donjeg dijela tablice nema isti sadržaj kao i jedini redak u gornjem dijelu tablice, $redak(b) \neq redak(\varepsilon)$ $(0 \neq 1)$.

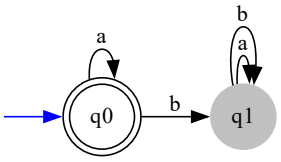
Zato *Učenik* dodaje b u skup S i gradi se nova opservacijska tablica $T_2 = (S, E, T)$. U gornji dio tablice se dodaje redak označen s b, a za njega se već zna da ne pripada U jer $T(b \cdot \varepsilon) = T(b) = 0$. Donji dio tablice $(S \cdot A)$ se sada sastoji od retka a te novih redaka ab i ba za koje treba ispitati pripadnost regularnom jeziku U. Treba primijetiti kako se u donjoj tablici preskače redak koji bi bio označen s $\varepsilon \cdot b = b$ jer se već nalazi u gornjem dijelu tablice. Učitelj javlja da ab ne pripada U $(T(ab \cdot \varepsilon) = T(ab) = 0)$, kao što ni ba ne pripada U $(T(ba \cdot \varepsilon) = T(ba) = 0)$. Izgrađena proširena opservacijska tablica T_2 je prikazana u tablici 3.14.

Tablica 3.14: Opservacijska tablica T_2

T_2	ε
ε	1
b	0
a	1
ba	0
bb	0

Opet se zaključuje da je T_2 konzistentna jer se u gornjem dijelu tablice dva različita retka, a vidi se da je T_2 i zatvorena jer svaki redak iz donjeg dijela tablice ima redak istog sadržaja i u gornjem dijelu tablice: $redak(a) = redak(\varepsilon)$, $redak(ba) = redak(b)$ i $redak(bb) = redak(b)$. Sada *Učenik* daje svoju pretpostavku M o nepoznatom regularnom jeziku U. Prema izrazima 3.5, 3.6, 3.7 i 3.8 gradi se DFA kao $M(S, E, T) = (Q, q_0, A, \delta, F)$ na slici 3.24 gdje je $Q = \{q_0, q_1\}$, $q_0 = q_0$, $A = \{a, b\}$, $F = \{q_0\}$, a funkcija prijelaza δ rezultira s prijelazima: $q_0 \xrightarrow{a} q_0$, $q_0 \xrightarrow{b} q_1$, $q_1 \xrightarrow{a} q_1$ i $q_1 \xrightarrow{b} q_1$. Na slici 3.24 je sivim osjenčano neregularno završno stanje iz kojeg

nema izlaska. *Učitelj* ocjeni pretpostavku M kao krivu i odgovori s negativnim protuprimjerom aa. To je riječ koja nije u U, a prihvaća ju pretpostavka M.



Slika 3.24: DFA prve pretpostavke $M = (S, E, T)$

Kreće se na novu iteraciju i *Učenik* sada mora dodati protuprimjer aa u S kao i sve njegove prefikse (a i ε) ako već nisu u S. Dakle, skup S je sada $S = \{\varepsilon, b, a, aa\}$. Izgrađuje se treća opservacijska tablica $T_3 = (S, E, T)$. Učenik mora zaključiti kako je aa negativan protuprimjer jer ga njegov pretpostavljeni M prihvaća, a ne bi smio jer aa nije dio jezika U, a za ostale nove nizove znakova iz $(S \cup (S \cdot A)) \cdot E$ treba ispitati njihovu pripadnost jeziku U. Niti jedan od novih nizova znakova (ab, aaa i aab) nije u U, a opservacijska tablica T_3 prikazana je u tablici 3.15.

Tablica 3.15: Opservacijska tablica T_3

T_3	ε
ε	1
b	0
a	1
aa	0
ba	0
bb	0
ab	0
aaa	0
aab	0

Vidi se da je T_3 zatvorena jer se za svaki redak u donjem dijelu tablice može naći redak istog sadržaja u gornjem dijelu tablice. Dalje se provjerava je li T_3 konzistentna. Prvo se vidi da vrijedi $redak(\varepsilon) = redak(a)$ $(1 = 1)$, ali $redak(\varepsilon \cdot a) \neq redak(a \cdot a)$ $(1 \neq 0)$. Traži se uzrok te nekonzistentnosti usporedbom vrijednosti polja u tablici i pronađeno je sljedeće: $T(\varepsilon \cdot (a \cdot \varepsilon)) \neq T(a \cdot (a \cdot \varepsilon))$, odnosno $T(a) \neq T(aa)$ $(1 \neq 0)$.

Dakle, *Učenik* dodaje $a \cdot \varepsilon = a$ u skup E i kreće izgradnja četvrte opservacijske tablice. Sada se dodaje novi stupac a u tablicu T_4 pa se povećava broj novih nizova znakova $(S \cdot (S \cdot A)) \cdot E$ za koje treba provjeriti pripadaju li jeziku U. *Učitelj* javi da je bba riječ iz U, a da riječi baa, aba, $aaaa$ i $aaba$ nisu riječi iz U. Trenutna opservacijska tablica T_4 prikazana je u tablici 3.16.

Tablica 3.16: Opservacijska tablica T_4

T_4	ε	a
ε	1	1
b	0	0
a	1	0
aa	0	0
ba	0	0
bb	0	1
ab	0	0
aaa	0	0
aab	0	0

Provjerom *Učenik* utvrdi kako T_4 nije zatvorena jer za redak bb iz donjeg dijela tablice $redak(bb) = 01$ nema redak identičnog sadržaja u gornjem dijelu tablice.

Zato se u skup S dodaje bb pa je sada $S = \{\varepsilon, b, a, aa, bb\}$ te kreće se s izgradnjom pete opservacijske tablice. U donjem dijelu tablice su redci označeni s $S \cdot A = \{ba, ab, aaa, aab, bba, bbb\}$. Provjera se pripadnost novih nizova znakova dobivenih iz $(S \cdot (S \cdot A)) \cdot E$, a *Učitelj* odgovara da $bbaa$, bbb i $bbba$ ne pripadaju nepoznatom regularnom jeziku U. Slijedi prikaz izgrađene opservacijske tablice T_5 u tablici 3.17.

Tablica 3.17: Opservacijska tablica T_5

T_5	ε	a
ε	1	1
b	0	0
a	1	0
aa	0	0
bb	0	1
ba	0	0
ab	0	0
aaa	0	0
aab	0	0
bba	1	0
bbb	0	0

Sada se vidi kako je T_5 zatvorena jer svaki redak iz donjeg dijela tablice (vrijednosti 00 i 10) imaju retke istog sadržaja u gornjem dijelu tablice ($redak(a) = 10$ i npr. $redak(b) = 00$). Dalje, vidi se kako su u gornjem dijelu tablice dva retka ista: $redak(b) = redak(aa)$ $(0 = 0)$ pa se treba provjeriti konzistentnost tablice T_5. Vrijedi $redak(b \cdot a) = redak(aa \cdot a)$ $(00 = 00)$, ali $redak(b \cdot b) \neq redak(aa \cdot b)$ $(01 = 00)$. Sada se traži uzrok nekonzistentnosti, prvo se vidi da

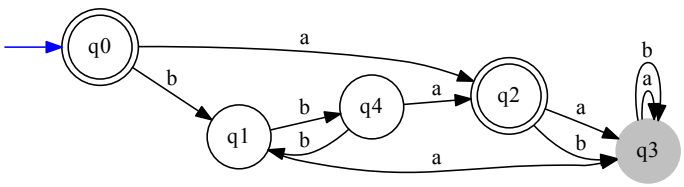
vrijedi $T(b\cdot(b\cdot\varepsilon)) = T(aa\cdot(b\varepsilon))$, odnosno $T(bb) = T(aab)$ $(0 = 0)$, ali zato $T(b\cdot(b\cdot a)) \neq T(aa\cdot(b\cdot a))$, odnosno $T(bba) \neq T(aaba)$ $(1 \neq 0)$.

Dakle, *Učenik* dodaje $b\cdot a = ba$ u skup E i on je sada $E = \{\varepsilon, a, ba\}$. Može se vidjeti da je ostao sufiksno-zatvoren jer su svi sufiksi od ba u E, a to su a i ε. Potom se kreće s konstrukcijom šeste opservacijske tablice provjerom pripadnosti svih novih nizova znakova iz T_5 jeziku U. Ukupno se ispita 6 novih nizova znakova: *baba*, *abba*, *aaaba*, *aabba*, *bbaba* ne pripadaju jeziku U, dok je riječ *bbbba* dio jezika U. U tablici 3.18 je prikazana trenutna opservacijska tablica T_6.

Tablica 3.18: Opservacijska tablica T_6

T_6	ε	a	ba
ε	1	1	0
b	0	0	1
a	1	0	0
aa	0	0	0
bb	0	1	0
ba	0	0	0
ab	0	0	0
aaa	0	0	0
aab	0	0	0
bba	1	0	0
bbb	0	0	1

Svaki redak u donjem dijelu tablice T_5 (vrijednosti 000, 100 i 001) ima redak identičnog sadržaja u gornjem dijelu tablice T_5: $redak(aa) = 000$, $redak(a) = 100$ i $redak(b) = 001$ pa je T_5 zatvorena. Također, T_5 je i konzistentna jer u gornjem dijelu tablice uopće nema dva retka s istim sadržajem. Dakle, *Učenik* može sada ponuditi *Učitelju* svoju drugu pretpostavku M o DFA za koji smatra da prihvaća jezik U. Prema opisanim pravilima dobije se $M = (Q, q_0, A, \delta, F)$ gdje je $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $q_0 = q_0$, $A = \{a, b\}$, $F = \{q_0, q_2\}$, a funkcija prijelaza δ rezultira s prijelazima: $q_0 \xrightarrow{a} q_2$, $q_0 \xrightarrow{b} q_1$, $q_1 \xrightarrow{a} q_3$, $q_1 \xrightarrow{b} q_4$, $q_2 \xrightarrow{a} q_3$, $q_2 \xrightarrow{b} q_3$, $q_3 \xrightarrow{a} q_3$, $q_3 \xrightarrow{b} q_3$, $q_4 \xrightarrow{a} q_2$ i $q_4 \xrightarrow{b} q_1$. Na slici 3.25 je prikazan DFA druge pretpostavke M.



Slika 3.25: DFA druge pretpostavke $M = (S, E, T)$

Učitelj analizira ovu pretpostavku M – vidi se da prihvaća ε i a te sve riječi koje započinju s parnim brojem simbola b i završavaju sa simbolom a. Ipak, brzo zaključuje da pretpostavka M nije točna jer pronalazi i dojavi pozitivan protuprimjer abb.

Zato *Učenik* mora uključiti riječ abb u S kao i sve njene prefikse (ab, a i ε) ako već nisu u S i kreće nova iteracija algoritma. Dakle, sada je skup $S = \{\varepsilon, b, a, aa, bb, ab, abb\}$. *Učenik* zaključuje da je abb pozitivan protuprimjer (pripada U, ali ga njegova pretpostavka M ne prihvaća). Uz to se mora provjeriti pripadnost jeziku U za sve ostale nove riječi iz $(S \cdot (S \cdot A)) \cdot E$. Rezultat tih upita je sljedeći: $abbba$, $abaa$, $ababa$, $abbaa$, $abbaba$, $abbb$ i $abbbba$ ne pripadaju jeziku U. Sada Učenik može izgraditi novu opservacijsku tablicu T_7 prikazanu tablicom 3.19.

Tablica 3.19: Opservacijska tablica T_7

T_7	ε	a	ba
ε	1	1	0
b	0	0	1
a	1	0	0
aa	0	0	0
bb	0	1	0
ab	0	0	0
abb	1	0	0
ba	0	0	0
aaa	0	0	0
aab	0	0	0
bba	1	0	0
bbb	0	0	1
aba	0	0	0
$abba$	0	0	0
$abbb$	0	0	0

Opet se mora provjeriti je li opservacijska tablica T_7 zatvorena i konzistentna. Lako se utvrdi da je zatvorena jer svaki redak iz donjeg dijela tablice T_7 (vrijednosti 000, 100 i 001) ima neki redak s istim sadržajem u gornjem dijelu tablice: $redak(aa) = 000$, $redak(b) = 001$ i npr. $redak(a) = 100$. Nadalje, postoje dva para redaka s istim sadržajem u gornjem dijelu tablice: $redak(a) = redak(abb)$ $(100 = 100)$ i $redak(aa) = redak(ab)$ $(000 = 000)$. Dakle, treba provjeriti konzistentnost T_7, a prvo se kreće od $redak(a) = redak(abb)$. Utvrdi se da vrijedi i $redak(a \cdot a) = redak(abb \cdot a)$, odnosno $redak(aa) = redak(abba)$ (000=000). Potom se vidi kako vrijedi i $redak(a \cdot b) = redak(abb \cdot b)$, odnosno $redak(ab) = redak(abbb)$ (000=000). Nakon toga se prijeđe na $redak(aa) = redak(ab)$ i pokaže se kako vrijedi $redak(aa \cdot a) = redak(ab \cdot a)$, odnosno $redak(aaa) = redak(aba)$ $(000 = 000)$. Ipak, na kraju se utvrdi kako $redak(aa \cdot b) \neq redak(ab \cdot b)$, odnosno $redak(aab) \neq redak(abb)$ $(000 \neq 100)$. Pronađena je

nekonzistentnost, a onda treba još pronaći njen uzrok. Uspoređuju se vrijednosti polja tablice T_7 i utvrđeno je: $T(aa \cdot (b \cdot \varepsilon)) \neq T(ab \cdot (b \cdot \varepsilon))$, odnosno kreće $T(aab) \neq T(abb)$ $(0 \neq 1)$.

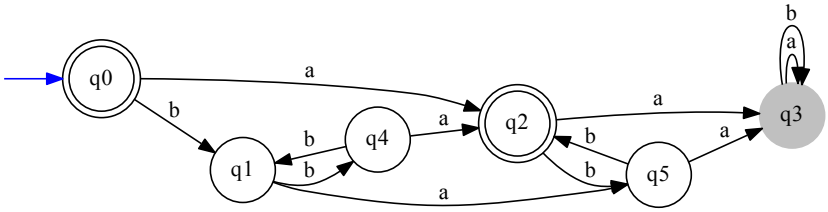
Učenik mora dodati $b \cdot \varepsilon = b$ u skup E (ostaje sufiksno-zatvoren) i krenuti s izgradnjom osme opservacijske tablice T_8. Sada treba ispitati pripadnost jeziku U svih novih nizova znakova koji su rezultat operacije $(S \cdot (S \cdot A)) \cdot E$. *Učitelj* odgovara da su *bab* i *abbbb* riječi koje pripadaju U, a ostali novi nizovi znakova ne pripadaju U (*aaab*, *aabb*, *bbab*, *bbbb*, *abab* i *abbab*). U tablici 3.20 se mogu vidjeti svi podaci iz opservacijske tablice T_8.

Tablica 3.20: Opservacijska tablica T_8

T_8	ε	a	ba	b
ε	1	1	0	0
b	0	0	1	0
a	1	0	0	0
aa	0	0	0	0
bb	0	1	0	0
ab	0	0	0	1
abb	1	0	0	0
ba	0	0	0	1
aaa	0	0	0	0
aab	0	0	0	0
bba	1	0	0	0
bbb	0	0	1	0
aba	0	0	0	0
$abba$	0	0	0	0
$abbb$	0	0	0	1

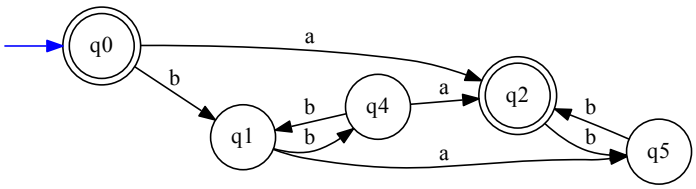
Opservacijska tablica T_8 je zatvorena jer za svaki redak iz donjeg dijela tablice (vrijednosti 0001, 0000, 1000, 0010) postoji barem jedan redak identičnog sadržaja u gornjem dijelu tablice: $redak(aa) = 0000$, $redak(ab) = 0001$, $redak(b) = 0010$ i npr. $redak(a) = 1000$. Na kraju se mora opet provjeriti i konzistentnost opservacijske tablice T_8. U gornjem dijelu tablice postoji samo jedan par redaka s istim sadržajem: $redak(a) = redak(abb)$ $(1000 = 1000)$. Zato se mora redom provjeriti vrijedi li $redak(a \cdot a) = redak(abb \cdot a)$ kao i $redak(a \cdot b) = redak(abb \cdot b)$. Prvo se vidi da vrijedi $redak(a \cdot a) = redak(abb \cdot a)$, odnosno $redak(aa) = redak(abba)$ $(0000 = 0000)$. Potom se utvrdi kako vrijedi i $redak(a \cdot b) = redak(abb \cdot b)$, odnosno $redak(ab) = redak(abbb)$ $(0001 = 0001)$. Dakle, opservacijska tablica T_8 je zatvorena i konzistentna pa *Učenik* može dati svoju treću pretpostavku M, gdje su prema pravilima algoritma elementi DFA $M(S,E,T) = (Q,q_0,A,\delta,F)$ sljedeći: skup stanja Q je $Q = \{q_0,q_1,q_2,q_3,q_4,q_5\}$ gdje je inicijalno stanje je q_0, a skup završnih stanja je $F = \{q_0,q_2\}$, alfabet je $A = \{a,b\}$, dok funkcija prijelaza δ generira prijelaze: $q_0 \xrightarrow{a} q_2$, $q_0 \xrightarrow{b} q_1$, $q_1 \xrightarrow{a} q_5$, $q_1 \xrightarrow{b} q_4$, $q_2 \xrightarrow{a} q_3$, $q_2 \xrightarrow{b} q_5$, $q_3 \xrightarrow{a} q_3$,

$q_3 \xrightarrow{b} q_3$, $q_4 \xrightarrow{a} q_2$, $q_4 \xrightarrow{b} q_1$, $q_5 \xrightarrow{a} q_3$ i $q_5 \xrightarrow{b} q_2$. Ova treća pretpostavka M o DFA koji bi odgovarao nepoznatom regularnom jeziku U prikazan je na slici 3.26.



Slika 3.26: DFA treće pretpostavke $M = (S, E, T)$

Može se primijetiti kako M na slici 3.26 prihvaća ε i a, kao i riječi koje počinju s parnim brojem simbola b i završavaju s a. Osim toga ovaj DFA prihvaća i sve riječi koje se sastoje od niza parnog broja simbola b, jednoga simbola a te opet niza parnog broja simbola b. Isto tako DFA prihvaća i sve riječi koje se sastoje od niza neparnog broja simbola b, jednoga simbola a te potom niza neparnog broja simbola b. Dakle, pretpostavljeni M prihvaća praznu riječ ε te sve riječi koje imaju točno jedan simbol a i nula ili paran broj simbola b, a to je upravo nepoznati regularni jezik U kako je zadan na početku primjera. Stoga *Učitelj* odgovara kako je *Učenikova* pretpostavka M točna i algoritam konačno završava. Može se još prikazati na slici 3.27 i sažeti oblik pretpostavke M kao DFA kojemu je radi preglednosti uklonjeno (skriveno) sivo osjenčano neregularno završno stanje q_3 kao i svi prijelazi koje vode u njega.



Slika 3.27: Sažeti oblik DFA točne pretpostavke $M = (S, E, T)$ sa slike 3.26

Na kraju je *Učenik* uspješno naučio nepoznati regularni jezik U iz trećeg pokušaja te nakon izgradnje osam opservacijskih tablica, a pritom je *Učitelj* ukupno odgovorio i na 37 upita o pripadnosti ispitivane riječi jeziku U.

Uz ovih 37 upita o pripadnosti *Učenik* je samostalno zaključio koji je status dva protuprimjera koje mu je *Učitelj* dao prilikom davanja krive prve i druge pretpostavke. Treba napomenuti kako neke implementacije L^* algoritma, poput one u okruženju *libalf* ne odlučuju o statusu

protuprimjera samostalno nego nakon dobivanja protuprimjera šalju *Učitelju* upit pripada li on nepoznatom regularnom jeziku U.

Nadogradnje L^* algoritma

Razvijene su različite nadogradnje izvornog L^* algoritma od kada je objavljen u [88] u svrhu optimizacije performansi algoritma kao i njegove primjene za učenje drugih vrsta konačnih automata.

Jedna od prvih optimizacija L^* algoritma bila je predložena u radu [91], a tiče se načina uključivanja protuprimjera u opservacijsku tablicu. Naime, autori su željeli osigurati da su redci u gornjem dijelu opservacijske uvijek različiti pa je zbog toga opservacijska tablica uvijek ostaje konzistentna i ne treba trošiti vrijeme na tu provjeru. U ovome pristupu se protuprimjer dodaje u skup oznaka stupaca E umjesto u skup oznaka redaka S.

Autori u [92] predlažu nadogradnju L^* algoritma za učenje Mealyjevih automata[15]. U svom radu predstavljaju dva algoritma, od kojih je prvi direktna prilagodba originalnog L^* algoritma na učenje Mealyjevih automata gdje vrijednosti svakog polja opservacijske tablice nije iz skupa $\{0,1\}$ nego je jednaka vrijednosti izlaza za odgovarajuću riječ. U drugome algoritmu predlažu dodatnu optimizaciju kod obrade protuprimjera (slično kao [91]) kako bi opservacijska tablica uvijek bila konzistentna. Iz protuprimjera se prvo izdvaja njegov najdulji prefiks među svim oznakama redaka, a potom se samo preostali dio protuprimjera upisuje zajedno sa svim svojim sufiksima u sufiksno-zatvoren skup E.

Nadogradnja L^* algoritma za učenje nedeterminističkih konačnih automata predložena je u radu [94]. Autori su pokazali kako su kod konačnih automata s većim brojem stanja (preko 40 stanja) svojim algoritmom naučili NFA automate koji su znatno manji od odgovarajućih DFA automata dobivenih s L^* algoritmom. U radovima [95] i [96] predlažu se algoritmi za učenje ω-regularnih skupova koji sadrže beskonačno duge ω-riječi. U oba rada predložena su novi algoritmi koja su proširenja izvornog L^* algoritma kojima se mogu sintetizirati različite klase ω-regularnih jezika.

Ipak, u ovom doktorskom istraživanju koristit će se izvorni L^* algoritam iz [88] kako je opisan u ovome potpoglavlju. Naime, pokazao se kao adekvatna i pouzdana metoda za stvaranje deterministčkih konačnih automata koji opisuju tijek procesa formativne provjere znanja na temelju nizova pitanja. Na kraju se pronađeni DFA se potom može automatski prevesti u odgovarajući simulacijski ili verifikacijski model koji se potom može simulirati ili verificirati alatom za provjeru modela *Spin*.

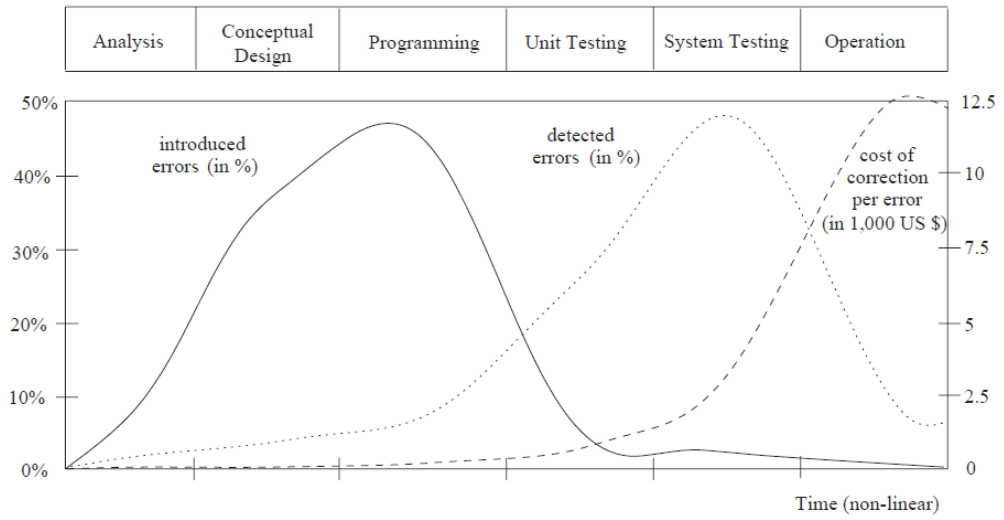
[15] Mealyjev automat je konačni automat koji za razliku od DFA ima zasebnu ulaznu i izlaznu abecedu, a umjesto skupa završnih stanja ima funkciju izlaza. Izlazna vrijednost ovisi o trenutnom stanju kao i o trenutnom ulazu – na svaki prijelaz se upisuje i vrijednost ulaza i izlaza [93].

3.5 Metoda provjere modela

Iznimno je važno korištenje pouzdanih i efikasnih postupaka za brzi pronalazak grešaka i problema u dizajnu ili implementaciji softverskih i hardverskih proizvoda. Svi ti postupci se mogu podvesti pod pojam verifikacije sustava, a u ovom istraživanju posebno će se obraditi verifikacija programske potpore korištenjem automatizirane metode provjere modela.

3.5.1 Verifikacija programske potpore

Glavni zadatak verifikacije programske potpore je što ranije otkrivanje i otklanjanje grešaka u životnom ciklusu nekog softverskog proizvoda ili usluge. Prema grafu na slici 3.28 preuzetom iz rada [97] manji dio grešaka nastaje već u početnoj fazi razvoja prilikom analize zahtjeva, a većina ih se stvara u fazi osmišljavanja dizajna rješenja te prilikom samoga programiranja odnosno implementacije plana rješenja. Greške se detektiraju različitim postupcima, ali veći dio njih se identificira tek u kasnijim fazama testiranja proizvoda. Neke greške prežive sve faze razvoja i otkrivaju se tek u radu nakon što je proizvod već isporučen klijentima. Najbitnije je na kraju spomenuti financijski trošak popravljanja otkrivenih grešaka koji raste s protokom vremena u životnom ciklusu proizvoda što znači da je posebno skupo otklanjati greške u produkcijskoj fazi kada proizvod već koriste klijenti. Takve greške potencijalno mogu dovesti do velikih gubitaka klijenata, a otklanjanje tih grešaka iziskuju i velike troškove proizvođača. Dakle, proizvođači su na ovaj način motivirani da primjenjuju postupke kojima će se što ranije i jeftinije otkloniti potencijalno skupe greške.



Slika 3.28: Generiranje i otkrivanje grešaka te troškovi popravaka u životnom ciklusu programske potpore [97]

Prije svega treba objasniti pojam verifikacije prema definiciji u vodiču za upravljanje projektima PMBOK [98] koji je standard prihvaćen od strane međunarodnog instituta inženjera

elektrotehnike i elektronike IEEE: *"Verifikacijom se ocjenjuje da li neki proizvod, usluga ili sustav udovoljava regulacijama, zahtjevima, specifikacijama ili zadanim uvjetima. To je često interni proces."* Često se informalno miješaju pojmovi verifikacije i validacije pa će se dati i definicija validacije, isto prema [98]: *''Validacijom se osigurava da proizvod, usluga ili sustav odgovara potrebama klijenata i drugih dionika. Validacija je često eksterni proces jer zahtjeva prihvaćanje od strane klijenata."* Umjesto ovih definicija često se koriste kraći opisi pojmova verifikacije i validacije pa se tako može reći kako verifikacija odgovara na pitanje: *"Gradimo li proizvod na ispravan način?"*, dok validacija daje odgovor na pitanje: *"Gradimo li ispravan proizvod?"*.

Treba isto tako napomenuti razlike između različitih postupaka za verifikaciju softverskih sustava prema [97]. Jedan od načina verifikacije softvera koji se često koristi u industriji je stručna provjera (engl. *peer review*). Provodi se tako da neovisni tim inženjera koji nije radio na razvoju proizvoda ručno provjerava njegov programski kod, odnosno vrši statičku analizu napisanog programskog koda. Ovakav način provjere softvera je vrlo čest i prilično je učinkovit što su potvrdile različite studije. Kao što se može pretpostaviti, stručnom provjerom kroz statičku analizu programskog koda teško se mogu pouzdano detektirati prikrivene greške i dublji problemi u programskoj logici, a pogotovo moguće poteškoće kod istovremenog izvođenja i međusobne komunikacije više programa.

Ranije je već spomenuto u potpoglavlju 3.2 kako značajan dio razvoja softvera odlazi na njegovo testiranje. Za razliku od stručne provjere programskog koda, testiranjem se dinamički provjeravaju ishodi izvođenja programa. Postupak testiranja se može dijelom automatizirati automatskim stvaranjem testova kao što je primjerice prikazano kod kombinatornog testiranja. Na kraju se uspoređuju izlazni podaci dobiveni testiranjem programa s očekivani izlaznim podacima specificiranim u dokumentaciji programa. Kao što se može pretpostaviti, u praksi je često neizvedivo i neisplativo stvaranje golemog skupa testova koji bi pokrio sve moguće puteve izvođenja programa. Zato se npr. kod kombinatornog testiranja generira samo podskup svih mogućih testova za koje se očekuje da mogu detektirati zadovoljavajući broj grešaka.

Kod hardverskih proizvoda je vrlo česta metoda verifikacije putem simulacije modela sustava, kojom se zapravo vrši testiranje modela sustava. Vidjet će se kasnije da je ovaj pristup moguć i za simuliranje modela programa korištenjem alata za provjeru modela *Spin*. Simuliranjem se može analizirati ponašanje modela sustava u različitim scenarijima te se pritom mogu uočiti moguće greške i problemi. Opet je vrlo često nemoguće osmisliti sve moguće scenarije koje bi trebalo simulirati kako bi se otkrile sve greške koje postoje u modelu odnosno odgovarajuće greške u implementaciji proizvoda.

Kao dodatna pomoć u verifikaciji sustava mogu se koristiti formalne metode, a to su postupci primijenjene matematike za modeliranje i analizu informacijskih i komunikacijskih sustava [97]. Modeliranjem se izgrađuje model koji na precizan način opisuje ponašanje sustava, a

pritom je model puno koncizniji i jednostavniji od stvarne implementacije sustava. Ipak se zbog velike kompleksnosti sustava obično modeliraju samo neki njegovi kritični dijelovi koje treba iscrpno provjeriti. Sama faza modeliranja sustava je kritična jer se iz krivog modela koji jasno ne odgovara stvarnom sustavu ili dijelu sustava mogu izvući krivi konačni zaključci. Nadalje, već kod modeliranja se mogu uočiti veći problemi i nedosljednosti u logici sustava što može pomoći u otklanjanju grešaka u ranoj fazi razvoja proizvoda. Na kraju treba naći postupak kojim se taj pojednostavljeni model stvarnog sustava može u potpunosti verificirati, a to omogućava metoda provjera modela [31].

3.5.2 Pregled metode provjere modela

Provjera modela (engl. *Model checking*) je automatizirana metoda formalne verifikacije kojim se utvrđuje zadovoljava li model hardverskog ili softverskog sustava neku traženu specifikaciju, odnosno ispituje se da li matematički model sustava M zadovoljava zadanu specifikaciju φ, $M \vDash \varphi$ [31, 97]. Matematički model sustava M gradi se kao usmjereni graf, u obliku labeliranog sustava s prijelazima ili konačnog automata. Specifikacija ili svojstvo φ definira se kao formula matematičke logike, npr. kao formula propozicijske ili temporalne logike. Glavni cilj provjere modela je identificiranje i ispravljanje nepoželjnih pojava u ponašanju sustava, a to su vrlo često prikrivene greške koje se jako rijetko događaju, ali onda dovode do zastoja sustava ili pada sustava s potencijalno katastrofalnim posljedicama. Provjera modela koristi se npr. za verifikaciju distribuiranih softverskih sustava, poput mrežnih protokola, ali i za verifikaciju kritičnih dijelova kompleksnih sustava.

Specifikacija se često zadaje kao pozitivno svojstvo modela sustava, dakle svojstvo koje mora vrijediti u svim izvođenjima modela sustava. Potom će alat za provjeru modela automatski verificirati model sustava za zadanu specifikaciju. Provest će se iscrpna provjera svih mogućih puteva izvođenja modela kako bi pokušao pronaći onaj u kojemu specifikacija nije zadovoljena, odnosno onaj u kojem je zadovoljena negacija zadane specifikacije. S druge strane, specifikacijom se može zadati i negativno, odnosno nepoželjno svojstvo modela sustava. U tom slučaju će alat za provjeru modela pokušati dokazati kako postoji barem jedan put izvođenja modela sustava u kojem je zadovoljena zadana specifikacija što bi potvrdilo da model sustava ima zadano nepoželjno svojstvo.

Primjena metode provjere modela može se raščlaniti na tri faze, a to su faza modeliranja, faza izvođenja i faza analize rezultata [97]. U početnoj fazi modeliranja treba odabrati prikladni alat za provjeru modela i u njegovom jeziku opisati model sustava kojega se želi ispitati. Alati za provjeru modela obično sadrže i simulator pa ga se može koristiti kao pomoć u fazi modeliranja. Kroz korištenje simulacija ekspert može pratiti u svakoj iteraciji izgradnje modela njegovo ponašanje i uskladiti ga s očekivanim osnovnim ponašanjem sustava kojega se modelira. Naravno, ako model sustava ne opisuje na odgovarajući način stvarni sustav onda

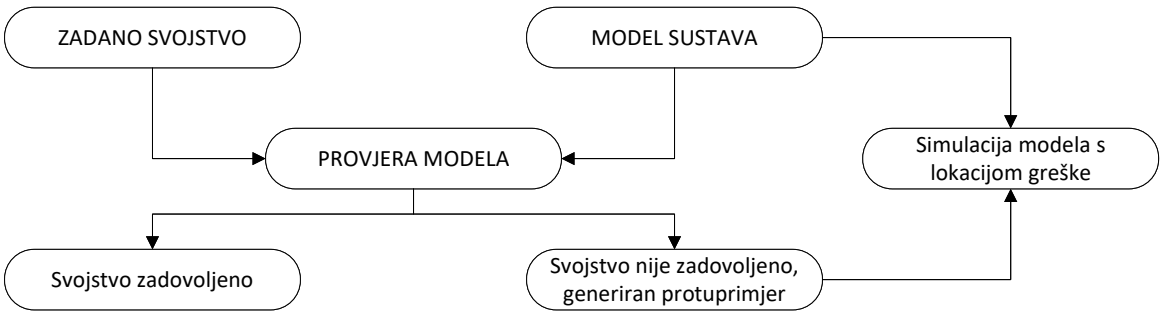
verifikacija takvoga nevaljanog modela sustava nije smislena.

Nakon izgradnje zadovoljavajućeg modela sustava treba definirati specifikacije kako bi se moglo na formalan način provjeriti da li ih model sustava zadovoljava. Pritom ekspert odabire prikladan način zadavanja specifikacija prema mogućnostima koje mu pruža korišteni alat za provjeru modela. U literaturi se najčešće specifikacije zadaju kao formule propozicijske ili temporalne logike kojima se iskazuje pozitivno svojstvo modela sustava.

U fazi izvođenja se pokreće odabrani alat za provjeru modela koji će provesti automatsku verifikaciju modela sustava za zadanu specifikaciju. Alat za provjeru modela može potvrditi kako model sustava zadovoljava zadanu specifikaciju ili će to opovrgnuti s protuprimjerom. Protuprimjer predstavlja instancu ponašanja modela sustava kojim se dokazuje da model sustava ne zadovoljava zadanu specifikaciju.

Zadnja faza je analiza rezultata u kojoj ekspert mora obraditi rezultate verifikacije i donijeti odluku o idućim koracima. U slučaju da je zadano svojstvo zadovoljeno prelazi se na verifikaciju idućeg svojstva, a ako ih više nema uspješno se završava proces provjere modela. Naprotiv, u slučaju da je svojstvo prekršeno ekspert treba provjeriti koji je uzrok tome. Korištenjem simulatora alat za provjeru modela može se točno prikazati trenutak u izvođenju modela sustava u kojem je došlo do greške kao i lista svi koraka koji su doveli do tog trenutka. Nakon utvrđivanja uzroka prekršenog svojstva ekspert će odlučiti treba li popraviti dizajn stvarnog sustava, model sustava ili zadanu specifikaciju te ponoviti cijelu proceduru provjere modela. Treba napomenuti kako alatu za provjeru modela može ponestati računalnih resursa (radne memorije) kako bi proveo verifikaciju do kraja. To se obično može dogoditi kada je model sustava vrlo složen ili kada je specifikacija vrlo široka. Ekspert mora u tom slučaju odlučiti kako će pojednostavniti model sustava ili zadati precizniju specifikaciju te ponoviti cijelu proceduru provjere modela.

Nakon što se osigura da su model sustava zadovoljava sva zadana svojstva proces provjere modela uspješno završava. Kao što se vidi proces provjere modela je iterativan i u pravilu zahtjeva različite prilagodbe modela sustava i pažljivoga odabira i zadavanja specifikacija kako bi ih se uspješno verificiralo. Ukratko se opisani proces provjere može prikazati sa slikom 3.29.



Slika 3.29: Opis procesa provjere modela

Navest će se i tipična svojstva sustava koja se često verificiraju provjerom modela, a to su svojstva dostupnosti, životnosti i sigurnosti. Svojstvom dostupnosti (engl. *reachability*) ispituje se je li moguće da sustav završi u potpunom zastoju (engl. *deadlock*) u kojem se različiti procesi međusobno blokiraju. U popularnim operacijskim sustavima na osobnim računalima u (rijetkim) situacijama kada dođe do zastoja operacijski sustav će problem *jednostavno riješiti* automatskim ponovnim pokretanjem računala. Svojstvo životnosti (engl. *liveness*) je tvrdnja da će se na kraju uvijek dogoditi neki željeni događaj. A s druge strane, svojstvo sigurnosti (engl. *safety*) provjera se tvrdnja da se nikada neće dogoditi neki neželjeni događaj.

Još treba podrobnije objasniti što sve može biti uzrok greške koja se identificirala nakon verifikacije nekog svojstva alatom za provjeru modela. Naime, moguće je da model sustava ne opisuje na zadovoljavajući način stvarni sustav pa treba popraviti ili ponovno izgraditi model sustava i onda ponoviti proces verifikacije svih svojstava. Nadalje, ako je model sustava na odgovarajući način opisuje stvarni sustav onda je moguće da je pronađena greška u modelu sustava odnosno stvarnom sustavu. U tom slučaju treba popraviti dizajn stvarnoga sustava te model sustava te ponoviti proces verifikacije svih svojstava. Na kraju, moguće je da je uzrok greške krivo ili neprecizno zadana specifikacija. U tom slučaju treba na pažljiviji način zadati specifikaciju i ponoviti verifikaciju provjerom modela samo za to promijenjeno svojstvo.

Mogući nedostatak provjere modela je potencijalno golemi prostor pretraživanja koji treba istražiti kod ispitivanja zadovoljivosti zadanih svojstava. Iz tog razloga alati za provjeru modela koriste različite metode optimizacije kako bi se taj prostor smanjio, ali uzimajući i to u obzir teško je efikasno verificirati složene modele sustava metodom provjere modela. Ipak, treba ponoviti kako provjera modela pruža automatiziran i učinkovit način otkrivanja mogućih problema u nekom modelu sustava, a pogotovo kod modela manjih, kritičnih dijelova većih sustava. Ovo može biti motivacija da se provjera modela koristi što ranije u fazi osmišljavanja rješenja dizajna proizvoda kako bi se pomoglo u brzom otkrivanju i otklanjanju grešaka.

3.5.3 LTL – linearna temporalna logika

Linearna temporalna logika (engl. *Linear temporal logic* ili LTL) je proširenje propozicijske logike s modalnim temporalnim operatorima s kojima je moguće ispitati međusobni poredak događaja u nekom beskonačnom linearnom nizu događaja [31, 97]. LTL logikom se mogu definirati temporalna svojstva koje neki model sustava mora zadovoljavati, primjerice "*naposljetku će se dogoditi zadani događaj*". Važno je naglasiti kako je pojam vremena u LTL logici linearan, dakle u nekom beskonačnom nizu događaja svaki trenutak ima samo jedan sljedeći trenutak. Postoji i druga vrsta temporalne logike kod koje se vrijeme ne promatra linearno, nego je u svakom trenutku moguće grananje vremena na više različitih puteva (CTL, engl. *Computation tree logic*). U ovom radu je fokus na LTL logici jer se ona češće koristi kod verifikacije softverskih sustava. Upotreba LTL logike za analizu i verifikaciju složenih distribuiranih sustava prvi

put je predložena i razmatrana u radu [99].

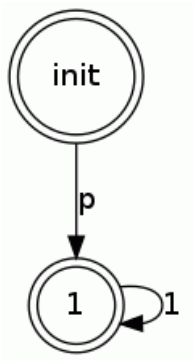
Dobro oblikovana LTL formula se gradi iz propozicijskih formula koje opisuju stanje sustava i temporalnih operatora. Sve formule stanja sustava, uključujući $\top$ i $\bot$ su dobro oblikovane temporalne formule. Ako je α unarni temporalni operator, β binarni temporalni operator, a p i q su dobro oblikovane temporalne formule, onda su dobro oblikovane temporalne formule i $\alpha\, p$, $p\, \beta\, q$, $p \wedge q$ te $\neg p$ [31].

Istinitost neke temporalne formule f se ispituje nad nekom beskonačnom ω-riječi σ. Pritom σ_i označava i-ti element beskonačnog niza elemenata σ, a $\sigma[1]$ označava sufiks σ koji započinje s elementom σ_1 [31]. Ako je temporalna formula f zadovoljena nad nekom ω-riječi σ to se označava sa sljedećim izrazom:

$$\sigma \models f \tag{3.9}$$

Za svaku formulu LTL logike postoji odgovarajući Büchijev automat koji prihvaća upravo one beskonačne ω-riječi koje zadovoljavaju tu LTL formulu. Zato će se uz definicije operatora LTL logike prema [31] prikazati i njihovi odgovarajući Büchijevi automati dobiveni alatom *ltl2ba* [100].

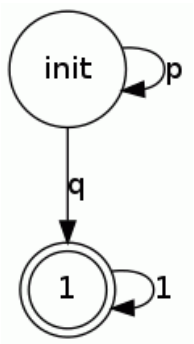
Definicija 3.26. Binarni operator W se naziva *slabi dok* (engl. *Weak Until*), a definira se na sljedeći način: $\sigma[i] \models (p\, W\, q) \Leftrightarrow \sigma_i \models q \vee (\sigma_i \models p \wedge \sigma[i+1] \models (p\, W\, q))$.



Slika 3.30: Büchijev automat za LTL formulu $p\, W\, q$

Na slici 3.30 prikazan je Büchijev automat koji odgovara LTL formuli $p\, W\, q$. Dakle, formula će biti zadovoljena ako je na nekom beskonačnom ω-putu izvođenja Büchijevog automata p istinit barem do trenutka kada q postane istinit, a ako q nikada ne postane istinit onda p mora uvijek na putu biti istinit.

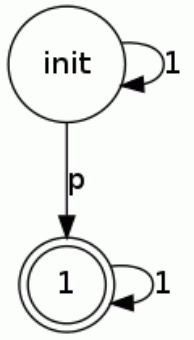
Definicija 3.27. Binarni operator U se naziva *dok* ili *jaki dok* (engl. *Until* ili *Strong Until*), a definira se na sljedeći način: $\sigma[i] \models (p\, U\, q) \Leftrightarrow \sigma[i] \models (p\, W\, q) \wedge \exists j,\, j \geq i,\, \sigma_j \models q$.



Slika 3.31: Büchijev automat za LTL formulu pUq

Büchijev automat koji odgovara LTL formuli pUq prikazan je na slici 3.31. Formula će biti zadovoljena ako je na nekom beskonačnom ω-putu izvođenja Büchijevog automata p istinit barem do trenutka kada q postane istinit, s tim da q mora postati u nekom trenutku istinit. Treba primijetiti kako se kod operatora W (slabija inačica operatora *dok*) uopće ne zahtijeva da q mora postati istinit, ali u tom slučaju p mora ostati istinit na cijelom putu.

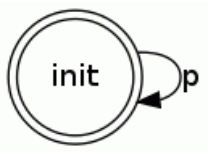
Definicija 3.28. Unarni operator $\Diamond$ se naziva *naposljetku* ili *konačno* (engl. *Eventually*), a definira se na sljedeći način: $\sigma \models \Diamond q \Leftrightarrow \sigma \models (\top U q)$.



Slika 3.32: Büchijev automat za LTL formulu $\Diamond p$

Slika 3.32 prikazuje Büchijev automat koji odgovara LTL formuli $\Diamond p$. Vidi se da će formula biti zadovoljena ako bilo gdje na nekom beskonačnom ω-putu izvođenja Büchijevog automata p postane istinit. Ovom LTL formulom se na jednostavan način može zadati svojstvo životnosti kojim se ispituje hoće li se na kraju uvijek doći do nekog zadanog događaja.

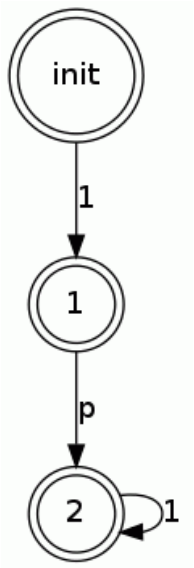
Definicija 3.29. Unarni operator $\Box$ se naziva *uvijek* ili *globalno* (engl. *Always*), a definira se na sljedeći način: $\sigma \models \Box p \Leftrightarrow \sigma \models (pW \bot)$.



Slika 3.33: Büchijev automat za LTL formulu $\Box p$

Na slici 3.33 prikazan je Büchijev automat koji odgovara LTL formuli $\Box p$. Dakle, formula će biti zadovoljena ako je na nekom beskonačnom ω-putu izvođenja Büchijevog automata p uvijek istinit. S ovom LTL formulom se može zadati nepromjenjivo svojstvo (*invarijanta*) koje mora biti zadovoljeno na cijelom putu izvođenja automata.

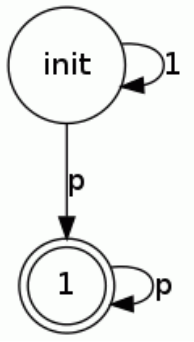
Definicija 3.30. Unarni operator X se naziva *sljedeći* (engl. *Next*), a definira se na sljedeći način: $\sigma[i] \models Xp \Leftrightarrow \sigma_{i+1} \models p$.



Slika 3.34: Büchijev automat za LTL formulu Xp

Büchijev automat koji odgovara LTL formuli Xp prikazan je na slici 3.34. Formula će biti zadovoljena ako je p istinit u sljedećem stanju u nekom beskonačnom putu izvođenja Büchijevog automata.

Korisno je analizirati i sljedeće dualne LTL formule: $\Diamond\Box p$ i $\Box\Diamond p$. S LTL formulom $\Diamond\Box p$ formalizira se sljedeće svojstvo: *"p naposljetku postane uvijek istinit"*. Büchijev automat za formulu $\Diamond\Box p$ prikazan je na slici 3.35.

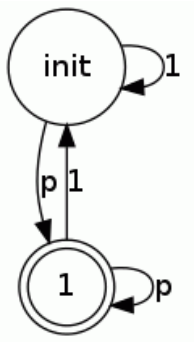


Slika 3.35: Büchijev automat za LTL formulu $\Diamond\Box p$

Dakle, na ω-putu izvođenja u nekom trenutku će p postati istinito te nakon toga p ostaje zauvijek istinito u svim sljedećim stanjima na putu izvođenja kao što se vidi iz Büchijevog automata

na slici 3.35, npr. vrijedit će $(\neg p, \neg p, \neg p, p, p, p, \ldots)$. Ovako se može iskazati svojstvo stabilnosti u kojem nema napretka (npr. kada program završi u beskonačnoj petlji u kojoj se nekoj varijabli naizmjence pridaju samo dvije različite vrijednosti).

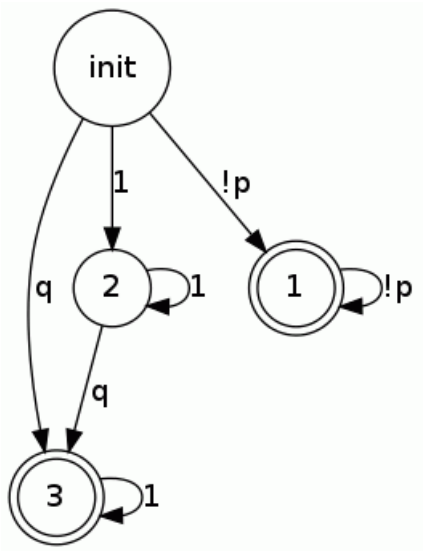
S dualnom formulom $\Box\Diamond p$ se formalno iskazuje svojstvo: *"p uvijek naposljetku postane iznova istinit"*. Njen odgovarajući Büchijev automat nalazi se na slici 3.36.



Slika 3.36: Büchijev automat za LTL formulu $\Box\Diamond p$

Analizirajući Büchijev automat na slici 3.36 vidi se kako na beskonačnom ω-putu izvođenja p beskonačno mnogo puta postati istinit, npr. vrijedit će $(\neg p, \neg p, p, \neg p, p, p, \neg p, p, \ldots)$. Vidi se kako ova LTL formula opisuje stanje napretka u sustavu, odnosno još jedno svojstvo životnosti (npr. semafor beskonačno mnogo puta iznova pokazuje crveno svjetlo).

Korištenjem logičke implikacije moguće je zadati svojstvo sustava $\Diamond p \rightarrow \Diamond q$, odnosno naposljetku p povlači naposljetku q. Drugim riječima, ako u beskonačnom ω-putu izvođenja u nekom trenutku vrijedi da je p istinito, onda će u nekom trenutku na tom putu i q biti istinito. Njen odgovarajući Büchijev automat prikazan je na slici 3.37.

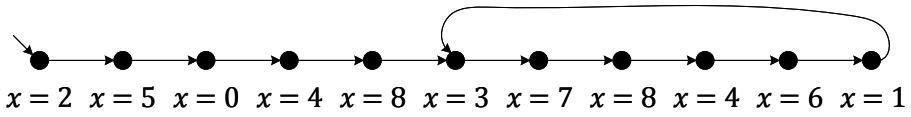


Slika 3.37: Büchijev automat za LTL formulu $\Diamond p \rightarrow \Diamond q$

Jedan način interpretacije ove formule je da će p prvi negdje na ω-putu izvođenja postati istinit,

a q će tek nakon toga postati istinit. Treba pripaziti da formula $\Diamond p \rightarrow \Diamond q$ ne isključuje i druge mogućnosti – prvo q postane istinito pa tek onda p ili i p i q postanu istiniti u istom trenutku. Dodatno treba obratiti pažnju na antecedent implikacije $(\Diamond p)$ jer će implikacija biti istinita u slučaju da je antecedent nije istinit. Dakle, LTL formula $\Diamond p \rightarrow \Diamond q$ će biti zadovoljena i za ω-puteve izvođenja gdje p nikada neće biti istinit bez obzira na to hoće li q postati istinit ili ne. Ovo se može ilustrirati i raspisivanjem razmatrane LTL formule: $\Diamond p \rightarrow \Diamond q \Leftrightarrow \neg\Diamond p \vee \Diamond q \Leftrightarrow \Box\neg p \vee \Diamond q$. To upravo odgovara Büchijevom automatu sa slike 3.37 iz kojeg se vidi kako su pokrivene sve mogućnosti spomenute u ovom kratkom razmatranju.

Primjer 3.2. Za ilustraciju provjere temporalnih svojstava korištenjem LTL formula analizirat će se tipični ispitni zadatak s predmeta *Formalne metode u oblikovanju sustava* na FER-u. Na slici 3.38 je zadan beskonačni ω-put izvođenja nekog automata, a u svakom stanju (označeno crnim kružićem) navedena je trenutna vrijednost varijable sustava x.



Slika 3.38: Zadani beskonačni ω-put izvođenja nekog automata

Treba ispitati da li navedeni put izvođenja automata zadovoljavaju zadane LTL formule:

1. $\Box p$, gdje je $p \equiv x > 1$: ispituje se je li x uvijek veći od 1. Formula nije zadovoljena zbog prvog stanja u kojem vrijedi $x \leq 1$, a to je treće stanje u nizu $(x = 0)$.
2. $\Diamond p$, gdje je $p \equiv x \leq 0$: ispituje se hoće li x naposljetku biti manji ili jednak 0. Formula je zadovoljena jer na naposljetku x postane jednak 0 (treće stanje u nizu).
3. $\Box\Diamond p$, gdje je $p \equiv x = 6$: ispituje se hoće li x beskonačno često naposljetku postati jednak 6. Formula je zadovoljena jer nakon ulaska u beskonačnu petlju ($x = 3$, $x = 7$, $x = 8$, $x = 4$, $x = 6$, $x = 1$, $x = 3$, $x = 7$ ...) uvijek iznova x postaje jednak 6.
4. $\Diamond\Box p$, gdje je $p \equiv x \geq 1$: ispituje se hoće li naposljetku uvijek x biti veći ili jednak 1. Formula je zadovoljena jer se naposljetku ulazi u gore spomenutu beskonačnu petlju u kojoj je x sigurno uvijek veći ili jednak 1.
5. $p\,U\,q$, gdje su $p \equiv x < 6$ i $q \equiv x > 7$: ispituje se je li x manji od 6 sve dok ne postane veći od 7. Formula je zadovoljena jer je u prva četiri stanja u nizu p istinito, a u petom stanju u nizu će q postaje istinito.
6. $p\,W\,q$, gdje su $p \equiv x \leq 5$ i $q \equiv x > 8$: ispituje se je li x manji ili jednak 5 dok ne postane veći od 8, a ako nikad nije veći od 8 onda mora barem uvijek biti manji ili jednak 5. Vidi se da q nikada na postane istinit jer niti u jednom stanju ne vrijedi $x > 8$. U tom slučaju u svim stanjima treba vrijediti p, a i to je prekršeno jer u petom stanju u nizu vrijedi $x = 8$. Zato ova formula nije zadovoljena.

Treba navesti i neka od osnovnih pravila ekvivalencije za linearnu temporalnu logiku pomoću kojih je moguće pojednostavniti ili negirati LTL formule [31, 97]:

$$\neg X\, p \Leftrightarrow X\, \neg p \tag{3.10}$$

$$\neg \Diamond p \Leftrightarrow \Box \neg p \tag{3.11}$$

$$\neg \Box p \Leftrightarrow \Diamond \neg p \tag{3.12}$$

$$\neg \Diamond \Box p \Leftrightarrow \Box \Diamond \neg p \tag{3.13}$$

$$\neg \Box \Diamond p \Leftrightarrow \Diamond \Box \neg p \tag{3.14}$$

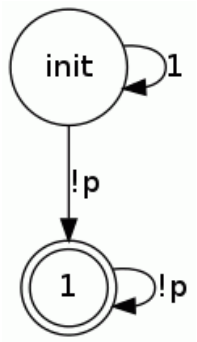
$$\Diamond (p \vee q) \Leftrightarrow \Diamond p \vee \Diamond q \tag{3.15}$$

$$\Box (p \wedge q) \Leftrightarrow \Box p \wedge \Box q \tag{3.16}$$

$$\neg (p\, U\, q) \Leftrightarrow \neg q\, W\, (\neg p \wedge \neg q) \tag{3.17}$$

$$\neg (p\, W\, q) \Leftrightarrow \neg q\, U\, (\neg p \wedge \neg q) \tag{3.18}$$

Primjerice može se potvrditi korištenjem alata *ltl2ba* kako su Büchijevi automati LTL formula $\neg \Box \Diamond p$ i $\Diamond \Box \neg p$ zaista istovjetni (slika 3.39).



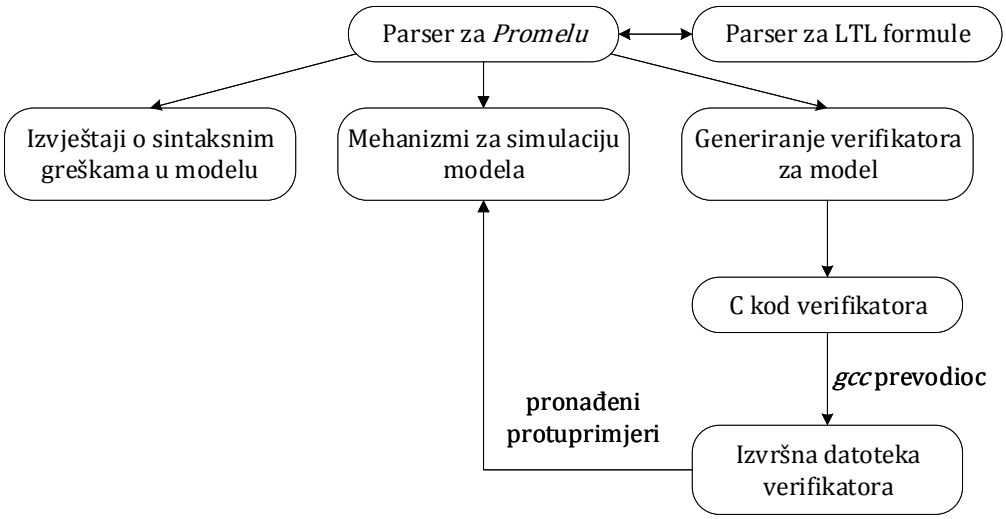
Slika 3.39: Büchijev automat za formulu $\neg \Box \Diamond p$, odnosno $\Diamond \Box \neg p$

Specijalizirani alati za provjeru modela poput alata *Spin* podržavaju zadavanje specifikacija putem LTL formula. Isto tako omogućuju automatsko negiranje ili pojednostavljivanje LTL formula tako da se to ne treba raditi ručno. U sljedećem potpoglavlju će se opisati kako alat *Spin* može provesti automatsku verifikaciju koja će pokazati zadovoljava li model sustava zadanu LTL formulu.

3.5.4 Alat za provjeru modela *Spin* i programski jezik *Promela*

Alat za provjeru modela *Spin* je jedan od industrijskih standarda za verifikaciju i simulaciju modela prvenstveno konkurentnih softverskih sustava. Prvu verziju alata *Spin* (skraćeno od

engl. *Simple Promela Interpreter*) razvio je krajem 1980-ih američki računarski znanstvenik G.J. Holzmann u korporaciji Bell Labs, a još od 1991. alat je slobodno dostupan te redovno se osvježava i unaprjeđuje [87]. U ovom istraživanju se koristio *Spin* kao alat naredbenog retka, ali postoji i verzija alata s grafičkim sučeljem (*iSpin*). Osnovna struktura alata za provjeru modela *Spin* prikazana je na slici 3.40 [31].



Slika 3.40: Osnovna struktura alata za provjeru modela *Spin*

Za definiranje modela koji opisuju ponašanje sustava alat *Spin* koristi vlastiti programski jezik *Promela* (skraćeno od engl. *Process Meta Language*). Modeliranje sustava u *Promeli* se svodi na izgradnju jednoga ili više procesnih modela koji opisuju zadanu funkcionalnost sustava. Jezik *Promela* omogućava modeliranje asinkronih procesa i nedeterminizama što rezultira različitim mogućim putevima izvođenja modela. Na kraju se izgrađeni model u obliku tekstualne datoteke (npr. *model.pml*) može simulirati ili verificirati primjenom alata *Spin*.

Alat *Spin* omogućava tri različita mehanizma simulacija kojima se olakšava analiza modela oblikovanih u *Promeli*, a to su slučajne simulacije, interaktivne simulacije i vođene simulacije:

1. *Slučajna simulacija:* to je osnovni tip simulacije, a može se pokrenuti iz naredbenog retka naredbom *spin.exe model.pml*. Ako je u model koji se simulira uveden nedeterminizam onda će se kod svakog pokretanja slučajne simulacije izvođenje modela odvijati slučajnim odabirom. Ovakve simulacije su vrlo korisne za dobivanje i analiziranje različitih mogućih sekvenci izvršenih naredbi u modelu.
2. *Interaktivna simulacija:* kod ovoga tipa simulacije se izvođenje modela ne odvija automatski slučajnim odabirom trenutno izvršnih naredbi nego se korisniku pruža mogućnost da sam odabere svaki mogući sljedeći korak u simulaciji. Ovaj tip simulacije se pokreće zadavanjem posebne zastavice kod pozivanja programa (*spin.exe -i model.pml*). Na ovaj način ekspert može npr. ručno analizirati i provjeriti neki zanimljivi put izvođenja modela koji bi se rijetko pojavio kod slučajne simulacije. Kroz interaktivnu simulaciju ekspert

može na detaljan način izvršiti inicijalnu provjeru izgrađenog modela prije daljnje verifikacije.

3. *Vođena simulacija:* ovaj tip simulacije se može izvesti tek nakon provedene verifikacije model za zadano svojstvo u slučaju da je pronađena greška u modelu i generiran protuprimjer. Dakle, nakon provedene verifikacije treba ponovno pozvati alat *Spin* naredbom *spin.exe -t model.pml* koji će onda simulirati generirani scenarij protuprimjera i prikazati sve korake izvođenja modela koji su na kraju doveli do identificirane greške. Na ovaj način ekspert može podrobnije analizirati generirani protuprimjer i pokušati utvrditi njegove uzroke kako bi ih u sljedećem krugu verifikacije nastojao otkloniti.

Ipak, treba naglasiti kako čak ni ogroman broj izvršenih slučajnih i interaktivnih simulacija uopće ne garantira detekciju greške u modelu, pogotovo ako se do nje dolazi rijetkim spletom okolnosti. Stoga, treba koristiti puno rigorozniju tehniku verifikacije modela koja će na efikasan način istražiti sve moguće puteve izvođenja kako bi se osiguralo da u definiranom modelu ne postoje kritične greške te da model zadovoljava zadana svojstva.

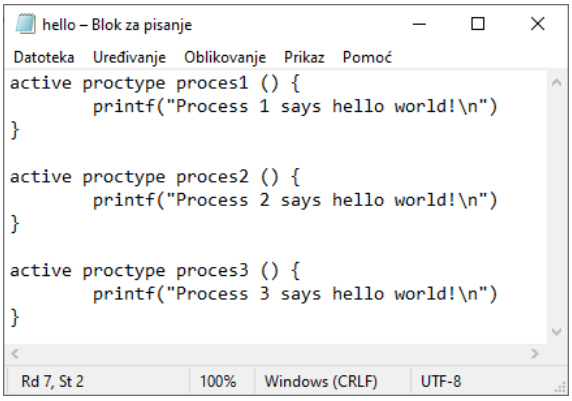
Glavna uloga alata za provjeru modela *Spin* je automatizirana verifikacija procesnih modela napisanih u *Promeli*. Nakon izgradnje modela sustava može se njega ugraditi i zadano svojstvo koje se mora zadovoljiti. Ako nije eksplicitno zadano nikakvo svojstvo prilikom verifikacije će se ipak provjeriti dolazi li model sustava u valjano završno stanje ili dolazi do zastoja. Potom se treba pozvati alat *Spin* naredbom *spin -a model.pml* čime se izgrađeni model u *Promeli* automatski pretvara u odgovarajući C program u tekstualnoj datoteci *pan.c* koji se naziva analizator procesa (engl. *process analyzer*). Nadalje je potrebno korištenjem standardnog prevodioca za jezik C (npr. *gcc*) naredbom *gcc -o pan.exe pan.c* stvoriti izvršnu datoteku analizatora procesa ili verifikatora *pan.exe*. I na kraju se njegovim pokretanjem (npr. naredbom *pan.exe* bez dodatnih parametara) automatski verificira zadani procesni model i generiraju rezultati verifikacije. Postupkom verifikacije se u potrazi za greškom efikasno istražuju svi mogući putevi izvođenja procesnog modela. Ako je greška pronađena (npr. zastoj sustava) analizator procesa generira datoteku s putem do greške (engl. *trail file*) u kojoj se zapisuju svi koraci u izvođenju modela koji su prethodili pojavi greške. Kao što je rečeno ranije, s vođenom simulacijom mogu se detaljno ispisati svi ti koraci.

Osnovni elementi jezika *Promela*

U nastavku će se opisati osnovni gradivni elementi u jeziku *Promela*, a to su asinkroni procesi (`proctype` blokovi), kanali s međuspremnikom i bez njega (`chan` tip podataka), labele, enumeracija (`mtype` blok), nedeterministička selekcija (`if` blok), nedeterministička selekcija s ponavljanjem (`do` blok), petlje (`for` blok), makronaredbe `#define` i `inline`, neutralna naredba preskoka (`skip` naredba), bezuvjetni skok (`goto` naredba), strukturirani podaci (`typedef` tip podataka), tvrdnje za provjeru invarijanti (`assert` blokovi) te deklaracije temporalnih tvrdnji

(`never` i `ltl` tvrdnje). Važno je naglasiti kako su naredbe u jeziku *Promela* izvršne ili blokirajuće. Sintaksa jezika *Promela* je slična standardnom jeziku C, ali semantika jezika *Promela* se bitno razlikuje od semantike jezika C kao što će se pokazati kasnije. Iz jezika C su preuzeti uobičajeni tipovi podataka s iznimkom realnih brojeva, dakle u *Promeli* nema tipova podataka `float` i `double`. Ipak, treba naglasiti kako je moguće u *Promela* modele direktno uključiti fragmente i blokove programskog koda u jeziku C korištenjem naprednijih naredbi kao što su `c_code` i `c_expr`.

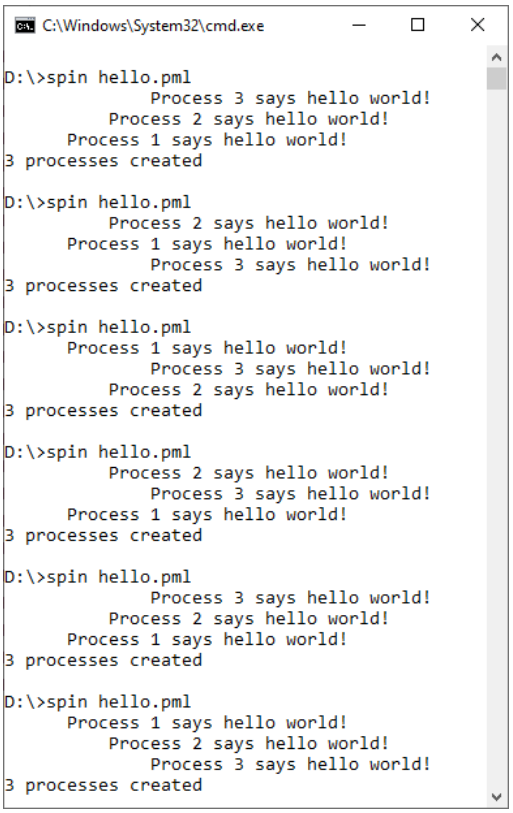
U pravilu svaki model mora imati barem jedan `proctype` blok koji definira proces kojim se opisuje ponašanje modela. Proces se mora pokrenuti kako bi se mogao izvoditi u simulaciji ili verifikaciji, a to se najjednostavnije postiže dodavanjem ključne riječi `active` prije `proctype` definicije procesa. Ako se unutar modela definira više procesa onda će se oni prilikom simulacije izvoditi asinkrono tako da se u svakom trenutku slučajno odabire za izvođenje samo jedna od svih naredbi koje su izvršne. Na ovaj način dolazi do ispreplitanja (engl. *interleaving*) pri izvođenju naredbi iz svih procesa, a to može rezultirati s drukčijim redoslijedom izvođenja naredbi u svakoj slučajnoj simulaciji. U slučaju da su u nekom trenutku sve naredbe u modelu blokirajuće onda se niti jedna ne može izvesti i dolazi do potpunog zastoja sustava. Kao uvodni primjer na slici 3.41 prikazan je jednostavni *Promela* model (datoteka *hello.pml*) s definicijom tri procesa od kojih svaki ispisuje po jednu poruku i potom završava izvođenje.



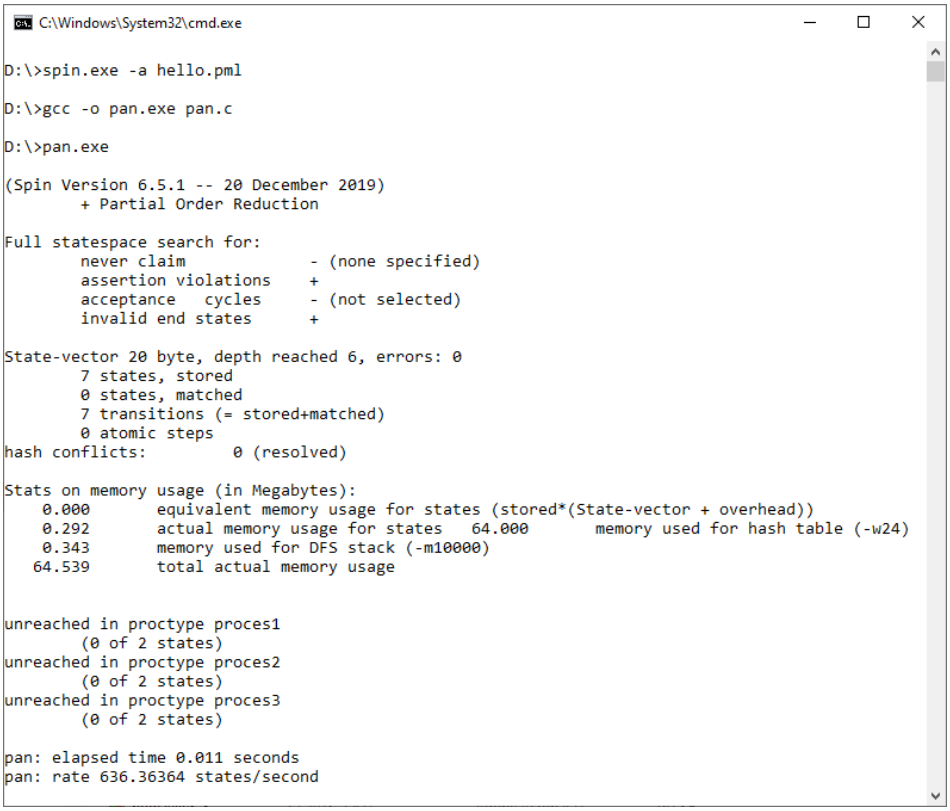
Slika 3.41: Jednostavni primjer *Promela* modela *hello.pml*

Model prikazan na slici 3.41 je simuliran šest uzastopnih puta slučajnim simulacijama u alatu *Spin*, a na slici 3.42 je ispis rezultata izvođenja tih simulacija. Iz prikazanog može se vidjeti kako je gotovo svako izvođenje slučajne simulacije rezultiralo s drukčijim redoslijedom izvođenja procesa, odnosno naredbi `printf`. Naredba `printf` je uvijek izvršna pa se u početnom trenutku izvođenja simulacije procesi mogu krenuti izvršavati bilo kojim mogućim redoslijedom. Ukupno postoji 6 različitih puteva izvođenja ovoga modela (broj permutacija skupa $\{1,2,3\}$), a sa slučajnim simulacijama na slici 3.42 pokriveno je pet od šest permutacija: $proces3 \rightarrow proces2 \rightarrow proces1$ (dva puta), $proces2 \rightarrow proces1 \rightarrow proces3$, $proces1 \rightarrow proces3 \rightarrow proces2$, $proces2 \rightarrow proces3 \rightarrow proces1$ i $proces1 \rightarrow proces2 \rightarrow proces3$. Preostao je još put izvođenja

*proces*3 → *proces*1 → *proces*2 koji bi se dobio u nekoj idućoj slučajnoj simulaciji.



Slika 3.42: Šest uzastopnih simulacije *Promela* modela sa slike 3.41



Slika 3.43: Postupak verifikacije *Promela* modela sa slike 3.41

Slika 3.43 prikazuje postupak verifikacije promatranog *Promela* modela *hello.pml*. U modelu nije eksplicitno navedeno svojstvo koje se treba ispitati pa se verifikacijom traže nevaljana završna stanja u modelu i provjerava jesu li prekršene invarijante zadane kao `assert` tvrdnje koje uvijek moraju biti istinite (u ovom modelu ih nema). Iz izvještaja se vidi kako nije pronađena greška u modelu (navedeno je `errors: 0`). U modelu nema nedostupnih stanja i svi procesi se uredno izvrše do kraja bez potpunog zastoja. Dodatno, izvještaj o verifikaciji navodi sve njene parametre, veličinu prostora stanja koji se pretražio, dosegnutu dubinu te mjerenje performansi kao što je utrošena radna memorija, trajanje verifikacije te brzina obrade stanja. Izvršena je iscrpna pretraga cijelog prostora stanja, a kako se radi o modelu bez deklariranih globalnih i lokalnih varijabli te jednostavne strukture dosegnuta je dubina od samo 6 koraka, a interni vektor stanja kojim verifikator prati različita stanja u sustavu ima veličinu od samo 20 B. Pri izvođenju verifikacije utrošeno je ukupno 64,539 MB, ali od toga je rezervirano uobičajenih 64 MB za tablicu raspršenog adresiranja. Kako bi se uštedjela radna memorija mogla se u ovom jednostavnom slučaju iskoristiti opcija verifikatora *-w* za smanjenje dimenzija tablice raspršenog adresiranja (uobičajena vrijednost je *-w24*). Trajanje verifikacije ovisi o složenosti modela i procesorskoj snazi računala na kojem se ona provodi, a ovdje je zbog jednostavnosti modela trajala vrlo kratko – samo 0,011 s.

Ako se neki proces sastoji od niza naredbi, onda se on izvodi sve dok se ne naiđe do naredbe koja blokira daljnje izvođenje tog procesa. Sintaksa i semantika naredbi inspirirana je Dijkstrinim blokirajućim naredbama (engl. *guarded commands*) oblika $G \rightarrow P$, gdje je G propozicija, a P naredba koja se može izvršiti samo ako je G istinit [31]. Npr. u *Promeli* se može zadati par naredbi `x == 2 -> x=3` sa sljedećim značenjem: sve dok varijabla *x* nije jednaka 2 uvjet `x == 2` će blokirati izvođenje sljedeće naredbe `x=3` kojom se varijabli *x* pridjeljuje vrijednost 3. U slučaju blokade jednoga procesa ne dolazi nužno do potpunog zastoja u sustavu jer se tada potencijalno još mogu izvršavati naredbe iz drugih procesa. Do potpunog zastoja dolazi kada sustav uđe u stanje u kojem su sve trenutne naredbe blokirane. U tom slučaju zaštićena globalna binarna varijabla sustava `timeout` postaje istinita što može pomoći u detekciji i otklanjanju zastoja.

Nedeterminizam se u modele može uvesti i korištenjem nedeterminističke selekcije (blok `if`) i nedeterminističke selekcije s ponavljanjem (blok `do`) koji se mogu vidjeti u tablici 3.21.

Tablica 3.21: Primjeri `if` i `do` blokova

`if` blok:	`do` blok:

`if`
`::  x == 2 -> x=1;`
`::  x < 3 -> x=2;`
`::  else -> x=1;`
`fi;`
	`do`
	`::  x == 2 -> x=3;`
	`::  x > 1 -> x=2;`
	`::  else -> break;`
	`od;`

Blok `if` može imati jednu ili više opcija, a svaka započinje s dvostrukom dvotočkom. Na početku svake opcije je blokirajuća naredba, a ako se ona može izvesti onda se ta opcija može odabrati za izvođenje. Ako su sve opcije blokirane onda se izvodi opcija koja započinje s `else`. Treba primijetiti kako je semantika selekcije *ako-onda-inače* u jeziku C drukčija jer se uvijek odabire prva opcija čiji je uvjet zadovoljen pa nema nedeterminizma. Pretpostavimo u primjeru u tablici 3.21 da je x globalna varijabla koja prije izvođenja `if` bloka ima vrijednost 2. Onda se mogu izvesti prva i druga opcija pa u *Spin* nedeterministički odabire jednu od njih i `if` blok se završava.

Blok `do` ima istu semantiku kao i blok `if` samo što se nakon prvog izvođenja ponavlja sve dok se iz petlje ne izađe s naredbom `break`. Ako opet pretpostavimo da je na početku globalna varijabla $x = 2$ onda će u svakoj iteraciji `do` bloka iz tablice 3.21 nedeterministički izabrati jedna od dvije prve opcije jer će uvijek biti izvršne. Izvođenjem ovog bloka će se varijabla x stalno mijenjati slučajnim odabirom između vrijednosti 2 i 3.

Može se razmotriti i slučaj *Promela* modela s globalnom varijablom $x = 2$ i dva procesa – u prvom je samo `if` blok, a u drugom samo `do` blok iz tablice 3.21. Sada je broj mogućnosti izvođenja modela mnogo veći jer se u svakom trenutku mogu ispreplitati redoslijedi izvođenja trenutno izvršnih naredbi iz oba procesa. U nekim simulacijama takvoga modela moguće je da drugi proces zauvijek ostane u petlji, ali u nekima je moguće i da se `do` blok prekine i tako oba procesa uredno završe. Naime, na početku svake simulacije je $x = 2$ pa se može izvršiti bilo koja od prvih dviju opcija u oba selekcijska bloka. Ako se prva izvede prva opcija `if` bloka onda se postavi $x = 1$ i prvi proces završava. U drugom procesu u tom trenutku prve dvije opcije blokiraju pa se aktivira treća `else` opcija te izvođenjem naredbe `break` za izlazak iz `do` bloka i drugi proces isto završava.

Ako to model zahtjeva neki niz naredbi se može spojiti u jednu složenu naredbu koja se potom izvodi nedjeljivo, odnosno bez ispreplitanja s trenutno izvršnim naredbama iz drugih procesa. Kako bi se to postiglo treba neki niz naredbi uklopiti u `atomic` ili `d_step` blok, a pritom je `d_step` blok stroži oblik složene naredbe jer se uvijek izvodi deterministički. Primjerice kada krene izvršavanje nedjeljivog bloka naredbi `atomic{x++; x-; x++;}` sigurno će se izvršiti ove tri operacije navedenim redom, a svi drugi procesi će čekati na svoj red izvođenja dok ovaj `atomic` blok ne završi.

Procesi u modelu mogu razmjenjivati podatke slanjem i primanjem poruka kroz komunikacijske kanale s i bez međuspremnika opisane s tipom podatka `chan`. Kanali koji imaju međuspremnik se definiraju se npr. kao `chan kanal1 = [N] of {mtype}`. U ovom slučaju zadani kapacitet kanala je N poruka, a svaka poruka se sastoji od jedne simboličke vrijednosti iz enumeracije `mtype` (engl. *message type*). Operatori `!` i `?` se koriste u naredbama za slanje i primanje poruka. Npr. ako je zadana enumeracija `mtype {poruka1,poruka2}` onda se naredbom `kanal1!poruka1;` šalje `poruka1` u kanal, a naredbom `kanal1?x;` neki drugi proces

može pročitati i maknuti tu poruku iz kanala te spremiti ju u lokalnu varijablu x tipa `mtype`. Ovakav način slanja poruka predstavlja asinkronu komunikaciju među procesima jer se poruka neće nužno odmah primiti i pročitati čim se pošalje u komunikacijski kanal. Ponašanje kanala s međuspremnikom se može opisati kao red čekanja u kojem će se prva poruka poslana u kanal i prva primiti (engl. *first in first out* ili *FIFO*). Ako dođe do prepunjenja kanala s porukama verifikator će javiti grešku, ali to se može izbjeći ako se kod stvaranja C programa verifikatora postavi posebna zastavica kojom će se sve poruke koje se šalju u popunjeni kanal izgubiti (*spin.exe -a -m model.pml*).

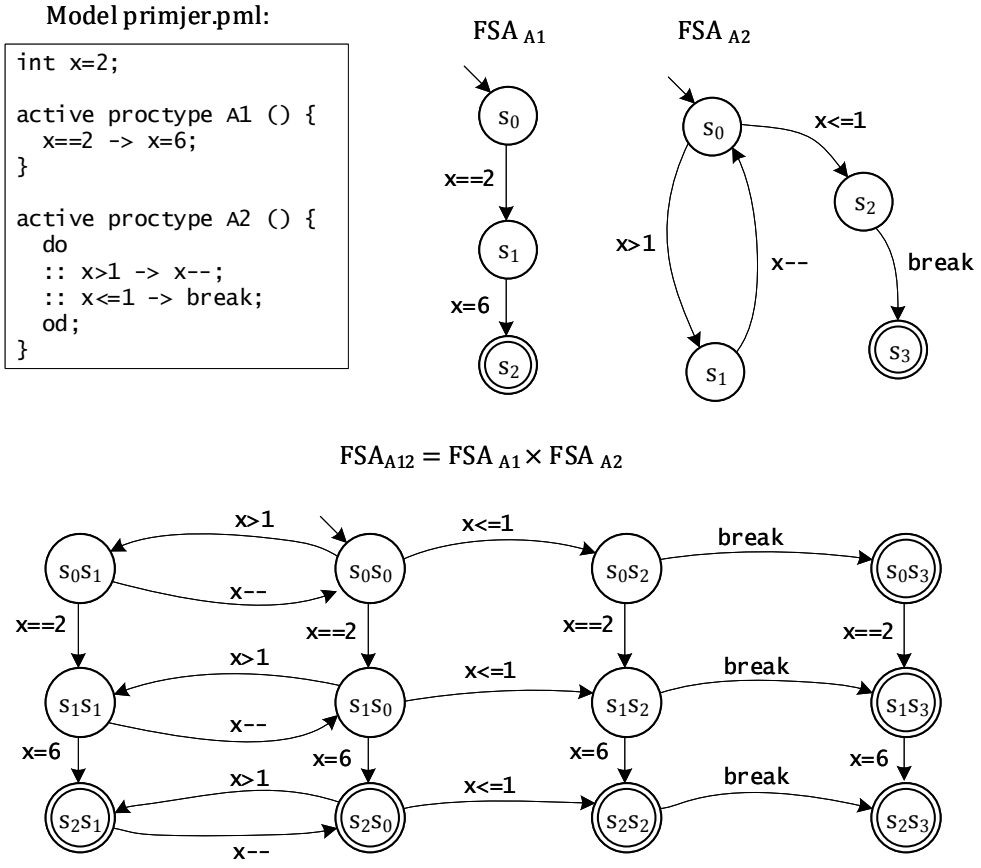
S druge strane kanali bez međuspremnika imaju kapacitet 0, odnosno ne mogu uopće spremati poruke. Takav kanal se može definirati npr. kao `chan kanal2 = [0] of {mtype}`. Ovi kanali se koriste za sinkronu komunikaciju među procesima jer se slanje i primanje poruke uvijek mora odviti u jednom koraku, sinkrono i nedjeljivo. Tek kada jedan proces može pročitati poruku onda ju drugi proces može poslati, a inače su naredbe slanja i primanja poruke blokirane.

Asinkrono izvođenje modela u *Promeli*

U prethodnim razmatranjima se nekoliko puta naglasilo kako se modeli u *Promeli* izvode s alatom *Spin* na asinkroni način, odnosno tako da se isprepliću sve trenutno izvršne naredbe. Ovo ponašanje se može formalnije iskazati kroz asinkroni produkt konačnih automata koji predstavlja graf dostupnosti svih stanja u modelu [31]. Prvo će se svaki `proctype` proces u modelu opisati kao konačni automat $FSA = (S, s_0, L, T, F)$, gdje je S skup stanja, s_0 inicijalno stanje, L skup labela (naredbi), T skup prijelaza između stanja $T \subseteq (S \times L \times S)$, F skup završnih stanja $F \subseteq S$. Izvođenjem svake naredbe se mijenja stanje sustava od nekog inicijalnog stanja s_0 do nekog od završnih stanja s_n. Svaki prijelaz se zapisuje kao trojka (s_1, l, s_2), gdje je s_1 trenutno stanje, s_2 iduće stanje, a l labela koja je jednaka naredbi koja se pritom izvodi. Ako se *Promela* model sastoji od više procesa onda će alat *Spin* izračunati asinkroni produkt svih njihovih konačnih automata i kroz njega tražiti sve moguće puteve izvođenja modela.

Asinkroni produkt konačnog skupa konačnih automata $A_1, \ldots, A_n$ je opet konačni automat $A = (S, s_0, L, T, F)$. Skup stanja $A.S$ jednak je kartezijevom umnošku skupova stanja automata $A_1, \ldots, A_n$ $(A_1.S \times \ldots \times A_n.S)$, a početno stanje $A.s_0$ je n-torka početnih stanja automata $A_1, \ldots, A_n$ $(A_1.s_0, \ldots, A_n.s_0)$. Nadalje, skup labela $A.L$ je unija svih skupova labela iz automata $A_1, \ldots, A_n$ $(A.L = A_1.L \cup \cdots \cup A_n.L)$, a skup prijelaza $A.T$ odgovara skupu trojki $((x_1, \ldots, x_n), l, (y_1, \ldots, y_n))$ takvih da $\exists i$, $1 \leq i \leq n$, $(x_i, l, y_i) \in A_i.T$ i $\forall j$, $1 \leq j \leq n$, $j \neq i \rightarrow (x_j \equiv y_j)$. Npr. ako je trenutno stanje automata A $(A_1.s_0, A_2.s_1, A_3.s_1)$ i postoji prijelaz (s_0, l, s_1) u skupu prijelaza automata A_1 onda će za tu labelu l automat A prijeći u stanje $(A_1.s_1, A_2.s_1, A_3.s_1)$. Dakle, sa svakim prijelazom u asinkronom produktu automata A mijenja se najviše jedan element u n-torci trenutnog stanja. Skup završnih stanja $A.F \subseteq A.S$ su sva stanja $(A_1.s, \ldots, A_n.s)$ gdje bilo koji element $A_i.s$ pripada skupu završnih stanja u automatu A_i $(A_i.s \in A_i.F)$. Kao

primjer je na slici 3.44 ilustriran asinkroni produkt dva konačna automata.



Slika 3.44: Asinkroni produkt dva konačna automata

Slika 3.44 prikazuje *Promela* model *primjer.pml* s dva procesa, njihove odgovarajuće konačne automate FSA_{A1} i FSA_{A2} kao i njihov asinkroni produkt $FSA_{A12} = FSA_{A1} \times FSA_{A2}$. Vidi se da se ovakav model može asinkrono izvesti na različite načine što se može pokazati slučajnim simulacijama s alatom *Spin*.

Svako izvođenje konačnog automata $FSA = (S, s_0, L, T, F)$ je uređeni skup prijelaza, odnosno niz prijelaza $\{(s_0, l_0, s_1), (s_1, l_1, s_2), (s_2, l_2, s_3), \ldots\}$. Izvođenje konačnog automata se prihvaća ako se sa zadnjim prijelazom u nizu ulazi u jedno od završnih stanja automata. U primjeru na slici 3.44 uz početno zadano globalnu varijablu $x = 2$ jedan od mogućih izvođenja asinkronog produkta FSA_{A12} je niz $\{((s_0,s_0), x > 1, (s_0,s_1)), ((s_0,s_1), x == 2, (s_1,s_1)), ((s_1,s_1), x --, (s_1,s_0)), ((s_1,s_0), x \leq 1, (s_1,s_2)), ((s_1,s_2), x = 6, (s_2,s_2)), ((s_2,s_2), break, (s_2,s_3))\}$. U ovom izvođenju varijabla x ima redom sljedeće vrijednosti: $x = 2$, $x = 1$ i na kraju $x = 6$. Treba primijetiti što bi se dogodilo u istom modelu ako je zadana početna vrijednost globalne varijable $x = 1$. Iz početnog stanja (s_0, s_0) bio bi moguć samo prijelaz $((s_0,s_0), x \leq 1, (s_0,s_2))$ jer su ostali prijelazi blokirani. Potom se još jedino može izvesti neblokirani prijelaz $((s_0,s_2), break, (s_0,s_3))$ i izvođenje staje u stanju (s_0, s_3), a sva ostala stanja se ne bi mogla nikada doseći.

Postupci verifikacije s alatom *Spin*

Osim zastoja sustava verifikacijom se mogu provjeriti i složenija svojstva sustava definiranjem invarijanti. To su tvrdnje koje moraju vrijediti za sve puteve izvođenja modela. Najjednostavniji način zadavanja invarijanti je kroz `assert` naredbe kojima se zadaje neka tvrdnja koja uvijek mora biti istinita. Npr. želimo zadati tvrdnju da studenti na predmetu *Osnove elektrotehnike* ne mogu ostvariti ukupni broj bodova veći od 100. U *Promeli* se to može zadati tvrdnjom `assert(x<=100)` uz definiranu varijablu x tipa *int* u koju se zapisuje ukupni broj bodova. Automatizirani postupak verifikacije će potom krenuti istraživati sve puteve izvođenja modela. Kad se u nekom od izvođenja tvrdnja prekrši verifikacija vraća protuprimjer $x > 100$.

Tablica 3.22: Primjeri provjere tvrdnji u *Promela* modelima

datoteka *model1.pml*:	datoteka *model2.pml*:

`int x=0;`
`active proctype proces1 () {`
`  do`
`  ::  x=74;`
`  ::  x=101;`
`  ::  x=112;`
`  od;`
`  assert(x<=100);`
`}`
	`int x=0;`
	`active proctype proces1 () {`
	`  do`
	`  ::  x=74;`
	`  ::  x=101;`
	`  ::  x=112;`
	`  od;`
	`}`
	`active proctype monitor () {`
	`  assert(x<=100);`
	`}`

Treba pažljivo odabrati mjesto u modelu u koji će postaviti `assert` naredba kako bi bili sigurni da će verifikacija otkriti moguću grešku. Za primjer razmotrit će se rezultati verifikacije naizgled sličnih modela prikazanih u tablici 3.22. Verifikacijom prvoga modela neće biti dojavljena greška o kršenju tvrdnje $x \leq 100$, ali će se dati upozorenje kako se naredba `assert(x<=100);` ne može doseći. Naime, očito je kako će varijabla x u nekim iteracijama `do` bloka poprimiti vrijednosti $x > 100$, ali nema izlaza iz `do` bloka pa se `assert` tvrdnja nikada ne izvede. U drugom modelu iz tablice 3.22 (*model2.pml*) definirana su dva procesa, u jednom je isti `do` blok kao i u prvom modelu, a u drugom je samo `assert` tvrdnja. Sada će verifikacija uspjeti pronaći grešku jer će se zbog isprepletanja trenutno izvršnih naredbi iz prvog i drugog procesa detektirati put izvođenja u kojem je $x > 100$, a potom se prekrši zadana tvrdnja. Štoviše, ako verifikator pokrenemo s naredbom *pan.exe -e* onda će se verifikacijom tražiti sve greške u modelu. U tom slučaju verifikacija će pronaći dva protuprimjera ($x = 101$ i $x = 112$) i generirati dva odgovarajuća puta izvođenja koji se mogu simulirati s alatom *Spin*.

Složenija svojstva sustava se mogu zadavati i LTL formulama koje su mnogo ekspresivnije od `assert` tvrdnji, a proces verifikacije se pritom odvija na rigorozniji način. Naime, u verifi-

kaciji će se s izvođenjem svake naredbe u modelu provjeravati vrijedi li svojstvo zadano LTL formulom. S druge strane, svojstva zadana kao `assert` tvrdnje će se provjeravati tek kada i ako dođu na red u putu izvođenja modela.

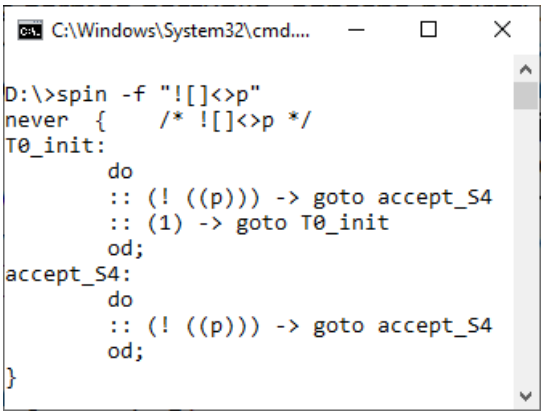
Kako bi se provjerila zadovoljivost neke formule LTL logike alat *Spin* je prvo automatski prevodi u odgovarajuću `never` tvrdnju. To je zaseban proces u *Promela* modelu kojem se može definirati temporalno svojstvo u obliku Büchijevog automata koji prihvaća beskonačne ω-riječi. Prema izvornoj semantici `never` tvrdnje to je opis negativnog svojstva koje model nikada ne smije zadovoljiti, a zato se i koristi engl. naziv *never*.

Uobičajeni postupak verificiranja modela za neko svojstvo zadano preko LTL formule uključuje sljedeće korake:

1. Zadavanje LTL formule kao pozitivnog temporalnog svojstva koje model sustava mora zadovoljiti.
2. Negiranje zadanog pozitivnog temporalnog svojstva i pretvaranje te negirane LTL formule u odgovarajući `never` blok koji se ubaci u *Promela* model.
   - Ovaj korak se može i automatizirati korištenjem `ltl` tvrdnje kojom se izravno zadaje pozitivno temporalno svojstvo sustava u model, a potom *Spin* prilikom verifikacije automatski negira zadanu LTL formulu te stvara njen odgovarajući `never` blok u zasebnoj datoteci.
3. Stvaranje verifikatora i pokretanje verifikacije modela za zadano svojstvo, a pritom verifikator traži upravo puteve izvođenja modela sustava u kojem je zadovoljena negacija zadane LTL formule iz 1. koraka.
   - Ako je pronađen put izvođenja u kojem je zadovoljena negacija LTL formule iz 1. koraka verifikacija se zaustavlja s dojavljenom greškom jer je `never` blok zadovoljen. Generira se protuprimjer kojim se pokazuje zadana LTL formula iz 1. koraka nije zadovoljena, kao i put izvođenja do te greške.
   - Ako nije pronađen put izvođenja u kojem je zadovoljena negacija LTL formule iz 1. koraka verifikacija uspješno završava bez prijavljene greške. Dakle, ako negacija LTL formule nije zadovoljena onda model sustava zadovoljava svojstvo zadano LTL formulom.
   - A ako verifikacija nije završila do kraja jer su prilikom iscrpne provjere svih puteva izvođenja sustava potrošeni svi memorijski resursi na raspolaganju *Spinu* onda treba ponovno stvoriti i pokrenuti verifikator korištenjem različitih dodatnih opcija za optimizaciju. Preporučuje se i pojednostavljenje modela ili redefiniranje zadane LTL formule kako bi se smanjio prostor pretrage svih stanja modela sustava.

Treba naglasiti kako na početku zadavanja LTL formule treba dobro razlučiti je li zadano svojstvo pozitivno ili negativno. Pozitivno svojstvo opisuje dobro i željeno ponašanje sustava, a negativno svojstvo daje opis lošeg i neželjenog načina ponašanja sustava. Samo ako je za-

dano svojstvo pozitivno onda treba provesti drugi korak iz gornjeg postupka, odnosno negirati zadanu LTL formulu. Inače se negativno svojstvo zadano LTL formulom treba izravno prevesti u `never` tvrdnju. Neka se uzme za primjer pozitivno temporalno svojstvo sustava definirano LTL formulom $\Box\Diamond p$ (uvijek će naposljetku iznova vrijediti p). Dakle, odgovarajuća `never` tvrdnja opisuje negaciju zadane LTL formule $(\neg\Box\Diamond p)$, a generiraju se automatski pozivanjem alata *Spin* naredbom *spin.exe -f "![]<>p"*. Dobivena `never` tvrdnja prikazana je na slici 3.45, a može se vidjeti kako odgovara Büchijevom automatu za istu LTL formulu sa slike 3.39.



Slika 3.45: Generiranje `never` tvrdnje za LTL formulu $\neg\Box\Diamond p$

Kao što je ranije spomenuto `never` tvrdnja se definira kao zaseban proces u *Promela* modelu. Ona se izvodi samo u postupku verifikacije, dok se kod slučajnih i interaktivnih simulacija uopće ne izvodi. Prilikom izvođenja *svake* naredbe, odnosno *svakog* prijelaza u modelu provjerava se je li zadovoljena `never` tvrdnja. Formalnije iskazano, prilikom verifikacije modela za pozitivno svojstvo zadano LTL formulom f računa se sinkroni produkt S od `never` tvrdnje B koja opisuje LTL formulu $\neg f$ i asinkronog produkta svih ostalih n automata u modelu $\prod_{i=1}^{n} A_i$, odnosno $S = B \otimes \prod_{i=1}^{n} A_i$. Na kraju verifikator traži protuprimjer, odnosno provjerava vrijedi li za neki put izvođenja σ od S: $\sigma \models \neg f$. Ako protuprimjer nije pronađen onda verifikacija završava bez pronađene greške pa zaključujemo da model sustava S zadovoljava pozitivno svojstvo zadano LTL formulom f, odnosno $S \models f$.

Verifikacijom se mogu prihvatiti i konačna izvođenja kao i beskonačna ω-izvođenja modela. Alat *Spin* koristi posebno pravilo o ponavljanju (engl. *stutter extension rule*) kojim se svako konačno izvođenje modela σ može proširiti u beskonačno ω-izvođenje [31]. Pritom se samo završno stanje s_n iz konačnog izvođenja σ beskonačno puta ponavlja s neutralnim prijelazom ε natrag u s_n, a tako se dobije odgovarajuće beskonačno ω-izvođenje $\sigma, (s_n, \varepsilon, s_n)^{\omega}$. U jeziku *Promela* je `skip` neutralna naredba preskoka.

Primjer 3.3. Kao složeniji primjer provest će se verifikacija konačnog automata na slici 3.25 iz poglavlja 3.4.2. Taj DFA predstavlja drugu pretpostavku u procesu učenja L^* algoritmom automata s alfabetom $\{a, b\}$ koji bi trebao prihvaćati sve riječi koje imaju samo jedan simbol a

i nula ili paran broj simbola b, a trebao bi prihvaćati i praznu riječ ε.

```
#define p (counter[0]+counter[1] <= 2000)
#define q ((counter[0] == 0 && counter[1] == 0) || (counter[0] == 1 && counter[1] % 2 == 0))

inline label(x, next_state_status) {
        atomic {
                if
                :: x == a -> counter[0]++; printf("a\n");
                :: x == b -> counter[1]++; printf("b\n");
                fi;
                end = next_state_status;
        }
}

mtype {a,b}
int end;
int counter[2];

ltl p1 {!(p U (end == 1 && !q))}
ltl p2 {!(p U (end == 2 && q))}

active proctype automaton () {
q0:     if
        :: label(a,1); goto q2;
        :: label(b,0); goto q1;
        :: goto end_OK;
        fi;
q1:     if
        :: label(a,2); goto q3;
        :: label(b,0); goto q4;
        fi;
q4:     if
        :: label(a,1); goto q2;
        :: label(b,0); goto q1;
        fi;
q2:     if
        :: label(a,2); goto q3;
        :: label(b,2); goto q3;
        :: goto end_OK;
        fi;
q3:     if
        :: label(a,2); goto q3;
        :: label(b,2); goto q3;
        fi;
end_OK: end = 1;
}
```

Slika 3.46: *Promela* model za verifikaciju DFA sa slike 3.25

Na slici 3.46 je izgrađeni *Promela* model *verifikacija2.pml* koji opisuje automat sa slike 3.25, a uključuje i zadana svojstva u obliku LTL formula. U *Promela* modelu je zadano polje brojača `counter` s dva elementa koji će prebrojavati broj simbola a i b u svakoj riječi. Kontrolna varijabla `end` čuva trenutni status stanja (1 za završno stanje, 2 za neregularno završno stanje i 0 za sva ostala stanja). Kod svakog prijelaza se izvodi makronaredba `label` kojom se broje pojavljivanja simbola i postavlja status stanja. Na početku su definirana dva uvjeta p i q. S uvjetom p se zadaje kako ukupna duljina riječi ne smije biti veća od 2000 simbola kako bi ograničili prostor pretrage. Uvjet q je zadovoljen ako je riječ prazna ili ako riječ ima točno jedan simbol a i nula ili paran broj simbola b. Zadana su i dva negativna temporalna svojstva s LTL formulama. Prvo svojstvo je zadano kao LTL formula $pU((end = 1) \wedge \neg q)$ koja provjerava vrijedi li da će riječ biti kraća od 2000 simbola dok se ne pronađe neka koja će završiti u završnom stanju, a

neće zadovoljavati uvjet q o prihvatljivim riječima. Drugo svojstvo je zadano kao LTL formula $pU((end = 2) \wedge q)$ koja provjerava vrijedi li da će riječ biti kraća od 2000 simbola dok se ne pronađe neka koja će završiti u neregularnom završnom stanju iako će zadovoljavati uvjet q o prihvatljivim riječima. U modelu se koriste `ltl` tvrdnje koje automatski negiraju zadanu LTL formulu i pretvaraju je u `never` blok pa su ovdje zadane s negacijom koja će se potom poništiti. Proces `automaton` definira sve prijelaze u DFA, a pritom se iz završnih stanja može izaći skokom na labelu `end_OK`.

Kako su u modelu zadana dva svojstva s LTL formulama verifikacija se mora izvesti u dva koraka. Izvođenjem naredbe *pan.exe -a -N p1* verificira se model za prvo svojstvo `p1` i nije pronađena greška, odnosno nije zadovoljena LTL formula $pU((end = 1) \wedge \neg q)$. Dakle, do završnih stanja dolaze samo one riječi koje su oblikovane prema zadanim pravilima. Prilikom ove verifikacije utrošeno je 148,816 MB, dosegnuta je dubina od 4005 koraka, a verifikacija je trajala 2 s. Nadalje, izvođenjem naredbe *pan.exe -a -N p2* verificira se model za drugo svojstvo `p2` i sada je pronađena greška, odnosno zadovoljena je LTL formula $pU((end = 2) \wedge q)$. Dakle, barem jedna riječ koja je oblikovana prema zadanim pravilima je završila u neregularnom završnom stanju umjesto u završnom stanju. Prilikom ove verifikacije utrošeno je 148,523 MB, dosegnuta je dubina od 4002 koraka, a verifikacija je trajala 1,98 s.

Na kraju će se vođenom simulacijom po generiranom putu do greške naredbom *spin.exe -t verifikacija2.pml* dobiti ispis pronađenog protuprimjera, a to je riječ *abb*. Dakle, pronađena je barem jedna riječ koju bi automat na slici 3.25 morao prihvaćati umjesto da je odbacuje. To je upravo pozitivan protuprimjer koji je *Učitelj* dao *Učeniku* u primjeru izvođenja L^* algoritma u potpoglavlju 3.4.2. Na sličan se način može provjeriti i treća pretpostavka koja predstavlja točan DFA sa slike 3.25. U ovom slučaju će verifikacija uspješno završiti u oba slučaja, dakle riječi oblikovane prema zadanim pravilima dolaze samo u završna stanja i ne mogu završiti u neregularnom završnom stanju.

Osim provjere svojstava modela zadanih s `assert` tvrdnjama i LTL formulama, alat *Spin* pruža i način provjere sekvenci poruka poslanih ili pročitanih kroz komunikacijske kanale s `trace` i `notrace` tvrdnjama [31]. Primjerice, korištenjem ovih tvrdnji mogu se zadavati svojstva sustava o ispravnim ili neispravnim redoslijedima poruka poslanih u neki komunikacijski kanal. S `trace` tvrdnjom želi se provjeriti pojavljuje li se u svim izvođenjima modela sustava točno zadan niz operacija slanja ili čitanja poruka. Analogno tome, s `notrace` tvrdnjom se želi provjeriti da se u nijednom izvođenju modela sustava ne pojavljuje takav niz operacija. U modelu je moguće zadati samo jednu `trace` ili `notrace` tvrdnju, a pritom se u modelu ne smiju istovremeno koristiti `ltl` i `never` tvrdnje. Isto tako, `trace` i `notrace` moraju biti determinističke, a u slučaju detektiranja nedeterminizma verifikator će prijaviti problem i neće izvršiti verifikaciju.

Ako verifikacija modela sa zadanom `trace` tvrdnjom uspješno završi onda je zadana `trace` tvrdnja zadovoljena i taj niz operacija slanja ili čitanja poruka se pojavljuje od početka svakoga izvođenja modela sustava. A u slučaju da verifikacija završi s greškom generirat će se protuprimjer – niz naredbi slanja ili čitanja poruka u nekom izvođenju modela sustava koji ne odgovara redoslijedu zadanom s `trace` tvrdnjom.

S druge strane, kod verifikacije modela sa zadanom `notrace` tvrdnjom generirat će se greška ako se tako zadan niz operacija čitanja ili slanja poruka ipak pojavljuje na početku barem jednog izvođenja modela sustava. Kada nema dojavljene greške onda se zaključuje kako u svim izvođenjima modela sustava nema niza operacija čitanja ili slanja poruka zadanog `notrace` tvrdnjom.

Kao primjer korištenja `trace` tvrdnje na slici 3.47 prikazan je *Promela* model *trace.pml* s dva procesa i sa zadanim nizom operacija slanja i primanja definiranih poruka.

```
mtype {a,b,c}
chan channelAB = [1] of {mtype}

active proctype automatonA () {
        channelAB!a;
        channelAB!c;
        channelAB!b;
}

active proctype automatonB () {
        do
        :: channelAB?a;
        :: channelAB?b;
        :: channelAB?c;
        :: timeout -> break;
        od;
}

trace {channelAB!a; channelAB?a;
channelAB!c; channelAB?c;
channelAB!b; channelAB?b;}
```

Slika 3.47: Primjer *Promela* modela s `trace` tvrdnjom

Model ima dva procesa koji komuniciraju razmjenom poruka iz skupa $\{a,b\}$ preko globalnog kanala s međuspremnikom kapaciteta 1, a s `trace` tvrdnjom je zadan očekivani redoslijed slanja i čitanja poruka u kanalu kao $a \rightarrow c \rightarrow b$. Verifikacija opisanog modela uspješno završava bez prijave greške, a to znači da se u svim izvođenjima modela pojavljuje niz slanja pa primanja poruka $a \rightarrow c \rightarrow b$. U ovom slučaju je verifikacija bila vrlo efikasna, dosegnuta je dubina od 9 koraka, trajanje verifikacije je $0,035$ s, a utrošeno je ukupno 64,577 MB memorije – tek nešto više od osnovnih 64 MB koji su rezervirani za svaku verifikaciju. Analizom modela vidi se da proces `automatonA` diktira redoslijed slanja poruka, a nakon toga odgovarajuće opcije `do` bloka za čitanje pojedinih poruka u procesu `automatonB` postaju izvršne. Nakon čitanja zadnje poslane poruke b u idućoj iteraciji `do` bloka sve su opcije za čitanje poruka blokirajuće pa

u modelu sustava dolazi do zastoja što se detektira varijablom `timeout` koja postaje istinita. Potom se `do` blok prekida čime i drugi proces uredno završava.

Kod stvaranja C programa verifikatora modela sa slike 3.47 može se posebnom zastavicom *-m* promijeniti ponašanje modela sustava (*spin -a -m trace.pml*). Pri uobičajenom ponašanju modela blokirano je slanje poruka u puni kanal dok je s aktiviranom zastavicom *-m* slanje poruka uvijek izvršno, a kod slanja u puni kanal poruka se izgubi. Ako se sada stvori izvršni program verifikatora te pokrene verifikacija dojavit će se greška jer je u barem jednom putu izvođenja prekršen niz slanja i primanja poruka zadan `trace` tvrdnjom. U ovom slučaju će pokretanje detaljne vođene simulacije naredbom *spin.exe -t -v trace.pml* otkriti protuprimjer: slanje poruke *a*, slanje poruke *c* dok *a* nije još pročitana (kanal je pun pa operacija slanja nije uspjela i poruka *c* se odbaci), potom čitanje poruke *a* i na kraju slanje poruke *b*. U tom trenutku je detektirana greška, a još je jedino moguće pročitati poruku *b*. Dakle, umjesto zadanog redoslijeda slanja i primanja poruka $a \rightarrow c \rightarrow b$ otkriven je krivi redoslijed $a \rightarrow b$.

```
mtype {a,b,c}
chan channelAB = [1] of {mtype}

active proctype automatonA () {
        channelAB!a;
        channelAB!c;
        channelAB!b;
}

active proctype automatonB () {
        do
        :: channelAB?a;
        :: channelAB?b;
        :: channelAB?c;
        :: timeout -> break;
        od;
}

notrace {channelAB?a; channelAB?b;}
```

Slika 3.48: Primjer *Promela* modela s `notrace` tvrdnjom

U ovom istraživanju se koristi i `notrace` tvrdnja pa će se i ona ilustrirati kroz verifikaciju modela *notrace.pml* sa slike 3.48. Model je isti kao i na slici 3.47 samo je sada zadana `notrace` tvrdnja kojom se traži da niti jedno izvođenje modela sustava ne smije započeti s primanjem poruke *a* i potom poruke *b*. I verifikacija ovoga modela uspješno završava jer se ni u jednom putu izvođenja ne pojavljuje zadani redoslijed primanja poruka $a \rightarrow b$. S druge strane, u slučaju zadavanja promjene ponašanja modela sustava kao u prethodnom razmatranju (*spin -a -m notrace.pml*) generirat će se verifikator čijim će se izvođenjem pronaći neželjeni niz primanja poruka $a \rightarrow b$. Objašnjenje je isto kao i u prethodnom primjeru, poruka *c* se pokušala poslati u puni kanal te je pritom izgubljena pa se jedino mogu primiti dvije poruke i to redom $a \rightarrow b$.

Na kraju treba ponoviti kako se verifikacijom modela alatom *Spin* iscrpno pretražuju svi mogući putevi izvođenja modela sve dok se ne detektira greška. Pritom prostor pretrage može potencijalno biti ogroman, što iziskuje velike memorijske i procesorske resurse kako bi se verifikacija uopće mogla dovršiti. Ako verifikacija nije izvršena u cijelosti jer su dosegnuti zadani limiti istražene dubine ili dostupne memorije sustava onda se ne može garantirati da model sustava sigurno zadovoljava zadano svojstvo. U tom slučaju treba povećati zadane limite alata *Spin* korištenjem različitih opcija za optimizaciju i pokušati ponoviti verifikaciju. Isto tako može se pokušati i pojednostavniti model sustava ili svojstvo zadano s LTL formulom te pokušati ograničiti najveću duljinu protuprimjera kojega se može naći verifikacijom. Npr. može se primijetiti kako je u primjeru 3.3 bila ograničena duljina riječi koje se žele provjeriti na 2000 simbola, a time se smanjio prostor pretrage i verifikacija se uspjela izvršiti do kraja.

Alat za provjeru modela *Spin* koristi različite mehanizme za efikasno istraživanje prostora pretrage stanja sustava. Verifikatori izgrađeni alatom *Spin* prvo pokušavaju identificirati one puteve izvođenja kod kojih je prekršeno zadano svojstvo modela sustava pa se pažljivim zadavanjem tog svojstva može utjecati na brzinu izvođenja verifikacije. Uobičajeni način verifikacije s alatom *Spin* je iscrpna pretraga svih stanja sustava, a kako bi se taj postupak optimizirao primjenjuje se strategija reduciranja parcijalnog poretka (engl. *partial order reduction*) [31]. Ovom strategijom se smanjuje broj stanja sustava koje treba istražiti u prostoru pretrage. Prilikom ispreplitanja naredbi u modelu traže se neovisne naredbe čiji će rezultat biti isti bez obzira na redoslijed izvođenja te međusobno ovisne naredbe čiji rezultat ovisi o njihovom redoslijedu izvođenja. Svi putevi izvođenja grupiraju se u disjunktne podskupove tako da svaki podskup čine oni putevi izvođenja koji dovode do istih rezultata. Onda se prilikom verifikacije provjerava samo po jedan primjer puta izvođenja iz svakoga podskupa čime se značajno ubrzava proces verifikacije, a pritom se iscrpno provjere svi ishodi izvođenja modela sustava. Dodatni mehanizam za optimizaciju postupka verifikacije je algoritam za stvaranje efikasne tablice raspršenog adresiranja (engl. *bitstate hashing*). Ovo može biti posebno korisno za verifikaciju vrlo složenih modela sustava jer se time značajno smanjuju potrebni memorijski resursi kao i ubrzava postupak verifikacije uz garanciju visoke prekrivenosti grafa dostupnosti [101].

Kako će se pokazati i u ovom istraživanju alat *Spin* pruža učinkovit i pouzdan način za verifikaciju različitih modela sustava. U poglavlju 7 pokazat će se na koji način je alat *Spin* iskorišten u predloženoj implementaciji *Učitelja* za automatiziranje izvođenja *L** algoritma. Isto tako predstavit će se automatizirane skripte kojima se olakšava izvođenje verifikacije i simulacije s alatom *Spin*, a dokumentirat će se rezultati simulacije i verifikacije otkrivenih modela procesa provjere znanja u sustavima za e-učenje.

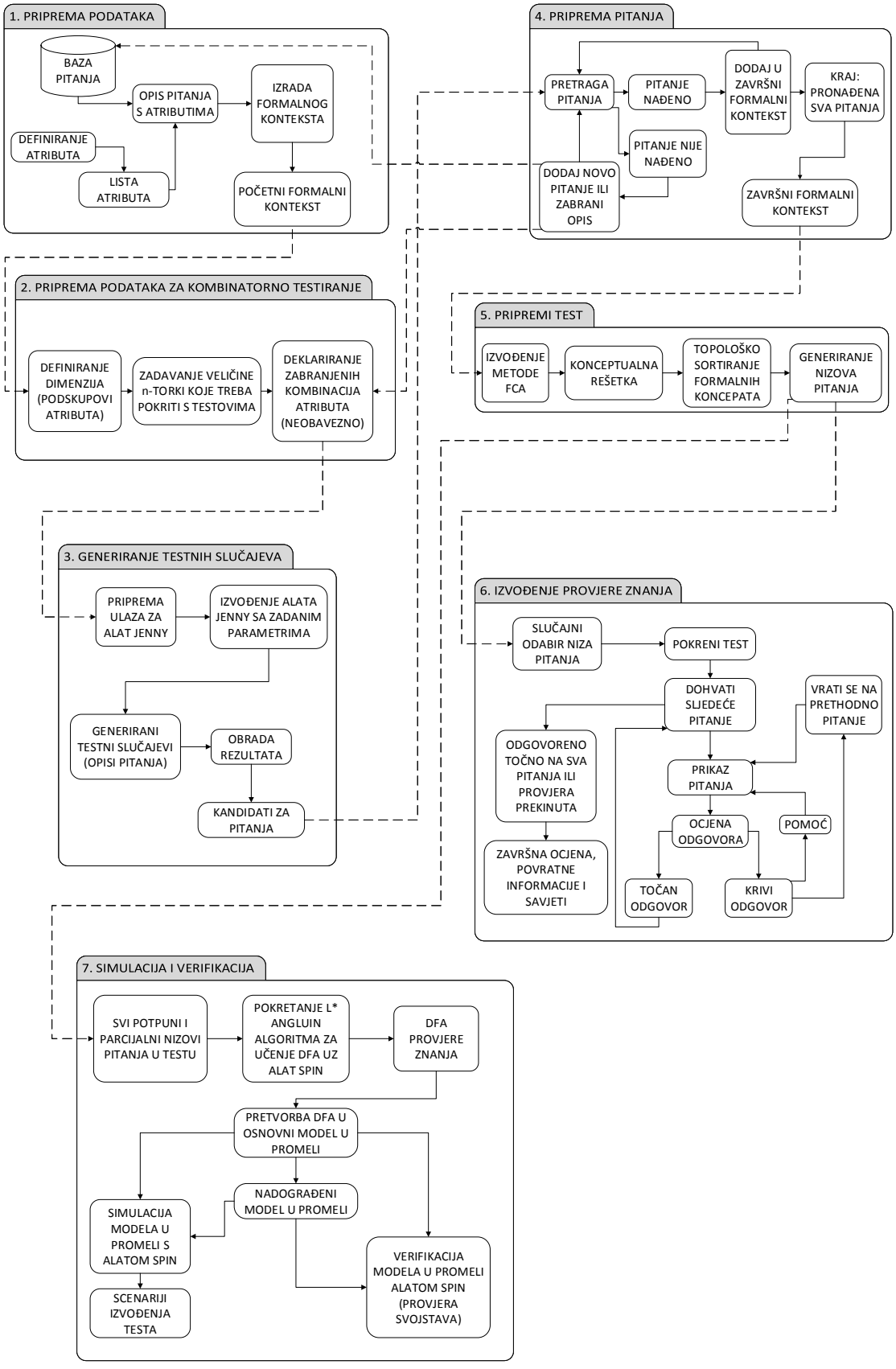
Poglavlje 4

Pregled modela sustava za automatiziranu provjeru znanja

Nakon što je u poglavlju 3 izložen teoretski pregled postojećih metoda i tehnika korištenih u istraživanju, u ovome poglavlju će se predstaviti predložena metoda za generiranje i izvođenje provjera znanja u sustavima za e-učenje. Glavni zadatak predložene metode je automatizirana priprema i odabir pitanja koja će se koristiti prvenstveno za formativne provjere znanja u sustavima za e-učenje. Pritom se traže ona pitanja koja povezuju više različitih pojmova iz zadane domene, a s kojima se nastoji na podrobniji način ispitati usvajanje i razumijevanje gradiva. Prilikom traženja pitanja treba pokriti sve dopuštene n-torke atributa zadane veličine $n = t$ s kojima se pitanja opisuju, a da pritom konačni skup odabranih pitanja bude što je moguće manji. Isto tako treba odrediti prikladan redoslijed pitanja kojim će se ona postavljati tijekom procesa provjere znanja. U ovome istraživanju će se pretpostaviti kako su pitanja koja se opisuju manjim brojem atributa jednostavnija odnosno lakša, a ona koja se opisuju većim brojem atributa složenija odnosno teža jer uključuju više pojmova iz gradiva koje treba poznavati. Prema tome, svaki odabrani niz pitanja za formativnu provjeru znanja treba biti poredan od pretpostavljeno lakših pitanja prema pretpostavljeno težim pitanjima. Na kraju se predložena metode za automatiziranu provjeru znanja treba verificirati uporabom formalne metode provjere modela.

Predložena metoda komplementarno koristi metodu FCA i tehniku kombinatornog testiranja te topološko sortiranje kako bi se na automatiziran način generirali prikladni nizovi označenih pitanja za formativne provjere znanja u sustavima za e-učenje. Nadalje, predložena metoda uključuje i postupak za vođenje formativne provjere znanja kroz odabir sljedećeg pitanja na temelju ocjene odgovora na prethodno pitanje, a pritom se omogućuju i različiti oblici pomoći kod krivih odgovora. Konačno, predložen je i automatizirani postupak za simulaciju i verifikaciju procesa provjere znanja korištenjem L^* algoritma za učenje DFA i alata za provjeru modela *Spin*. Na slici 4.1 prikazan je razvijeni model sustava koji implementira predloženu metodu za automatizirano generiranje i izvođenje provjere znanja te postupke za njenu verifikaciju, a koji

je dijelom predstavljen u sklopu ovog doktorskog istraživanja u članku [102].



Slika 4.1: Model sustava za automatiziranu provjeru znanja

U nastavku će se ukratko predstaviti pojedini dijelovi razvijenog modela sustava za automatiziranu provjeru znanja prikazanog na slici 4.1, a u narednim poglavljima će se svaki od njih detaljnije obraditi. Prva četiri naznačena dijela modela sustava implementiraju korake predložene metode kojima se pronalazi prikladni sažeti skup pitanja, dok se peti i šesti dio modela sustava odnose na pripremu provjere znanja, odnosno na sam postupak formativne provjere znanja u sustavima za e-učenje. Dodatno, sedmi dio modela sustava opisuje postupak za verifikaciju predložene metode.

Glavni preduvjet za korištenje predložene metode za automatiziranu provjeru znanja je postojanje baze pitanja u sustavu za e-učenje u kojoj je svako pitanje označeno s nekim odgovarajućim podskupom atributa koji opisuju zadano nastavno gradivo. Ako ovaj preduvjet nije na početku ispunjen treba izvršiti pripremu podataka prema 1. koraku predložene metode sa slike 4.1. U tom slučaju prvo treba definirati skup atributa s kojima se može na prikladan način opisati pitanja u bazi. Definirani skup atributa predstavlja skup jasnih pojmova iz čitavog nastavnog gradiva ili dijela nastavnog gradiva, a može se npr. dobiti pregledom kazala ili ključnih riječi u odgovarajućem udžbeniku ili drugom nastavnom materijalu. Nakon toga se treba označiti svako pitanje iz baze s nekim odgovarajućim podskupom definiranih atributa. U ovom koraku treba se osloniti na domensko znanje i iskustvo eksperta, odnosno nastavnika. Nakon što su sva potrebna pitanja označena prelazi se na automatsku izgradnju formalnog konteksta kojim će se opisati domensko znanje sadržano u nastavnom gradivu ili nekom njegovom manjem dijelu. Formalni kontekst je binarna dvodimenzionalna matrica u kojoj svaki redak predstavlja jedno od pitanja iz baze, a svaki stupac odgovara jednom od definiranih atributa. Iz baze označenih pitanja automatski se puni matrica formalnog konteksta s vrijednostima 0 i 1 – ako neko pitanje ima određeni atribut na sjecištu tog retka i stupca unosi se vrijednost 1, a inače 0. Na kraju se dobije početni formalni kontekst koji je ulazni podatak za metodu FCA kako je objašnjeno u potpoglavlju 3.1.

Nakon pripreme početnog formalnog konteksta u 2. koraku predložene metode prelazi se na pripremu podataka za kombinatorno testiranje. Domenski ekspert, odnosno nastavnik treba odlučiti na koji će način grupirati pojedine atribute u disjunktne podskupove atributa. Svaki podskup atributa predstavljat će jedan parametar ili dimenziju, a elementi svakog podskupa (pojedini atributi) su vrijednosti tog parametra odnosno značajke te dimenzije. Dakle, ovaj pristup je u skladu s postavkama kombinatornog testiranja iz potpoglavlja 3.2. Potom domenski ekspert treba odabrati traženu veličinu n-torki različitih vrijednosti parametara odnosno značajki koje treba pokriti s testnim slučajevima. U predloženoj metodi svaki testni slučaj generiran kombinatornim testiranjem smatra se opisom ispitnog pitanja. Dodatno, nastavnik može zadati i ograničenja na zabranjene kombinacije atributa koje se ne smiju pojavljivati u generiranim testnim slučajevima.

U idućem, 3. koraku predložene metode provodi se automatizirano kombinatorno testira-

nje gdje se uz pomoć alata *Jenny* predstavljenog u potpoglavlju 3.2 generiraju testni slučajevi odnosno opisi ispitnih pitanja. U ovom koraku automatski se pripremaju ulazni podaci za alat *Jenny*, a nakon izvođenja alata njegovi rezultati se također automatski obrađuju i transformiraju u oblik pogodan za daljnju analizu. Pritom je svaki generirani testni slučaj traženi opis pitanja sa zadanim značajkama odnosno atributima. Ne mora nužno značiti da u bazi već postoji odgovarajuće pitanje za svaki opis pitanja generiran kombinatornim testiranjem. Stoga, svi generirani opisi pitanja predstavljaju tek moguće kandidate za stvarna pitanja.

Kako bi se provjerilo jesu li generirani opisi pitanja već implementirani kao postojeća pitanja u bazi provodi se 4. korak metode za automatiziranu provjeru znanja, odnosno priprema skupa pitanja za provjeru znanja. Cilj ovog koraka je punjenje inicijalno praznog završnog formalnog konteksta s odgovarajućim sažetim skupom pitanja. Sada se za svaki generirani opis pitanja traže odgovarajuća pitanja u bazi i ako su pronađena onda se uključuju u završni formalni kontekst. S druge strane, ako niti jedno odgovarajuće pitanje nije pronađeno u bazi onda nastavnik treba odlučiti hoće li stvoriti novo pitanje koje će implementirati generirani opis pitanja ili neće jer se u njemu nalazi neka kombinacija atributa koju treba zabraniti. U prvom slučaju će nastavnik osmisliti jedno ili više novih pitanja koja pokrivaju generirani opis, označit će ih atributima te dodati u početni formalni kontekst. U drugom slučaju nastavnik treba deklarirati zabranjenu kombinaciju atributa iz generiranog opisa pitanja. U oba slučaja će se potom isprazniti završni formalni kontekst te ponovno pokrenuti 3. korak odnosno iznova generirati testni slučajevi u obliku opisa pitanja. Kao što se može primijetiti ovaj postupak kombinatornog testiranja je iterativan i nastavit će se sve dok se za svaki generirani opis pitanja ne pronađe barem po jedno pitanje u bazi koje ga pokriva. U tom trenutku postupak kombinatornog testiranja uspješno završava s popunjenim završnim formalnim kontekstom koji sadrži sažeti skup pitanja kojim su pokriveni svi generirani opisi pitanja.

U 5. koraku metode za automatiziranu provjeru znanja izvodi se neki od specijaliziranih alata za metodu FCA poput programa *ConExp 1.3* kojim se automatski izgrađuje konceptualna rešetka iz završnog formalnog konteksta u kojemu su ispitna pitanja označena s atributima. Konceptualna rešetka se sastoji od čvorova, odnosno formalnih koncepata koji su međusobno povezani usmjerenim granama iz kojih se mogu očitati njihovi međusobni odnosi *natkoncept-potkoncept*. Svaki formalni koncept je par nekog skupa objekata – pitanja te skupa njihovih dijeljenih atributa. U kontekstu e-učenja svaki takav formalni koncept može se promatrati kao elementarna točka znanja ili *EKP* točka (engl. *elementary knowledge point*). Dakle, svaka *EKP* točka sadrži neki skup pojmova iz gradiva te listu odgovarajućih pitanja kojima se može ispitati znanje studenta o tim pojmovima. Objedinjeni skup svih povezanih *EKP* točaka u konceptualnoj rešetci čini ontologiju odabranog dijela gradiva predmeta. Nakon generiranja konceptualne rešetke pokreće se program *tsort* kojim se ona topološki sortira kako je opisano u potpoglavlju 3.3. Ovako se dobije odgovarajući linearni poredak formalnih koncepata ili *EKP* točaka iz

konceptualne rešetke, poredanih od onih općenitijih s vrha konceptualne rešetke koji u pravilu sadrže manje atributa, do onih specifičnijih s dna konceptualne rešetke koji u pravilu sadrže više atributa. Potom se iz tog automatski generiranog linearnog poretka mogu izvući odgovarajuće sekvence pitanja sadržanih u formalnim konceptima. Analogno poretku formalnih koncepata i generirani nizovi pitanja su poredani od općenitijih, odnosno pretpostavljeno lakših pitanja prema specifičnijim, odnosno pretpostavljeno težim pitanjima. Upravo ova karakteristika čini ovako generirane nizove pitanja prikladnima za primjenu u formativnim provjerama znanja u kojima student može postupno samotestirati usvojeno znanje.

Nakon što se na opisani način generira prikladan niz pitanja za provjeru znanja prelazi se na 6. korak predložene metode kojim se definira način i tijek procesa provjere znanja. Ovaj korak je implementiran kroz modul za formativnu provjeru znanja koji je dio vlastitog sustava za e-učenje. Nakon prijave na sustav za e-učenje studentima je omogućen pristup vođenim formativnim provjerama znanja koje pokrivaju dio gradiva ili čitavo gradivo predmeta. U ovom istraživanju odabrana je domena predmeta *Osnove elektrotehnike*, a prvenstveno se koriste pitanja s odabirom, točnije pitanja višestrukog izbora odgovora sa samo jednim točnim odgovorom. Nadalje, kako se radi o vođenoj formativnoj provjeri znanja student u svakom trenutku vidi samo svoje trenutno pitanje na ekranu. Dakle, nakon odabira formativne provjere prvo se na slučajan način odabere jedan od mogućih automatski pripremljenih nizova pitanja te provjera započinje s prikazom prvoga pitanja iz niza. Naravno, omogućen je i nastavak rješavanja provjere sa zadnjim pitanjem iz ranije prekinute sesije. Uz tekst pitanja te popratnu sliku, ako je ona zadana, ispisuje se i slučajno izabrana permutacija ponuđenih odgovora. Osim statičkih pitanja s jednom verzijom teksta, slike i odgovora, omogućena su i varijabilna pitanja kojima se generira veći broj različitih inačica iz osnovnog predloška pitanja, a na slučajan način se jedna od tih inačica prikazuje studentu. Nakon što se pitanje prikaže i student na njega da svoj odgovor sustav ga treba ocijeniti. Ako je odgovor točan prelazi se na sljedeće pitanje u nizu, a inače sustav nudi pomoć, prvenstveno u obliku automatski generiranih poveznica na odgovarajuće interaktivne i ostale nastavne materijale dostupne u sustavu za e-učenje. Studentu se potom daje još jedna prilika da odgovori na isto pitanje. Ako ponovno odgovori netočno sustav će ga vratiti na prethodno, pretpostavljeno lakše pitanje iz niza, a ako student da točan odgovor još jednom će dobiti drukčiju inačicu istog ili sličnog pitanja. Tek kada točno odgovori i na ovu varijantu pitanja sustav će studentu postaviti sljedeće pitanje iz niza. Cijeli proces provjere znanja se provodi sve dok student ne odgovori točno na sva pitanja u generiranom nizu. Studentima je omogućeno privremenog zaustavljanje rješavanja formativne provjere znanja u bilo kojem trenutku te kasniji nastavak rješavanja. Kada provjera znanja uspješno završi ili student odustane od daljnjeg rješavanja sustav će mu dati opisnu završnu ocjenu u obliku statistike ukupnog broja odgovora na postavljena pitanja, broja ponavljanja pitanja i vraćanja na prethodna pitanja te efektivnog trajanja provjere. Osim toga sustav će studentu kao povratnu informaciju dati

automatski generiranu listu dijelova gradiva s kojim je imao više problema tijekom rješavanja provjere. Ovakav modul za provjeru znanja je prvenstveno namijenjen za samotestiranje jer svaki student može dobiti duži niz pitanja koji pokriva odabrani dio gradiva ili čak cjelokupno gradivo predmeta, a samo rješavanje provjere može se odvijati s prekidima kroz duži vremenski period.

Važan dio ovog doktorskog istraživanja je verifikacija predložene metode za automatizirano generiranje i izvođenje provjere znanja u sustavima za e-učenje. U tu svrhu osmišljen je postupak formalne verifikacije metodom provjere modela i primjene *L** algoritma Dane Angluin za učenje deterministčkih konačnih automata koji su predstavljeni u potpoglavljima 3.5, odnosno 3.4.2. Ovaj postupak verifikacije opisan je u sedmom dijelu modela sustava za automatiziranu provjeru znanja prikazanog na slici 4.1, a implementiran je kao prototipni sustav za verifikaciju i simulaciju procesa provjere znanja. Nastavnici i eksperti za oblikovanje sustava za e-učenje mogu uz pomoć implementiranog prototipnog sustava detaljno ispitati prikladnost generiranih nizova pitanja za formativnu provjeru znanja te provjeriti način i tijek izvođenja procesa formativne provjere znanja.

Ulazni podaci za izvođenje verifikacije predložene metode su isti oni prikladni nizovi pitanja koji se koriste i za izvođenje formativne provjere znanja, a generirani su automatski iz topološkog sortiranja konceptualne rešetke. Svako pitanje je predstavljeno sa svojom jedinstvenom identifikacijskom oznakom koja se može smatrati i simbolom koji jednoznačno označava to pitanje. Uzima se da skup svih simbola koji označavaju sva pitanja sadržana u konceptualnoj rešetci čini abecedu, a svaki generirani niz pitanja može se onda smatrati riječju sastavljenom od simbola te abecede. Dodatno, uzet će se u obzir i svi parcijalni nizovi pitanja kao svi prefiksi kompletnih nizova pitanja. Tako se uzima u obzir i mogućnost da student u svakom trenutku odustane od rješavanja pokrenute formativne provjere.

Prvi zadatak postupka za verifikaciju je automatska izgradnja deterministčkog konačnog automata koji će prihvaćati sve generirane kompletne i parcijalne nizove pitanja. U tu svrhu koristi se *L** algoritam Dane Angluin kao i alat za provjeru modela *Spin*. Korištena je vlastita implementacija *L** algoritma u jeziku *Python* koja u potpunosti definira ulogu *Učenika* koji kroz dijalog sa sveznajućim *Učiteljem* želi naučiti inicijalno nepoznati DFA. U *L** algoritmu nije detaljno definirana uloga *Učitelja* nego je samo ukratko specificiran način na koji *Učitelj* odgovara na upite *Učenika*. Stoga, kako bi se postupak učenja nepoznatog DFA automatiziraо osmišljen je algoritam *Učitelja* te je implementiran kao skripta u jeziku *Python*. Kako bi se na automatski način ispitivale pretpostavke *Učenika* o nepoznatom DFA *Učitelj* koristi alat za provjeru modela *Spin*. Pritom *Učitelj* svaku pretpostavku o nepoznatom DFA automatski transformira u više tipova odgovarajućih procesnih modela u *Promeli* te ih potom automatski formalno verificira s alatom *Spin*. Cilj verifikacije je pronalazak greške u *Učenikovoj* pretpostavci o nepoznatom DFA. Ako je greška pronađena generira se protuprimjer, a *Učitelj* će ga

potom obraditi i proslijediti *Učeniku* kao svoj odgovor. Treba naglasiti kako se traže i pozitivni i negativni protuprimjeri, a zato se koriste različiti pristupi poput `notrace` tvrdnje ili `never` tvrdnje za definiranje temporalnog svojstva LTL formulom. Na kraju će u konačnom broju koraka *Učenik* uspjeti naučiti DFA koji prihvaća sve kompletne i parcijalne nizove generiranih pitanja.

Nakon što se pronađe traženi DFA koji opisuje proces provjere znanja *Učitelj* će automatski generirati odgovarajuće procesne modele u *Promeli* s kojima se može na različite načine verificirati, ali i simulirati proces provjere znanja. Uz to generirat će se i nadograđeni procesni modeli u *Promeli* koji na precizniji način opisuju stvarni tijek formativne provjere znanja. U nadograđenim procesnim modelima dodaju se povratne grane čime se modelira ponavljanje pitanja ili vraćanje na prethodno pitanje u nizu nakon krivog odgovora.

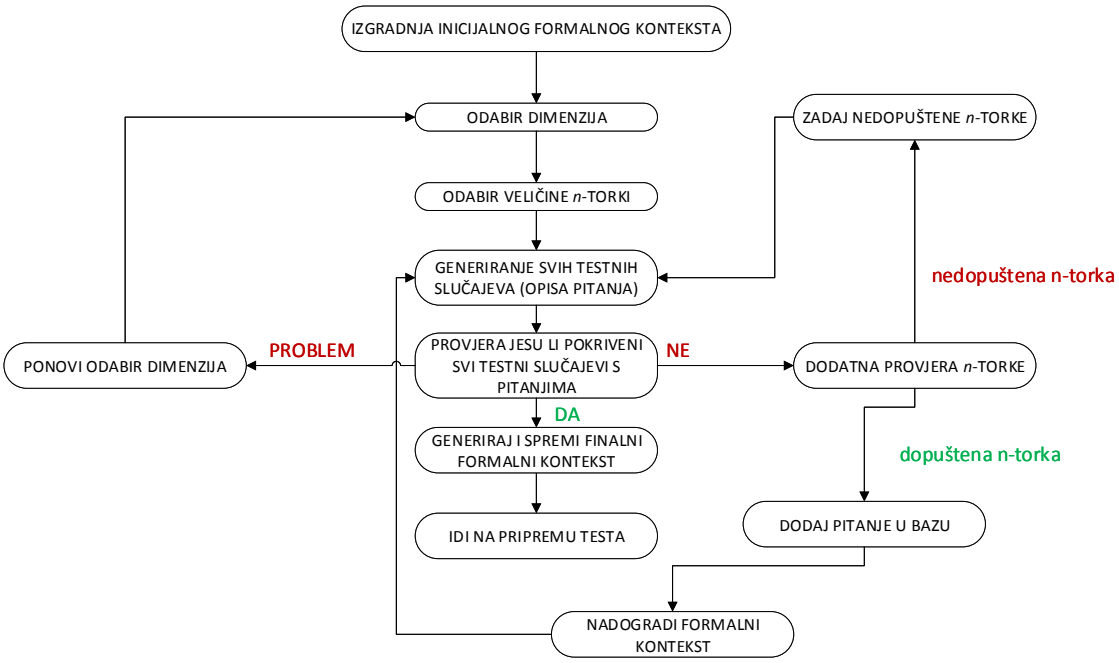
Kako bi se olakšao postupak verifikacije napravljena je jednostavna web aplikacija koja služi kao sučelje prema skriptama u *Pythonu* koje implementiraju uloge *Učenika* i *Učitelja*. Nastavnici i eksperti za oblikovanje sustava za e-učenje mogu kroz ovo sučelje jednostavno zadati obavezne i dodatne ulazne parametre te pokrenuti proces učenja DFA i na kraju preuzeti sve rezultate za daljnju analizu. Među rezultatima se preuzima i dodatna skripta u *Pythonu* kojom se olakšava i automatizira postupak verifikacije i simulacije pronađenih modela procesa provjere znanja u *Promeli*. Na kraju se pomoću simulacija osnovnog i nadograđenog modela dobiju različiti scenariji izvođenja procesa provjere znanja, a verifikacijom se može pouzdano ispitati zadovoljavaju li ti modeli zadana svojstva.

U narednim poglavljima će se detaljnije opisati pojedini dijelovi modela sustava za automatiziranu provjeru znanja prikazanog na slici 4.1 te predstaviti, analizirati i raspraviti dobiveni rezultati.

Poglavlje 5

Postupak kombinatornog testiranja

U fokusu ovoga poglavlja su prva četiri dijela modela za automatiziranu provjeru znanja prikazanog na slici 4.1. Detaljnije će se predstaviti kako se kombinatorno testiranje može iskoristiti za odabir sažetog skupa pitanja kojim će se pokriti sve moguće n-torke definiranih atributa zadane veličine $n = t$. Na početku se iz skupa pitanja označenih definiranim atributima izgrađuje početni formalni kontekst, a nakon toga se izvrši postupak kombinatornog testiranja kojim se traži odgovarajući sažeti skup pitanja. Ovdje će se podrobnije predstaviti predložena metoda za kombinatorno testiranje kao i njena implementacija. Potom će se navesti studijski primjer većeg skupa označenih ispitnih pitanja i razmotrit će se različite strategije definiranja parametara za kombinatorno testiranje kao i odabira najprikladnije veličine n-torki koje se treba pokriti s generiranim testnim slučajevima. Naposljetku će se predstaviti i raspraviti rezultati izvođenja predložene metode kombinatornog testiranja nad studijskim primjerom.



Slika 5.1: Predložena metoda kombinatornog testiranja

Na slici 5.1 je dan općeniti pregled predložene metode kombinatornog testiranja. Iz te slike vidi se da predložena metoda započinje s izgradnjom početnog formalnog konteksta i završava kada je svaki generirani testni slučaj u potpunosti pokriven s barem jednim pitanjem iz početnog formalnog konteksta. Naravno, prije izvođenja procesa kombinatornog testiranja domenski ekspert treba definirati broj dimenziji ili parametara te odabrati njihove vrijednosti. U ovom slučaju nastavnik treba odlučiti kako grupirati atribute ili značajke u željeni broj disjunktnih podskupova. Uz to treba odrediti koja je tražena snaga međusobnog pokrivanja t kojom bi se pokrile sve n-torke vrijednosti različitih parametara ili dimenzija veličine $n = t$. Kao što je navedeno u potpoglavlju 3.2 uobičajeno se zadaju vrijednosti $t = 2$ i $t = 3$ s kojima će generirani testni slučajevi pokriti sve moguće parove odnosno trojke značajki.

Svaki generirani testni slučaj u ovom kontekstu predstavlja opis ispitnog pitanja. Prilikom izvođenja postupka kombinatornog testiranja pokazat će se kako su neki generirani testni slučajevi već pokriveni s označenim pitanjima u početnom formalnom kontekstu. Nastavnik potom treba detaljno razmotriti preostale generirane testne slučajeve koji nisu pokriveni s postojećim pitanjima u početnom formalnom kontekstu. Ako se u nepokrivenom testnom slučaju nalazi neka kombinacija značajki koja je zanimljiva i ulazi u gradivo predmeta onda se takav opis pitanja treba implementirati kao jedno ili više novih pitanja i uključiti takva novostvorena pitanja u početni formalni kontekst. Naprotiv, ako se u generiranom testnom slučaju uoči kombinacija značajki koja nema smisla ili ne ulazi u gradivo predmeta onda se takva kombinacija mora zabraniti prije iduće iteracije kombinatornog testiranja.

Postupak kombinatornog testiranja uspješno završava u trenutku kada je svaki generirani testni slučaj u potpunosti pokriven s barem jednim pitanjem iz početnog formalnog konteksta. Nakon toga izgrađuje se završni formalni kontekst u kojeg ulaze samo ona pitanja iz početnog formalnog konteksta koja odgovaraju generiranim testnim slučajevima. Time je dobiven sažeti skup označenih pitanja koji pokriva sve moguće dozvoljene n-torke značajki zadane veličine $n = t$. Treba podsjetiti kako će se iz završnog formalnog konteksta potom izgraditi odgovarajuća konceptualna rešetka primjenom metode FCA, a onda topološkim sortiranjem dobiti prikladni nizovi pitanja za formativnu provjeru znanja u sustavima za e-učenje (peti i šesti dio modela sustava na slici 4.1).

Treba posebno naglasiti kako rezultati kombinatornog testiranja uvelike ovise o početno zadanom broju dimenzija te odabiru i rasporedu značajki po dimenzijama, kako će se vidjeti u potpoglavljima 5.2 i 5.3. Ako se kombinatornim testiranjem ne generiraju korisni testni slučajevi onda treba dobro razmisliti kako iznova definirati broj i sadržaj svake dimenzije te potom pokušati ponovno provesti cijeli postupak kombinatornog testiranja.

Prije nastavka treba napomenuti razlike u predloženom pristupu kombinatornog testiranja te postupka eksplorativne analize atributa (engl. *attribute exploration*) korištenjem metode FCA [103]. Eksploratorna analiza atributa je ručni i interaktivni postupak za izgradnju formalnog

konteksta i njegove konceptualne rešetka. Na početku se zadaje skup atributa, ali je skup objekata prazan. Korištenjem alata za metodu FCA može se pokrenuti proces eksploratorne analize atributa. Alat predlaže moguće implikacije atributa, a onda traži od domenskog eksperta da ih potvrdi ili opovrgne. Ovisno o odgovorima eksperta alat za metodu FCA dodaje u formalni kontekst nove objekte koji su u skladu s ispitanim implikacijama atributa. Ovaj interaktivni postupak može postati vrlo kompleksan i vremenski zahtjevan, pogotovo kod većih skupova atributa (npr. 50-ak atributa). Može biti posebno teško opovrgnuti predloženu implikaciju atributa, jer u tom slučaju domenski ekspert treba vrlo pažljivo ponuditi protuprimjer koji ne smije proturječiti prethodno potvrđenim implikacijama atributa. Nadalje, u kontekstu pripreme pitanja za provjere znanja u sustavima za e-učenje i domenski ekspert treba na kraju uspješnog postupka eksploratorne analize atributa dobro provjeriti svaki generirani objekt odnosno opis potencijalnog pitanja. Ako u bazi pitanja otprije ne postoje pitanja koja odgovaraju nekom generiranom opisu pitanja iz formalnog konteksta onda se prema tom opisu treba osmisliti i sastaviti novo pitanje. Naravno, taj zadatak će često biti težak jer s novim pitanjem treba pokriti isključivo zadani podskup atributa.

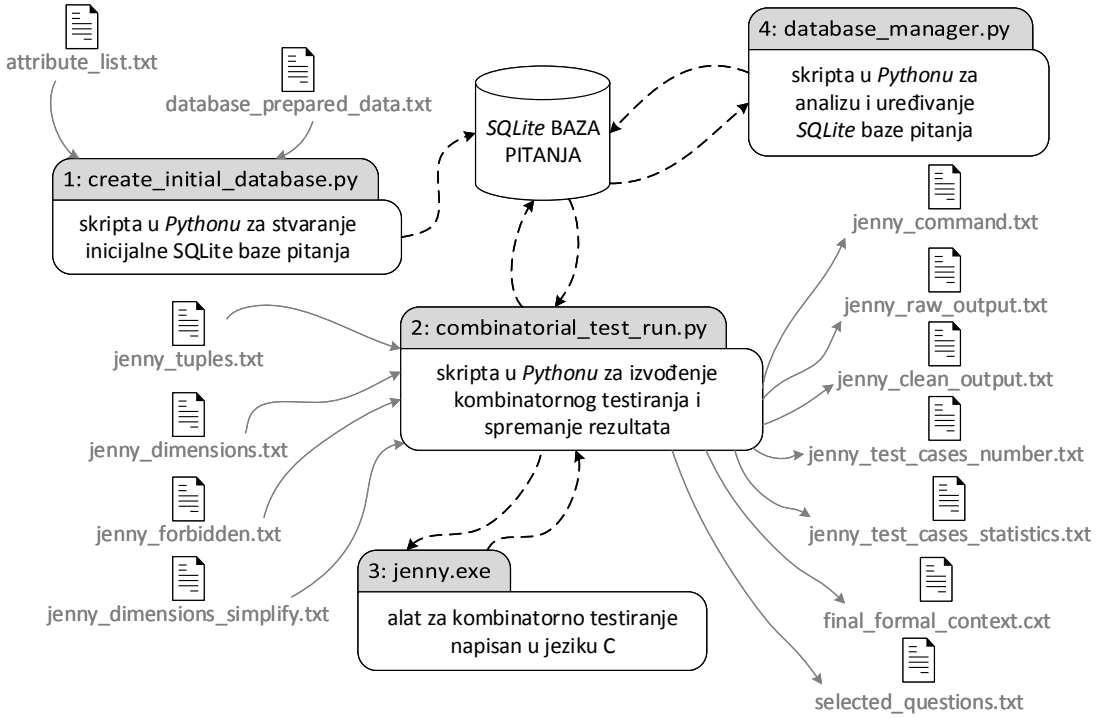
S druge strane, predložena metoda kombinatornog testiranja automatski pronalazi sažeti skup testnih slučajeva odnosno potencijalna ispitna pitanja koji pokrivaju sve moguće parove ili trojke značajki iz zadanih dimenzija. Nakon toga se automatski provjerava jesu li generirani potencijala pitanja s postojećim pitanjima u bazi. Samo u slučaju kada neko od generiranih potencijalnih pitanja još ne postoji u bazi domenski ekspert mora odlučiti hoće li ga implementirati kao novo pitanje ili će zabraniti neku od uočenih neželjenih kombinacija značajki. Povrh toga, pri sastavljanju novih pitanja nastavnici imaju veću fleksibilnost jer mogu uz sve značajke iz testnog slučaja uključiti i druge značajke. Na taj se način može iz svakog generiranog testnog slučaja stvoriti i veći broj srodnih novih pitanja.

Na kraju ove kratke usporedbe s eksploratornom analizom atributa treba ponovno naglasiti kako će testni slučajevi generirani kombinatornim testiranjem sigurno pokriti sve moguće parove ili trojke značajki. Ova odlika kombinatornog testiranja je u skladu s temeljnim zadatkom ovoga istraživanja, a to je pronalazak sažetog skupa pitanja koja u sebi povezuju različite kombinacije pojmova. S takvim skupom pitanja se nastoji provjeriti dublje razumijevanje usvojenog gradiva i studentima na vrijeme ukazati s kojim dijelovima gradiva imaju potencijalnih poteškoća.

5.1 Modul za kombinatorno testiranje

Metoda kombinatornog testiranja sa slike 5.1 implementirana je u modulu za kombinatorno testiranje. Ovaj modul se sastoji od *SQLite* baze podataka [104] i tri skripte u *Pythonu* za automatiziranu pripremu i izvođenje kombinatornog testiranja kao i za obradu i analizu dobivenih

rezultata. Korištena *SQLite* baza podataka pruža veliku fleksibilnost prilikom spremanja, pretraživanja i analiziranja podataka sa standardnim *SQL* upitima bez potrebe za implementacijom složenih podatkovnih struktura u *Pythonu*. Nadalje, skripte u *Pythonu* automatski pokreću alat za kombinatorno testiranje *Jenny* (predstavljen u potpoglavlju 3.2) kojim se generiraju testni slučajevi. Glavni elementi implementiranog modula za kombinatorno testiranje prikazani su na slici 5.2.



Slika 5.2: Pregled modula za kombinatorno testiranje

Na početku se izvodi *Python* skripta *create_initial_database.py* koja automatski stvara *SQLite* bazu podataka koja će sadržavati opise ispitnih pitanja kao i rezultate kombinatornog testiranja. Ova skripta stvara sljedeće tri tablice u bazi podataka: *questions*, *testcases* i *questions_testcases*. Prva od njih, tablica *questions* sadržavat će podatke iz formalnog konteksta odabranog skupa pitanja. Tablica ima jednu identifikacijsku oznaku, a to je primarni ključ *IDquestion* (cjelobrojni tip podataka). Uz njega je i *n* polja (cjelobrojni tip podataka) imenovanih prema atributima iz formalnog konteksta. Svako od tih *n* može imati samo vrijednost 0 (ako pitanje nema taj atribut) ili vrijednost 1 (ako pitanje ima taj atribut). Imena atributa korištenih u formalnom kontekstu učitavaju se iz posebno pripremljene tekstualne datoteke *attribute_list.txt* (svaki redak sadrži ime atributa bez razmaka), a potom se automatski sastavi i izvede odgovarajuća `CREATE TABLE` naredba u SQL-u za stvaranje tablice *questions*. Nakon toga, skripta popuni tablicu *questions* s podacima iz formalnog konteksta odabranog skupa pitanja. Zato skripta treba učitati pripremljenu tekstualnu datoteku *database_prepared_data.txt* s korištenim formalnim kontekstom u kojoj svaki redak sadrži jedinstvenu identifikacijsku oznaku pitanja te razmakom razdvojenu

listu vrijednosti svih atributa (0 ili 1). Opet se prema učitanim podacima automatski stvaraju i pokreću odgovarajuće `INSERT` naredbe u SQL-u. Druga tablica u bazi podataka je *testcases* i ona ima jedan primarni ključ *IDtestcase* (cjelobrojni tip podataka), koji služi kao jedinstvena identifikacijska oznaka svakog generiranog testnog slučaja te tekstualno polje *description* koja će sadržavati listu svih značajki tog testnog slučaja. Analiza rezultata kombinatornog testiranja se upisuje u treću tablicu *questions_testcases* u kojoj se zapisuje koja pitanja u potpunosti ili djelomično pokrivaju generirane testne slučajeve. Ova tablica ima tri polja: polje *IDquestion* (cjelobrojni tip podataka), polje *IDtestcase* field (cjelobrojni tip podataka) – to su redom identifikacijske oznake ispitnog pitanja i generiranog testnog slučaja, kao i polje *full_match* (cjelobrojni tip podataka), koje bilježi pokriva li pitanje u potpunosti generirani testni slučaj (vrijednost 1) ili pitanje samo djelomično pokriva testni slučaj (vrijednost 0). Tablice *testcases* i *questions_testcases* se prije izvođenja svake iteracije kombinatornog testiranja prazne te se u njih potom upisuju rezultati provedenog testiranja.

Središnji dio modula za kombinatorno testiranje je implementiran kroz *Python* skriptu *combinatorial_test_run.py*. Ova skripta služi kao sučelje prema alatu za kombinatorno testiranje *Jenny* kojim se pripremaju ulazni podaci, poziva program *Jenny* te dodatno obrađuju rezultati testiranja. Nadalje, analizira generirane testne slučajeve i sprema sve rezultate eksperimenta, odnosno svake iteracije kombinatornog testiranja. Prije pozivanja ove skripte domenski eksperti trebaju pripremiti sljedeće ulazne tekstualne datoteke kojim se zadaju obavezni i neobavezni parametri alata *Jenny*:

- datoteka *jenny_tuples.txt* (obavezna) sadrži samo jedan broj – željenu veličinu *n*-torke značajki koje se trebaju pokriti s generiranim testnim slučajevima.
- datoteka *jenny_dimensions.txt* (obavezna) sadrži broj i sadržaj svake od dimenzija. Kao što je prethodno naglašeno dimenzije su disjunktni podskupovi atributa. Svaki redak u toj datoteci je tabulatorom razdvojena lista značajki koja predstavlja jednu dimenziju.
- datoteka *jenny_forbidden.txt* (neobavezna) sadrži zabranjene kombinacije značajki koje se ne smiju pojaviti niti u jednom generiranom testnom slučaju. Svaki redak datoteke je razmakom razdvojena lista značajki iz različitih dimenzija koje se ne smiju pojavljivati zajedno u testnom slučaju (to su uglavnom parovi značajki).

Dodatno, predložena metoda uključuje i opciju imenovanja grupa značajki što može biti jako korisno kada se radi s većim skupovima značajki odnosno atributa. Ako se ova mogućnost koristi onda treba navesti imena svake grupe značajki kao i njihove elemente u posebnoj tekstualnoj datoteci *jenny_dimensions_simplify.txt*. Svaki redak datoteke sadrži podatke o jednoj grupi značajki: ime grupe značajki, znak "#" i razmakom razdvojenu listu značajki odnosno atributa. Smatrat će se da ispitno pitanje pokriva grupu značajki ako ima barem jednu njenu značajku. Kasnije se imena grupa značajki mogu koristiti kod zadavanja dimenzija ili kod navođenja zabranjenih kombinacija značajki.

Nakon čitanja ovih ulazni tekstualnih datoteka, skripta *combinatorial_test_run.py* automatski priprema odgovarajuću naredbu za pokretanje alata *Jenny* te ga s njom pokreće i na kraju olakša prikaz dobivenih rezultata tako što zamjeni implicitna imena značajki alata *Jenny* (vide se npr. na slici 3.16) s imenima značajki ili grupa značajki navedenih u ulaznim tekstualnim podacima. Skripta isto tako dodatno sprema i korištenu naredbu za pokretanje alata *Jenny* (datoteka *jenny_command.txt*), rezultate izvođenja kombinatornog testiranja u izvornom obliku (tekstualna datoteka *jenny_raw_output.txt*) kao i dodatno obrađene rezultate (tekstualna datoteka *jenny_clean_output.txt*) te sažeti rezultat o izvršenom kombinatornom testiranju – u datoteci *jenny_test_cases_number.txt* se zapiše ukupan broj generiranih testnih slučajeva.

Potom, skripta *combinatorial_test_run.py* poziva funkciju *find_test_cases()* koja implementira glavni algoritam predložene metode kombinatornog testiranja (algoritam 3). Na početku funkcija provjera jesu li korištene grupe značajki. Ako ih je domenski ekspert naveo onda se stvara rječnik s parovima *ključ-vrijednost*: ključ je ime grupe značajki *feature group name*, a vrijednost je segment SQL naredbe s *N* elemenata iz grupe značajki: (*attribute1*=1 OR *attribute2*=1 $\cdots$ OR *attributeN*=1).

Algoritam 3 Algoritam funkcije *find_test_cases()*

inicijaliziraj brojač testnih slučajeva: $test\_case\_counter \leftarrow 0$

Ako su definirane grupe značajki u *jenny_dimensions_simplify.txt*:
  stvori rječnik $jenny\_simplified\_dict$ s imenima grupa značajki i značajkama svake grupe
  za svaki redak $line$ u datoteci *jenny_dimensions_simplify.txt*:
    podjeli $line$ kod znaka "#" i spremi rezultate kao $data[0]$ i $data[1]$
    dodaj par *ključ-vrijednost* u $jenny\_simplified\_dict$:
      key=$data[0]$ i $value$= "(" + $data[1]$.replace(" ","=1 OR ") + "=1)"

pozovi funkciju koja briše podatke iz tablica *testcases* i *questions_testcases*

za svaki redak $line$ (generirani testni slučaj) iz očišćene datoteke *output jenny_output_clean.txt*:
  $test\_case\_counter \leftarrow test\_case\_counter + 1$
  pozovi funkciju koja sprema testni slučaj u tablicu *testcases*
  ako postoji rječnik $jenny\_simplified\_dict$:
    za svaki *key* u $jenny\_simplified\_dict$:
      nađi u retku $line$ ključ *key* i zamjeni ga s njegovom vrijednošću *value*

provjeri je li testni slučaj u potpunosti pokriven s nekim pitanjem(ima) te spremi rezultate u tablicu *questions_testcases*

provjeri je li testni slučaj djelomično pokriven s nekim pitanjem(ima) te spremi rezultate u tablicu *questions_testcases*

pozovi funkciju koja generira statistiku o testnim slučajevima te generira i sprema finalni formalni kontekst

Nakon toga funkcija *find_test_cases()* isprazni tablice *testcases* i *questions_testcases*, i zatim prolazi kroz obrađene rezultate kombinatornog testiranja alatom *Jenny* te sprema svaki generirani testni slučaj s automatski izgrađenom `INSERT` naredbom u SQL-u.

U sljedećem koraku funkcija *find_test_cases()* za svaki spremljeni generirani testni slučaj automatski gradi složeniju `INSERT INTO SELECT` naredbu u SQL-u kojom se u tablicu *questions_testcases* spremaju podaci o ispitnim pitanjima koja u potpunosti ili djelomično pokrivaju taj testni slučaj. Smatra se da je testni slučaj u potpunosti pokriven ako postoji jedno ili više pitanja koja među svojim značajkama (atributima) imaju sve značajke iz tog testnog slučaja. Kako bi se provjerilo je li neki testni slučaj djelomično pokriven s ispitnim pitanjima koristi se globalna varijabla *partially_covered_limit* kojom se zadaje donja granica broja značajki koja treba biti pokrivena s pitanjima. U skladu s time, smatra se kako je testni slučaj djelomično pokriven ako postoji jedno ili više pitanja koja među svojim značajkama imaju barem $n = partially\_cover\_limit$ značajki testnog slučaja. Na slikama 5.3 i 5.4 prikazuju se predlošci SQL naredbi za dinamičku izgradnju naredbi za traženje i spremanje podataka o potpuno pokrivenim, odnosno djelomično pokrivenim testnim slučajevima.

```
INSERT INTO questions_testcases
(IDquestion, IDtestcase, full_match)
SELECT question_ID, test_case_ID, 1
FROM
questions
WHERE
test_case_feature_1 = 1 AND test_case_feature_2 = 1 AND
··· AND test_case_feature_N = 1
```

Slika 5.3: Predložak za SQL naredbu za traženje i spremanje podataka o potpuno pokrivenim testnim slučajevima

```
INSERT INTO questions_testcases
(IDquestion, IDtestcase, full_match)
SELECT question_ID, test_case_ID, 0
FROM
questions
WHERE
( (test_case_feature_1 = 1) + (test_case_feature_2 = 1)
+ ··· + (test_case_feature_N = 1) )
>= partially_cover_limit
```

Slika 5.4: Predložak za SQL naredbu za traženje i spremanje podataka o djelomično pokrivenim testnim slučajevima

Treba napomenuti kako u sustavu za upravljanje bazama podataka *SQLite* nema posebnog *Boolean* tipa podataka pa je stoga *True* jednak broju 1, a *False* je jednak broju 0. Ova karakteristika programa *SQLite* je iskorištena u SQL naredbi na slici 5.4 kojom se jednostavno identificiraju sva pitanja koja djelomično pokrivaju generirani testni slučaj. Npr. ako se postavi vrijednost donje granice na $partially\_cover\_limit = 1$ onda će `WHERE` klauzula sa slike 5.4 biti

istovjetna sljedećem logičkom izrazu (*test_case_feature_1*=1 OR *test_case_feature_2*=1 OR $\cdots$ OR *test_case_feature_N*=1).

Nakon provedene iteracije kombinatornog testiranja funkcija *find_test_cases()* pozove pomoćnu funkciju *run_sql_test_cases_statistics()* koja provodi analizu rezultata kombinatornog testiranja i sprema je u tekstualnu datoteku *jenny_test_cases_statistics.txt*. Ta pomoća funkcija za statističku obradu rezultata izvodi redom šest složenijih `SELECT` naredbi u SQL-u koje filtriraju i agregiraju podatke o dobivenim testnim slučajevima.

Detaljnije će se prikazati prva od tih SQL naredbi kojom se generira lista svih testnih slučajeva, zajedno s brojem pitanja koje u potpunosti ili djelomično pokrivaju svaki testni slučaj. Ova SQL naredba se može formalno prikazati s izrazima relacijske algebre 5.1, 5.2 i 5.3 nad relacijama *testcases*(*IDtestcase*, *description*) i *questions_testcases*(*IDquestion*, *IDtestcase*, *full_match*). Korištena je notacija proširene relacijske algebre iz [76].

Upiti definirani izrazima relacijske algebre 5.1 i 5.2 su virtualne relacije (engl. *views*) koje koriste agregatnu funkciju γ operaciju lijevoga vanjskog spajanja $\rtimes$ između relacija *testcases* i *questions_testcases* kako bi se redom prebrojila pitanja koja u potpunosti, odnosno djelomično pokrivaju svaki testni slučaj.

$$\begin{aligned} &Full \leftarrow \Pi_{x.IDtestcase\,\text{as}\,ID,\,full\_count}\,(_{x.IDtestcase}\gamma_{\text{count}(y.IDtestcase)\,\text{as}\,full\_count} \\ &(\rho_x(testcases) \rtimes_{x.IDtestcase=y.IDtestcase \wedge y.full\_match=1} \rho_y(questions\_testcases))) \end{aligned} \tag{5.1}$$

$$\begin{aligned} &Partial \leftarrow \Pi_{x.IDtestcase\,\text{as}\,ID,\,partial\_count}\,(_{x.IDtestcase}\gamma_{\text{count}(y.IDtestcase)\,\text{as}\,partial\_count} \\ &(\rho_x(testcases) \rtimes_{x.IDtestcase=y.IDtestcase \wedge y.full\_match=0} \rho_y(questions\_testcases))) \end{aligned} \tag{5.2}$$

Zadnji izraz relacijske algebre 5.3 primjenjuje operaciju projekcije Π operaciju unutarnjeg spajanja $\bowtie$ između relacija *testcases* i virtualnih relacija *Full* i *Partial* kako bi se prikazali uz svaki generirani testni slučaj i podaci dobiveni izrazima 5.1 i 5.2.

$$\begin{aligned} &\Pi_{IDtestcase, description, full\_count, partial\_count} \\ &(testcases \bowtie_{testcases.IDtestcase=Full.ID} Full \bowtie_{Full.ID=Partial.ID} Partial) \end{aligned} \tag{5.3}$$

Ukratko će se opisati i preostalih pet SQL naredbi korištenih za stvaranje izvještaja o provedenom kombinatornom testiranju. Druga i treća SQL naredba odvojeno ispisuju listu testnih slučajeva koji su potpuno pokriveni i testnih slučajeva koji su samo djelomično pokriveni. Ove liste mogu biti korisne domenskim ekspertima u kasnijim analizama. Četvrta SQL naredba računa osnovnu statistiku generiranih testnih slučajeva – ispisuje broj testnih slučajeva te broj i postotak testnih slučajeva koji su potpuno odnosno djelomično pokriveni s postojećim pitanjima. Nakon toga peta SQL naredba ispisuje sva pitanja koja u potpunosti pokrivaju testne slučajeve, zajedno s brojem testnih slučajeva koje svako od tih pitanja pokriva. Kako pitanja imaju veći broj značajki moguće je da isto pitanje pokriva više od jednoga testnog slučaja. Zad-

nja, šesta SQL naredba ispisuje formalni kontekst koji sadrži samo pitanja koja su u potpunosti pokrivena s testnim slučajevima. Na kraju postupka kombinatornog testiranja, kada je svaki testni slučaj u potpunosti pokriven to će postati završni formalni kontekst koji će se kasnije obraditi metodom FCA.

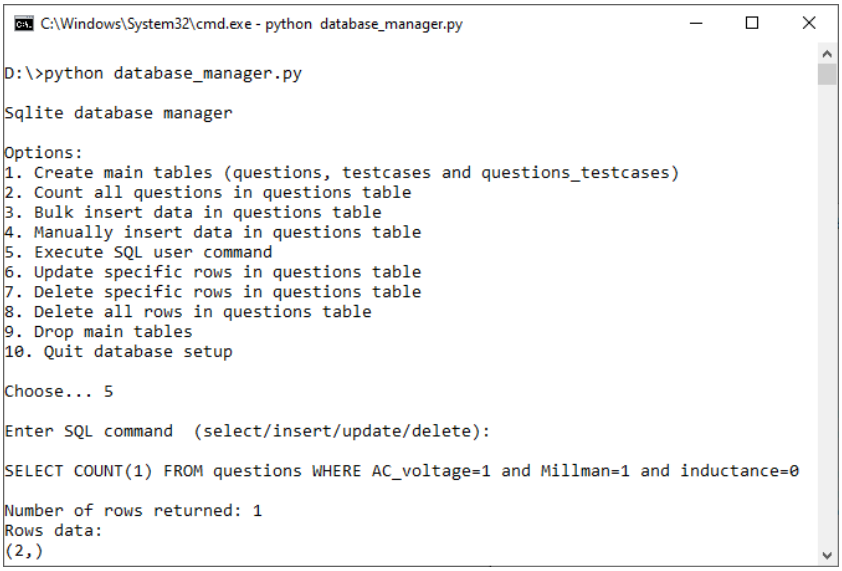
Nakon provedene statističke obrade rezultata funkcija *run_sql_test_cases_statistics()* automatski gradi finalni formalni kontekst koji sadrži podskup svih pitanja koja su potpuno pokrivena s generiranim testnim slučajevima. Finalni formalni kontekst se potom automatski sprema kao tekstualna datoteka *final_formal_context.cxt* u posebnom ulaznom formatu *.cxt* kojeg koristi alat za metodu FCA *Concept Explorer 1.3*. Uz to generira se i tekstualna datoteka *selected_questions.txt* u kojoj je u svakom retku zapisna identifikacijska pitanja iz završnog formalnog konteksta i potom lista atributa tog pitanja. Kasnije će se vidjeti kako se ove dvije datoteke koriste za automatiziranu pripremu formativnih provjera znanja u sustavima za e-učenje. Dodatno, kako bi se olakšalo naknadno analiziranje svih podataka automatski se sprema i inicijalni formalni kontekst sa svim pitanjima iz korištenog skupa u formatu *.cxt* kao tekstualna datoteka *initial_formal_context.cxt*.

Prije nego što skripta *combinatorial_test_run.py* završi s radom kopiraju se sve korištene ulazne i generirane izlazne tekstualne datoteke zajedno sa *SQLite* bazom pitanja u poseban direktorij označen trenutnim vremenskim žigom. Na taj se način mogu kasnije lakše analizirati rezultati svakog provedenog eksperimenta.

Modul za kombinatorno testiranje sadrži i posebnu skriptu u *Pythonu* pod imenom *database_manager.py*, koja se može koristiti za upravljanje *SQLite* bazom pitanja. Ova skripta pruža jednostavno sučelje u obliku izbornika kojom se omogućava pregled, traženje i spremanje podataka u *SQLite* bazu pitanja. U svim navedenim *Python* skriptama korišten je modul *sqlite3* za upravljanje i komunikaciju s bazom podataka *SQLite*.

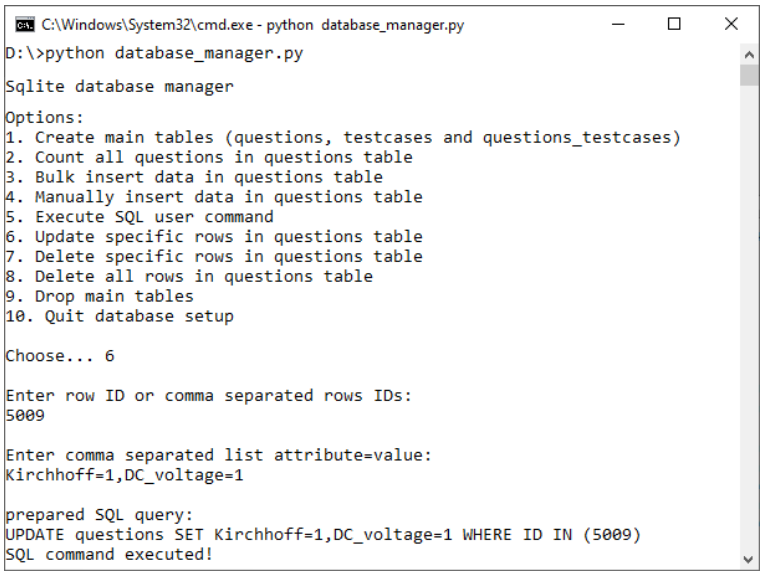
Korištenjem ove skripte korisnici mogu ručno izvesti naredbe za stvaranje ili brisanje prethodno opisanih tablica i dobiti osnovne statističke podatke o njima. Skripta isto tako olakšava dodavanje novih opisa pitanja u tablicu ispitnih pitanja *questions*, npr. nakon što se osmisli novo pitanje koje implementira prethodno nepokriveni testni slučaj. Nadalje, omogućava korisnicima izvođenje proizvoljnih SQL naredbi. Ovo može biti posebno korisno kod analize rezultata kombinatornog testiranja, npr. ako nema postojećeg pitanja koje u potpunosti pokriva zadani testni slučaj može se napisati `SELECT` naredba u SQL-u koja će pronaći pitanja čiji su opisi najbliži značajkama testnog slučaja. Isto tako na ovaj način se može eventualno ispraviti ili proširiti opis svakoga pitanja, npr. ako se utvrdi da vrijednosti atributa pitanja nisu bila dobro postavljena. Dakle, za domenske eksperte s naprednijim znanjem o SQL-u ova skripta za upravljanje bazom može biti zanimljiva jednostavna alternativa ostalim složenijim alatima za upravljanje *SQLite* bazama podataka, kao što je alat u naredbenom retku *Sqlite3* [105] ili aplikacija s grafičkim sučeljem *SQLiteStudio* [106].

Prikazat će se dva kratka primjera izvođenja skripte za upravljanje bazom podataka. Prvi primjer je prikazan na slici 5.5, a u njemu se izvodi korisnička `SELECT` naredba u SQL-u kojom se u tablici *questions* prebrojila pitanja označena sa zadanom konfiguracijom tri različita atributa. Iz rezultata se vidi kako su u bazi pronađena dva pitanja koja imaju atribute *AC_voltage* i *Millman*, a nemaju atribut *inductance*.



Slika 5.5: Primjer izvođenja korisničke SQL naredbe sa skriptom *database_manager.py*

U drugom primjeru sa slike 5.6 korisnik može ažurirati oznake odabranih pitanja. Iz zadanih podataka se automatski gradi i izvede odgovarajuća `UPDATE` naredba u SQL-u. Korisnik je u ovom slučaju zadao da pitanje 5009 treba imati atribute *Kirchhoff* i *DC_voltage*.



Slika 5.6: Primjer ažuriranja podataka o pitanjima sa skriptom *database_manager.py*

5.2 Studijski primjer

U ovom potpoglavlju će se predstaviti studijski primjer nad kojim će se izvoditi predložena metoda za kombinatorno testiranje. Kao studijski primjer korist će se skup od 473 označena ispitna pitanja s predmeta *Osnove elektrotehnike* na prvoj godini preddiplomskog studija na FER-u. Pitanja su preuzeta iz vlastitog zavodskog sustava za e-učenje *WebOE*[1]. Ovaj skup od 473 ispitna pitanja je ručno označen s predefiniranim skupom od 50 atributa prikazanih na slici 5.7 koje pokrivaju cjelokupno nastavno gradivo predmeta. Prethodne inačice ovoga označenoga skupa pitanja već su se koristila u prethodnim vlastitim istraživanjima [102, 108], a navedeni skup predstavlja samo manji dio od ukupno preko 3800 osnovnih inačica ispitnih pitanja pohranjenih u sustavu *WebOE*.

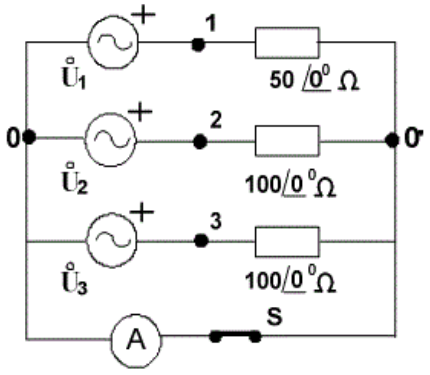
```
01: force_electric
02: field_electric
03: potential_electric
04: energy_electric
05: charge
06: capacitance
07: capacitors
08: resistance
09: DC_current
10: DC_voltage
11: DC_power
12: current_source
13: voltage_source
14: real_source
15: ideal_source
16: force_magnetic
17: field_magnetic
18: energy_magnetic
19: flow_magnetic
20: induced_voltage
21: Ohm
22: Kirchhoff
23: Thevenin
24: Norton
25: Millman
26: superposition
27: bridge
28: delta_star
29: impedance
30: AC_current
31: AC_voltage
32: AC_power
33: inductance
34: inductive_coupling
35: sine_wave
36: vectors
37: phasors
38: sine_sources
39: frequency_dependence
40: non_sinusoidal_sources
41: harmonics
42: transients
43: three_phase_phasor_diagram
44: three_phase_sources
45: three_phase_star_balanced
46: three_phase_star_unbalanced
47: three_phase_delta_balanced
48: three_phase_delta_unbalanced
49: three_phase_neutral
50: three_phase_break
```

Slika 5.7: Predefinirani skup od 50 atributa koje opisuju gradivo predmeta *Osnove elektrotehnike*

[1] *WebOE* (https://osnove.tel.fer.hr) je sustav za e-učenje koji se koristi na predmetu *Osnove elektrotehnike*, a razvija se na Zavodu za osnove elektrotehnike i električka mjerenja na FER-u [107].

U sustavu *WebOE* su podržani različiti tipovi pitanja (npr. pitanja s višestrukim izborom odgovora, pitanja s izborom odgovora *točno/krivo*, pitanja s unosom kratkog tekstualnog odgovora te računska pitanja s unosom numeričkih odgovora), iako su u odabranom skupu sva pitanja tipa višestrukog izbora odgovora (tri do pet odgovora, samo jedan je točan) i računskoga tipa gdje studenti trebaju unijeti od jednoga do tri numerička odgovora koji se automatski ocjenjuju. Kao primjer, na slici 5.8 je prikazano jedno od pitanja s višestrukim izborom odgovora na engleskom jeziku kao i lista atributa s kojima je pitanje označeno. Treba naglasiti kako su izvorna pitanja na hrvatskom jeziku, ali su mnoga od njih dostupna i na engleskom s obzirom na to da se zadnjih godina na FER-u izvodi i nastavni program na engleskom jeziku.

Pitanje ID2909 (*varijanta na engleskom jeziku*)
Figure shows a three-phase network where the ammeter measures 4 A when the switch S is closed. Calculate total real power *P* of the three-phase load.



Ponuđeni odgovori:
A) 1200 W B) 2400 W C) 3600 W D) 6400 W E) 12800 W
(*točan odgovor D*)

Oznake pitanje (atributi):
potential_electric, voltage_source, ideal_source, Ohm, Kirchhoff, impedance, AC_current, AC_voltage, AC_power, vectors, phasors, sine_sources, three_phase_phasor_diagram, three_phase_sources, three_phase_star_unbalanced, three_phase_neutral

Slika 5.8: Primjer pitanja s višestrukim odabirom odgovora

Iako se tekst pitanja na slici 5.8 doima vrlo kratkim i jasnim, rješenje zadanog problema nije trivijalno. Važni podaci su sadržani u samoj slici koja prati tekst pitanja. Iz priložene sheme može se vidjeti kako je trofazni izvor spojen u zvijezdu, a da su u trofaznom trošilu sva trošila omska i da su spojena u nesimetričnu zvijezdu. Na kraju je bitno primijetiti kako ampermetar uz zatvorenu sklopku mjeri efektivnu jakost struje kroz nul-vodič koji povezuje zvjezdišta izvora i trošila. Za efikasan izračun ovoga zadatka treba primijeniti osnovne zakone elektrotehnike (Ohmov zakon i Kirchhoffov zakon za struje), a isto tako treba znati kako se računaju struje,

naponi i snage u izmjeničnim krugovima korištenjem fazorskog ili vektorskog računa. Na kraju se vidi kako je pitanje dosta specifično, a samim time je označeno s opsežnom listom atributa.

Testiranjem korištenog skupa pitanja pomoću metode za kombinatorno testiranje želi se osigurati da odabrana pitanja pokrivaju sve parove ili trojke atributa. Dakle, tako odabrana pitanja će povezivati različite pojmove iz nastavnog gradiva čime se nastoji pomoći studentima da podrobnije usvoje nastavno gradivo. Kako bi se postigli optimalni rezultati treba eksplicitno navesti koje su kombinacije atributa zabranjena. Neke od takvih kombinacija nisu smislene – npr. par (*phasors*, *non_sinusoidal_sources*), a neke su previše složene i nisu pokrivene s nastavnim gradivom – npr. par (*three_phase_sources*, *harmonics*). Nakon automatskog generiranja testnih slučajeva treba provjeriti jesu li oni već pokriveni s postojećim ispitnim pitanjima u skupu, a inače treba razmotriti mogućnost dodavanja novi pitanja u skup koja će implementirati do sada nepokrivene testne slučajeve. Isto tako postupak kombinatornog testiranja može detektirati greške u označavanju pitanja. Primjerice, ako se primijeti da neki generirani testni slučaj nema smisla, odnosno nije izvediv, a ipak je pokriven s nekim od pitanja iz skupa može se zaključiti kako je takvo pitanje označeno na krivi način.

Prije izvođenja predložene metode kombinatornog testiranja treba pripremiti sve potrebne ulazne datoteke kako je opisano u potpoglavlju 5.1 – lista atributa sa slike 5.7 mora se pohraniti u datoteku *attribute_list.txt*, a formalni kontekst odabranog skupa pitanja treba biti pohranjen u datoteci *database_prepared_data.txt*. U ovom slučaju se formalni kontekst automatski se izgradio iz baze podataka sustava za e-učenje, a mali uzorak podataka iz pripremljene ulazne datoteke *database_prepared_data.txt* prikazan je na slici 5.9. Kao što se vidi s te slike u svakom retku je jedinstvena identifikacijska oznaka pitanja, a potom lista od 50 binarnih vrijednosti razdvojenih razmakom kojima se određuje ima li to pitanje atribute i to redom kako su navedeni na slici 5.7.

```
2937 0 0 1 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
2938 0 0 1 0 0 0 1 1 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0
2939 0 0 1 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2940 0 0 1 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 1 0 0 1 1 1 0 0 0 0 1 1 0 0 0 1 0 0
2941 0 0 1 0 1 1 1 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2942 0 0 1 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
2943 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2944 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0
2945 0 0 1 0 0 0 0 1 1 1 1 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
```

Slika 5.9: Uzorak formalnog konteksta iz datoteke *database_prepared_data.txt*

Treba naglasiti kako će metoda FCA iz pripremljenog formalnog konteksta s 473 pitanja i 50 atributa generirati konceptualnu rešetku s ukupno 3504 formalna koncepta ili *EKP* točke i ukupno 15284 usmjerenih grana. Može se odmah primijetiti kako je takva konceptualna rešetka izuzetno velika i složena. Upravo je zadatak metode kombinatornog testiranja je pronalazak sažetog skupa pitanja koji na kraju može rezultirati s manjim formalnim kontekstom, a samim time i jednostavnijom konceptualnom rešetkom.

5.2.1 Predložene strategije za kombinatorno testiranje

U sljedećem važnom koraku treba zadati na koji će se način provoditi postupak kombinatornog testiranja podataka sadržanih u formalnom kontekstu. Glavni zadatak je definiranje broja dimenzija te određivanje kako će se odabrane značajke rasporediti po dimenzijama. Kao što je već rečeno ranije svaki atribut iz formalnog konteksta smatra se jedinstvenom značajkom, a dimenzije su disjunktni skupovi značajki. Način na koji se odabere broj i sadržaj dimenzija ima veliki utjecaj na broj i kvalitetu generiranih testnih slučajeva. Isto tako raspored značajki po dimenzijama može utjecati na broj zabranjenih kombinacija značajki koje se zadaju tijekom procesa kombinatornog testiranja. U ovome istraživanju cilj je generirati relativno mali broj testnih slučajeva uz što manji broj potrebnih zabranjenih kombinacija značajki. Pritom bi generirani testni slučajevi trebali biti kvalitetni, tako da su ili već pokriveni s postojećim pitanjima ili predstavljaju smislene opise prema kojima nastavnici mogu sastaviti i dodati nova korisna pitanja u bazu.

Stoga su se kroz ovo istraživanje predložile četiri osnovne strategije za definiranje dimenzija koje će se predstaviti u nastavku, a potom se s njima provelo kombinatorno testiranje i to za različite veličine n-torki značajki koje treba pokriti testovima kao i uz dodatno korištenje zabranjenih kombinacija značajki.

Strategija 1

U predloženoj Strategiji 1 značajke su grupirane u četiri različite dimenzije. Prva dimenzija se sastoji od 17 značajki vezanih uz elektrostatiku, magnetizam i složenije pojave u krugovima. Druga dimenzija pokriva osnovne zakone i teoreme elektrotehnike, a sastoji se od 8 značajki. Treća dimenzija se sastoji od 13 značajki koje pokrivaju idealne i realne naponske i struje izvore kao i trofazne sustave. Posljednja, četvrta dimenzija ima 12 značajki koje pokrivaju istosmjerne i izmjenične električne veličine, fazore i frekvencijske ovisnosti. Popis svih dimenzija je dan na slici 5.10.

```
DIMENZIJA 1:
force_electric, field_electric, potential_electric, energy_electric, charge, capacitance,
capacitors, inductance, inductive_coupling, force_magnetic, field_magnetic, energy_magnetic,
flow_magnetic, induced_voltage, non_sinusoidal_sources, harmonics, transients

DIMENZIJA 2:
Ohm, Kirchhoff, bridge, delta_star, superposition, Thevenin, Norton, Millman

DIMENZIJA 3:
current_source, voltage_source, real_source, ideal_source, sine_sources,
three_phase_phasor_diagram, three_phase_sources, three_phase_star_balanced,
three_phase_star_unbalanced, three_phase_delta_balanced, three_phase_delta_unbalanced,
three_phase_neutral, three_phase_break

DIMENZIJA 4:
DC_current, DC_voltage, DC_power, resistance, AC_current, AC_voltage, AC_power, impedance,
sine_wave, vectors, phasors, frequency_dependence
```

Slika 5.10: Popisi definiranih dimenzija u Strategiji 1

Na ovaj način je ukupno moguće generirati najviše $17 \cdot 8 \cdot 13 \cdot 12 = 21216$ različitih testnih slučajeva. Treba odmah napomenuti kako ova strategija ima određenih manjkavosti, npr. interesantna kombinacija značajki (*Thevenin*, *Millman*) neće se naći u generiranim testnim slučajevima jer te obije značajke pripadaju drugoj dimenziji.

Strategija 2

U drugoj strategiji predlaže se potpuno drukčiji pristup pripremi ulaznih podataka za kombinatorno testiranje. U ovom pristupu se atributi ne grupiraju u manji broj dimenzija nego se svaki od 50 atributa na slici 5.7 uzima kao zasebna dimenzija. Dakle, sada je definirano 50 dimenzija, a svaka dimenzija ima samo dvije vrijednosti: *True* ili *False*. Stoga će vrijednost *True* biti označena sa značajkom *attribute_name_1* (onda testni slučaj ima tu značajku), a vrijednost *False* će biti označena sa značajkom *attribute_name_0* (onda testni slučaj nema tu značajku). U usporedbi sa Strategijom 1 sada su moguće sve kombinacije atributa pa se zato zbog kombinatorne eksplozije enormno poveća najveći broj svih različitih testnih slučajeva na čak 2^{50}, odnosno na preko bilijardu testnih slučajeva.

Strategija 3

U ovoj strategiji se predlaže kao i u Strategiji 1 korištenje četiri dimenzije, samo što se u ovom slučaju značajke rasporede po dimenzijama na slučajan način. Takve slučajno generirane dimenzije mogu se vidjeti na slici 5.11, a u korištenoj slučajnoj konfiguraciji imaju redom po 12, 12, 17 i 9 značajki.

```
DIMENZIJA 1:
three_phase_star_balanced, inductive_coupling, real_source, flow_magnetic, induced_voltage,
three_phase_sources, capacitance, delta_star, sine_wave, voltage_source, AC_power,
frequency_dependence
DIMENZIJA 2:
inductance, three_phase_delta_balanced, Kirchhoff, ideal_source, current_source,
three_phase_break, three_phase_star_unbalanced, charge, three_phase_phasor_diagram,
Thevenin, Ohm, DC_voltage
DIMENZIJA 3:
Millman, DC_current, field_electric, vectors, three_phase_delta_unbalanced, impedance,
DC_power, energy_magnetic, AC_voltage, superposition, AC_current, capacitors,
energy_electric, force_electric, phasors, resistance, force_magnetic
DIMENZIJA 4:
field_magnetic, transients, potential_electric, harmonics, bridge, sine_sources, Norton,
non_sinusoidal_sources, three_phase_neutral
```

Slika 5.11: Popisi definiranih dimenzija u Strategiji 3

Najveći broj svih različitih testnih slučajeva je prema Strategiji 3 jednak $12 \cdot 12 \cdot 17 \cdot 9 = 22032$, a to je usporedivo s najvećim brojem testnih slučajeva u Strategiji 1. Opet treba naglasiti kako se mogu uočiti određeni nedostaci ove strategije, npr. vrlo česta kombinacija značajki (*Ohm*, *DC_voltage*) neće se pojaviti niti u jednom testnom slučaju jer oba pripadaju drugoj

dimenziji. S druge strane, ranije spomenuta kombinacija značajki (*Thevenin*, *Millman*), koja se ne pojavljuje u Strategiji 1 ovdje će biti pokrivena s barem jednim testnim slučajem. Vidi se kako je razlog tome to što su te značajke sada nalaze u različitim dimenzijama.

Strategija 4

I u ovom predloženom pristupu kreće se od dimenzija definiranih u Strategiji 1 (slika 5.10), ali se potom srodne značajke u svakoj dimenziji spajaju u grupe značajki. Korištene grupe značajki su prikazane na slici 5.12, a ovako pojednostavljene dimenzije su dane na slici 5.13.

```
electrostatics:
force_electric
field_electric
energy_electric
charge
capacitors

DC_values:
DC_current
DC_voltage
resistance

AC_values:
AC_current
AC_voltage
impedance

magnetism:
inductive_coupling
force_magnetic
field_magnetic
energy_magnetic
flow_magnetic
induced_voltage

sources:
current_source
voltage_source
real_source
ideal_source
sine_sources
non_sinusoidal_sources

phasor_diagrams:
sine_wave
vectors
phasors

three_phase_easier:
three_phase_phasor_diagram
three_phase_sources

three_phase_star:
three_phase_star_balanced
three_phase_star_unbalanced

three_phase_delta:
three_phase_delta_balanced
three_phase_delta_unbalanced

three_phase_harder:
three_phase_neutral
three_phase_break
```

Slika 5.12: Grupe značajki koje se koriste u Strategiji 4

```
DIMENZIJA 1:
electrostatics, magnetism, potential_electric, capacitance, inductance, harmonics,
transients
DIMENZIJA 2:
Ohm, Kirchhoff, bridge, delta_star, superposition, Thevenin, Norton, Millman
DIMENZIJA 3:
sources, three_phase_easier, three_phase_star, three_phase_delta, three_phase_harder
DIMENZIJA 4:
DC_values, AC_values, DC_power, AC_power, phasor_diagrams, frequency_dependence
```

Slika 5.13: Popisi definiranih dimenzija u Strategiji 4

Na ovaj način su se značajno reducirale veličine pojedinih dimenzija i sada imaju redom po 7, 8, 5 i 6 značajki. Zbog toga je i najveći mogući broj različitih testnih slučajeva drastično manji i sada je jednak $7 \cdot 8 \cdot 5 \cdot 6 = 1680$.

Kao što je ranije navedeno prilikom traženja pitanja koja pokrivaju generirani testni slučaj smatrat će se kako je neka grupa značajki sa slike 5.12 pokrivena ako pitanje ima barem jednu značajku iz te grupe. Ovime se daje veća fleksibilnost nastavnicima i drugim domenskim ekspertima prilikom kombinatornog testiranja ispitnih pitanja. Npr. testni slučaj koji u svom opisu ima grupu značajki *phasor_diagram* može na kraju biti pokriven s pitanjima iz skupa koji imaju bilo koji od sljedećih atributa: *sine_wave*, *vectors* i/ili *phasors*.

5.3 Rezultati i rasprava

U ovom potpoglavlju će se predstaviti rezultati izvođenja predložene metode kombinatornog testiranja nad razmatranim skupom koji se inicijalno sastoji od 473 označena pitanja preuzeta iz baze podataka vlastitog zavodskog sustava za e-učenje *WebOE*. U svakoj iteraciji procesa kombinatornog testiranja se generira lista testnih slučajeva, a potom se na automatiziran način putem *Python* skripte provjerava je li svaki generirani slučaj u potpunosti ili barem djelomično pokriven s nekim od postojećih pitanja u inicijalnom skupu, kao što je detaljno objašnjeno u potpoglavlju 5.1. Na kraju procesa kombinatornog testiranja će se svi nepokriveni dozvoljeni testni slučajevi implementirati kao nova pitanja i dodati u inicijalni skup od 473 pitanja.

Prvo će se dati rezultati inicijalnog kombinatornog testiranja korištenjem sve četiri strategije definirane u potpoglavlju 5.2. Svakom od strategija se proveo postupak kombinatornog testiranja četiri puta, tako da se pokriju parovi značajki (postavljanjem tražene veličine n-torki značajki na $t = 2$) te sve trojke značajki (uz postavljanje $t = 3$) iz različitih dimenzija i to bez i s eksplicitno navedenom listom zabranjenih kombinacija značajki koje se ne smiju pojavljivati u generiranim testnim slučajevima. Isto tako treba naglasiti kako će se u svim izvođenjima kombinatornog testiranja smatrati kako je generirani testni slučaj djelomično pokriven s postojećim pitanjem iz baze ako to pitanje ima barem 3 značajke iz testnog slučaja (zadano $partially\_cover\_limit = 3$ u predlošku na slici 5.4). Nakon analize inicijalnih rezultata dobivenih s navedenim strategijama odabrat će se najprikladnija strategija kojom će se izvršiti cjelokupni postupak kombinatornog testiranja.

5.3.1 Rezultati kombinatornog testiranja prema Strategiji 1

U Strategiji 1 koristile su se četiri dimenzije definirane prema slici 5.10, a rezultati testiranja su prikazani u tablici 5.1. Svaki podatkovni stupac u tablici odgovara izvođenju kombinatornog testiranja s drukčije zadanim ulaznim parametrima.

U prvom izvođenju kombinatornog testiranja, postavlja se veličina *n*-torki značajki na $t = 2$ i nisu zadane zabranjene kombinacije značajki. Broj generiranih testnih slučajeva je bio 225. Među njima je ukupno 102 testna slučaja (45,33%) koja su djelomično pokrivena s postojećim pitanjima, ali samo 11 testnih slučajeva je (4,89%) je u potpunosti pokriveno s barem jednim pitanjem. Naizgled je puno pitanja (124 od 473 pitanja, odnosno 26,22%) potpuno pokriveno s barem jednim testnim slučajem, ali od toga je čak 107 pitanja potpuno pokriveno sa samo jednim generiranim testnim slučajem: (*potential_electric*, *Ohm*, *sine_sources*, *impedance*).

Tablica 5.1: Izvođenje kombinatornog testiranja prema Strategiji 1

Izvođenje testa	1	2	3	4
Parametri testa	$t = 2$	$t = 2$ i 3 zabranjene kombinacije	$t = 3$	$t = 3$ i 3 zabranjene kombinacije
Broj testnih slučajeva	225	223	2670	2641
Potpuno pokriveni testni slučajevi	11 [4,89%]	17 [7,62%]	153 [5,73%]	163 [6,17%]
Djelomično pokriveni testni slučajevi	102 [45,33%]	109 [48,88%]	1280 [47,94%]	1282 [48,54%]
Potpuno pokrivena pitanja	124 [26,22%]	113 [23,89%]	234 [49,47%]	257 [54,33%]

Nadalje, u drugom izvođenju kombinatornog testiranja opet je zadana veličina *n*-torki značajki na $t = 2$, ali je sada zadana i kratka lista od tri zabranjene kombinacije značajki, a to su: (*DC_current*, *harmonics*), (*DC_current*, *induced_voltage*) i (*force_electric*, *Millman*). Zbog toga je broj generiranih testnih slučajeva pao na 223, a nešto više testnih slučajeva je u potpunosti pokriveno (7,62%).

U trećem i četvrtom izvođenju kombinatornog testiranja je zadana veličina *n*-torki značajki kao $t = 3$. U trećem izvođenju testiranja nisu korištene zabranjene kombinacije značajki, dok je u četvrtom izvođenju korištena gore spomenuta lista od tri zabranjene kombinacije značajki. Sada je bilo generirano puno više testnih slučajeva i to 2670 u trećem izvođenju, a 2641 u četvrtom izvođenju testiranja. Postotak potpuno pokrivenih testnih slučajeva je u oba slučaja bio dosta malen, redom 5,73% i 6,17%. Zbog većeg broj generiranih testnih slučajeva narastao je i broj pitanja koja su potpuno pokrivena na 49,47% u trećem izvođenju te 54,33% u četvrtom izvođenju testiranja.

Kao što se može vidjeti iz tablice 5.1 ova strategija inicijalno generira mnogo testnih slučajeva, a pogotovo kada se žele pokriti sve trojke značajki iz različitih dimenzija. Isto tako

može se primijetiti kako je u sva četiri izvođenja testa broj potpuno pokrivenih testnih slučajeva vrlo mali. Jedno od rješenja bi moglo biti sastavljanje puno veće liste zabranjenih kombinacija značajki, ali to je vrlo delikatan i vremenski zahtjevan posao s obzirom na to da je definirano ukupno 50 značajki. Povrh toga, kao što je spomenuto u potpoglavlju 5.2 ovom strategijom se neće generirati testni slučajevi koji uključuju neke korisne kombinacije značajki jer se nalaze u istoj dimenziji.

Treba ukratko spomenuti kako je samo izvođenje kombinatornog testiranja alatom *Jenny* bilo vrlo brzo, čak i na prosječnom stolnom računalu korištenom za testiranja (*Intel i3-540* procesor i 14 GB radne memorije uz SSD disk). Vremena izvođenja su se kretala od 27,52 ms u drugom izvođenju testa ($t = 2$ i zadane tri zabranjene kombinacije značajki) do 376,46 ms u trećem izvođenju testa ($t = 3$ i bez zabranjenih kombinacija značajki). Prilikom mjerenja vremena izvođenja kombinatornog testiranja korištena je naredba *Measure-Command* dostupna u proširenom naredbenom retku *PowerShell* u operacijskim sustavima *Windows* [109].

5.3.2 Rezultati kombinatornog testiranja prema Strategiji 2

U sljedećem pristupu korištena je Strategija 2 za postupak kombinatornog testiranja. Sada je definirano 50 dimenzija (jedna za svaki atribut), a svaka ima dvije značajke (ima atribut ili nema atribut). Rezultati četiri provedena inicijalna kombinatorna testiranja navedena su u tablici 5.2.

Tablica 5.2: Izvođenje kombinatornog testiranja prema Strategiji 2

Izvođenje testa	1	2	3	4
Parametri testa	$t = 2$	$t = 2$ i 85 zabranjenih kombinacija	$t = 3$	$t = 3$ i 85 zabranjenih kombinacija
Broj testnih slučajeva	14	17	36	46
Potpuno pokriveni testni slučajevi	0 [0,0%]	0 [0,0%]	0 [0,0%]	0 [0,0%]
Djelomično pokriveni testni slučajevi	14 [100,0%]	17 [100,0%]	36 [100,0%]	44 [95,65%]
Potpuno pokrivena pitanja	0 [0,0%]	0 [0,0%]	0 [0,0%]	0 [0,0%]

Primjenom ove strategije nisu se dobili korisni rezultati. U sva četiri izvođenja kombinatornog testiranja, korištenjem veličine n-torki značajki od $t = 2$ i $t = 3$ te bez i s većom listom od 85 zabranjenih kombinacija atributa nije bilo niti jednog potpuno pokrivenog generiranog

testnog slučaja. S obzirom na to da ima 50 dimenzija, svaka sa samo dvije značajke (ima atribut definiran imenom dimenzije ili ga nema) kombinatorno testiranje će vrlo učinkovito generirati izuzetno mali skup testnih slučajeva koji će pokriti sve dozvoljene parove ili trojke značajki (od 14 testnih slučajeva u prvom izvođenju do 46 testnih slučajeva u četvrtom izvođenju testiranja). Nažalost, gotovo svi tako generirani testni slučajevi predstavljaju kombinacije atributa koje su ili prekompleksne za nastavno gradivo predmeta ili kontradiktorne i neizvedive. Čak i kada se koristila lista od 85 zabranjenih kombinacija atributa opet nije bilo kvalitetnijih testnih slučajeva. Kako se generirao mali broj testnih slučajeva, a ukupni broj dimenzija je 50 vidi se kako je u inicijalnom skupu teško pronaći pitanja čiji bi opis u potpunosti odgovarao nekom od testnih slučajeva. Nadalje, stvaranje novih pitanja temeljenih na generiranim testnim slučajevima u ovom slučaju je vrlo zahtjevno jer bi nova pitanja trebala imati veliki broj raznorodnih značajki. Jedan od načina kako bi se mogla povećati kvaliteta generiranih testnih slučajeva je stvaranje puno duže liste zabranjenih kombinacija značajki, ali zbog velikog broja dimenzija to uopće nije lako izvedivo. Isto tako mogla bi se povećati tražena veličina *n*-torki značajki koje treba pokriti s testnim slučajevima na $t = 4$ ili i više, čime bi se povećao broj generiranih testnih slučajeva, ali bi bilo izglednije da su neki od tih testnih slučajeva smisleni. Na kraju treba primijetiti kako su gotovo svi testni slučajevi djelomično pokriveni s pitanjima u inicijalnom skupu. Dakako, ovo nije previše iznenađujuće jer svaki testni slučaj ima do 50 značajki pa je bilo relativno lako pronaći pitanja koja imaju barem tri značajke.

Što se tiče vremena izvođenja kombinatornog testiranja ona su nešto duža nego u Strategiji 1, od 40,82 ms u prvom izvođenju testa ($t = 2$ i bez zabranjenih kombinacija značajki) do 724,16 ms u četvrtom izvođenju testa ($t = 3$ i 85 zabranjenih kombinacija značajki). Može se primijetiti kako uz ovako zadane parametre alatu *Jenny* treba više vremena kako bi pronašao i generirao odgovarajuće testne slučajeve iako je njihov broj značajno manji nego u Strategiji 1.

5.3.3 Rezultati kombinatornog testiranja prema Strategiji 3

S predloženom Strategijom 3 korišten je slučajno generiran raspored značajki u četiri dimenzije sa slike 5.11. Opet su provedena četiri inicijalna izvođenja kombinatornog testiranja s različitim ulaznim parametrima, a svi dobiveni rezultati prikazani su u tablici 5.3.

Usporedbom rezultata inicijalnih izvođenja kombinatornih testova iz tablice 5.3 s rezultatima prema Strategiji 1 u tablici 5.1 vidi se kako su se u oba slučaja dobili kvantitativno slični rezultati. Razlog tome je isti broj dimenzija iako su se zbog različitog rasporeda značajki po dimenzijama generirali najčešće potpuno drukčiji testni slučajevi. U drugom i četvrtom izvođenju testiranja korištene su sljedeće zabranjene kombinacije značajki: (*real_source*, *field_electric*), (*induced_voltage*, *harmonics*), (*three_phase_delta_balanced*, *DC_current*) kao i (*three_phase_star_balanced*, *DC_current*).

Tablica 5.3: Izvođenje kombinatornog testiranja prema Strategiji 3

Izvođenje testa	1	2	3	4
Parametri testa	$t = 2$	$t = 2$ i 4 zabranjene kombinacija	$t = 3$	$t = 3$ i 4 zabranjene kombinacija
Broj testnih slučajeva	213	209	2480	2443
Potpuno pokriveni testni slučajevi	13 [6,1%]	13 [6,22%]	161 [6,49%]	162 [6,63%]
Djelomično pokriveni testni slučajevi	88 [41,31%]	88 [42,11%]	1088 [43,87%]	1057 [43,27%]
Potpuno pokrivena pitanja	127 [26,85%]	164 [23,89%]	227 [47,99%]	234 [49,47%]

Iz rezultata proizlazi kako nije opaženo povećanje broja potpuno pokrivenih testnih slučaja s korištenjem zabranjenih kombinacija značajki kao što se ipak u određenoj mjeri dogodilo u drugom i četvrtom izvođenju kombinatornog testiranja prema Strategiji 1 (tablica 5.1). Pažljivijim pregledom generiranih testnih slučajeva po Strategiji 3 vidjelo bi se kako ih je mnogo presloženo za gradivo predmeta *Osnove elektrotehnike*, npr. takav je testni slučaj br. 173 iz prvoga izvođenja testiranja ($t = 2$ i bez zabranjenih kombinacija značajki): (*three_phase_sources*, *three_phase_break*, *AC_voltage*, *harmonics*). S druge strane, u četvrtom izvođenju testiranja ($t = 3$ i četiri zabranjene kombinacije značajki) mogu se pronaći i neki vrlo korisni opisi pitanja kojih nema među rezultatima prema Strategiji 1, kao što je npr. testni slučaj br. 1700 (*real_source*, *current_source*, *DC_current*, *Norton*).

Vrijeme generiranja testnih slučajeva je bilo slično kao u Strategiji 1 – od 33,01 ms do 380,69 ms, a što je u skladu s prethodnim razmatranjima.

5.3.4 Rezultati kombinatornog testiranja prema Strategiji 4

Kod izvođenja kombinatornog testiranja prema Strategiji korištene četiri pojednostavljene dimenzije prikazane na slici 5.13, zajedno s grupama značajki sa slike 5.12. Dakle, u ovom pristupu se koristilo grupiranje srodnih značajki kako bi se reducirale veličine dimenzija, a samim time bi se trebao smanjiti i broj testnih slučajeva.

Isto tako sada je inicijalno zadana veća lista od 30 zabranjenih kombinacija značajki koje su prikazane na slici 5.14. A iz sažetka rezultata prikazanih u tablici 5.4 vidi se kako se ovaj put u sva četiri izvođenja kombinatornog testiranja s različitim ulaznim parametrima zaista generiralo značajno manje testnih slučajeva nego u prvoj i trećoj strategiji.

```
(electrostatics, Thevenin)
(electrostatics, Norton)
(electrostatics, Millman)
(electrostatics, three_phase_easier)
(electrostatics, three_phase_star)
(electrostatics, three_phase_delta)
(electrostatics, three_phase_harder)
(electrostatics, AC_values)
(electrostatics, DC_power)
(electrostatics, AC_power)
(electrostatics, phasor_diagrams)
(electrostatics, frequency_dependence)
(magnetism, three_phase_easier)
(magnetism, three_phase_star)
(magnetism, three_phase_delta)
(magnetism, three_phase_harder)
(harmonics, superposition)
(harmonics, three_phase_easier)
(harmonics, three_phase_star)
(harmonics, three_phase_delta)
(harmonics, three_phase_harder)
(harmonics, phasor_diagrams)
(transients, three_phase_easier)
(transients, three_phase_star)
(transients, three_phase_delta)
(transients, three_phase_harder)
(transients, AC_values)
(transients, AC_power)
(transients, phasor_diagrams)
(transients, frequency_dependence)
```

Slika 5.14: Početna lista od 30 zabranjenih kombinacija značajki u Strategiji 4

Tablica 5.4: Izvođenje kombinatornog testiranja prema Strategiji 4

Izvođenje testa	1	2	3	4
Parametri testa	$t = 2$	$t = 2$ i 30 zabranjenih kombinacija	$t = 3$	$t = 3$ i 30 zabranjenih kombinacija
Broj testnih slučajeva	56	66	352	321
Potpuno pokriveni testni slučajevi	9 [16,07%]	14 [21,21%]	55 [15,63%]	76 [23,68%]
Djelomično pokriveni testni slučajevi	40 [71,43%]	55 [83,33%]	223 [63,35%]	262 [81,62%]
Potpuno pokrivena pitanja	142 [26,85%]	80 [16,91%]	246 [52,01%]	253 [53,49%]

U prvom izvođenju testiranja (pokrivanje svih parova značajki bez zabranjenih kombinacija značajki) generirano je 56 testnih slučajeva, a u drugom izvođenju (opet $t = 2$, ali sada s većom listom zabranjenih kombinacija značajki) generirano je 66 testnih slučajeva. Nadalje, kod pokrivanja svih trojki značajki iz različitih dimenzija automatski su generirano 352 testna slučaja u trećem izvođenju kombinatornog testiranja ($t = 3$ i bez zabranjenih kombinacija značajki) te 321 testni slučaj u četvrtom izvođenju ($t = 3$ i lista zabranjenih kombinacija).

Sada je značajno veći udio generiranih testnih slučajeva pokriven s postojećim pitanjima iz inicijalnog skupa od 473 pitanja, i to 16,07% u prvom izvođenju testiranja i 15,63% u trećem izvođenju testa. Nadalje, kada su se navele navedene zabranjene kombinacije značajki onda je potpuna pokrivenost testnih slučajeva porasla na 21,21% u drugom izvođenju testa sve do 23,68% u četvrtom izvođenju testa. Treba primijetiti kako je u prvom i drugom izvođenju testa gdje se tražila pokrivenost svih dozvoljenih parova značajki iz različitih dimenzija ostalo nepokriveno $56 - 9 = 47$ testnih slučajeva, odnosno $66 - 14 = 52$ testna slučaja. Ako ovi testni

slučajevi predstavljaju korisne opise pitanja onda ih nastavnici trebaju implementirati kao nova pitanja koja će se nakon sastavljanja dodati u inicijalni skup od 473 pitanja. U trećem i četvrtom izvođenju kombinatornog testiranja vidi se kako je apsolutni broj testnih slučajeva koji nisu potpuno pokriveni puno veći i redom je jednak $352-55=297$ i $321-76=245$. Ipak, može se odmah vidjeti kako su te brojke puno manje nego u rezultatima trećih i četvrtih izvođenja kombinatornog testiranja prema Strategijama 1 i 3. Stoga nastavnici mogu u ovom slučaju i dalje ručno detaljno provjeriti te nepokrivene testne slučajeve i eventualno prema njima sastaviti nova pitanja.

S obzirom na to da su korištene dimenzije značajno manje nego u prvoj ili trećoj strategiji i performanse kombinatornog testiranja su sada značajno bolje. Vrijeme izvođenja kombinatornog testiranja prema Strategiji 4 kreće se od 12,7 ms kod prvog izvođenja do 40,55 ms kod četvrtog izvođenja testa.

5.3.5 Rasprava

Iz predstavljenih rezultata inicijalnih izvođenja kombinatornog testiranja za sve četiri predložene strategije može se primijetiti kako broj generiranih testnih slučajeva značajno varira u ovisnosti o zadanoj snazi međusobnog pokrivanja, odnosno o odabiru veličine n-torki značajki koje treba pokriti s generiranim testnim slučajevima kao i o broju i veličini dimenzija. Naravno, kako bi se što bolje iskoristio postupak kombinatornog testiranja nastavnici i drugi domenski eksperti trebaju posebno pažljivo odabrati broj dimenzija te na koji način će odabrane značajke rasporediti po tim dimenzijama. Potom je posebno važno odrediti sve zabranjene kombinacije značajki iz različitih dimenzija što je težak zadatak, ali se upravo pomoću postupka kombinatornog testiranja mogu lakše identificirati iz generiranih testnih slučajeva. Treba naglasiti kako su sve kombinacije značajki iz iste dimenzije implicitno zabranjene jer se u svakom generiranom testnom slučaju pojavljuje samo po jedna značajka iz svake dimenzije. Npr. može se vidjeti da implicitno zabranjena kombinacija značajki (*DC_values*, *frequency_dependence*) iz četvrte dimenzije u Strategiji 3 na slici 5.13.

Analizom rezultata svih provedenih izvođenja kombinatornog testiranja utvrđeno je kako su veliki podskupovi pitanja iz inicijalnog skupa od 473 pitanja potpuno pokriveni s vrlo malim brojem generiranih testnih slučajeva. Primjerice, u trećem izvođenju kombinatornog testiranja prema Strategiji 4 ($t=3$ i bez zabranjenih kombinacija značajki) nađeno je u inicijalnom skupu 246 pitanja koja su pokrivena s barem jednim testnim slučajem. Pritom su dominantni testni slučajevi bili (*potential_electric*, *Ohm*, *sources*, *DC_values*) koji je pokrio 207/246 pitanja te testni slučaj (*potential_electric*, *Ohm*, *sources*, *AC_values*) koji je pokrio 139/246 pitanja. Dakako, ovo je bilo i očekivano jer je glavni fokus predmeta *Osnove elektrotehnike* na analizi krugova istosmjerne i izmjenične struje. S druge strane, može se vidjeti kako je samo jedno od 246 pitanja potpuno pokriveno s testnim slučajem (*inductance*, *Ohm*, *three_phase_delta*, *AC_values*).

Nastavnici bi zbog toga mogli razmotriti o potrebi stvaranja većeg broja o induktivnim trofaznim trošilima spojenim u trokut spoj, a u slučaju sustava *WebOE* prvo bi ih mogli tražiti u preostalom dijelu baze pitanja koja sadrži preko 3800 osnovnih varijanti različitih pitanja.

Nakon analize rezultata inicijalnih kombinatornih testiranja prema svim predloženim strategijama izabrana je Strategija 4 kao najprikladniji kandidat za demonstraciju čitavoga procesa kombinatornog testiranja koji će uspješno završiti tek kada su svi generirani testni slučajevi pokriveni s barem jednim pitanjem iz zadanog skupa. Kao što se može vidjeti iz rezultata, uz Strategiju 4 je postignut najviši postotak inicijalno potpuno pokrivenih testnih slučajeva i generiran je relativno manji broj nepokrivenih testnih slučajeva koji se mogu ručno detaljno provjeriti. Povrh toga, korištenjem pojednostavljenih dimenzija s grupama srodnih značajki sa slike 5.12 daje nastavnicima dodatnu fleksibilnost kod traženja prikladnih pitanja za nepokrivene testne slučajeve ili kod stvaranja novih pitanja na temelju takvih testnih slučajeva.

Isto tako treba napomenuti kako će se potpunim kombinatornim testiranjem prema Strategiji 4 zahtijevati pokrivanje svih dozvoljenih parova značajki iz različitih dimenzija, odnosno zadat će se veličina n-torki kao $t = 2$. To je jedna od uobičajenih vrijednosti snage međusobne pokrivenosti koja se koristi u praksi, a pokazat će se kako se tako dobije gotovo minimalan broj korisnih testnih slučajeva bez potrebe za predugačkim listama zabranjenih kombinacija značajki.

U svim provedenim inicijalnim kombinatornim testiranjima može se opaziti kako alat *Jenny* na zadovoljavajuće brz način generira testne slučajeve, čak i na prosječnim stolnim uredskim računalima. Ipak, bitno je primijetiti kako se za složenije konfiguracije s većim brojem dimenzija produžuje vrijeme potrebno za generiranje svih odgovarajućih testnih slučajeva. Primjerice, brzina izvođenja alata *Jenny* uz zadavanje $t = 3$ i korištenje Strategije 2 kojom se definira 50 dimenzija (svaka po dvije značajke) je ipak gotovo dvostruko sporija nego npr. kod Strategije 1 gdje su definirane samo četiri dimenzije (svaka od 8 do 17 značajki). Ipak, treba uzeti u obzir kako se objektivno može samo izmjeriti automatizirano izvođenje svake iteracije kombinatornog testiranja, ali se ne i trajanje detaljne analize generiranih testnih slučajeva kao i provođenje odluka koje će se potom donijeti, a to je ili zabrana kombinacije značajki ili implementacija testnog slučaja kao novog pitanja. Isto tako važno je naglasiti koliko može kombinatorno testiranje smanjiti broj testnih slučajeva uz zadovoljavanje uvjeta o pokrivanju svih n-torki značajki iz različitih dimenzija. Primjerice, da je sa Strategijom 2 odabrano pokrivanje svih četvorki značajki iz različitih dimenzija ($t = 4$) bez zadavanja zabranjenih kombinacija značajki generirala bi se 92 testna slučaja što je ogromno smanjenje u odnosu na ukupan broj svih testnih slučajeva (2^{50}). Ipak treba primijetiti kako bi u ovom slučaju generiranje ta 92 testna slučaja trajalo 12,5 s što je puno duže od oko 0,5 s koliko je potrebno za pokrivanje svih trojki značajki iz različitih dimenzija.

U budućem radu će se izvršiti detaljna usporedba svih rezultata dobivenih s alatom *Jenny* s

drugim etabliranim slobodnim alatima za kombinatorno testiranje. Isto tako je trenutno predložena metoda za kombinatorno testiranje implementirana samo kao program u naredbenom retku. Stoga se u budućem radu namjerava izgraditi web sučelje kojim bi se nastavnicima i drugim domenskim ekspertima olakšalo korištenje i primjene predložene metode za kombinatorno testiranje.

5.3.6 Potpuni postupak kombinatornog testiranja prema Strategiji 4

Za demonstraciju cjelokupnog postupka izvođenja predložene metode kombinatornog testiranja odabrana je Strategija 4 jer se s njom generira relativno mali broj testnih slučajeva, a time je olakšano i detektiranje i zadavanje zabranjenih kombinacija značajki iz različitih dimenzija. Kao što je već ranije opisano koriste se četiri dimenzije sa slike 5.13 pojednostavljene primjenom grupa srodnih značajki prikazanih na slici 5.12. Radi jednostavnosti i sažetosti prikaza rezultata zadano je pokrivanje svih dozvoljenih parova značajki iz različitih dimenzija, odnosno zadana je snaga međusobnog pokrivanja od $t = 2$. Isto tako je korištena inicijalna lista od 30 zabranjenih kombinacija značajki iz različitih dimenzija sa slike 5.14.

Tablici 5.5 sadrži sažetak cjelokupnog izvođenja predložene metode kombinatornog testiranja nad odabranim skupom pitanja koji inicijalno ima 473 elementa.

Tablica 5.5: Cjelokupni postupak kombinatornog testiranja sa Strategijom 4 ($t = 2$)

Izvođenje testa	1	2	3	4	5
Broj pitanja u bazi	473	473	473	473	522
Broj zabranjenih kombinacija	30	38	39	42	42
Broj generiranih testnih slučajeva	66	71	71	72	72
Potpuno pokriveni testni slučajevi	14 [21,21%]	16 [22,54%]	16 [22,54%]	23 [31,94%]	72 [100,0%]
Djelomično pokriveni testni slučajevi	55 [83,33%]	65 [91,55%]	65 [91,55%]	65 [90,28%]	72 [100,0%]
Potpuno pokrivena pitanja	80	56	70	90	139

U tablici 5.5 su u podatkovnim stupcima prikazani skupni rezultati svake od pet iteracija izvođenja kombinatornog testiranja. Iz prikazanih podataka mogu se pratiti promjene u broju generiranih testnih slučajeva te njihovoj pokrivenosti, promjene veličine podskupa pitanja koja

su potpuno pokrivena s jednim ili više testnih slučajeva, kao i rast broja zabranjenih kombinacija značajki iz različitih dimenzija te na kraju povećanje inicijalnog skupa pitanja.

Nakon svake iteracije kombinatornog testiranja se provjerio svaki generirani testni slučaj kako bi se detektirale eventualne nedozvoljene kombinacije značajki iz različitih dimenzija, prije svega one kontradiktorne ili ponekad one koje su presložene za gradivo predmeta *Osnove elektrotehnike*. Ako se takve neželjene kombinacije značajki pronađu onda se dodaju u listu zabranjenih značajki iz različitih dimenzija te se pokreće nova iteracija kombinatornog testiranja. Predzadnja iteracija kombinatornog testiranja je ona u kojoj su svi generirani testni slučajevi ili već pokriveni s pitanjima u inicijalnom skupu ili su nepokriveni, ali predstavljaju dozvoljene i zanimljive opise potencijalnih novih pitanja. Potom se prije zadnje iteracije kombinatornog testiranja u inicijalni skup dodaju nova pitanja koja implementiraju nepokrivene testne slučajeve. Isto tako se inicijalni formalni kontekst nadopunjuje s novim označenim pitanjima. Nakon toga slijedi zadnja iteracija kombinatornog testiranja koja će pokazati da je svaki generirani testni slučaj pokriven s barem jednim pitanjem i time postupak kombinatornog testiranja uspješno završava.

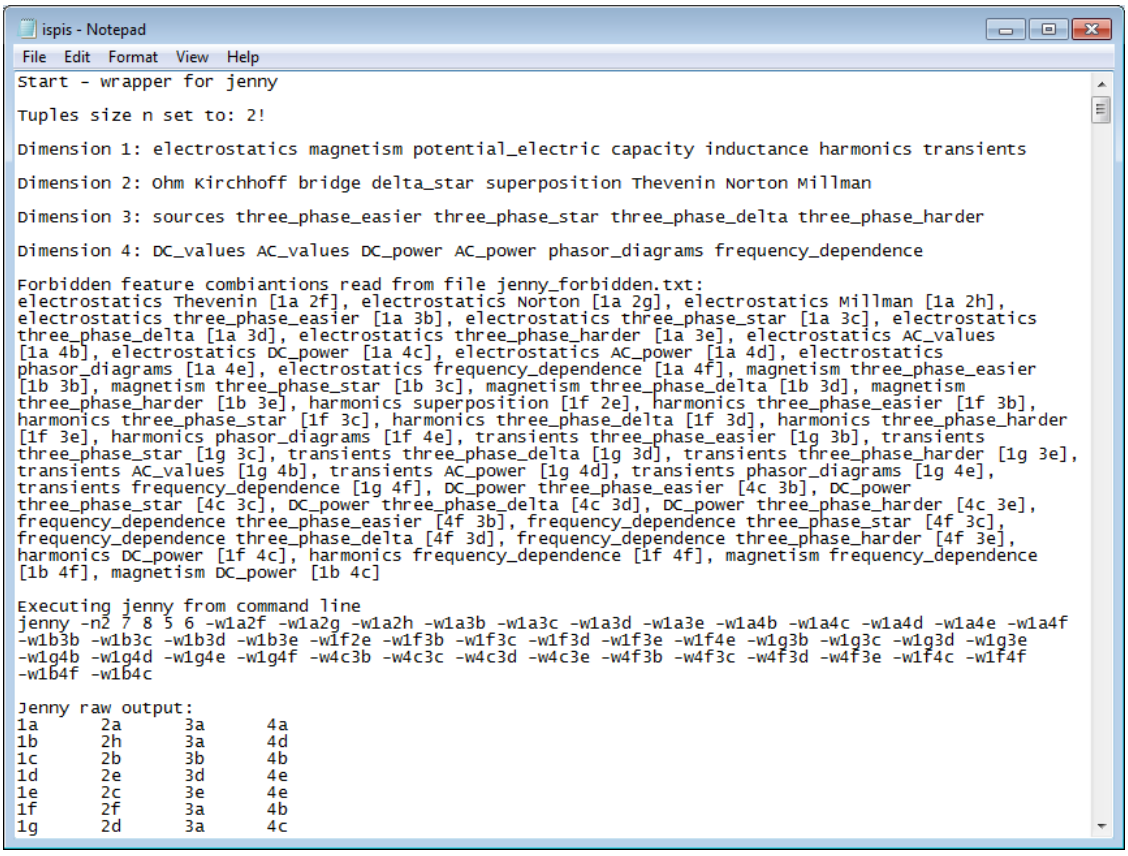
Treba primijetiti kako je ovdje prvo izvođenje kombinatornog testiranja istovjetno drugom inicijalno izvođenju postupka kombinatornog testiranja iz tablice 5.4 jer su korišteni isti ulazni parametri, veličina n-torki je zadana kao $t = 2$, a korištena je ista početna lista od 30 zabranjenih kombinacija značajki iz različitih dimenzija. U drugom i trećem retku tablice mogu se pratiti trenutne vrijednosti veličine skupa svih pitanja te broja zabranjenih kombinacija značajki iz različitih dimenzija. Pritom se može opaziti kako se broj kombinacija zabranjenih značajki sukcesivno povećavao nakon prve tri iteracije kombinatornog testiranja. Krenulo se od 30 zabranjenih značajki, a prije četvrte iteracije ukupno se došlo do 42 zabranjene značajke iz različitih dimenzija koliko ih je ostalo i u petoj iteraciji kombinatornog testiranja. Sve dodatne zabranjene kombinacije značajki koje su se dodavale prije druge, treće i četvrte iteracije kombinatornog testiranja mogu se vidjeti na slici 5.15. Opet treba naglasiti kako je kroz sam postupak kombinatornog testiranja nastavnicima i drugim domenskim ekspertima uvelike olakšana identifikacija neželjenih kombinacija značajki.

```
UKLJUČENI KOD 2. IZVOĐENJA TESTIRANJA:
(DC_power, three_phase_easier)
(DC_power, three_phase_star)
(DC_power, three_phase_delta)
(DC_power, three_phase_harder)
(frequency_dependence, three_phase_easier)
(frequency_dependence, three_phase_star)
(frequency_dependence, three_phase_delta)
(frequency_dependence, three_phase_harder)

UKLJUČEN KOD 3. IZVOĐENJA TESTIRANJA:
(harmonics, DC_power)

UKLJUČEN KOD 4. IZVOĐENJA TESTIRANJA:
(harmonics, frequency_dependence)
(magnetism, frequency_dependence)
(magnetism, DC_power)
```

Slika 5.15: Zabranjene kombinacije značajki dodane na početnu listu sa slike 5.14

Rastom broja zabranjenih kombinacija značajki iz različitih dimenzija povećavao se broj zadanih ograničenja za generirane testne slučajeve što je postupno rezultiralo s rastućim postotkom potpuno i/ili djelomično pokrivenih testnih slučajeva. Ipak, nakon uklanjanja svih neželjenih kombinacija značajki u četvrtoj iteraciji testiranja vidjelo se kako je samo 23 od ukupno 72 testna slučaja u potpunosti pokriveno s barem jednim pitanjem iz početnoga skupa. Iz tog razloga je bilo potrebno nakon četvrte iteracije kombinatornog testiranja osmisliti i sastaviti barem 49 novih pitanja koja će implementirati svaki od 49 interesantnih i korisnih trenutno nepokrivenih testnih slučajeva. Korištenjem *Python* skripte za upravljanje bazom podataka *database_manager.py* dodano je 49 novih označenih pitanja u inicijalni formalni kontekst, a time se povećao ukupni broj pitanja na 522. Potom se u petom izvođenju kombinatornog testiranja utvrdilo kako je svaki od 72 generirana testna slučaja pokriven s barem jednim od tih 522 pitanja. U tom trenutku postupak kombinatornog testiranja je završen, a glavna *Python* skripta *combinatorial_test_run.py* još generira detaljni statistički izvještaj kao i završni formalni kontekst u koji je uključeno samo 139 od ukupno 522 pitanja koja su pokrivena s barem jednim generiranim testnim slučajem.

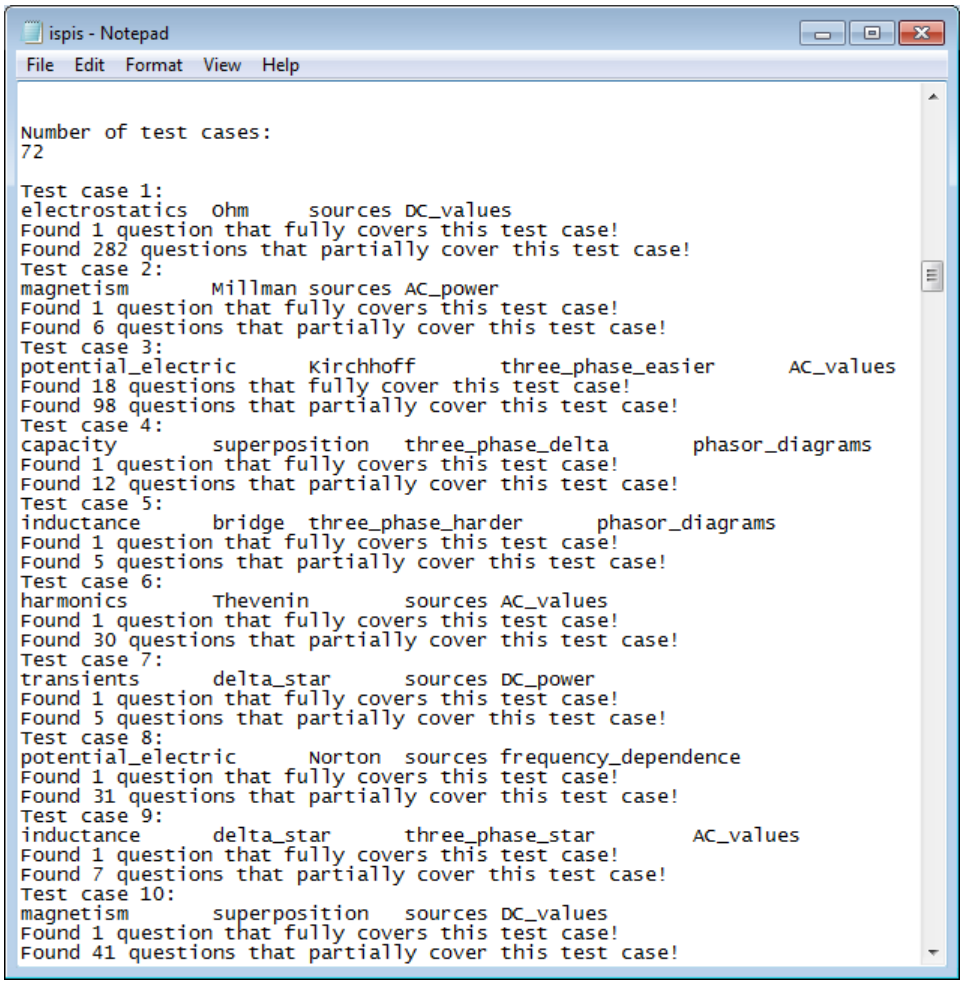
Prikazat će se nekoliko isječaka iz detaljnog ispisa koji se generira pri izvođenju pete iteracije kombinatornog testiranja sa skriptom skripta *combinatorial_test_run.py*. Na slici 5.16 može se vidjeti početak ispisa nakon učitavanja ulaznih podataka.



Slika 5.16: Isječak ispisa (početni dio) petog izvođenja skripte *combinatorial_test_run.py*

U tom dijelu ispisa prikazana je zadana veličina t s kojom će se izvoditi kombinatorno testiranje, sadržaj dimenzija kao i popis zabranjenih kombinacija značajki i za svaku od njih automatski generiran ulazni parametar za alat *Jenny*. Iz prikazanog isječka se vidi i odgovarajuća naredba za pokretanje alata *Jenny* te početak ispisivanja testnih slučajeva s implicitno imenovanim značajkama.

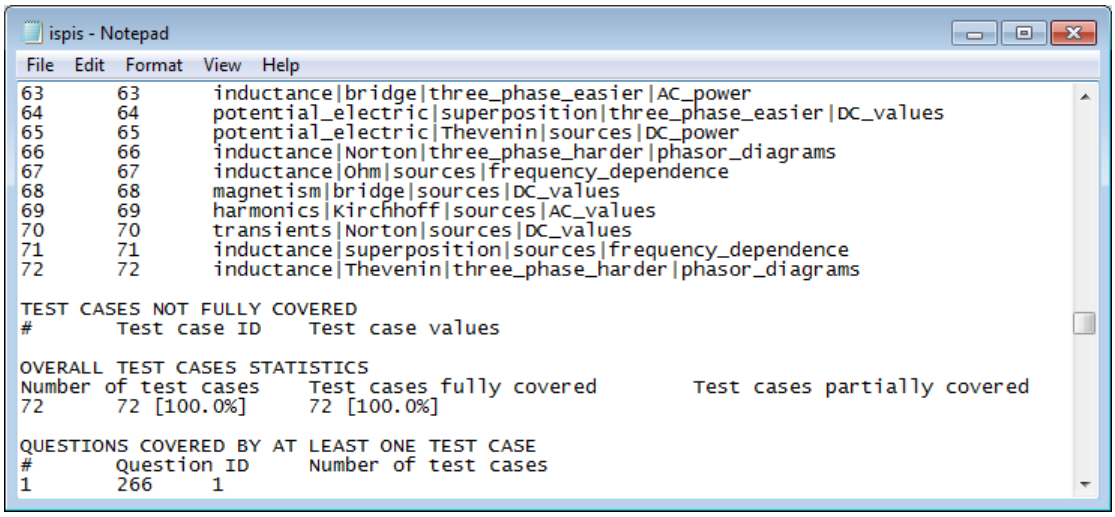
Slika 5.17 prikazuje dio ispisa pete iteracije kombinatornog testiranja gdje se redom provjerava svaki od 72 generirana testna slučaja. Za svaki od testnih slučajeva se prema predloženoj metodi kombinatornog testiranja ispituje je li u potpunosti pokriven s nekim od postojećih pitanja u bazi, a zatim se dodatno provjerava je li možda barem djelomično pokriven s nekim postojećim pitanjima. Kao što je ranije naglašeno u tom slučaju traži se poklapanje tri od četiri značajke testnog slučaja s atributima nekog pitanja.



Slika 5.17: Isječak ispisa (obrada testnih slučajeva) petog izvođenja skripte *combinatorial_test_run.py*

Još je prikazan na slici 5.18 isječak završnog dijela ispisa pete iteracije kombinatornog testiranja gdje se navode osnovni statistički podaci o generiranim testnim slučajevima. Vidi se kako nema testnih slučajeva koji nisu u potpunosti pokriveni s nekim od postojećih pitanja u bazi, što znači kako je s ovom iteracijom uspješno završen proces kombinatornog testiranja. Isto

tako vidi se početak ispisa samo onih pitanja iz baze koja su pokrivena barem s jednim testnim slučajem, a za svako takvo pitanje navodi se i broj različitih testnih slučajeva koji ga pokrivaju.



Slika 5.18: Isječak ispisa (statistički sažetak) petog izvođenja skripte *combinatorial_test_run.py*

Na idućim stranicama je prvo dan grafički pregled svih generiranih testnih slučajeva iz zadnjeg, petog izvođenja kombinatornog testiranja na slici 5.19, a potom su dani osnovni podaci za svaki od 72 generirana testa slučaja redom u tablicama 5.6, 5.7 i 5.8. U tablicama su dane i napomene o ranije nepokrivenim testnim slučajevima koji su u prije zadnjeg izvođenja kombinatornog testiranja implementirani kao nova pitanja. Isto tako naglašeno je koji testni slučajevi predstavljaju česte kombinacije značajki koje su u potpunosti pokrivene s većim brojem pitanja.

Iz grafičkog pregleda testnih slučajeva vidi se kako su generirani testni slučajevi često djelomično pokriveni s većim brojem pitanja, odnosno kako se dosta lako može pronaći preklapanje 3 od 4 značajke generiranog testnog slučaja s atributima pitanja iz završnog formalnog konteksta. S druge strane, odmah se uočava kako je vrlo mali broj testnih slučajeva u potpunosti pokriven s većim brojem pitanja, odnosno kako je većina testnih slučajeva potpuno pokrivena tek s jednim pitanjem iz završnog formalnog konteksta. Ovo može biti još jedna smjernica nastavnicima i drugim domenskim ekspertima kako bi u budućnosti bilo korisno stvoriti veći broj različitih pitanja za svaki takav generirani testni slučaj.

Među 139 pitanja iz završnog formalnog konteksta postoje ona koja potpuno pokrivaju više od jednog generiranog testnog slučaja, kao što su 24 pitanja potpuno pokrivena s dva testna slučaja, četiri pitanja koja su potpuno pokrivena s tri testna slučaja te pet pitanja koja pokrivaju čak četiri testna slučaja. Kao primjer navest će se opis složenijeg pitanja ID2940: (*potential_electric*, *resistance*, *voltage_source*, *ideal_source*, *Ohm*, *Kirchhoff*, *sine_sources*, *AC_current*, *AC_voltage*, *AC_power*, *vectors*, *phasors*, *impedance*, *inductance*, *capacitance*, *three_phase_phasor_diagram*, *three_phase_sources*, *three_phase_delta_unbalanced*), koje je pokriveno s četiri različita testna slučaja: ID3, ID11, ID18 i ID54 sa slike 5.19.

Slika 5.19: Pregled testnih slučajeva iz petog izvođenja testiranja iz tablice 5.5

Tablica 5.6: Detaljni pregled testnih slučajeva ID1 – ID24 iz petog izvođenja testiranja iz tablice 5.5

ID	Značajke testnog slučaja	Potpuno pokrivena pitanja	Djelomično pokrivena pitanja	Komentar
1	(*electrostatics*, *Ohm*, *sources*, *DC_values*)	1	282	
2	(*magnetism*, *Millman*, *sources*, *AC_power*)	1	6	1. novo pitanje
3	(*potential_electric*, *Kirchhoff*, *three_phase_easier*, *AC_values*)	18	98	*česta kombinacija*
4	(*capacitance*, *superposition*, *three_phase_delta*, *phasor_diagrams*)	1	12	2. novo pitanje
5	(*inductance*, *bridge*, *three_phase_harder*, *phasor_diagrams*)	1	5	3. novo pitanje
6	(*harmonics*, *Thevenin*, *sources*, *AC_values*)	1	30	4. novo pitanje
7	(*transients*, *delta_star*, *sources*, *DC_power*)	1	5	5. novo pitanje
8	(*potential_electric*, *Norton*, *sources*, *frequency_dependence*)	1	31	6. novo pitanje
9	(*inductance*, *delta_star*, *three_phase_star*, *AC_values*)	1	7	7. novo pitanje
10	(*magnetism*, *superposition*, *sources*, *DC_values*)	1	41	
11	(*capacitance*, *Ohm*, *three_phase_easier*, *AC_power*)	2	57	
12	(*electrostatics*, *Kirchhoff*, *sources*, *DC_values*)	1	149	8. novo pitanje
13	(*electrostatics*, *bridge*, *sources*, *DC_values*)	1	35	9. novo pitanje
14	(*magnetism*, *Thevenin*, *sources*, *phasor_diagrams*)	2	22	
15	(*magnetism*, *Norton*, *sources*, *AC_values*)	1	5	10. novo pitanje
16	(*potential_electric*, *Millman*, *three_phase_harder*, *DC_values*)	1	22	
17	(*capacitance*, *Thevenin*, *sources*, *frequency_dependence*)	4	55	
18	(*inductance*, *Ohm*, *three_phase_delta*, *AC_power*)	1	27	
19	(*inductance*, *Kirchhoff*, *sources*, *frequency_dependence*)	12	75	*česta kombinacija*
20	(*capacitance*, *bridge*, *sources*, *frequency_dependence*)	1	41	11. novo pitanje
21	(*electrostatics*, *delta_star*, *sources*, *DC_values*)	1	37	12. novo pitanje
22	(*potential_electric*, *Norton*, *three_phase_star*, *phasor_diagrams*)	1	39	13. novo pitanje
23	(*electrostatics*, *superposition*, *sources*, *DC_values*)	1	65	14. novo pitanje
24	(*inductance*, *Millman*, *sources*, *frequency_dependence*)	1	38	15. novo pitanje

Tablica 5.7: Detaljni pregled testnih slučajeva ID25 – ID48 iz petog izvođenja testiranja iz tablice 5.5

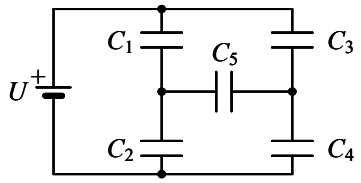
ID	Značajke testnog slučaja	Potpuno pokrivena pitanja	Djelomično pokrivena pitanja	Komentar
25	(*magnetism*, *Ohm*, *sources*, *phasor_diagrams*)	3	151	
26	(*capacitance*, *Kirchhoff*, *sources*, *DC_power*)	1	52	16. novo pitanje
27	(*potential_electric*, *bridge*, *three_phase_delta*, *AC_values*)	1	14	17. novo pitanje
28	(*potential_electric*, *delta_star*, *three_phase_harder*, *AC_power*)	1	26	18. novo pitanje
29	(*capacitance*, *superposition*, *three_phase_harder*, *AC_values*)	1	12	19. novo pitanje
30	(*potential_electric*, *Thevenin*, *three_phase_star*, *AC_power*)	1	43	20. novo pitanje
31	(*inductance*, *Norton*, *sources*, *DC_power*)	1	1	21. novo pitanje
32	(*capacitance*, *Millman*, *three_phase_star*, *AC_values*)	1	7	
33	(*capacitance*, *delta_star*, *three_phase_easier*, *phasor_diagrams*)	1	12	22. novo pitanje
34	(*capacitance*, *Norton*, *three_phase_easier*, *DC_values*)	1	7	23. novo pitanje
35	(*harmonics*, *Ohm*, *sources*, *AC_values*)	4	170	
36	(*transients*, *Ohm*, *sources*, *DC_power*)	1	32	24. novo pitanje
37	(*inductance*, *Kirchhoff*, *three_phase_harder*, *DC_values*)	1	58	25. novo pitanje
38	(*potential_electric*, *superposition*, *sources*, *DC_power*)	4	36	
39	(*harmonics*, *Norton*, *sources*, *AC_power*)	1	3	26. novo pitanje
40	(*transients*, *Thevenin*, *sources*, *DC_values*)	1	67	27. novo pitanje
41	(*capacitance*, *Kirchhoff*, *three_phase_star*, *DC_values*)	1	61	28. novo pitanje
42	(*potential_electric*, *bridge*, *three_phase_star*, *AC_power*)	1	29	29. novo pitanje
43	(*magnetism*, *Kirchhoff*, *sources*, *phasor_diagrams*)	3	65	
44	(*inductance*, *superposition*, *three_phase_star*, *AC_power*)	1	8	30. novo pitanje
45	(*inductance*, *Millman*, *three_phase_easier*, *phasor_diagrams*)	1	4	
46	(*potential_electric*, *Millman*, *three_phase_delta*, *DC_values*)	1	15	31. novo pitanje
47	(*harmonics*, *delta_star*, *sources*, *DC_values*)	1	11	32. novo pitanje
48	(*transients*, *Millman*, *sources*, *DC_power*)	1	5	33. novo pitanje

Tablica 5.8: Detaljni pregled testnih slučajeva ID49 – ID72 iz petog izvođenja testiranja iz tablice 5.5

ID	Značajke testnog slučaja	Potpuno pokrivena pitanja	Djelomično pokrivena pitanja	Komentar
49	(*transients*, *bridge*, *sources*, *DC_power*)	1	5	34. novo pitanje
50	(*inductance*, *Thevenin*, *three_phase_delta*, *AC_values*)	1	22	35. novo pitanje
51	(*potential_electric*, *Ohm*, *three_phase_harder*, *DC_values*)	18	229	*česta kombinacija zbog: resistance* $\in$ *DC_values*
52	(*transients*, *Kirchhoff*, *sources*, *DC_values*)	15	127	*česta kombinacija*
53	(*harmonics*, *Millman*, *sources*, *AC_power*)	1	5	36. novo pitanje
54	(*inductance*, *Kirchhoff*, *three_phase_delta*, *AC_power*)	1	16	
55	(*harmonics*, *bridge*, *sources*, *AC_power*)	1	3	37. novo pitanje
56	(*potential_electric*, *delta_star*, *three_phase_delta*, *phasor_diagrams*)	4	11	
57	(*transients*, *superposition*, *sources*, *DC_values*)	1	57	38. novo pitanje
58	(*capacitance*, *Thevenin*, *three_phase_easier*, *AC_values*)	1	30	39. novo pitanje
59	(*inductance*, *Norton*, *three_phase_delta*, *AC_values*)	1	3	40. novo pitanje
60	(*capacitance*, *delta_star*, *sources*, *frequency_dependence*)	1	40	41. novo pitanje
61	(*magnetism*, *delta_star*, *sources*, *AC_power*)	1	7	42. novo pitanje
62	(*inductance*, *Ohm*, *three_phase_star*, *AC_values*)	2	128	
63	(*inductance*, *bridge*, *three_phase_easier*, *AC_power*)	1	2	43. novo pitanje
64	(*potential_electric*, *superposition*, *three_phase_easier*, *DC_values*)	1	65	44. novo pitanje
65	(*potential_electric*, *Thevenin*, *sources*, *DC_power*)	4	46	
66	(*inductance*, *Norton*, *three_phase_harder*, *phasor_diagrams*)	1	3	45. novo pitanje
67	(*inductance*, *Ohm*, *sources*, *frequency_dependence*)	32	111	*česta kombinacija*
68	(*magnetism*, *bridge*, *sources*, *DC_values*)	1	11	46. novo pitanje
69	(*harmonics*, *Kirchhoff*, *sources*, *AC_values*)	1	71	47. novo pitanje
70	(*transients*, *Norton*, *sources*, *DC_values*)	1	28	48. novo pitanje
71	(*inductance*, *superposition*, *sources*, *frequency_dependence*)	3	44	
72	(*inductance*, *Thevenin*, *three_phase_harder*, *phasor_diagrams*)	1	21	49. novo pitanje

Na kraju će se kao primjer prikazati jedno od 49 novih pitanja na slici 5.20. Prikazan je tekst pitanja na engleskom jeziku, prateća shema kao i lista ponuđenih odgovora uz napomenu o točnom odgovoru. Isto tako je navedena i lista svih atributa s kojima je pitanje označeno. Ovo novo pitanje pokriva testni slučaj ID13 (*electrostatics*, *bridge*, *sources*, *DC_values*), a grupe srodnih značajki sa slike 5.12 pružile su fleksibilnost pri sastavljanju pitanja. Grupa značajki *electrostatics* pokrivena je s atributima *capacitors*, *charge* i *energy_electric*, grupa značajki *sources* pokrivena je s atributima *ideal_source* i *voltage_source*, a grupa značajki *DC _values* pokrivena je s atributom *DC_voltage*. Treba primijetiti da se pitanje može označiti i s dodatnim atributima, npr. s atributom *capacitance* i *delta_star*, ali će tako i dalje biti potpuno pokriveno s testnim slučajem ID13.

Pitanje ID5009 (*varijanta na engleskom jeziku*)
For a capacitor network shown in the figure calculate the electric energy stored in capacitor C_5 if U=12 V, C_1=8 μF, C_2=4 μF, C_3=6 μF, C_4=3 μF and C_5=5 μF. All capacitors are initially uncharged.



Ponuđeni odgovori:
A) 48 μJ B) 64 μJ C) 96 μJ D) 128 μJ E) 0 J
(*točan odgovor E*)

Oznake pitanja (atributi):
energy_electric, charge, capacitors, potential_electric, capacitance, voltage_source, ideal_source, DC_voltage, Kirchhoff, bridge, delta_star

Slika 5.20: Novo pitanje s višestrukim odabirom odgovora koje implementira testni slučaj ID13

Nakon što je uspješno završen postupak kombinatornog testiranja završni formalni kontekst sa 139 pitanja koja u potpunosti pokrivaju generirane testne slučajeve. Među tih 139 pitanja je 90 postojećih pitanja kao i 49 novih pitanja dodanih tijekom postupka kombinatornog testiranja. Vidi se kako je većina generiranih testnih slučajeva potpuno pokrivena s jednim do četiri pitanja, ali postoji i nekoliko testni slučajeva s čestim kombinacijama značajki koji su potpuno pokriveni s većim brojem pitanja. To su primjerice testni slučaj ID67 (*inductance*, *Ohm*, *sources*, *frequency_dependence*) koji je potpuno pokriven s 32 različita pitanja ili testni slučaj ID3 (*potential_electric*, *Kirchhoff*, *three_phase_easier*, *AC_values*) koji je potpuno pokriven s 18 različitih pitanja.

Prikazanim predloženim postupkom kombinatornog testiranja na kraju se iz većeg skupa od 522 pitanja izdvojio sažeti podskup od 139 pitanja koja pokrivaju sve dozvoljene parove

značajki iz različitih dimenzija. Tako su identificirana ona pitanja koja povezuju više različitih pojmova iz nastavnoga gradiva, a time bi mogla pomoći studentima u boljem razumijevanju cjelokupnog gradiva. U idućem poglavlju predstavit će se postupak kojim obraditi pronađeni sažeti skup pitanja s metodom FCA te topološkim sortiranjem kako bi se dobili prikladni nizovi pitanja za formativnu provjeru znanja u sustavima za e-učenje.

Poglavlje 6

Postupak za pripremu i vođenje provjere znanja

Nakon primjene predložene metode kombinatornog testiranja generirao se sažeti skup pitanja kojima se pokrivaju svi dozvoljeni parovi pojmova iz zadanog gradiva. U ovom poglavlju će se opisati daljnji postupak obrade tog generiranog skupa pitanja kako bi se iz njega pripremili prikladni nizovi pitanja za formativnu provjeru znanja u sustavima za e-učenje.

Kao što je već ranije naglašeno potrebno je na automatiziran način pripremiti listu pitanja koja će se formirati tako da se prvo u listu uključe najopćenitija pitanja koja sadrže manji broj atributa, a potom se u listu mogu postupno dodavati sve specifičnija pitanja koja su opisana s više atributa. Isto tako, osim toga važno je poštivati međusobnu hijerarhiju skupova atributa kojima su opisana pitanja kako bi se s redoslijedom zadavanja pitanja pratili međusobni odnosi među pojmovima iz gradiva. Primjerice, može se postaviti neko pitanje sa skupom atributa A samo ako su već ranije postavljena sva pitanja koja među svojim atributima sadrže podskupove od A.

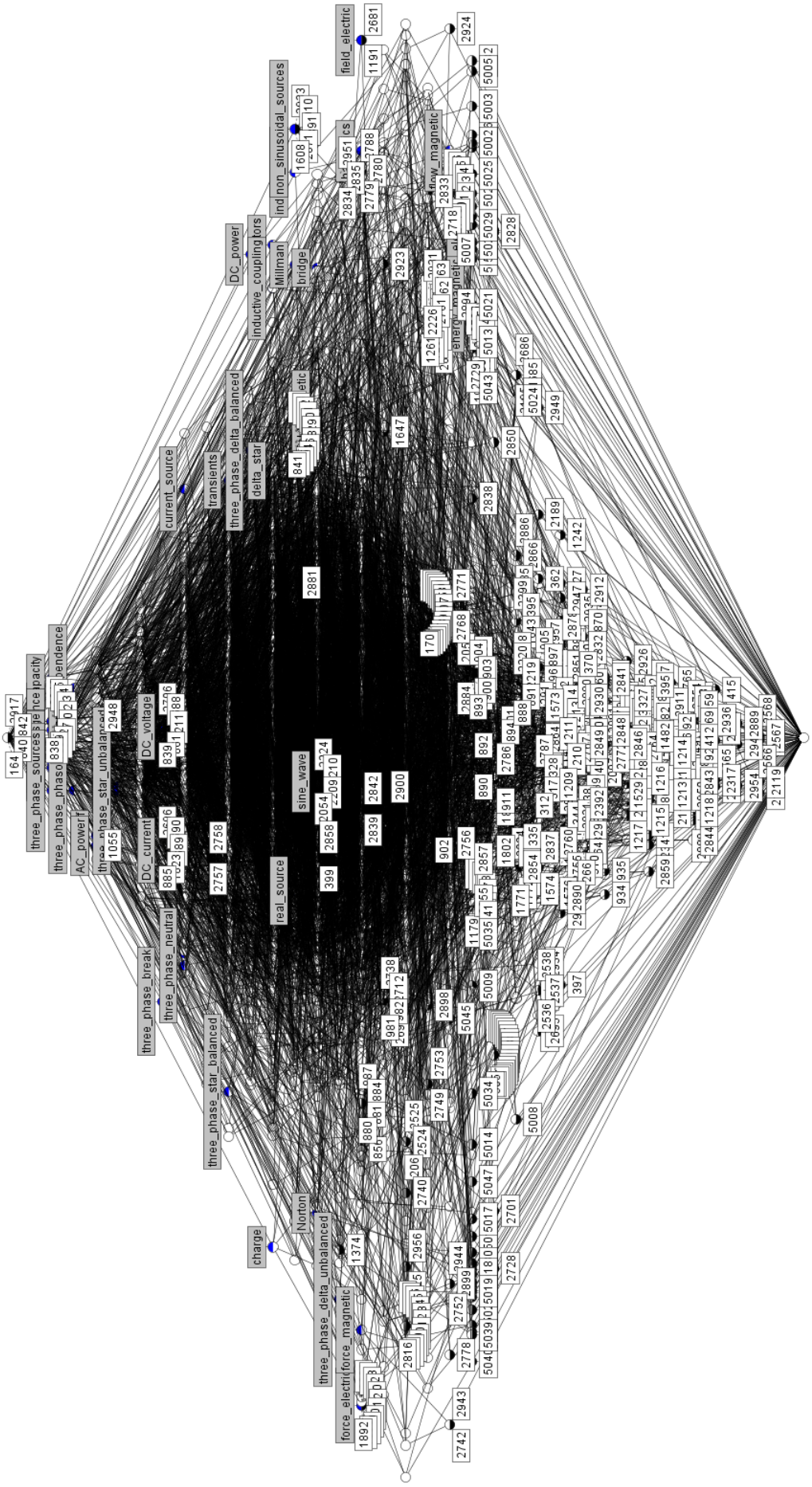
U ostvarivanju ovog cilja predlaže se korištenje formalne analize koncepata, metode strojnog učenja koja je detaljnije opisana u potpoglavlju 3.1. Treba podsjetiti kako se s metodom FCA može iz skupa objekata označenog s binarnim atributima, odnosno formalnog konteksta automatski pronaći skup svih formalnih koncepata. Svaki formalni koncept predstavlja par skupa objekata i skupa atributa u kojem svi objekti iz skupa objekata dijele sve atribute iz skupa atributa i obratno. Ovdje će se za formalni koncept koristiti i ranije uvedeni naziv *EKP* točka, odnosno elementarna točka znanja jer svaka *EKP* točka jednoznačno određuje skup pitanja i skup njihovih zajedničkih atributa. Nadalje, metoda FCA automatski gradi konceptualnu rešetku koja predstavlja ontologiju domene zadane ulaznim podacima, a pritom su u konceptualnoj rešetci sačuvani svi međusobni hijerarhijski odnosi između formalnih koncepata. Uz konceptualnu rešetku alati za izvođenje metode FCA mogu automatski generirati skupove implikacija atributa kao i asocijacijskih pravila među atributima u konceptualnoj rešetci.

Formalna analiza koncepata pruža pouzdan i matematički utemeljen postupak za analizu i vizualizaciju podataka pripremljenih u obliku formalnog konteksta. Kao što se vidjelo u prethodnom poglavlju upravo su ispitna pitanja vrlo pogodna za korištenje u formalnom kontekstu jer nastavnik ili drugi domenski ekspert može za svako pitanje odrediti ima li ono neki od definiranih atributa ili nema.

Konceptualna rešetka se može predstaviti kao parcijalno uređeni skup u obliku usmjerenog acikličkog grafa u kojem je svaki čvor jedan formalni kontekst, a međusobni odnosi *natkoncept-potkoncept* mogu se pratiti kroz grane jer je svaka usmjerena od natkoncepta prema potkonceptu. Važno je ponoviti da je konceptualna rešetka struktura potpune rešetka jer se za svaki podskup formalnih koncepata iz konceptualne rešetke može pronaći njihov natkoncept i njihov potkoncept. Stoga se na vrhu konceptualne rešetke nalazi najopćenitiji formalni koncept koji sadrži skup svih objekata i obično prazan skup atributa koje dijele baš svi objekti. S druge strane, na dnu konceptualne rešetke nalazi se najspecifičniji formalni koncept koji sadrži skup svih atributa kao i obično prazan skup objekata koji imaju sve atribute. Treba naglasiti kako veličina konceptualne rešetke može rasti i gotovo eksponencijalno s rastom formalnog konteksta. Posljedica toga su često velike i nepregledne konceptualne rešetke koje je teško vizualno analizirati, ali korištenjem alata za metodu FCA može se dobiti tekstualni zapis strukture konceptualne rešetke koji se potom može strojno obraditi. U ovom radu će se predložiti primjena metode topološkog sortiranja kako bi se iz ponekad vrlo složenih konceptualnih rešetki mogli dobiti potpuno uređeni skupovi formalnih koncepata, odnosno linearno sortirani niz formalnih koncepata u kojem su sačuvani svi međusobni odnosi *natkoncept-potkoncept*. Iz takvih potpuno uređenih skupova formalnih koncepata će se potom automatski generirati nizovi pitanja. Takvi nizovi pitanja će biti prikladni za formativnu provjeru znanja jer njihov redoslijed strogo prati utvrđen linearni poredak formalnih koncepata nakon topološkog sortiranja konceptualne rešetke. Ako pretpostavimo kako su pitanja iz općenitijih formalnih koncepata pri vrhu konceptualne rešetke lakša, a ona iz specifičnijih formalnih koncepata pri dnu konceptualne rešetke teža onda se može reći kako će pitanja biti poredana po težini, počevši od onih lakših.

Nakon pripreme nizova pitanja predstavit će se predloženi postupak za vođenje formativne provjere znanja u sustavima za e-učenje. Predloženi postupak je implementiran kao modul za formativnu provjeru znanja u vlastitom sustavu za e-učenje. Modul se sastoji od skripte za dinamičko generiranje web stranice kroz koju se rješava formativna provjera te svih potrebnih novih tablica u postojećoj bazi podataka sustava za e-učenje u koje će se spremati podaci o samim formativnim provjerama znanja kao i svi zapisi o rješavanju provjera.

Na slici 6.1 će se prikazati konceptualna rešetka skupa pitanja iz studijskog primjera nakon što je proširen s još 49 pitanja tijekom procesa kombinatornog testiranja. Tako se dobio skup od 522 pitanja označen s 50 atributa, a takav formalni kontekst je rezultirao s vrlo složenom i nepreglednom konceptualnom rešetkom.



Slika 6.1: Konceptualna rešetka za prošireni skup pitanja iz studijskog primjera (522 pitanja i 50 atributa)

Konceptualna rešetka sa slike 6.1 sastoji se od 3740 formalnih koncepata ili *EKP* točaka i 16010 usmjerenih grana, a generirana je korištenjem programa *Concept Explorer 1.3*, jednoga od etabliranih alata za metodu FCA. Ne treba naglašavati kako bi ručna priprema ovakvog formalnog konteksta od 522 objekta i 50 atributa kroz grafičko sučelje programa bila izuzetno vremenski zahtjevna i podložna greškama. Zato se iskoristio tekstualni format *.cxt* kojim se u alatu *Concept Explorer 1.3* može zadati formalni kontekst. Kako je objašnjeno u prethodnom poglavlju 5 prilikom kombinatornog testiranja automatski se generiraju inicijalni formalni kontekst kao i završni formalni kontekst kao tekstualne datoteke u zadanom *.cxt* formatu. Na slici 6.2 je mali uzorak datoteke inicijalnog formalnog konteksta (*initial_formal_context.cxt*) koja se sastoji od popisa 522 objekata ili pitanja, popisa od 50 atributa te samog zapisa formalnog konteksta u kojem svaki znak "." označava da objekt iz tog retka ima atribut iz tog stupca, a svaki znak "x" da objekt iz tog retka ima atribut iz tog stupca.

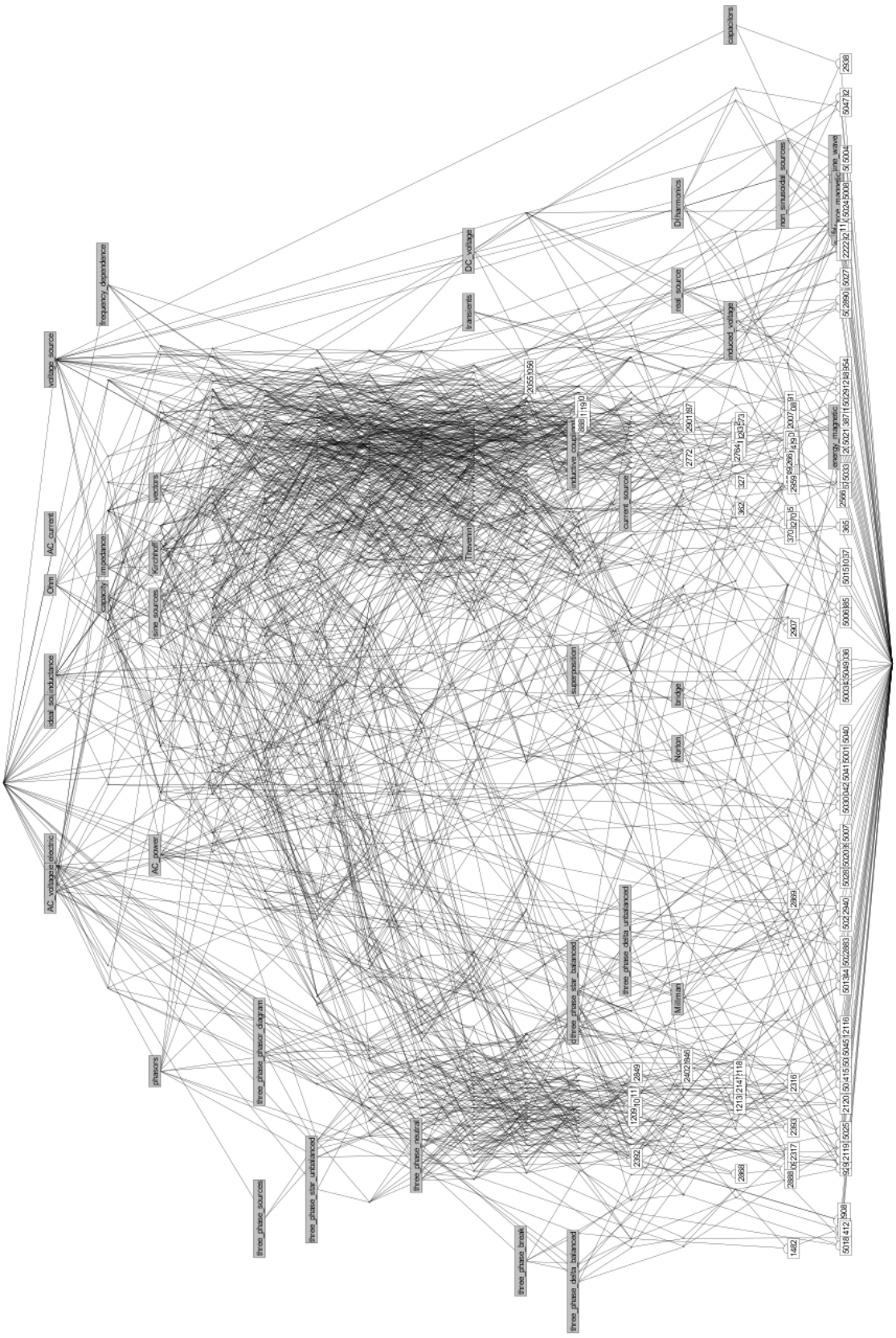
```
three_phase_delta_unbalanced
three_phase_neutral
three_phase_break
..................................................
..x....xxx..x.x.....x.............................
....xxx..x.........................................
..x....xxx..x.x.....xx..........x........x........
..x..x.x....x.x.....x.......xxx.x..x.xx...........
..x..x.x....x.x.....x.......xxx....xxx............
.....x.x....x.x.....x.......xxx.x..x.x............
..x....xxx..x.x.....xx..........x........x........
.....x.x....................x.x.x.x...x...........
```

Slika 6.2: Uzorak iz tekstualne datoteke inicijalnog formalnog konteksta od 522 pitanja i 50 atributa

Osim konceptualne rešetke alat *Concept Explorer 1.3* može generirati i listu asocijacijskih pravila za zadane mjere pouzdanosti i podrške, ali uz to može i odrediti bazu implikacija atributa po Duquenneu i Guiguesu koja u slučaju konceptualne rešetka sa slike 6.1 sadrži 752 osnovne implikacije atributa iz kojih se mogu izvesti sve ostale implikacije atributa kako je objašnjeno u potpoglavlju 3.1.

Ranije je u potpoglavlju 5.2 napomenuto kako se odabrani inicijalni skup od 473 pitanja označen s 50 atributa pretvara metodom FCA u konceptualnu rešetku s 3504 formalna konteksta ili EKP točke i 15284 usmjerene grane. Dakle, s povećanjem skupa pitanja za 49 na 522 na kraju se dobila konceptualna rešetka sa slike 6.1 koja ima 236 formalnih koncepata više i još 726 dodatnih usmjerenih grana. Odmah se može vidjeti kako ni 473, a pogotovo 522 pitanja nije optimalan broj za formativnu provjeru znanja. Zato je i glavni zadatak ovoga istraživanja bio u pronalaženju manjeg skupa prikladnih pitanja s kojima će studenti samotestirati svoje znanje.

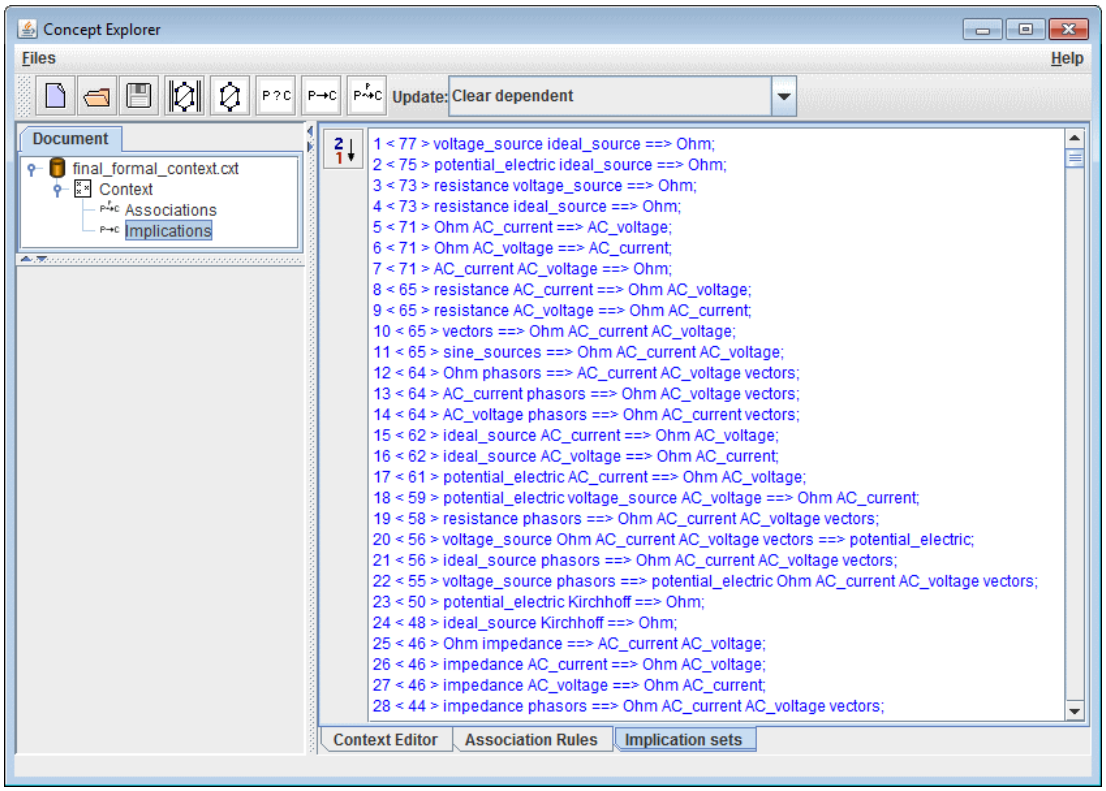
Kako se prikazano u prethodnom poglavlju s predloženom metodom kombinatornog testiranja skup pitanja se reducirao s 522 na 139, a da su pritom pokriveni svi dozvoljeni parovi atributa. Iz automatski generiranog završnog formalnog konteksta (*final_formal_context.cxt*) sa 139 pitanja i 50 atributa se metodom FCA izgradila konceptualna rešetka prikazana na slici 6.3.



Slika 6.3: Konceptualna rešetka za završni formalni kontekst iz studijskog primjera (169 pitanja i 50 atributa)

Vidi se kako je i konceptualna rešetka sa slike 6.3 vrlo složena, ali ipak je puno jednostavnija od one početne konceptualne rešetke sa slike 6.1. Opet se s alatom *Concept Explorer 1.3* dobiju točne dimenzije konceptualne rešetke sa slike 6.3, a to je 1037 formalnih koncepata ili *EKP* točaka te ukupno 3413 usmjerenih grana. Iz slike se može primijetiti kako je odabran reducirani prikaz atributa i objekata, tako da je svaki atribut naveden samo u najvišem formalnom konceptu u kojem se pojavljuje (pretpostavlja se da je sadržan i u svim njegovim potkonceptima), a svaki objekt ili pitanje je navedeno samo u najnižem formalnom konceptu u kojem se pojavljuje (pretpostavlja se da se nalazi i u svim njegovim natkonceptima). Vidjet će se kako se ovaj način prikaza može učinkovito iskoristiti za identificiranje odgovarajućih nizova pitanja nakon topološkog sortiranja *EKP* točaka u konceptualnoj rešetci.

Treba napomenuti kako je u ovom slučaju alat *Concept Explorer 1.3* generirao bazu implikacija atributa po Duquenneu i Guiguesu koja se sastoji od 421 relacije koje vrijede u čitavom završnom formalnom kontekstu (imaju pouzdanost od 100%). Na slici 6.4 je navedeno prvih 28 od ukupno 421 osnovne implikacije atributa.

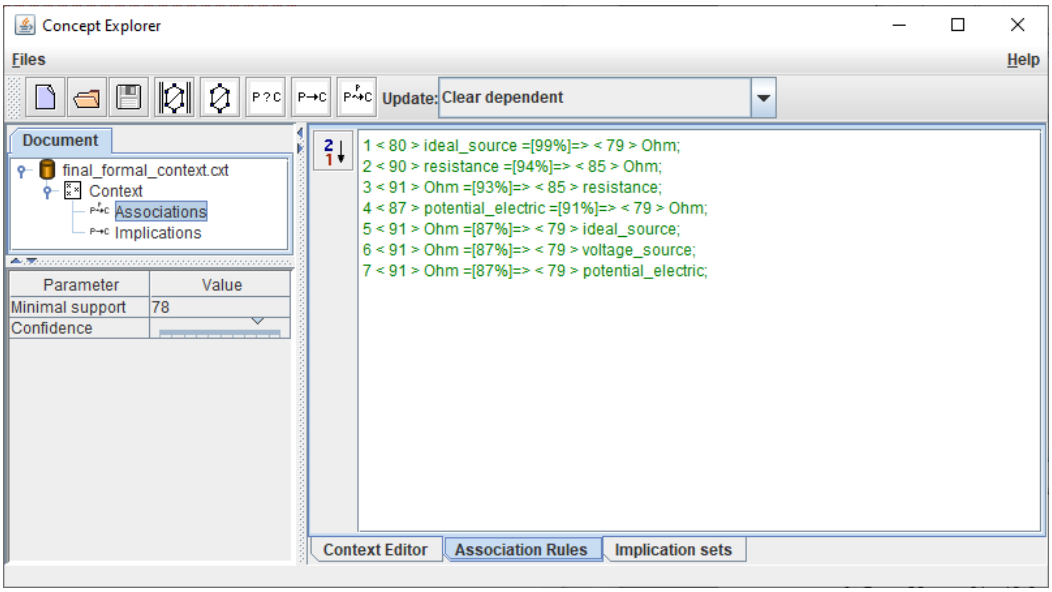


Slika 6.4: Dio baze implikacija atributa po Duquenneu i Guiguesu za konceptualnu rešetku na slici 6.3

Iz prikazane baze osnovnih implikacija atributa može se npr. razmotriti implikacija atributa navedena pod brojem 5: $\{Ohm, AC\_current\} \Rightarrow AC\_voltage$, tj. sva pitanja koje imaju atribute *Ohm* i *AC_current* imaju i atribut *AC_voltage*. Od svih 139 pitanja pronađeno je 71 pitanje koje među svojim atributima ima i trojku $\{Ohm, AC\_current, AC\_voltage\}$. Dakle, podrška ove implikacije atributa se računa kao $(71/139) \cdot 100\% \approx 51{,}1\%$. Može se primijetiti sa slike 6.4

kako su utvrđene implikacije atributa poredane padajući po vrijednosti podrške, a kako najveću podršku imaju pitanja koja pokrivaju krugove izmjenične i istosmjerne struje što je i očekivano jer je fokus predmeta *Osnove elektrotehnike* upravo na tom nastavnom gradivu. Od implikacija atributa s manjom podrškom može se spomenuti, primjerice, ona pod brojem 243: {*resistance*, *inductive_coupling*} $\Rightarrow$ {*potential_electric*, *Ohm*, *Kirchhoff*, *inductance*}. Njena podrška je puno manja ($\approx 2{,}9\%$) jer se u samo četiri pitanja pojavljuje unija svih atributa iz te implikacije atributa. Opet, može se reći kako je to očekivan rezultat s obzirom na to da je međuinduktivna veza jedna specifičnija tema i manja cjelina unutar cjelokupnog gradiva predmeta. Nastavnici i drugi domenski eksperti mogu analizom implikacija atributa dobro procijeniti koliko je uravnotežen odabrani skup pitanja i koliko je u skladu s nastavnim gradivom. U ovom slučaju je potvrđeno kako su glavne nastavne teme pokrivene s većim brojem pitanja i kako je broj pitanja o nekoj nastavnoj temi u skladu s udjelom te teme u ukupnom nastavnom gradivu.

Osim implikacija atributa alat *Concept Explorer 1.3* može generirati i asocijacijska pravila za zadane minimalne vrijednosti pouzdanosti i podrške. Primjerice, na slici 6.5 prikazan je popis asocijacijskih pravila za konceptualnu rešetku sa slike 6.3 uz zadane minimalne vrijednosti podrške od 56,12% (pravilo podržava barem 78 od 139 pitanja) te minimalne pouzdanosti od 80%.



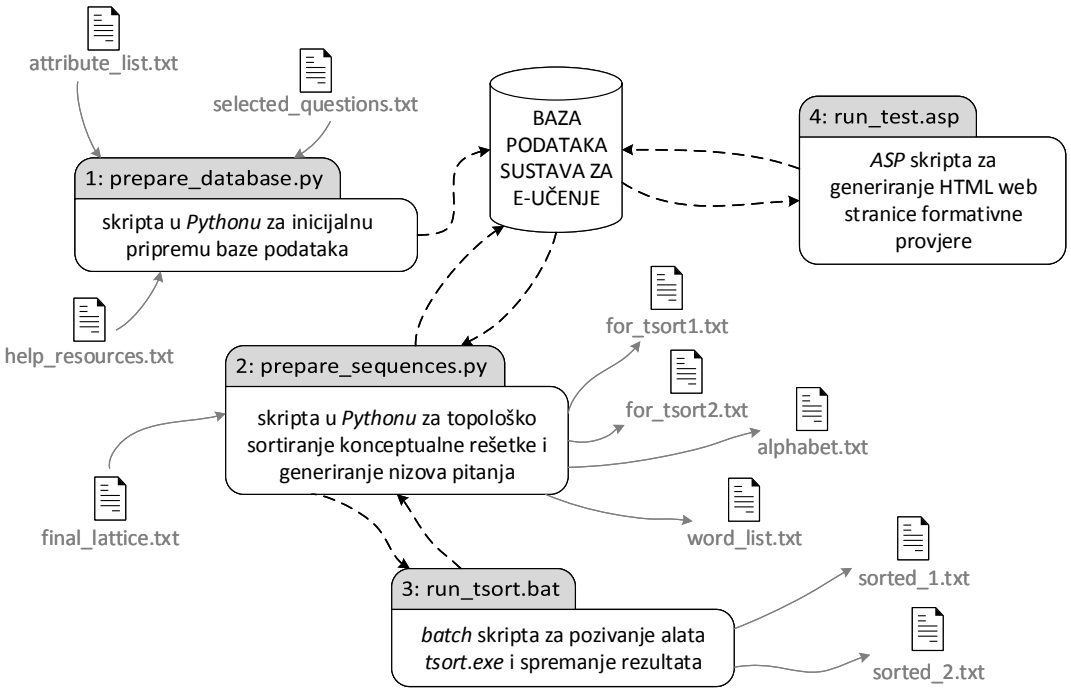
Slika 6.5: Asocijacijska pravila za konceptualnu rešetku sa slike 6.3 (min. podrška 56,12%, min. pouzdanost 80%)

Generirana asocijacijska pravila mogu opet pomoći nastavnicima i drugim domenskim ekspertima u procjeni kvalitete odabranog skupa pitanja. Npr. domenski ekspert može detaljnije provjeriti asocijacijsko pravilo pod brojem 2: *resistance* $\rightarrow$ *Ohm* (pitanje koje ima atribut *resistance* ima i atribut *Ohm*). Vidi se da je pouzdanost tog pravila 94%, odnosno od 90 pitanja koje imaju atribut *resistance* 85 ih ima i atribut *Ohm*. Pažljivijim pregledom preostalih 5 pitanja

koja su označena pojmom *otpora*, ali ne i *Ohmovog zakona* može se vidjeti kako se radi npr. o pitanjima vezanim o temperaturni koeficijent otpora ili električnu otpornost. Isto tako kod obrnutog asocijacijskog pravila pod brojem 3: *Ohm* $\rightarrow$ *resistance* pouzdanost opet nije 100% jer postoji 6 od 91 pitanja koja su označena s pojmom *Ohmovog zakona*, ali ne i pojmom *otpora*. U ovom slučaju radi se o pitanjima iz izmjeničnih krugova ili trofaznih sustava u kojima su trošila samo idealne zavojnice i/ili idealni kondenzatori. Vidi se kako asocijacijska pravila mogu dati vrijedni kontrolni mehanizam za provjeru kvalitete označavanja pitanja. Ako se pritom otkrije krivo označeno pitanje u bazi ta greška se treba ispraviti prije nastavka pripreme pitanja za formativnu provjeru znanja. Ako se nekom pitanju naknadno doda propušteni atribut onda se ne treba nužno ponavljati cijeli proces kombinatornog testiranja jer će i dalje svaki dozvoljeni par atributa biti pokriven s barem jednim pitanjem. S druge strane ako se nekom pitanju izbriše krivo dodijeljeni atribut preporučuje se ponoviti cijeli postupak kombinatornog testiranja kako bi bili sigurni da je svaki dozvoljeni par atributa pokriven s barem jednim pitanjem.

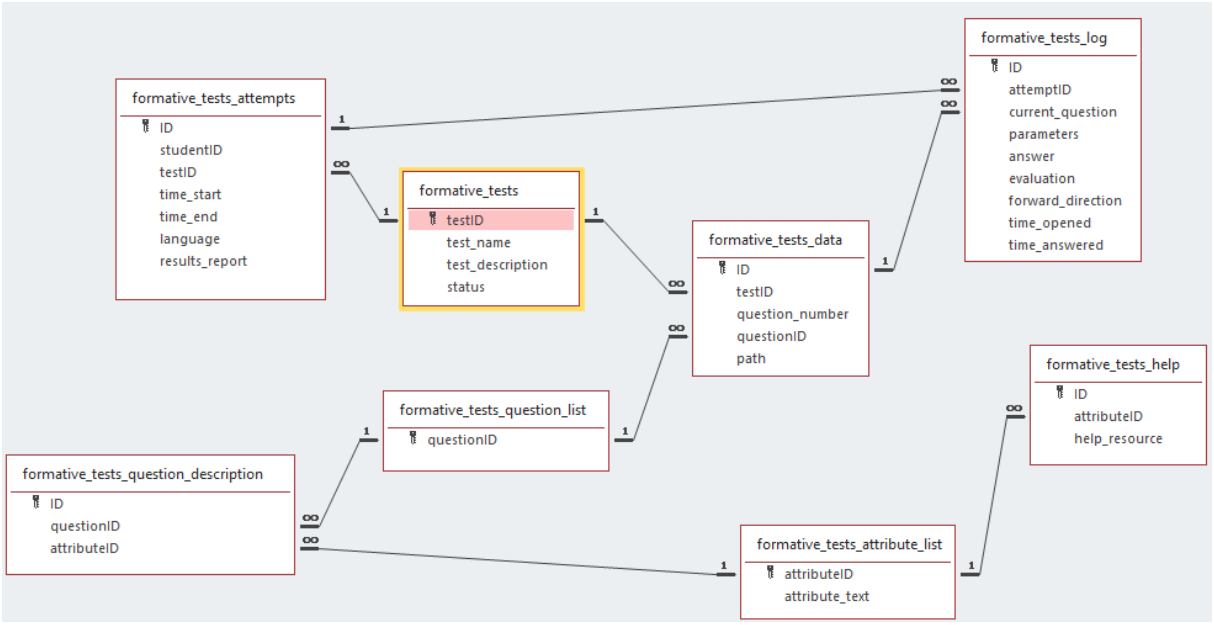
6.1 Modul za pripremu i vođenje formativne provjere znanja

Nakon što se skup odabranih pitanja provjeri može se pristupiti generiranju željenog redoslijeda odabranih pitanja kojim će se postavljati tijekom formativne provjere znanja. U vlastitom sustavu za e-učenje je dodan novi modul za pripremu i izvođenje formativne provjere znanja koji će automatizirati postupak generiranja odgovarajućih nizova pitanja kroz koje će se voditi formativna provjera znanja. Struktura implementiranog modula prikazana je na slici 6.6.



Slika 6.6: Pregled modula za pripremu i vođenje formativne provjere znanja

U modulu za pripremu i vođenje formativne provjere znanja sve četiri skripte prikazane na slici 6.6 direktno ili indirektno komuniciraju s glavnom bazom podataka u sustavu za e-učenje. Kako bi se spremili svi podaci vezani uz formativne provjere znanja bilo je potrebno osmisliti i integrirati nove tablice u postojeću *Microsoft Access* bazu podataka vlastitog sustava za e-učenje *WebOE*. Na slici 6.7 prikazan je model novoga dijela baze podataka u obliku dijagrama entiteta i veza preuzet iz programa *Access*.



Slika 6.7: Model novog dijela baze podataka za formativne provjere znanja

Može se vidjeti kako je u bazu podataka ugrađeno osam novih tablica u koje se čitaju i spremaju svi podaci potrebni za pripremu i izvođenje formativne provjere znanja. Od toga se šest tablica odnosi na pripremu podataka za formativnu provjeru znanja:

- tablica *formative_tests_question_list* sadrži samo primarni ključ *questionID* (cjelobrojni tip podataka) koji je jedinstvena identifikacijska oznaka pitanja. U tablicu će se upisati identifikacijske oznake svih pitanja iz završnog formalnog konteksta, a učitavaju se iz tekstualne datoteke *selected_questions.txt* koja se automatski generira u postupku kombinatornog testiranja.
- u tablicu *formative_tests_attribute_list* upisuju se svi atributi iz završnog formalnog konteksta, a pritom se svakom atributu redom zadaje identifikacijska oznaka. Dakle, tablica se sastoji od dva polja – primarnog ključa *attributeID* (cjelobrojni tip podataka) i tekstualnog polja *attribute_text* s nazivom atributa. U ovu tablicu se učitavaju podaci iz tekstualne datoteke *attribute_list.txt* koja se koristila i kao ulaz za kombinatorno testiranja.
- sljedeća tablica je *formative_tests_question_description* koja ima primarni ključ *ID* (automatski brojač) te još dva polja: *questionID* i *attributeID*, strani ključevi iz tablica *formative_tests_question_list* odnosno *formative_tests_attribute_list*. U ovoj tablici se za svako

pitanje navede lista identifikacijskih oznaka svih njegovih atributa, a podaci se opet učitavaju iz automatski generirane tekstualne datoteke *selected_questions.txt*. U svakom retku će biti naveden po jedan od atributa svakog pitanja, a naravno dopušteno je da jedno pitanje ima više atributa kao i da različita pitanja mogu imati isti(e) atribut(e).

- tablica *formative_tests_help* sadrži pripremljene podatke o opcijama za pomoć prilikom rješavanja formativne provjere znanja. Ova tablica ima tri polja, od toga je prva identifikacijska oznaka *ID* (automatski brojač), a potom polje *attributeID* (strani ključ iz tablice *formative_tests_attribute_list*) te tekstualno polje *help_resource* u koje se zapisuje opcija za pomoć, najčešće kao poveznica na odgovarajuće interaktivne nastavne materijale u sustavu *WebOE*. I ovdje se dozvoljava da isti atribut ima više opcija pomoći, ali i da jednu opciju pomoći mogu imati različiti atributi.
- u tablici *formative_tests* zapisuju se osnovni podaci o formativnim provjerama znanja. Ova tablica ima četiri polja – primarni ključ *testID* (cjelobrojni tip podataka), tekstualna polja *test_name* s imenom formativne provjere znanja i *test_description* s detaljnijim opisom. Četvrto polje je *status* (*Boolean* tip podataka) u kojem se zapisuje trenutni status formativne provjere – ako je postavljen na `True` onda je ta formativna provjera dostupna za rješavanje.
- šesta tablica, *formative_tests_data* sadrži podatke o pitanjima u svakoj formativnoj provjeri znanja. Ova tablica ima pet polja – primarni ključ *ID* (automatski brojač), identifikacijska oznaka testa *testID* (strani ključ iz tablice *formative_tests*), redni broj pitanja u nizu (cjelobrojni tip podataka) koji označava pripadnu *EKP* točku i identifikacijska oznaka pitanja *questionID* (strani ključ iz tablice *formative_tests_question_list*). Zadnje, četvrto polje je *path* (cjelobrojni tip podataka) u kojem se zapisuje identifikacijska oznaka puta kroz pitanja. Primjerice neka odabrana formativna provjera znanja može imati više od jednoga definiranog puta rješavanja, odnosno redoslijeda i izbora pitanja.

S druge strane, u preostale dvije tablice *formative_tests_attempts* i *formative_tests_log* se spremaju podaci o izvođenju formativne provjere znanja:

- u tablicu *formative_tests_attempts* se zapisuju osnovni podaci o svakom pokušaju rješavanja formativne provjere znanja. Tablica ima sedam polja i to primarni ključ *ID* (automatski brojač), tekstualno polje *studentID* – identifikacijska oznaka studenta (JMBAG) u sustavu *WebOE*, datumska polja *time_start* i *time_end* za početak i kraj rješavanja formativne provjere, tekstualno polje *language* za odabrani jezik prikaza provjere te tekstualno polje *results_report* u koje se sprema generirani izvještaj o rezultatima nakon završetka provjere znanja.
- i zadnja tablica, *formative_tests_log* sadrži detaljne dnevničke zapise o svakom pitanju na koje je student odgovarao. Primarni ključ je *ID* (automatski brojač), polje *attemptID* označava identifikacijsku oznaku pokušaja rješavanja (strani ključ *ID* iz tablice *forma-*

tive_test_attempts), polje *current_question* je identifikacijska oznaka trenutnog pitanja (strani ključ *ID* iz tablice *formative_tests_data*), u tekstualno polje *parameters* se spremaju slučajno generirani podaci za dinamički prikaz pitanja, a tekstualno polje *answer* sadrži odgovor studenta na pitanja. Nadalje, u polje *evaluation* (*Boolean* tip podataka) zapisuje se ocjena studentovog odgovora, a u preostala dva datumska polja *time_opened* i *time_answered* sadrže podatke o trenutku otvaranja pitanja i trenutku slanja odgovora na ocjenu. U preostalo polje *forward_direction* (*Boolean* tip podataka) upisuje se trenutni smjer rješavanja provjere: True (idi na iduće pitanje u nizu – uobičajena vrijednost), False (vrati se na prethodno pitanje u nizu).

Nakon pojašnjenja novih tablica u bazi podataka opisat će se i funkcionalnost skripti u modulu za pripremu i vođenje formativne provjere znanja sa slike 6.6. S prve tri skripte *prepare_database.py*, *prepare_sequences.py* i pomoćna *run_tsort.bat* pripremaju se ulazni podaci i automatski generiraju i spremaju formativne provjere znanja, a s četvrtom skriptom *test_run.asp* se izvodi sama formativna provjera znanja.

Priprema baze podataka

Inicijalna *Python* skripta *prepare_database.py* poziva se nakon provedenog kombinatornog testiranja samo jednom kako bi se u bazu podataka spremili svi podaci o definiranim atributima i odabranom sažetom skupu pitanja te o njihovim opisima. Glavni ulazni podaci za ovu skriptu su datoteke korištene u postupku kombinatornog testiranja: tekstualna datoteka *attributes_list.txt* s popisom svih definiranih atributa te automatski generirana tekstualna datoteka *selected_questions.txt* koja sadrži opise svih pitanja iz završnog formalnog konteksta. Nadalje, nastavnici i drugi domenski eksperti mogu u dodatnu ulaznu tekstualnu datoteku *help_resources.txt* zapisati opcije pomoći za svaki definirani atribut. U svaki redak te datoteke se prvo upisuje atribut, a onda nakon znaka tabulatora opcija pomoći kao kraći blok teksta ili češće kao poveznica na interaktivne nastavne materijale u sustavu *WebOE*. Za isti atribut se može zadati i više opcija za pomoć, samo je onda potrebno za svaku opciju dodati novi redak u datoteku *help_resources.txt*. U skripti se redom pozivaju tri funkcije koje vrše upis podataka iz ulaznih datoteka u tablice u bazi podataka:

- funkcija *store_attributes_to_database()* otvara datoteku *attributes_list.txt* i redak po redak zapisuje atribute u tablicu *formative_tests_attribute_list* s pridijeljenom identifikacijskom oznakom izvršavanjem dinamički pripremljene `INSERT` naredbe u SQL-u.
- funkcija *store_questions_to_database()* otvara datoteku *selected_questions.txt* i za svaki redak zapisuje identifikacijske oznake pitanja u tablicu *formative_tests_question_list*, a potom se automatski grade i redom izvršavaju `INSERT INTO SELECT` naredbe u SQL-u kojima se zapisuju atribute pitanja u tablicu *formative_tests_question_description*.

- na kraju funkcija *store_help_resources_to_database()* otvara datoteku *help_resources.txt* i automatski generiraju potrebne `INSERT INTO SELECT` naredbe u SQL-u kojima se zapisuju opcije pomoći za svako pitanje u tablicu *formative_tests_help*.

Generiranje nizova pitanja

Potom se poziva središnja *Python* skripta *prepare_sequences.py* koja pokreće topološko sortiranje konceptualne rešetke i tako automatski pronalazi odgovarajuće redoslijede pitanja te ih sprema u bazu kao nove formativne provjere znanja. U nastavku će se prikazati algoritam glavne funkcije *topological_sort()* u skripti koja vodi proces topološkog sortiranja konceptualne rešetke.

Algoritam 4 Algoritam funkcije *topological_sort()*

Učitaj argument funkcije *variant* (označava iteraciju topološkog sortiranja)
Inicijaliziraj praznu listu opcija pitanja: $sequences\_options \leftarrow []$

Učitaj tekstualni zapis konceptualne rešetke *final_lattice.txt* u listu redaka $lines$
Na slučajan način permutirati elemente liste $lines$
Za svaki $element$ u listi $lines$:
  Ako $element$ počinje s "Edge: "
    obradi i dodaj $element$ u varijablu $tsort\_input$
  Ako $element$ počinje s "Object: "
    obradi i dodaj $element$ u listu $objects$

Spremi $tsort\_input$ kao datoteku pod nazivom "for_tsort_" + *variant* + ".txt"
Pozovi skriptu *run_tsort.bat* koja pokreće topološko sortiranje

Učitaj linearni poredak formalnih koncepata iz "sorted_" + *variant* + ".txt" u listu $lines$
Obrni redoslijed elemenata liste $lines$
Za svaki $element$ iz liste $lines$:
  pronađi pitanja iz liste $objects$ i dodaj ih u kao listu u listu $sequences\_options$
Slučajnim odabirom pronađi tri elementa iz liste $sequences\_options$ s dva ili više pitanja:
  na slučajan način u svakom od tih elemenata ostavi točno dva pitanja
U svim ostalim elementima iz $sequences\_options$ na slučajan način ostavi samo jedno pitanje

Napravi Kartezijev produkt svih elemenata u listi $sequences\_options$ i dodaj ih u globalnu listu $complete\_sequences$
Za svaki $element$ u listi $complete\_sequences$:
  proširi globalnu listu $alphabet$ s novim identifikacijskim oznakama pitanja iz liste $element$

Za svaki $element$ u listi $complete\_sequences$:
  Dok je $length(element) > 0$:
    dodaj $element$ u globalnu listu $partial\_sequences$ ako ga u njoj nema
    makni zadnju identifikacijsku oznaku pitanja iz liste $element$

Glavni ulazni podatak za skriptu *prepare_sequences.py* je tekstualna datoteka *final_lattice.txt* koja sadrži zapis strukture konceptualne rešetke stvorene iz završnog formalnog konteksta. Ovu datoteku je potrebno spremiti iz alata *Concept Explorer 1.3* nakon učitavanja završnog formalnog konteksta i generiranja njegove konceptualne rešetke. U toj datoteci se nalaze grafički podaci o položaju svakog formalnog koncepta ili *EKP* točke, lista svih usmjerenih grana između formalnih koncepata te popis objekata i atributa po formalnim konceptima. Ovdje se automatski izdvajaju podaci o usmjerenim granama te o popisu objekata po formalnim konceptima dok se ostatak datoteke *final_lattice.txt* zanemaruje. Treba ponoviti kako se za parcijalno uređeni graf kao što je konceptualna rešetka može dobiti više ispravnih topoloških sortiranja formalnih koncepata (primjer na slici 3.20). To je moguće jer se neki formalni koncepti koji sadrže potpuno različite atribute ne mogu postaviti u međusobni odnos *natkoncept-potkoncept*, a najčešće se takvi formalni koncepti prikazuju na istoj razini konceptualne rešetke. Sa svakim pokretanjem ove skripte tražit će se dva potencijalno različita ispravna topološka sortiranja iste konceptualne rešetke. Iz ta dva topološka sortiranja će se generirati različiti putevi rješavanja iste formativne provjere znanja. A sa svakim idućim pokretanjem skripte *prepare_sequences.py* može se generirati nova formativna provjera znanja s potencijalno drukčijim putevima rješavanja.

Kako bi se proveo postupak topološkog sortiranja pomoću *batch* skripte *run_tsort.bat* pokreće se standardni *Unix* program naredbenog retka *tsort*. Program *tsort* je dostupan i za sustave *Windows* u sklopu paketa *GnuWin* [110]. Može se lako provjeriti da će program *tsort* za datoteku *final_lattice.txt* generirati samo jedno topološko sortiranje. Kako bi eventualno dobili drukčije topološko sortiranje treba na slučajan način permutirati retke u datoteci *final_lattice.txt*. Ovdje će se zato kod svakog pokretanja skripte *prepare_sequences.py* generirati po dva potencijalno drukčija topološka sortiranja konceptualne rešetke iz završnog formalnog konteksta.

Važno je napomenuti kako se zbog zadanog formata tekstualnog zapisa konceptualne rešetke generirane s alatom *Concept Explorer 1.3* nakon provedenog topološkog sortiranja treba obrnuti generirani linearni poredak formalnih koncepata. Tek će se tako dobiti traženi redoslijed od najopćenitijih do najspecifičnijih formalnih koncepata ili *EKP* točaka. Nakon toga će se prolaskom kroz potpuno uređeni skup *EKP* točaka izdvojiti samo one u kojima su zapisana pitanja. Treba podsjetiti kako se zbog reduciranog zapisa atributa i objekata u programu *Concept Explorer 1.3* svako pitanje eksplicitno pojavljuje samo u najspecifičnijem formalnom konceptu, a implicitno se nalazi i u svim njegovim natkonceptima. Naravno, u nekim *EKP* točkama može se nalaziti i više pitanja s istim atributima. To mogu biti vrlo slična pitanja ili više varijanti istog pitanja, ali ponekad su to sasvim različita pitanja koja zbog ograničenog broja definiranih atributa imaju isti opis. Zatim se na slučajan način odabire najviše po dva pitanja kao predstavnike svakog formalnog koncepta ili *EKP* točke u kojima se eksplicitno pojavljuju pitanja. Time će se smanjiti ukupni broj pitanja u formativnoj provjeri znanja, ali će ona i dalje pokrivati sve dozvoljene parove atributa što je bilo osigurano kombinatornim testiranjem.

Na kraju će se korištenjem *Pythonove* funkcije `itertools.product()` izračunati Kartezijev produkt svih odabranih skupova pitanja iz pojedinih *EKP* točaka. Tako se dobije uređena n-torka gdje je n broj *EKP* točaka, a svaki element te n-torke je redom neki od identifikacijskih oznaka pitanja iz svake *EKP* točke. Svaka takva uređena n-torka predstavlja jedan mogući redoslijed pitanja u formativnoj provjeri znanja, odnosno jedan mogući put rješavanja te provjere. Treba primijetiti kako se i s uvedenim ograničenjem od najviše dva pitanja po formalnom konceptu ovako može generirati ogroman broj različitih redoslijeda pitanja. Primjerice, ako postoji 100 *EKP* točaka, a svaka ima po dva pitanja onda će se generirati 2^{100} različitih redoslijeda pitanja. Zbog toga je uvedeno dodatno ograničenje kojim se najviše iz tri slučajno odabrane *EKP* točke uzima najviše po dva slučajno odabrana pitanja, a iz svih ostalih će se slučajno odabrati najviše po jedno pitanje. Ovako će se generirati maksimalno 8 različitih redoslijeda pitanja za svaki linearni poredak formalnih koncepata ili *EKP* točaka.

Nakon generiranja redoslijeda pitanja u skripti *prepare_sequences.py* se pozivaju funkcije za stvaranje dvije tekstualne datoteke: *alphabet.txt* s popisom svih pitanja iz generirane provjere te *word_list.txt* s popisom svih potpunih i djelomičnih nizova pitanja (svih prefiksa potpunih nizova pitanja) zajedno s praznom riječju ε. Ove dvije izlazne datoteke koristit će se kasnije za automatsku izgradnju modela formativne provjere znanja koje će se potom moći verificirati i simulirati s alatom za provjeru modela *Spin*.

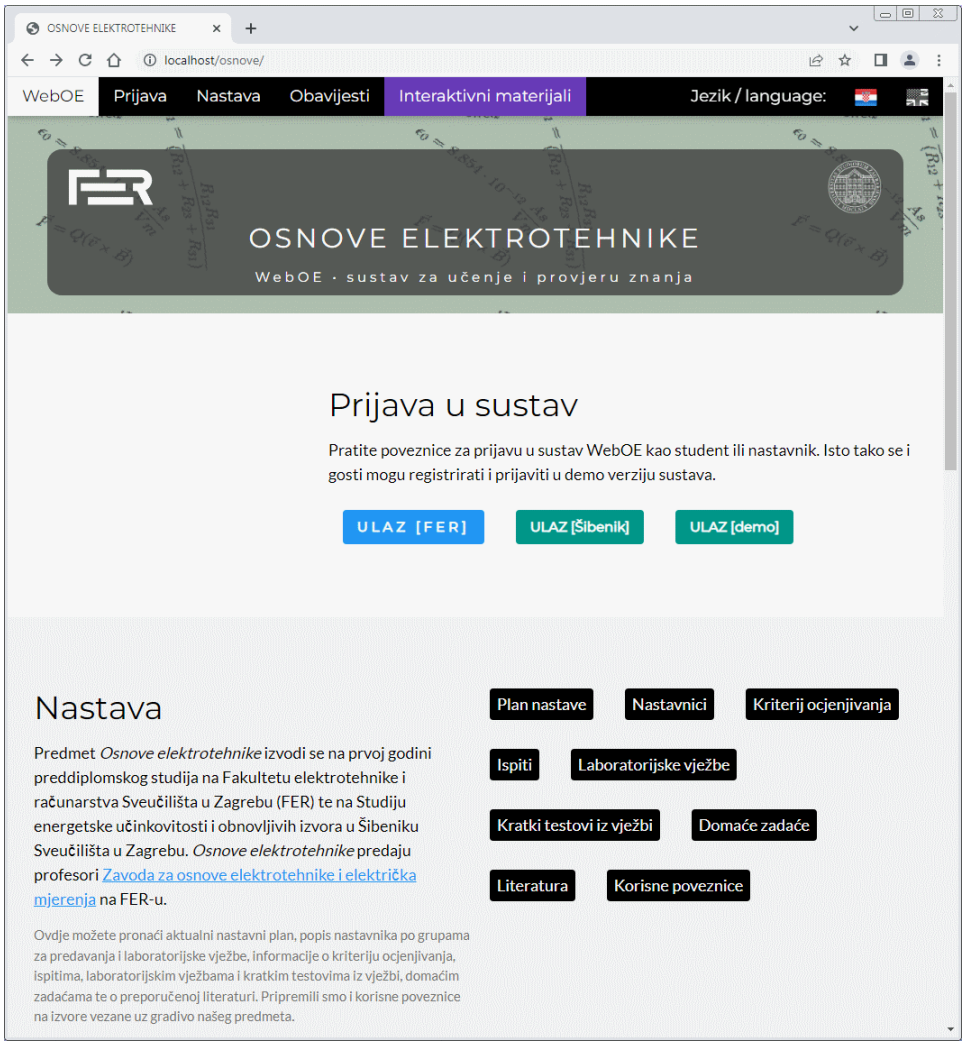
A na kraju skripte *prepare_sequences.py* se korisniku ispisuju osnovni podaci o generiranim redoslijedima pitanja (broj različitih djelomičnih i potpunih redoslijeda te broj korištenih pitanja). Korisniku se potom nudi mogućnost spremanja generiranih redoslijeda pitanja kao nove formativne provjere znanja. Ako korisnik odgovori potvrdno onda treba unijeti kratko ime provjere znanja kao i opis provjere znanja. Potom se stvara nova formativna provjera znanja u tablici *formative_tests*, a potom se svi njeni generirani redoslijedi pitanja upisuju u tablicu *formative_tests_data* tako da se za svaki redni broj pitanja iz jednog topološkog sortiranja konceptualne rešetke upišu moguće identifikacijske oznake pitanja. Treba spomenuti kako je za komunikaciju s *Access* bazom podataka te za izvođenje i spremanje rezultata upita korišten *Pythonov* modul *pyodbc* [111].

Vođenje formativne provjere znanja

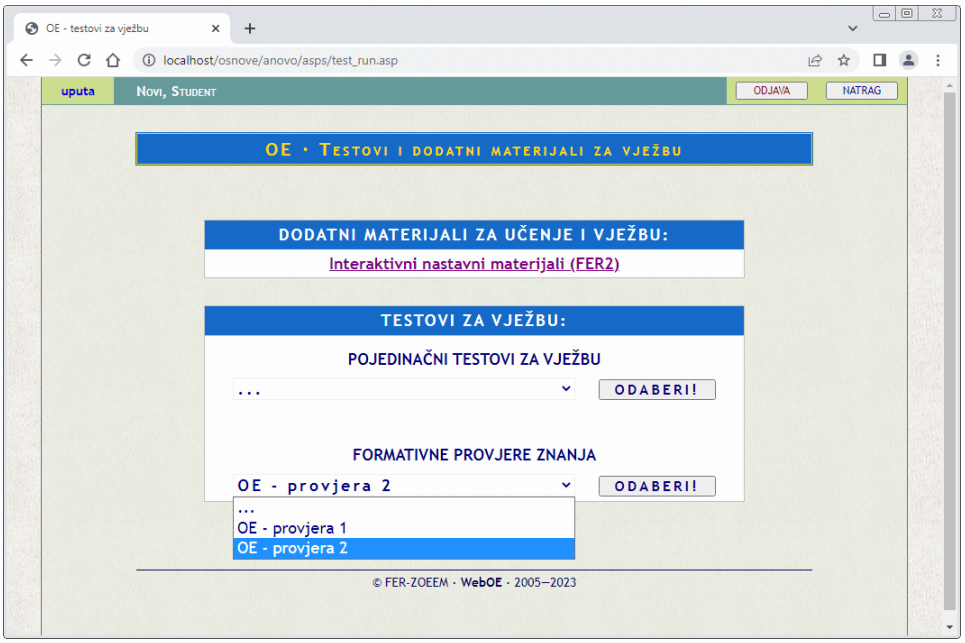
Zadnja, četvrta skripta *test_run.asp* koristi se za pokretanje i vođenje formativne provjere znanja. Skripta je radi lakše integracije u postojeći sustav za e-učenje *WebOE* napisana u poslužiteljskom skriptnom jeziku *ASP* za generiranje dinamičkih web stranica u HTML-u. Za prikaz formativne provjere znanja koriste uobičajeni web standardi poput HTML-a i CSS-a te klijentskog skriptnog jezika *Javascript* i *jQuery* [112]. Dodatno se po potrebi koristi *Javascript* biblioteka za crtanje grafova *JSXGraph* [113], grafički format SVG [114] te *Javascript* biblioteka *MathJax* za matematičku notaciju u *LaTeX* formatu [115].

Skripta pristupa bazi podataka sustava *WebOE* iz koje se traže podaci o formativnoj provjeri znanja, a nakon odabira tražene testa pokreće se proces formativne provjere znanja i pritom se zapisuju osnovni podaci o otvorenom pokušaju rješavanja provjere u tablicu *formative_tests_attempts*, a nakon otvaranja i odgovaranja na svako pitanje u odabranom nizu svi potrebni podaci se upisuju u dnevničku tablicu *formative_tests_log*. Uvijek se u zadnjem zapisu u toj tablici čuva smjer napretka u tom pokušaju provjere znanja (polje *forward_direction*). Već ranije je naglašeno kako će se u slučaju krivog odgovora na postavljeno pitanje student preusmjeriti na prethodna pitanja u nizu, a s točnim odgovorima student će napredovati kroz odabrani niz pitanja sve do kraja. Na kraju formativne provjere znanja student će dobiti izvještaj o broju odgovora na pitanja, prosječnom trajanju odgovaranja na pitanja, vremenskom okviru u kojem je provjera rješavana te će se ponovno navesti pojmovi iz gradiva kod kojih je bilo više krivih odgovora i vraćanja na prethodna pitanja.

Na slici 6.8 je prikazana početna stranica sustava *WebOE*, a nakon prijave na sustav studenti mogu odabrati poveznicu za testove za vježbu gdje mogu pristupiti pripremljenim formativnim provjerama znanja kako je prikazano na slici 6.9.

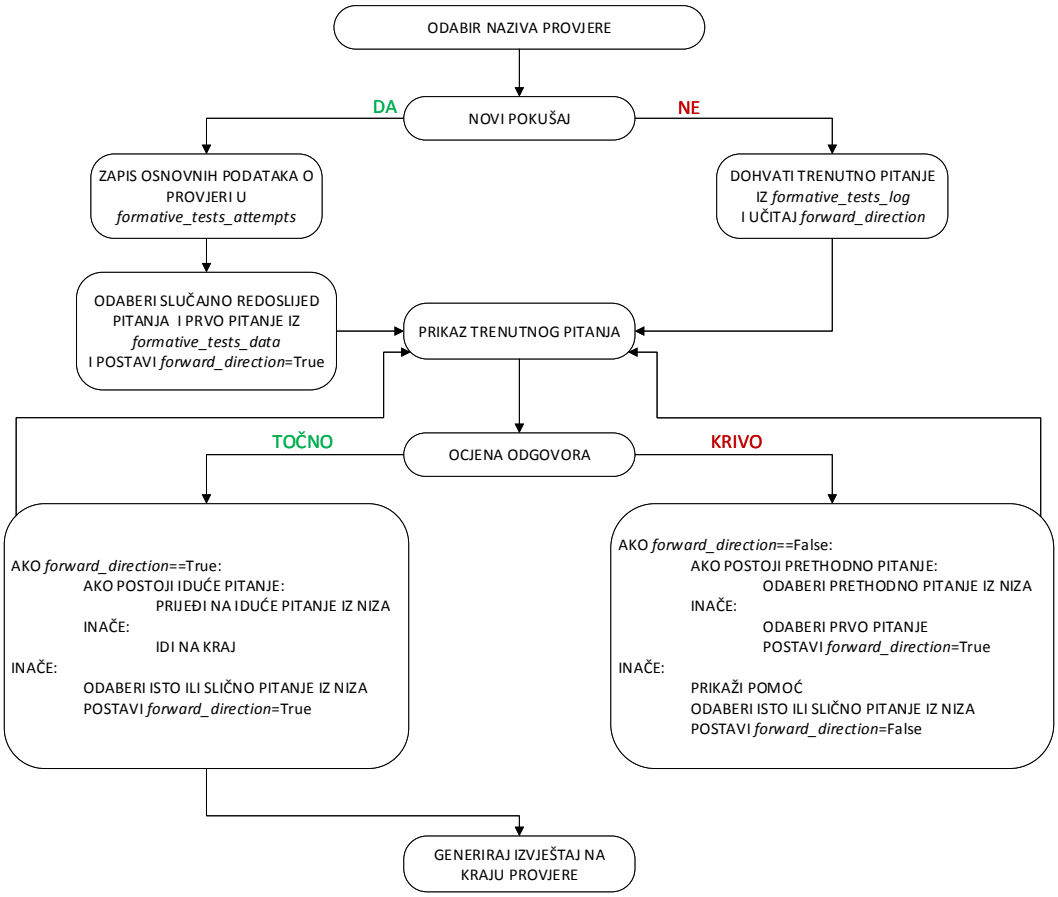


Slika 6.8: Početna stranica sustava *WebOE*



Slika 6.9: Odabir dostupnih formativnih provjera znanja

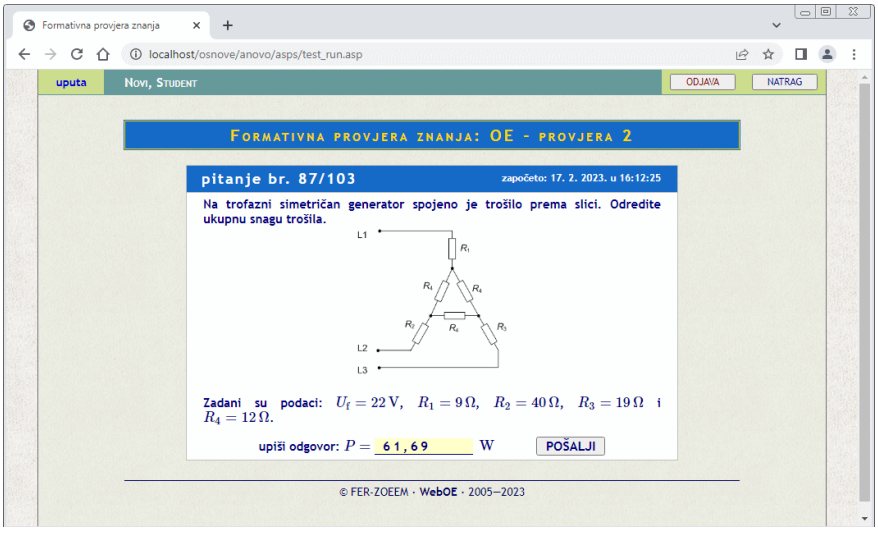
U nastavku će se detaljnije opisati model poslužiteljske skripte za vođenje formativne provjere znanja *test_run* prikazan na slici 6.10.



Slika 6.10: Model poslužiteljske skripte za vođenje formativne provjere znanja *test_run*

Nakon odabira jedne od dostupnih formativnih provjera znanja započinje ili novi pokušaj rješavanja provjere s upisom osnovnih podataka u tablicu *formative_tests_attempts* ili se nastavlja prethodni nezavršeni pokušaj rješavanja te provjere. Ako započinje novi pokušaj rješavanja prvo se na slučajan način odabire jedan od dva moguća puta rješavanja provjere iz tablice *formative_tests_data* i pronalazi se prvo pitanje u nizu. Pritom se po potrebi slučajno odabire prvo pitanje ako prvoj *EKP* točki pripada više od jednog pitanja. Nakon toga se svi generirani podaci za prvo pitanje upisuju u tablicu *formative_tests_log* i kreće se s rješavanjem provjere. S druge strane, ako se nastavlja prethodno prekinuto rješavanje provjere onda se iz tablice *formative_tests_log* pronalazi zadnje postavljeno pitanje i prikazuje se studentu, a rješavanje provjere se time nastavlja.

Još je potrebno detaljnije objasniti tijek formativne provjere znanja u slučaju krivoga odgovora. Dakle, kada student krivo odgovori na postavljeno pitanje sustav će mu ponuditi automatski generiranu poruku pomoći prema oznakama pitanja i ranije pripremljenim opcijama za pomoć za svaki od definiranih atributa u tablici *formative_tests_help*. Potom sustav ponovno postavi studentu isto pitanje (uz drukčije odabrane ulazne parametre ili drugu permutaciju ponuđenih odgovora) ili slično pitanje koje pripada istoj *EKP* točki. Status smjera rješavanja provjere u zapisu pitanja u tablici *formative_tests_log* se tada postavlja na $forward\_direction = \text{False}$. U slučaju ponovnog krivog odgovora sustav vodi studenta na prethodno pitanje u odabranom redoslijedu pitanja. Tek kada student ponovno odgovori točno na neko pitanje prvo će dobiti ponovno drugu varijantu istog ili sličnog pitanja, a smjer rješavanja se vrati na uobičajenu vrijednost $forward\_direction = \text{True}$. Ako student ponovno odgovori točno i na to pitanje prijeći će na iduće pitanje u odabranom nizu. Nakon što student prođe kroz čitav odabrani redoslijed pitanja formativna provjera znanja uspješno završava, a studenti dobivaju detaljniji statistički prikaz svojih rezultata provjere, a kao dodatnu povratnu informaciju generira se i popis pojmova s kojima su imali najviše poteškoća prilikom rješavanja provjere.



Slika 6.11: Prikaz jednog pitanja iz formativne provjere znanja

Na slici 6.11 je prikazan unos rješenja za jedno pitanje u generiranom nizu koje pokriva složenije teme iz izračuna snage nesimetričnih trofaznih trošila. Pritom treba naglasiti kako se kod prikaza pitanja s unosom numeričkog odgovora na slučajan način zadaju ulazni parametri, a uneseni rezultat se automatski ocjenjuje s pripremljenom skriptom sa zadanim postupkom rješenja. Slučajno generirani ulazni podaci se pritom spremaju u polje *parameters* u zapis o pitanju u tablici *formative_tests_log*, i to u formatu "*parametar1#parametar2#...*". U sustavu su podržana pitanja s unosnom jednog, dva ili tri numerička odgovora. Zbog mogućeg zaokruživanja u računu sustav će studentima prihvatiti odgovore unutar 5% od tražene točne vrijednosti. Pritom se uneseni odgovori spremaju u tekstualno polje *answer* u zapisu pitanja u tablici *formative_tests_log* i to u formatu "*odgovor1#odgovor2#...*".

Kao što je ranije spomenuto većina pitanja u studijskom primjeru su pitanja s višestrukim izborom odgovora od kojih je samo jedan ponuđeni odgovor točan. Kod takvih pitanja isto postoji nekoliko stupnjeva dinamičkog prikaza. U prvom stupnju se permutira redoslijed ponuđenih odgovora što se isto tako sprema u polje *parameters* u zapisu pitanja u tablici *formative_tests_log* kao permutacija oznaka odgovora, npr. "*ADCBE*". Osim toga postoje i parametrizirana pitanja kod kojih su neki manji ili veći dijelovi teksta pitanja ili odgovora varijabilni, a isto tako svaka varijanta može imati drukčiju sliku uz tekst. U tom slučaju se kod prikaza pitanja generira slučajan ključ koji se sprema zajedno s permutacijom oznaka odgovora (npr. "*ADCBE#18*") u isto polje *parameters* u zapisu pitanja u tablici *formative_tests_log*. Treba naglasiti kako se kod parametriziranih pitanja uglavnom mijenjaju diskretne vrijednosti numeričkih parametara u tekstu kao i ponuđenih odgovora. Nakon odabira i slanja odgovora sprema se oznaka odabranog odgovora u tekstualno polje *answer* u zapisu pitanja u tablici *formative_tests_log*.

6.2 Rezultati i rasprava

U uvodnom dijelu ovoga poglavlja već je detaljno analizirana konceptualna rešetka završnog formalnog konteksta sa 139 pitanja i 50 atributa koji je generiran s postupkom kombinatornog testiranja u poglavlju 5. A u ovom potpoglavlju prikazat će se rezultati postupka za pripremu formativne provjere znanja temeljene na toj konceptualnoj rešetci kao i uzorci podataka o rješavanim formativnim provjerama znanja.

Na početku će se ukratko opisati rezultati provođenja inicijalne faze pripreme baze podataka sustava *WebOE*. Prvo su se sa *Python* skriptom *prepare_database.py* automatski unijeli nazivi definiranih 50 atributa u tablicu *formative_tests_attribute_list* kao i popis identifikacijskih oznaka 139 odabranih pitanja i njihove opise u tablicama *formative_tests_question_list* i *formative_tests_question_description*. Kao primjer je na slici 6.12 prikazan segment tablice *formative_tests_attribute_list* sa spremljenim nazivima atributa kao i s generiranim identifikacijskim oznakama svakog atributa.

formative_tests_attribute_list

attributeID	attribute_text
1	force_electric
2	field_electric
3	potential_electric
4	energy_electric
5	charge
6	capacity
7	capacitors
8	resistance
9	DC_current
10	DC_voltage
11	DC_power
12	current_source
13	voltage_source
14	real_source
15	ideal_source
16	force_magnetic
17	field_magnetic
18	energy_magnetic
19	flow_magnetic
20	induced_voltage
21	Ohm
22	Kirchhoff
23	Thevenin

Slika 6.12: Segment iz tablice *formative_tests_attribute_list*

Kasnije će se te identifikacijske oznake atributa koristiti kod opisa pitanja i kod zadavanja različitih opcija za pomoć kod krivih odgovora na pitanja. Nakon spremanja opisa svih pitanja u tablici *formative_tests_question_description* ima 1438 redaka, odnosno u prosjeku svako od 139 pitanja opisano je u prosjeku s preko 10 različitih atributa. Na slici 6.13 je prikazan manji uzorak podataka iz te tablice.

formative_tests_question_description

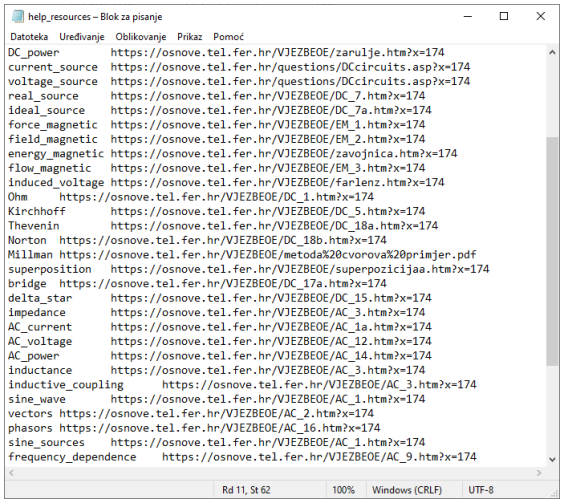
ID	questionID	attributeID
15819	266	3
15820	266	8
15821	266	9
15822	266	10
15823	266	13
15824	266	15
15825	266	21
15826	266	22
15827	266	33
15828	266	42
15829	287	3
15830	287	6
15831	287	8
15832	287	13
15833	287	15
15834	287	21
15835	287	29
15836	287	30
15837	287	31
15838	287	33
15839	287	36
15840	287	38
15841	287	39

Slika 6.13: Uzorak iz tablice *formative_tests_question_description*

Jedan od ciljeva formativne provjere znanja je da ponudi studentima korisnu povratnu informaciju u slučaju krivog odgovora. Zato nastavnici i drugi domenski eksperti za predloženi postupak formativne provjere znanja mogu navesti različite izvore i opcije pomoći za svaki od atributa iz završnog formalnog konteksta, odnosno za temu gradiva koju taj atribut pokriva. Opcije pomoći mogu biti poveznice na nastavne materijale ili druge izvore na webu, ali i pripremljeni blokovi teksta sa statičkim slikama ili npr. interaktivnim grafovima s bibliotekom *JSXGraph*. U ovom istraživanju su se kao opcije pomoći ponudile poveznice na vlastite inte-

raktivne nastavne materijale, javno dostupne u sustavu za e-učenje *WebOE*[1]. Među njima su sada značajno proširene i modernizirane verzije ranijih nastavnih materijala mr.sc. Ivana Felje, a veliki dio je nastao kao originalan rezultat ovoga doktorskog istraživanja. Treba spomenuti kako su se kod izrade ovih nastavnih materijala koristile različite tehnologije, a najviše *Javascript* biblioteke *JSXGraph* i *MathJax*.

Ovdje je za svaki od 50 atributa navedena po jedna poveznica na najprikladnije vlastite interaktivne nastavne materijale, a ti podaci su pripremljeni kao ulazna datoteka *help_resources.txt* prikazana na slici 6.14. Potom su automatski spremljeni sa skriptom *prepare_database.txt* u tablicu *formative_tests_help* čiji se sadržaj može vidjeti na slici 6.15.



Slika 6.14: Dio datoteke *help_resources.txt* s opcijama pomoći za svaki atribut

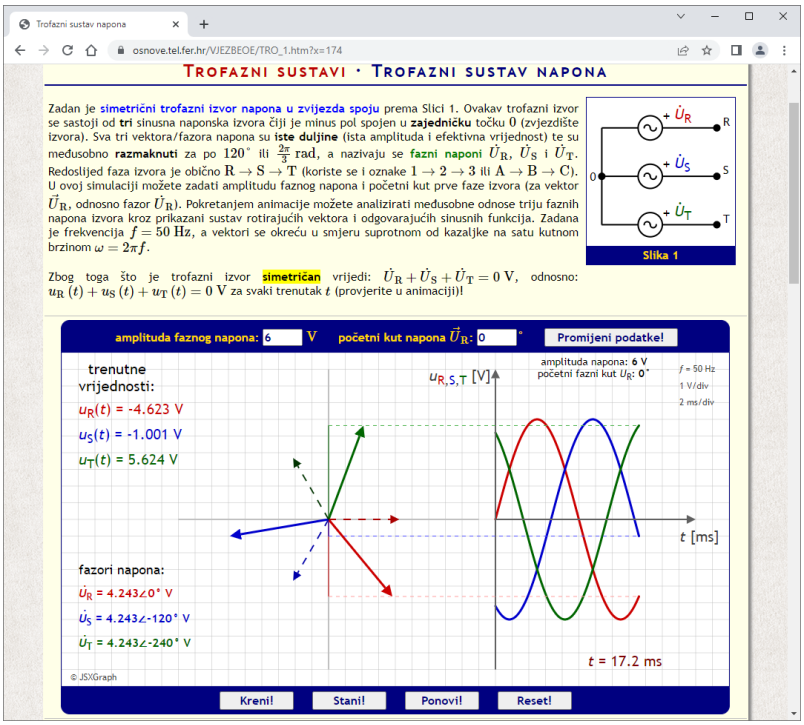
formative_tests_help

ID	attributeID	help_resource
61	11	https://osnove.tel.fer.hr/VJEZBEOE/zarulje.htm?x=174
62	12	https://osnove.tel.fer.hr/questions/DCcircuits.asp?x=174
63	13	https://osnove.tel.fer.hr/questions/DCcircuits.asp?x=174
64	14	https://osnove.tel.fer.hr/VJEZBEOE/DC_7.htm?x=174
65	15	https://osnove.tel.fer.hr/VJEZBEOE/DC_7a.htm?x=174
66	16	https://osnove.tel.fer.hr/VJEZBEOE/EM_1.htm?x=174
67	17	https://osnove.tel.fer.hr/VJEZBEOE/EM_2.htm?x=174
68	18	https://osnove.tel.fer.hr/VJEZBEOE/zavojnica.htm?x=174
69	19	https://osnove.tel.fer.hr/VJEZBEOE/EM_3.htm?x=174
70	20	https://osnove.tel.fer.hr/VJEZBEOE/farlenz.htm?x=174
71	21	https://osnove.tel.fer.hr/VJEZBEOE/DC_1.htm?x=174
72	22	https://osnove.tel.fer.hr/VJEZBEOE/DC_5.htm?x=174
73	23	https://osnove.tel.fer.hr/VJEZBEOE/DC_18a.htm?x=174
74	24	https://osnove.tel.fer.hr/VJEZBEOE/DC_18b.htm?x=174
75	25	https://osnove.tel.fer.hr/VJEZBEOE/metoda%20cvorova%20primjer.pdf
76	26	https://osnove.tel.fer.hr/VJEZBEOE/superpozicijaa.htm?x=174
77	27	https://osnove.tel.fer.hr/VJEZBEOE/DC_17a.htm?x=174
78	28	https://osnove.tel.fer.hr/VJEZBEOE/DC_15.htm?x=174
79	29	https://osnove.tel.fer.hr/VJEZBEOE/AC_3.htm?x=174
80	30	https://osnove.tel.fer.hr/VJEZBEOE/AC_1a.htm?x=174
81	31	https://osnove.tel.fer.hr/VJEZBEOE/AC_12.htm?x=174
82	32	https://osnove.tel.fer.hr/VJEZBEOE/AC_14.htm?x=174
83	33	https://osnove.tel.fer.hr/VJEZBEOE/AC_3.htm?x=174
84	34	https://osnove.tel.fer.hr/VJEZBEOE/AC_3.htm?x=174
85	35	https://osnove.tel.fer.hr/VJEZBEOE/AC_1.htm?x=174
86	36	https://osnove.tel.fer.hr/VJEZBEOE/AC_2.htm?x=174
87	37	https://osnove.tel.fer.hr/VJEZBEOE/AC_16.htm?x=174
88	38	https://osnove.tel.fer.hr/VJEZBEOE/AC_1.htm?x=174
89	39	https://osnove.tel.fer.hr/VJEZBEOE/AC_9.htm?x=174

Slika 6.15: Uzorak iz tablice *formative_tests_help*

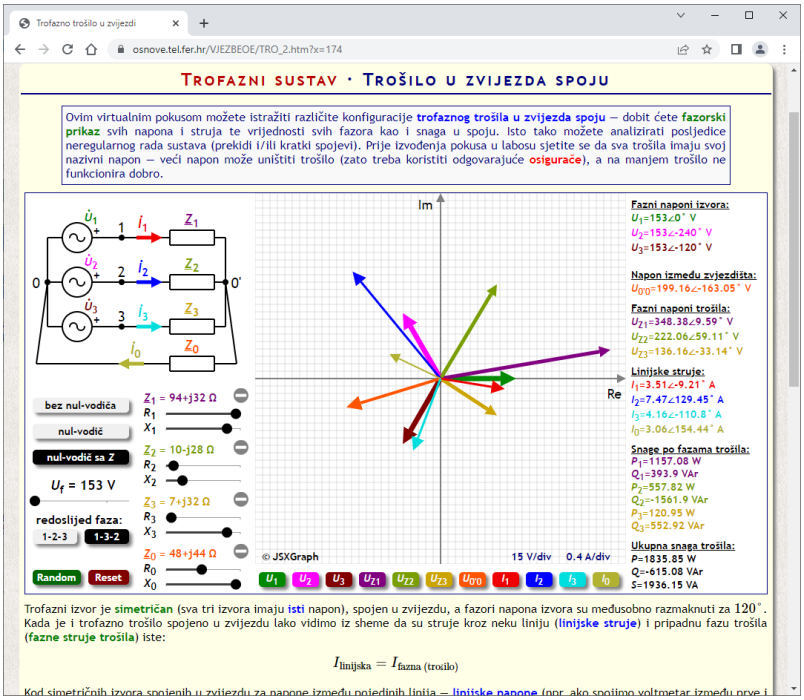
[1]https://osnove.tel.fer.hr/VJEZBEOE/vjezbe.asp

U nastavku će se prikazati nekoliko primjera interaktivnih nastavnih materijala u sustavu *WebOE*. Na slici 6.16 prikazana je uvodna animacija sinusoida napona, rotirajućih vektora kao i fazora napona u simetričnom trofaznom izvoru u zvijezda spoju.



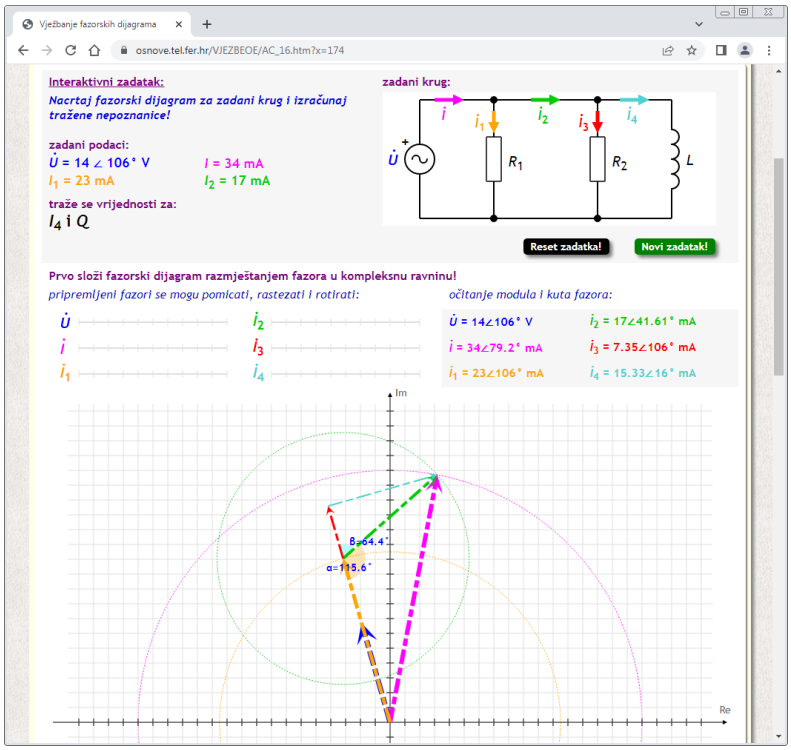
Slika 6.16: Isječak iz interaktivnih materijala o trofaznim sustavima napona

Iduća slika 6.17 prikazuje dio interaktivnih nastavnih materijala o trofaznim trošilima u zvijezda spoju, a s kojim se automatski rješavaju zadane konfiguracije trošila.



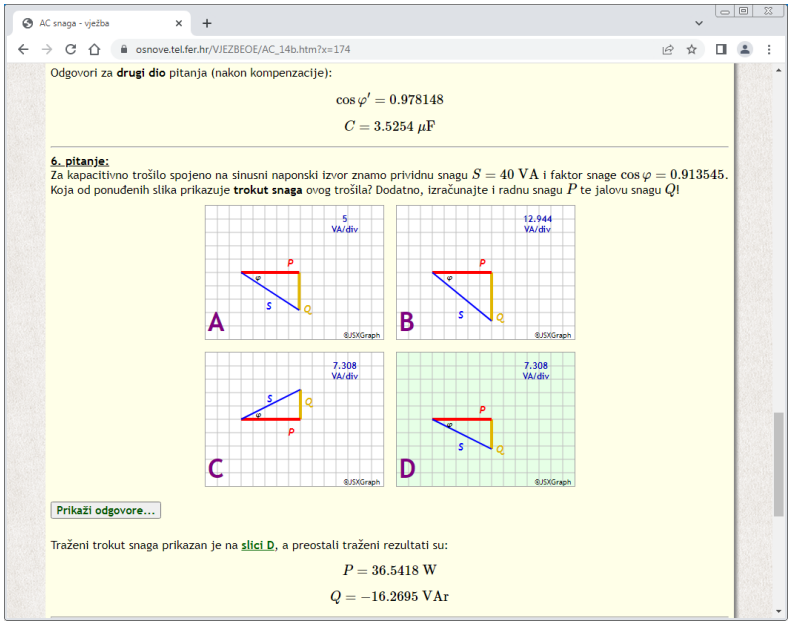
Slika 6.17: Isječak iz interaktivnih materijala o trofaznim trošilima u zvijezda spoju

Na slici 6.18 je prikazan manji dio interaktivnog zadataka za vježbanje crtanja fazorskog dijagrama. Tekst i slika zadataka se dinamički generiraju iz zadanih predložaka, a student treba izračunati tražene veličine i namjestiti i postaviti sve fazore u kompleksnu ravninu. Ovdje je prikazano točno rješenje uz aktiviranu pomoć kod crtanja fazora.



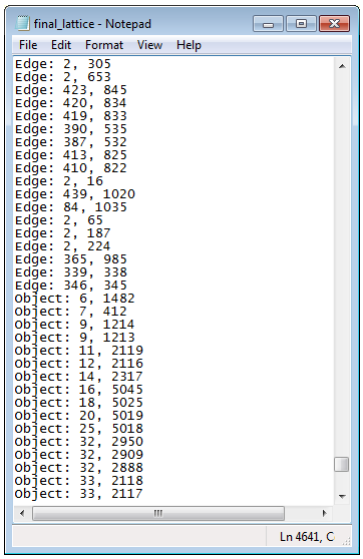
Slika 6.18: Isječak interaktivnih materijala o vježbanju crtanja fazorskih dijagrama

Na kraju je na slici 6.19 prikazan isječak automatski generiranih zadataka za vježbu iz snage u izmjeničnim krugovima s otvorenim točnim odgovorima.



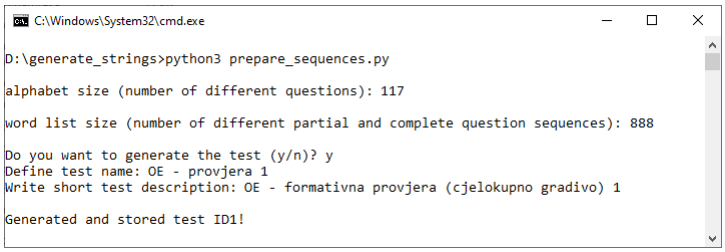
Slika 6.19: Isječak automatski generiranih zadataka za snagu u izmjeničnim krugovima

U sljedećem koraku se prelazi na topološko sortiranje velike konceptualne rešetke prikazane na slici 6.3 koja ima ukupno 1037 formalnih koncepata ili *EKP* točaka, a pritom se svaki formalni koncept sastoji od nekog podskupa od 139 pitanja i nekog podskupa od 50 atributa. Pritom je najbitniji tekstualni zapis izgrađene konceptualne rešetke koji se automatski dobije iz korištenog alata za metodu FCA *Concept Explorer 1.3*. Na slici 6.20 je prikazan mali dio te tekstualne datoteke *final_lattice.txt* koja sadrži 4641 redak s podacima o koordinatama svake *EKP* točke, svim usmjerenim granama te o svakoj *EKP* točki u kojoj je eksplicitno zapisan neki od objekata ili atributa.



Slika 6.20: Manji dio tekstualnog zapisa konceptualne rešetke sa slike 6.3

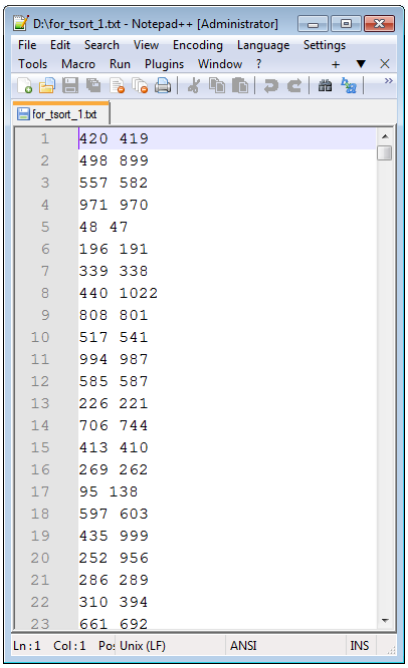
Primjer izvođenja automatskog provođenja topološkog sortiranja konceptualne rešetke u datoteci *final_lattice.txt* s *Python* skriptom *prepare_sequences.py* te pomoćnom *batch* skriptom *run_tsort.py* prikazan je na idućoj slici 6.21.



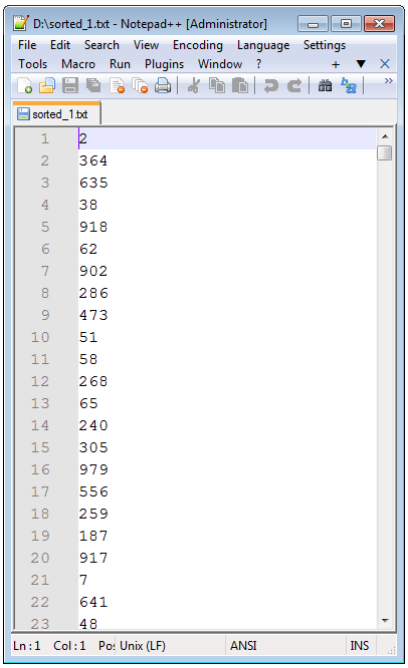
Slika 6.21: Primjer izvođenja topološkog sortiranja sa skriptom *prepare_sequences.py*

Izvođenjem skripte *prepare_sequences.py* automatski su se dobila dva topološka sortiranja iste konceptualne rešetke. Time su definirana dva potpuno uređena skupa *EKP* točaka iz konceptualne rešetke, a svaki od ta dva skupa ima po 1037 *EKP* točaka. A kako bi se dobila dva ispravna topološka sortiranja iste konceptualne rešetke na početku se na slučajan način permutirali redci datoteke *final_lattice.txt*. Primjerice, za prvo topološko sortiranje koristila se automatski gene-

rirana datoteka *for_tsort_1.txt* prikazana na slici 6.22, a dobiveno topološko sortiranje s programom *tsort* automatski je spremljeno je u datoteku *sorted_1.txt* i prikazano na slici 6.23. Treba naglasiti kako zbog formata tekstualnog zapisa konceptualne rešetke u programu *Concept Explorer 1.3* nakon topološkog sortiranja treba obrnuti redoslijed *EKP* točaka kako bi se dobio traženi poredak od najopćenitijih do najspecifičnijih *EKP* točaka.



Slika 6.22: Pregled datoteke *for_tsort_1.txt* sa slučajno izmiješanim popisom usmjerenih grana iz datoteke *final_lattice.txt*



Slika 6.23: Pregled datoteke *sorted_1.txt* s rezultatom topološkog sortiranja parcijalno uređenog poretka na slici 6.22

Ako se iz generiranih linearnih poredaka *EKP* točaka filtriraju samo one *EKP* točke u kojima su eksplicitno zapisana pitanja preostaje u svakom skupu samo po 103 *EKP* točke. U ovom istraživanju će svaki od tih filtriranih skupova *EKP* točaka predstavljati jedan osnovni put izvođenja formativne provjere znanja. Kako pojedine *EKP* točke sadrže više od jednog pitanja opisanih s istim skupom atributa tako se iz svakog osnovnog puta izvođenja može izvesti više odgovarajućih redoslijeda pitanja. Kako se ne bi generiralo previše mogućih redoslijeda pitanja koristila su se ograničenja na broj pitanja u svakoj *EKP* točki. Zato su se na slučajan način kod svakog izvođenja skripte *prepare_sequences.py* odabrala najviše tri *EKP* točke u kojima se broj pitanja ograniči na dva, dok u svim ostalim *EKP* točkama ostaje samo po jedno pitanje. Tako su se u ovom slučaju dobila dva osnovna puta izvođenja formativne provjere znanja, svaki s 8 različitih potpunih nizova od 103 pitanja, koji su ukupno iskoristili 117 od 139 pitanja u završnom kontekstu. Osim toga generirano je ukupno 888 različitih potpunih i djelomičnih nizova pitanja (svi prefiksi od 16 potpunih nizova pitanja) uključujući i prazan niz pitanja. Svi podaci o dva osnovna generirana puta izvođenja formativne provjere znanja spremljeni su u bazu podataka sustava *WebOE* u tablicu *formative_tests_data*, a uzorak iz te tablice je prikazan na slici 6.24.

formative_tests_data

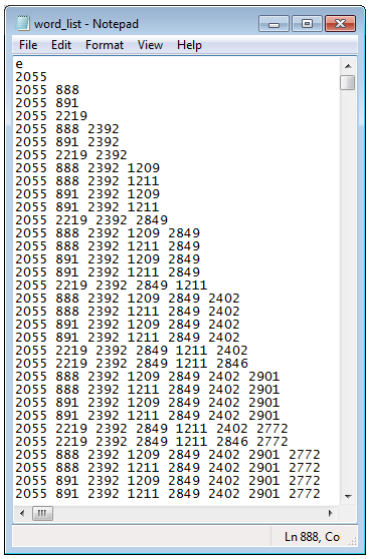
ID	testID	question_number	questionID	path
2894	1	86	5022	1
2895	1	87	5024	1
2896	1	88	2892	1
2897	1	88	2872	1
2898	1	89	5008	1
2899	1	90	2222	1
2900	1	91	2890	1
2901	1	92	5035	1
2902	1	93	5033	1
2903	1	94	5049	1
2904	1	95	922	1
2905	1	96	2954	1
2906	1	97	2568	1
2907	1	98	5047	1
2908	1	99	2908	1
2909	1	100	5012	1
2910	1	101	415	1
2911	1	102	5001	1
2912	1	103	5015	1
2913	1	1	2055	2
2914	1	2	2219	2
2915	1	3	2392	2
2916	1	4	2849	2
2917	1	5	1211	2
2918	1	6	2846	2
2919	1	6	2402	2
2920	1	7	2772	2
2921	1	8	2901	2
2922	1	9	287	2
2923	1	10	335	2
2924	1	11	362	2
2925	1	12	2117	2
2926	1	13	1213	2
2927	1	14	2868	2

Slika 6.24: Uzorak iz tablice *formative_tests_data*

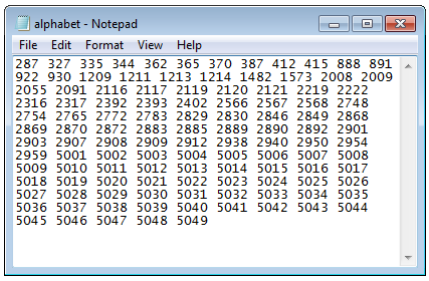
Iz prikazanog uzorka može se vidjeti kako 88. pitanje u prvom osnovnom nizu *EKP* točaka (polje $path = 1$) može biti jedno od dva pitanja s identifikacijskim oznakama 2892 i 2872. Isto tako, u drugom osnovnom nizu (polje $path = 2$) kao 6. pitanje će se postaviti jedno od dva pitanja s identifikacijskim oznakama 2846 i 2402. Može se provjeriti kako će u svakom od dva osnovna niza *EKP* točaka u ovom slučaju biti po 106 različitih pitanja (tri *EKP* točke s po

dva pitanja i 100 *EKP* s po jednim pitanjem). Dakle, prilikom rješavanja formativne provjere znanja u ovisnosti o broju krivih odgovora i vraćanja na prethodna pitanja student će proći kroz minimalno 103, a maksimalno 106 različitih osnovnih varijanti pitanja iz odabranog niza *EKP* točaka.

Na slici 6.25 prikazan je dio od 888 generiranih djelomičnih i potpunih različitih nizova identifikacijskih oznaka pitanja iz automatski generirane datoteke *word_list.txt*, a na slici 6.26 se nalazi lista svih 117 različitih identifikacijskih oznaka pitanja iz datoteke *alphabet.txt* koja su se pojavljivala u jednom i/ili drugom osnovnom nizu *EKP* točaka.



Slika 6.25: Pregled datoteke *word_list.txt* sa svim potpunim i djelomičnim nizovima pitanja



Slika 6.26: Pregled datoteke *alphabet.txt* sa svim korištenim pitanjima

Ove dvije tekstualne datoteke sa slika 6.25 i 6.26 koristit će se kasnije u poglavlju 7 kao ulazni podaci za izradu modela za verifikaciju predložene metode za automatizirano generiranje formativnih provjera znanja.

Nakon predstavljanja rezultata pripreme formativne provjere znanja može se prikazati i jedan primjer zapisa izvođenja te provjere. Na slici 6.27 je prikazan uzorak podataka iz tablice *formative_tests_log* iz kojeg se može pratiti tijek jednog probnog pokušaja rješavanja formativne provjere znanja. U ovom pokušaju je odabrana prva formativna provjera znanja (*OE* -

provjera 1 sa slike 6.21), a na početku je slučajno odabran prvi od generirana dva osnovna puta izvođenja procesa provjere.

formative_tests_log

ID	attemptID	current_question	parameters	answer	evaluation	forward_direction	time_opened	time_answered
2	2	2807	CBA	B	☑	☑	15.2.2023. 10:33:22	15.2.2023. 10:34:11
3	2	2808	BAC	A	☐	☑	15.2.2023. 10:34:11	15.2.2023. 10:34:27
5	2	2809	CAB	C	☑	☐	15.2.2023. 10:34:27	15.2.2023. 10:34:31
6	2	2808	ABC	B	☑	☑	15.2.2023. 10:34:31	15.2.2023. 10:34:40
7	2	2810	17#14#28#53#10#10	205,33	☑	☑	15.2.2023. 10:34:40	15.2.2023. 10:47:12
8	2	2811	7#4#221#50	55,1#4,3#17,88	☐	☑	15.2.2023. 10:47:12	15.2.2023. 11:04:43
9	2	2811	7#2#222#50	110#2,94#24,1	☑	☐	15.2.2023. 11:04:43	15.2.2023. 11:19:12
10	2	2811	6#3#215#50	72,2#3,27#23,19	☑	☑	15.2.2023. 11:19:12	15.2.2023. 11:29:16
11	2	2813	ABDCE#8	C	☑	☑	15.2.2023. 11:29:16	16.2.2023. 9:33:16
14	2	2814	EDCAB	C	☑	☑	16.2.2023. 9:33:16	16.2.2023. 9:35:22
15	2	2815	BADCE	B	☑	☑	16.2.2023. 9:35:22	16.2.2023. 9:36:04
16	2	2816	ABCED#6	C	☑	☑	16.2.2023. 9:36:04	17.2.2023. 11:10:23
17	2	2817	BACD	B	☑	☑	17.2.2023. 11:10:23	17.2.2023. 11:14:17
18	2	2818	CAB		☐	☑	17.2.2023. 11:14:17	

Slika 6.27: Uzorak iz tablice *formative_tests_log*

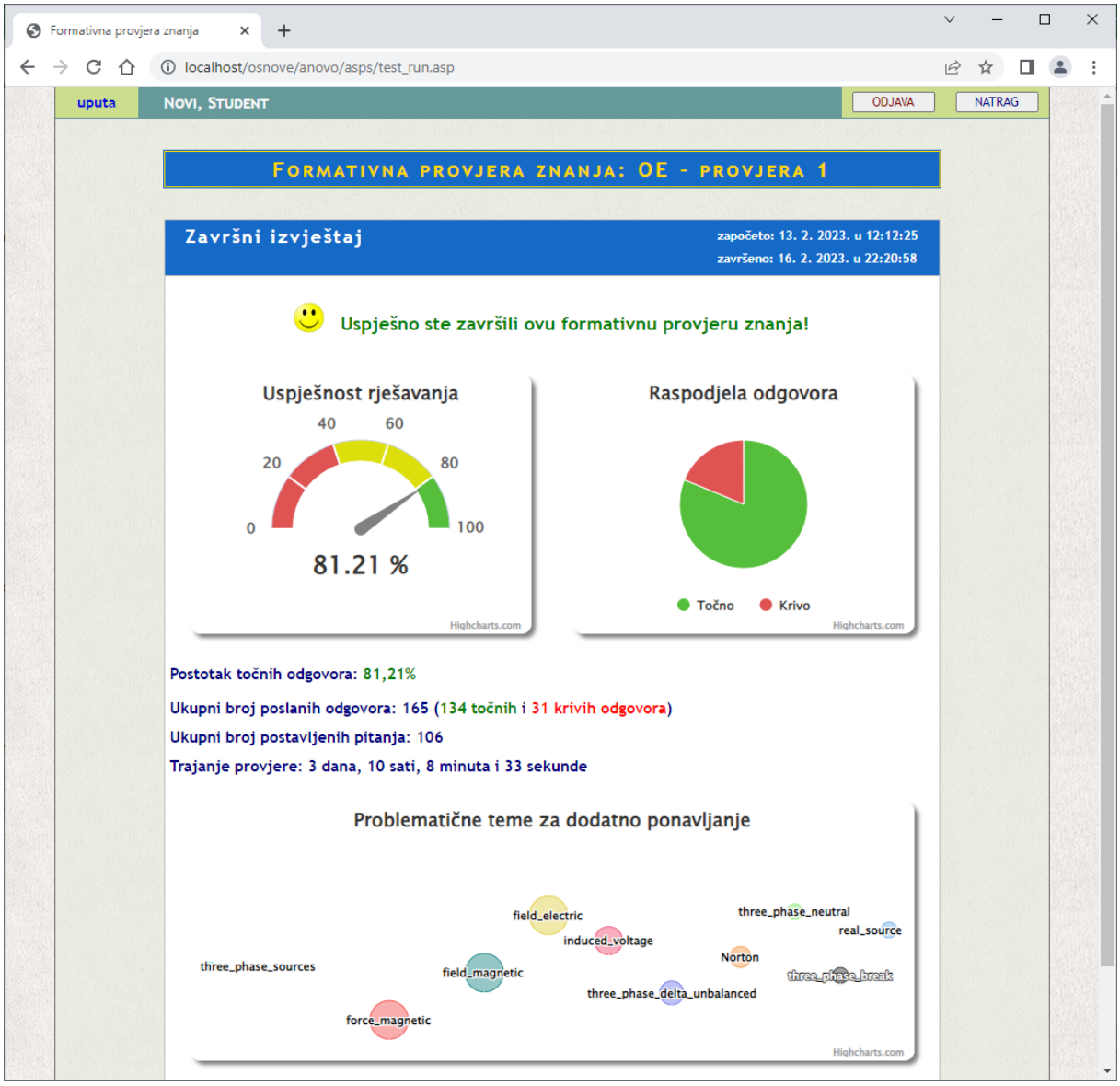
Iz prikazanog uzorka može se primijetiti kako je odabrani probni student rješavao formativnu provjeru znanja kroz više dana, a svaki put se provjera nastavila na mjestu gdje je prethodni put završila. Iz zadnjeg retka ID18 u uzorku sa slike 6.27 zaključuje se kako student još nije odgovorio na zadano pitanje, odnosno kako će ga to pitanje opet čekati kad ponovno pokrene ovu formativnu provjeru znanja. Isto tako, iz prikazanih rezultata se može se iščitati ranije predstavljeni način upravljanja s procesom provjere znanja. Naime, nakon krivog odgovora studentu će se ponoviti jedna varijanta istog ili srodnog pitanje iz iste *EKP* točke, a ako točno odgovori na to pitanje ponovno će mu se ponoviti neka slučajno odabrana varijanta tih pitanja. Tek kada drugi put odgovori točno nakon ranijeg krivog odgovora sustav će studentu postaviti jednu moguću varijantu idućeg pitanja u odabranom nizu. Na slici 6.27 mogu se vidjeti ovakvi scenariji iz redaka ID3 do ID7 te redaka ID8 do ID11.

Iz slučajno zadanih ulaznih parametara spremljenih u polju *parameters* može se uvijek rekonstruirati točna varijanta pitanja kako je bila postavljena studentu. Nadalje, iz vrijednosti zapisa polja *parameters* i *answers* vidi se kako se u izdvojenom dijelu niza pitanja na slici 6.27 pojavljuju sva tri tipa pitanja koja su ranije objašnjena – pitanja s višestrukim izborom odgovora, parametrizirana pitanja s višestrukim izborom odgovora te računska pitanja s unosom od jednog do tri numerička odgovora.

Treba naglasiti kako će računsko pitanje s numeričkim odgovorima u ovoj implementaciji modula za formativnu provjeru znanja biti ocijenjeno kao točno samo ako su svaki rezultat unutar zadane tolerancije od $\pm 5\%$ od tražene vrijednosti. U budućem razvoju moglo bi biti korisno razviti fleksibilniji sustav za ocjenjivanje ovakvih pitanja. Primjerice, ako su dva odgovora točna, a samo treći nije točan (redak ID7 sa slike 6.27) može se prvo ponuditi ispravak krivoga odgovora, a tek ako je on i dalje netočan ponovila bi se neka varijanta istog ili srodnog pitanja.

Na kraju uspješnog završetka formativne provjere studentu se automatski generira završni izvještaj. U njemu se navode osnovni statistički pokazatelji o provedenoj provjeri kao što su

postotak točnih odgovora, ukupan broj poslanih odgovora i ukupan broj postavljenih osnovnih varijanti pitanja te ukupno trajanje provjere. Dodatno se navodi do 10 atributa koji predstavljaju nastavne teme s kojima je student imao najviše problema tijekom rješavanja formativne provjere. Pritom poslužiteljska skripta *test_run* izračuna frekvenciju atributa svih pitanja na koje je student krivo odgovorio, kao i frekvenciju atributa svih pitanja na koje je student točno odgovorio. Potom se izračuna kvocijent tih dviju frekvencija i pronalazi se 10 atributa s najvećim vrijednostima kvocijenta. Primjer jednog takvog završnog izvještaja je prikazan na slici 6.28. Za prikaz statističkih rezultata i popisa nastavnih tema koje treba dodatno utvrditi korištena je *Javascript* biblioteka *Highcharts* [116].



Slika 6.28: Primjer završnog izvještaja

Detaljnijim pregledom automatski generiranih nizova pitanja primjećuje se kako u sažetom skupu od 139 mogućih pitanja ima vrlo malo trivijalnih i lakih pitanja. Treba napomenuti kako

se početni skup sastoji od 473 pitanja preuzetih iz starijih sumativnih provjera poput pismenih ispita, testova na laboratorijskim vježbama i domaćim zadaćama. Nakon provođenja kombinatornog testiranja taj skup se dodatno filtrirao jer je stavljen naglasak na pitanja koja povezuju različite nastavne teme. U budućem radu će se razmotriti stvaranje manjih skupova jednostavnijih pitanja koji bi bili pogodniji za kraće formativne provjere znanja koje ne bi pokrivale čitavo gradivo predmeta nego samo pojedine nastavne cjeline.

Poglavlje 7

Verifikacija metode za automatiziranu provjeru znanja

7.1 Uvod

U ovom poglavlju predstavit će se postupak verifikacije predložene metode za automatiziranu provjeru znanja kroz implementaciju nadograđenog algoritma L^* Dane Angluin i korištenje alata za provjeru modela *Spin*. Kako je već objašnjeno u poglavljima 5 i 6, postupkom kombinatornog testiranja se pronašao sažeti skup označenih pitanja, potom se metodom FCA iz generirala konceptualna rešetka tog skupa pitanja, a na kraju su se postupkom topološkog sortiranja konceptualne rešetke dobili prihvatljivi nizovi pitanja za formativne provjere znanja.

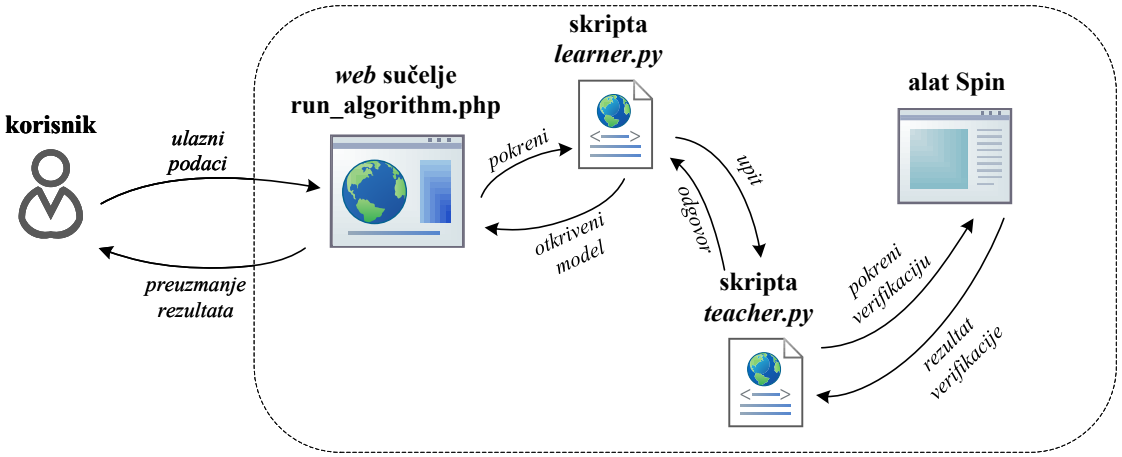
Ovdje će se upravo na temelju tih generiranih potpunih i djelomičnih nizova pitanja automatski izgraditi model procesa formativne provjere znanja u obliku konačnog automata. Cilj je da se tim konačnim automatom vjerodostojno opiše osnovno ponašanje modula za automatiziranu provjeru znanja iz poglavlja 6. Kroz izvođenje otkrivenog konačnog automata moći će se analizirati tijek procesa provjere znanja – redoslijed zadavanja pitanja te putevi kojima se može doći od početnog pitanja sve do zadnjeg pitanja u nizu. Alfabet konačnog automata jednak je skupu jedinstvenih identifikacijskih oznaka svih pitanja, a svako stanje automata odgovara nekom stanju procesa provjere znanja. Nadalje, svaki označeni prijelaz u automatu predstavlja zadavanje nekog pitanja, a svi prihvatljivi nizovi pitanja su riječi koje konačni automat mora prihvaćati. Naposljetku će se omogućiti automatska verifikacija i simulacija pronađenog konačnog automata kako bi se osiguralo da se tijek procesa provjere znanja odvija na ispravan način te kako bi se mogli analizirati različiti načini izvođenja automata. Preciznije rečeno, verifikacijom konačnog automata će se potvrditi ili opovrgnuti odgovara li model procesa provjere znanja zadanim specifikacijama, a slučajnim simulacijama konačnog automata dobit će se različiti scenariji izvođenja procesa provjere znanja.

Model procesa provjere znanja se u ovome radu izgrađuje primjenom L^* algoritma Dane

Angluin za učenje deterministčkih konačnih automata čiji je formalni opis predstavljen u potpoglavlju 3.4.2). Ovaj algoritam definira ulogu *Učenika* koji u interakciji sa sveznajućim *Učiteljem* (eng. *oracle*) otkriva inicijalno nepoznati deterministički konačni automat. U ovom radu se predlaže nadogradnja L^* algoritma s novim algoritmom *Učitelja* koji će na sva pitanja *Učenika* odgovarati samostalno uz pomoć alata za provjeru modela *Spin*. Na kraju, treba isto tako naglasiti da će se alat *Spin* koristiti i za automatsku verifikaciju i simulaciju pronađenog konačnog automata koji će se prije toga automatski prevesti u odgovarajući procesni model u jeziku *Promela*.

7.2 Otkrivanje modela procesa provjere znanja

U radu je razvijen automatizirani modul za otkrivanje modela procesa provjere znanja u obliku konačnog automata kroz implementaciju nadograđenog L^* algoritma te korištenjem alata za provjeru modela *Spin*. Glavne funkcionalnosti modula su prikazane na slici 7.1.



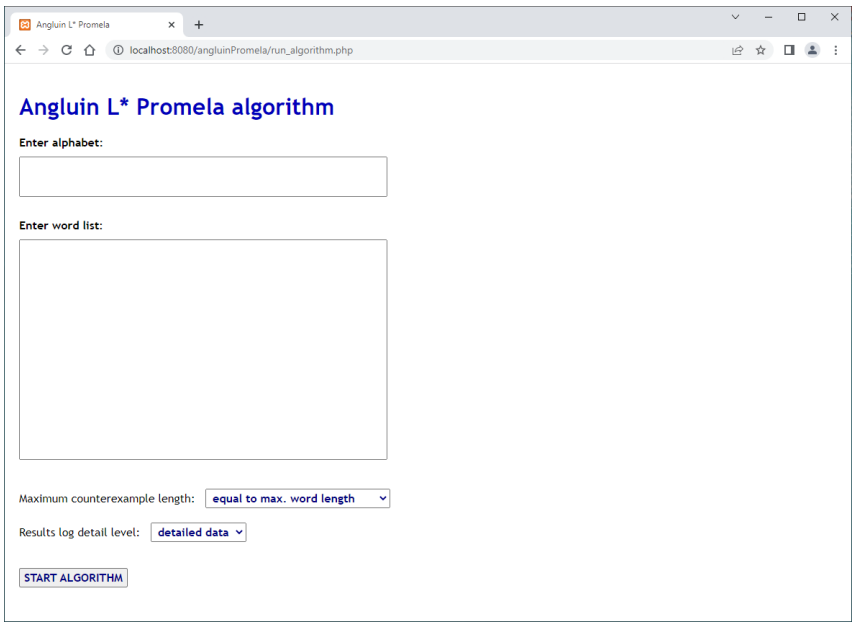
Slika 7.1: Modul za otkrivanje modela procesa provjere znanja

L^* algoritam opisuje postupak pronalaženja nepoznatog DFA ako je inicijalno poznat samo njegov alfabet ili abeceda. U predloženom pristupu abecedu DFA će predstavljati identifikacijske oznake pitanja za provjeru znanja, a svi generirani mogući potpuni i djelomični nizovi pitanja smatrat će se riječima koje DFA mora prihvaćati. Dakle, sa svakim idućim pitanjem iz niza konačni automat prelazi iz trenutnog stanja u sljedeće, a pritom prolazi put od početnog stanja do nekog završnog odnosno prihvaćajućeg stanja.

Sam postupak traženja nepoznatog DFA je u L^* algoritmu definiran kao dijalog između dva sudionika – *Učenika* i *Učitelja*. Uloga *Učenika* je u potpunosti opisana s L^* algoritmom, dok se za ulogu *Učitelja* samo navodi kako mora istinito odgovarati na sve upite *Učenika* i na koji način mu treba slati odgovore. Naravno, ulogu *Učitelja* može preuzeti korisnik – do-

menski ekspert koji će na upite *Učenika* npr. ručno odgovarati kroz sučelje naredbenog retka. Ipak, kako bi se interakcija između ova dva aktera automatizirala u ovom radu je razvijen i implementiran vlastiti algoritam koji opisuje ulogu *Učitelja*. Pritom je napravljena i vlastita implementacija *L** algoritma. Razvijeni modul za otkrivanje modela procesa provjere znanja sastoji se od glavne *Python* skripte *learner.py* koja opisuje ulogu *Učenika* te dodatne pomoćne *Python* skripte *teacher.py* koji implementira predloženi novi algoritam *Učitelja*. Pritom skripta *teacher.py* automatski poziva alat za provjeru modela *Spin* kako bi uz njegovu pomoć mogla autonomno i istinito odgovoriti na upite *Učenika*. Kroz ovo rješenje se automatski pronalazi model procesa provjere znanja u obliku DFA koji prihvaća sve generirane nizove pitanja.

Osim korištenje modula kroz naredbeni redak omogućen mu je pristup i putem web preglednika. Razvijeno je web sučelje napisano u jeziku *PHP* kojim se omogućava da modul za otkrivanje modela procesa provjere znanja bude dostupan korisnicima u obliku web usluge (slika 7.2). Kroz ovo sučelje korisnici – nastavnici i drugi domenski eksperti mogu zadati ulazne podatke (generirane nizove pitanja) i neobavezne dodatne parametre te na kraju preuzeti konačni rezultat – otkriveni model procesa provjere znanja koji se potom može simulirati i verificirati alatom *Spin*.



Slika 7.2: Web sučelje modula za otkrivanje modela procesa znanja

7.2.1 Implementacija *L** algoritma *Učenika*

Kroz *Python* skriptu *learner.py* u ovom radu je implementirana uloga *Učenika* kao i tijek njegove interakcije s *Učiteljem* sve dok se ne pronađe traženi deterministički konačni automat. Jedini potrebni ulazni podatak za ovu skriptu je skup svih elemenata abecede traženog DFA koji se treba upisati u tekstualnu datoteku *alphabet.txt*. U ovoj implementaciji nema posebnih

ograničenja na elemente abecede, tako da mogu uključivati i spojene nizove znakova. Jedino nije dozvoljen razmak (koristi se za odvajanje elemenata u riječima), malo slovo "e" kojim se označava prazna riječ ε te kontrolni nizovi znakova "yes" i "no". Kao što je ranije napomenuto, u ovom pristupu će se elementima abecede smatrati identifikacijske oznake svih pitanja iz neke automatski generirane provjere znanja. Kako je prikazano u poglavlju 6 datoteka *alphabet.txt* se automatski generira prilikom stvaranja nizova pitanja za formativnu provjeru znanja sa skriptom *prepare_sequences.py*. Dodatni neobavezni ulazni podatak se zadaje s datotekom *log_details_level.txt*, a određuje opširnost ispisa tijeka izvođenja algoritma i sadrži samo jednu od dvije vrijednosti: 1 (za ispis svih podataka) i 0 (za ispis samo osnovnih podataka). Treba napomenuti kako se ovi ulazni podaci mogu unijeti i kroz web sučelje prikazano na slici 7.2.

Kroz skriptu *learner.py Učenik* iterativno šalje upite *Učitelju* i analizira dobivene odgovore. Kroz upite o pripadnosti *Učenik* ispituje *Učitelja* prihvaća li traženi DFA pojedine riječi (nizove pitanja). Na temelju tih odgovora *Učenik* gradi svoju bazu znanja u obliku opservacijske tablice, a kada utvrdi da je opservacijska tablica zatvorena i konzistentna *Učenik* će pokušati pogoditi traženi automat. Poslat će *Učitelju* upit o ekvivalenciji sa svojom pretpostavkom o nepoznatom automatu u tekstualnom *.dot* formatu koji se koristi u alatu za vizualizaciju grafova *Graphviz* [117]. Ako *Učenikova* pretpostavka o traženom automatu nije točna onda će mu *Učitelj* poslati protuprimjer – riječ koju predloženi automat još ne prihvaća ili ne smije prihvaćati i postupak se nastavlja dalje. Nakon konačnog broja iteracija *Učitelj* će prihvatiti *Učenikovu* pretpostavku o traženom automatu i time će se otkrivanje modela procesa provjere znanja uspješno završiti.

Treba napomenuti kako skripta *learner.py* sprema sve izlaze i zapisuje tijek kompletne interakcije učenika i učitelja u posebnu mapu *algorithm_run_data*. Svaka pretpostavka o traženom automatu se automatski sprema u tekstualnom *.dot* formatu, a na kraju se točna pretpostavka sprema i kao SVG, PDF ili PNG datoteka (zadani format je SVG). Cjelokupni skup datoteka generiran prilikom izvođenja *L** algoritma se sprema arhivu koju korisnik može preuzeti kroz web sučelje u obliku komprimirane *zip* datoteke.

Skripta *learner.py* vjerno prati izvorni *L** algoritam Dane Angluin (prikazan kao algoritam 2 u potpoglavlju 3.4.2), ali je nadograđen s dodatnom funkcijom *process_counterexample(word)* za obradu protuprimjera te s pomoćnim funkcijama za ispis opservacijske tablice i čitavog tijeka izvođenja algoritma te za spremanje pretpostavki o traženom DFA različitim oblicima (vlastiti tekstualni zapis, zapis grafa u *.dot* formatu ili slika u SVG ili drugom grafičkom formatu). Treba napomenuti kako se baza znanja o pripadnosti pojedinih riječi traženom DFA sprema u riječnik *T* (ključ je riječ, a vrijednost je 1 ako traženi DFA prihvaća tu riječ, a 0 inače), a struktura opservacijske tablice se dobije kroz operacije nad listama: *listA* (abeceda), *listS* (nazivi redaka gornjeg dijela opservacijske tablice), *listSA* (nazivi redaka donjeg dijela opservacijske tablice) i *listE* (nazivi stupaca opservacijske tablice).

S dodatnom funkcijom *process_counterexample(word)* omogućava se da *Učenik* samostalno

odluči je li dobiveni protuprimjer (riječ *word*) pozitivan ili negativan. Ovaj korak se u izvornom L^* algoritmu implicitno pretpostavlja bez navođenja načina provedbe te odluke. Štoviše, u nekim implementacijama L^* algoritma kao npr. u *libalf* okruženju ovaj korak je preskočen i zamijenjen s dodatnim upitom o pripadnosti s kojim se ispituje prihvaća li traženi DFA protuprimjer *word*. Ovdje će funkcija *process_counterexample(word)* pokušati provesti protuprimjer *word* kroz pretpostavljeni DFA u vlastitom tekstualnom formatu. Ako se na kraju niza dođe u završno stanje onda pretpostavljeni DFA prihvaća protuprimjer *word* iako ne bi smio. Zato se u ovom slučaju radi o negativnom protuprimjeru pa se automatski u bazu znanja upisuje $T[word] = 0$. S druge strane, ako na kraju niza *word* ne dođe u završno stanje, onda ga pretpostavljeni DFA ne prihvaća iako bi ga trebao prihvaćati. Dakle, onda je *word* pozitivan protuprimjer pa se u bazu znanja odmah upisuje $T[word] = 1$. Naravno, nakon obrade dobivenog protuprimjera nastavlja se s izvođenjem algoritma sve dok *Učenik* ne pogodi traženi DFA.

U skripti *learner.py* koriste se i sljedeće važnije funkcije: *answer_membership_query(word)* s kojom *Učitelj* odgovara prihvaća li traženi DFA zadanu riječ *word*, *check_closed()* koja provjerava je li opservacijska tablica zatvorena (ako nije onda se vraća se uzrok), *check_consistent()* koja provjerava je li opservacijska tablica konzistentna (ako nije onda se vraća uzrok), *answer_equivalence_query* kojom *Učitelj* odgovara je li pretpostavka o traženom DFA točna i *proposedDFA()* koja iz zatvorene i konzistentne opservacijske tablice gradi pretpostavku o traženom DFA. Sada se može prikazati algoritam glavne funkcije u skripti *learner.py*:

Algoritam 5 Algoritam glavne funkcije skripte *learner.py*

Inicijaliziraj rječnik $T \leftarrow \{\}$ te liste $listS \leftarrow [\varepsilon]$ i $listE \leftarrow [\varepsilon]$
Dohvati abecedu u listu *listA*
Glavna petlja:
  izračunaj donji dio opservacijske tablice $listSA = listS \cdot listA$
  izračunaj $column1 = listA + listSA$
  Za svaki x u $column1$:
    Za svaki y u $listE$:
      nađi $z = x \cdot y$
      ako $z \notin T$:
        $T[z] = answer\_membership\_query(z)$
  Ako $check\_closed() \neq$ "":
    dodaj u listu $listS$ niz znakova $check\_closed()$ i idi na novu iteraciju glavne petlje
  Ako $check\_consistent() \neq$ "":
    dodaj u listu $listE$ niz znakova $check\_consistent()$ i idi na novu iteraciju glavne petlje
  pozovi funkciju *proposedDFA()* i pretpostavi traženi DFA
  pozovi funkciju *answer_equivalence_query()*
    Ako je vraćen protuprimjer *word*:
      pozovi funkciju *process_counterexample(word)*
    Inače ako je odgovor "yes":
      pozovi funkcije za spremanje rezultata
      zaustavi algoritam

Na kraju treba napomenuti kako će funkcija *proposedDFA()* nakon izgradnje pretpostavke o traženom DFA izgraditi i njegovu sažetu varijantu iz koje će se automatski ukloniti sva neregularna završna stanja iz kojih nema izlaza, baš kao i svi prijelazi koji vode u neregularna završna stanja. Ovako će se dobiti jasniji prikaz konačnog automata, a i dalje se implicitno pretpostavlja da svi nedopušteni prijelazi završavaju u sada skrivenom neregularnom završnom stanju. U potpoglavlju 3.4.2 je prikazan primjer potpunog DFA (slika 3.26) kao i odgovarajućeg DFA u sažetom prikazu (slika 3.27).

7.2.2 Predloženi algoritam *Učitelja* i njegova implementacija

Na početku ovoga istraživanja u prvim verzijama *Python* skripte *learner.py* ulogu *Učitelja* bi preuzeo korisnik – domenski ekspert koji bi odgovarao na upite *Učenika* nakon ručne provjere pripadnosti neke riječi ili ispravnosti pretpostavke o traženom DFA. Kasnije se predložio postupak za formaliziranje i automatiziranje uloge *Učitelja* u obliku *Python* skripte *teacher.py*. Ona se kao modul uključi u skriptu *learner.py* i potom *Učenik* može pozivati funkcije *Učitelja answer_membership_query(word)* i *answer_equivalence_query()* navedene u algoritmu 5.

Kako je specificirano u *L** algoritmu *Učitelj* mora uvijek istinito odgovarati na upite *Učenika*, a kako bi to mogao napraviti na početku mu se mora dati abeceda traženog DFA kao i sve riječi koje DFA treba prihvaćati. U ovom slučaju će abeceda biti skup svih identifikacijskih oznaka pitanja, a riječi će biti svi generirani potpuni i djelomični nizovi pitanja. Opet će se za to koristiti automatski generirane i pripremljene datoteke *alphabet.txt* i *word_list.txt* dobivene sa skriptom *prepare_sequences.py* u postupku automatiziranog generiranja nizova pitanja za formativnu provjeru znanja. Dodatni ulazni podatak je tekstualna datoteka *counterexample_length.txt* s vrijednosti 0 ili 1 kojom se zadaje može li traženi DFA prihvaćati i riječi duže od najduže riječi navedene u *word_list.txt*. Uobičajeno se to zabranjuje (vrijednost 0), ali po potrebi se ovo može dozvoliti s postavljanjem vrijednosti u datoteci *counterexample_length.txt* na 1.

Kod upita o pripadnosti neke riječi traženom DFA *Učitelj* će samo ispitati listu svih riječi iz datoteke *word_list.txt* pa je potom jednostavno točno odgovoriti na taj upit *Učeniku*. Nadalje, kako bi se pouzdano i istinito odgovaralo na *Učenikove* upite o ekvivalenciji u ovom radu se predlaže korištenje alata za provjeru modela *Spin*. Pritom će skripta *teacher.py* iz *Učenikove* pretpostavke o traženom DFA automatski generirati odgovarajuće procesne modele u *Promeli*, a potom će se pozivati alat *Spin* kako bi se provela verifikacija generiranih procesnih modela. Na kraju će se obraditi svi rezultati verifikacije i *Učitelj* će autonomno odgovoriti *Učeniku* s pronađenim protuprimjerom ili s potvrdom da je pogodio traženi DFA čime se algoritam završava.

U predloženom algoritmu *Učitelja* koriste se dva osnovna tipa procesnih modela za verifikaciju u *Promeli* s kojima se mogu pronaći pozitivni ili negativni protuprimjeri. U oba tipa

se koristi mehanizam asinkrone komunikacije s kanalom veličine 1 u koji se šalju i čitaju poruke. Koristi se enumeracija `mtype` kako bi se definirale poruke kao elementi zadane abecede. Primjerice ako su elementi abecede {0, 1, 2, 3} onda će se stvoriti odgovarajuća enumeracija `mtype {m_0, m_1, m_2, m_3}`. Nadalje, kod transformiranja DFA u *.dot* formatu u procesni model u *Promeli* se svako stanje iz DFA modelira kao `if` blok s labelom koja je ista kao i oznaka tog stanja u DFA. Potom se svaki prijelaz iz tog stanja u DFA preslika u grane `if` bloka, a labela svakog od tih prijelaza iz DFA se modelira kao slanje odgovarajuće poruke u asinkroni komunikacijski kanal u *Promeli*. Na kraju svake opcije dolazi naredba skoka `goto` koja vodi na labelu idućega `if` bloka, a odgovara odredišnom stanju tog prijelaza iz DFA. Ako je neko stanje u DFA završno, onda se kod preslikavanja u *Promela* model u `if` blok tog stanja dodaje još jedna opcija koja izvodi naredbu skoka na posebno regularno završno stanje označeno s labelom `end_q`. Na prikazani način se dobije osnovni kostur *Promela* modela koji se potom može automatski nadograditi s kontrolnim varijablama ili zadavanjem traženih svojstava koje model treba zadovoljiti, npr. u obliku `never` ili `notrace` tvrdnji.

Primjena `notrace` tvrdnji

Prvi tip procesnog modela za verifikaciju temelji se na korištenju `notrace` tvrdnji (primjer na slici 3.48). Kako je objašnjeno ranije `notrace` tvrdnja ispituje je zadovoljena ako se niz slanja/primanja poruka zadan tom tvrdnjom nikada ne pojavljuje u izvođenju modela. Ovaj pristup se primjenjuje kod traženja pozitivnih protuprimjera. Pritom će se inicijalno pretpostaviti kako *Promela* model *Učenikove* pretpostavke o traženom DFA ne prihvaća riječi zadane u datoteci *word_list.txt*, a onda će se verifikacijom utvrditi je li to bila istina. Unutar `notrace` tvrdnje se formira `if` blok u koji će se za svaku još nepokrivenu riječ iz datoteke *word_list.txt* dodati nova opcija s nizom slanja poruka koje redom odgovaraju elementima iz te riječ koja završava sa slanjem kontrolne poruke "END" koja označava kraj riječi, odnosno ulazak u završno stanje.

```
mtype {m_0,m_1,END}
chan ch = [1] of {mtype}
active proctype automaton () {
q0:    if
       :: ch!m_0; ch?m_0; goto q1;
       :: ch!m_1; ch?m_1; goto q1;
       :: goto end_q;
       fi;
q1:    if
       :: ch!m_0; ch?m_0; goto q1;
       :: ch!m_1; ch?m_1; goto q0;
       :: goto end_q;
       fi;
end_q: ch!END;
}
notrace {
    if
    :: ch!m_0; ch!m_0; ch!m_0; ch!END;
    :: ch!m_0; ch!m_0; ch!m_1; ch!END;
    :: ch!m_1; ch!m_0; ch!END;
    :: ch!END;
    fi;
}
```

Slika 7.3: Početni primjer modela za verifikaciju korištenjem `notrace` tvrdnje

Kao primjer razmotrit će se početni model *notrace1.pml* sa slike 7.3 u kojem se želi provjeriti na opisani način mogu li se u modelu pojaviti riječi 0 0 0, 0 0 1 i 1 0 te prazna riječ ε. Nakon izvođenja niza naredbi za verifikaciju ovog primjera (*spin -a notrace1.pml*, *gcc -o pan.exe pan.c* te konačno *pan.exe -e* kako bi se detektirale sve greške) verifikator će javiti sistemsku grešku jer u *notrace* tvrdnji postoji nedopušteni nedeterminizam (poruke 0 0 0 i 0 0 1) i prekinuti proces verifikacije. Jedan način rješenja ovoga problema je izgradnja tri *Promela* modela, svaki sa samo jednim nizom slanja poruka u `notrace` tvrdnji. Ovako bi se dobili traženi rezultati, ali bi cijeli postupak mogao biti vrlo dug kod ispitivanja velikog broja riječi jer se svaki put treba provesti prevođenje *Promela* modela u verifikator u jeziku C, potom prevesti verifikator u izvršnu datoteku pa je tek onda pokrenuti. Zato je u ovom istraživanju predložen efikasniji pristup s kojim se eliminira nedeterminizam u `notrace` tvrdnji. Naime, u model će se dodati još jedan kontrolni komunikacijski kanal *ch_start* veličine 1 u koji će se samo upisati početna kontrolna startna poruka (cijeli broj). Broj kontrolnih startnih poruka odgovara broju riječi u `notrace` tvrdnji, a potom će svaka riječ u `notrace` tvrdnji dobiti kao prefiks jedinstvenu kontrolnu startnu poruku. Na slici 7.4 je prikazan popravljeni model *notrace2.pml* u kojem se opet želi provjeriti pojavljuju li se u modelu riječi 0 0 0, 0 0 1 i 1 0 te prazna riječ ε.

```
mtype {m_0,m_1,END}
chan ch = [1] of {mtype}
chan ch_start = [1] of {int}
active proctype automaton () {
       if
       :: ch_start!1;
       :: ch_start!2;
       :: ch_start!3;
       :: ch_start!4;
       fi;

q0:    if
       :: ch!m_0; ch?m_0; goto q1;
       :: ch!m_1; ch?m_1; goto q1;
       :: goto end_q;
       fi;
q1:    if
       :: ch!m_0; ch?m_0; goto q1;
       :: ch!m_1; ch?m_1; goto q0;
       :: goto end_q;
       fi;
end_q: ch!END;
}
notrace {
    if
    :: ch_start!1; ch!m_0; ch!m_0; ch!m_0; ch!END;
    :: ch_start!2; ch!m_0; ch!m_0; ch!m_1; ch!END;
    :: ch_start!3; ch!m_1; ch!m_0; ch!END;
    :: ch_start!4; ch!END;
    fi;
}
```

Slika 7.4: Popravljeni primjer modela za verifikaciju korištenjem `notrace` tvrdnje

Sada će se nakon izvođenja istog niza naredbi za verifikaciju pronaći četiri greške u modelu jer je alat *Spin* dokazao da se sve četiri riječi zaista mogu pojaviti kod izvođenja modela. Ovaj osnovni predloženi princip će se koristiti, uz dodatne optimizacije, kako bi se provođenjem jedne verifikacije otkrile sve riječi koje model *Učenikove* pretpostavke o traženom DFA još ne prihvaća. Dakle, ako verifikator ne javi grešku za neku od riječi u `notrace` tvrdnji to znači da

Učenikova pretpostavka o traženom DFA nije točna jer ne prihvaća sve zadane riječi, a jedna od tih nepokrivenih riječi može se odabrati kao pozitivni protuprimjer koji će se dojaviti *Učeniku*.

Osim prikazanog principa pronalaženja pozitivnih protuprimjera s `notrace` tvrdnjom se mogu pokušati pronaći i negativni protuprimjeri. Naime, u slučaju da je zadano kako traženi DFA ne smije prihvatiti riječ dulju od najduže riječi u datoteci *word_list.txt* onda se može provjeriti s `notrace` tvrdnjom da li to isto vrijedi i za *Učenikovu* pretpostavku o traženom DFA. Pritom će se iskoristiti naredba `ch!_` kojom se pri detektiranju odgovarajućeg niza poruka u `notrace` tvrdnji ignorira sadržaj poslane poruke. Na slici 7.3 je primjer koji se nastavlja na model sa slike 7.3, a želi se provjeriti kako model ne prihvaća nijednu riječ dulju od tri elementa.

```
mtype {m_0,m_1,END}
chan ch = [1] of {mtype}
active proctype automaton () {
q0:    if
       :: ch!m_0; ch?m_0; goto q1;
       :: ch!m_1; ch?m_1; goto q1;
       :: goto end_q;
       fi;
q1:    if
       :: ch!m_0; ch?m_0; goto q1;
       :: ch!m_1; ch?m_1; goto q0;
       :: goto end_q;
       fi;
end_q: ch!END;
}
notrace {ch!_; ch!_; ch!_; ch!_; ch!END;}
```

Slika 7.5: Primjer generiranja negativnog protuprimjera s `notrace` tvrdnjom (*notrace3.pml*)

Nakon pokretanja verifikacije (niz naredbi *spin -a notrace3.pml*, *gcc -o pan.exe pan.c* i *pan.exe*) verifikator će dojaviti pronađenu grešku – u modelu se zaista može pojaviti riječ duljine četiri znaka (bez kontrolnog znaka "END"). Pokretanjem naredbe *spin -t -s notrace3.pml* može se prikazati izvođenje u kojem dolazi do te greške, a to je riječ 0 0 0 0. Ovaj postupak će se iskoristiti kada više nema pronađenih pozitivnih protuprimjera kako bi se osiguralo da pretpostavljeni DFA ne prihvaća i neku dulju riječ ako je to na početku bilo zabranjeno.

Primjena `never` tvrdnji i LTL formula

S drugim tipom verifikacijskih procesnih modela pronalaze se samo negativni protuprimjeri, odnosno riječi koje prihvaća DFA koji je predložio *Učenik*, a traženi DFA ih ne prihvaća. Osnovna struktura modela je ista, iz DFA se svaki prijelaz iz jednog stanja u drugo preslika u prijelaz iz jednog u drugi labelirani `if` blok u *Promela* modelu, a pritom se u komunikacijski kanal pošalje poruka koja odgovara labeli s prijelaza u DFA. Sada se koriste dodatne globalne kontrolne varijable s kojima se prati je li dosegnuto završno stanje *end_q* ($end = 1$) te kontrolni zbroj (*current*) koji pomaže u razlikovanju stanja u sustavu. Svakim prolaskom kroz neki prijelaz se povećava kontrolna suma pa se po njoj mogu razlikovati riječi koje dođu u regularno završno

stanje. Isto tako se cijelo vrijeme kontrolira trenutna duljina riječi (niza poslanih poruka) i ako je riječ preduga onda se odustaje od daljnje potrage za protuprimjerom u tom putu izvođenja. Vremensko svojstvo koje model treba zadovoljiti je zadano s LTL formulom $p\,U\,q$ u obliku odgovarajuće `never` tvrdnje gdje je p definiran kao izraz `length < max` (duljina riječi je manja od maksimalno dopuštene dužine), a q je konjunkcija `end == 1` i svih zadanih ograničenja na iznos kontrolne sume [118]. Sa svakim ograničenjem zadaje se preskakanje već ranije otkrivene regularne riječi koja pripada traženom DFA. Na ovaj način će se sa svakim izvođenjem verifikacije pronaći samo jedan negativni protuprimjer koji će doći do završnog stanja (`end=1`) i pritom će biti duljine najviše `length == max`, a preskočit će se sve druge ranije prihvaćene regularne riječi iz postupka verifikacije korištenjem `notrace` tvrdnje. Demonstracijski primjer na slici 7.6 će se opet nastaviti na model sa slike 7.3. Treba primijetiti kako se sada koristi makronaredba `inline` kako bi se kod svakog slanja poruke kontroliralo stanje sustava.

```
#define p (length < 3)
#define q (end == 1 && current != 0 && current != 105 && current != 152 && current != 169)

inline check(x,y,next_state_status) {
        atomic {
                current = current + (x+3)*(x+7);
                length = length + 1;
                if
                :: length > 3 -> goto end_KO;
                :: else -> ch!y; ch?msg_read
                fi;
                end = next_state_status;
        }
}

mtype {m_0,m_1}
bit end = 0;
int current;
int length;
mtype msg_read;
chan ch = [1] of {mtype}
active proctype automaton () {
q0:    if
       :: check(1,m_0,1) goto q1;
       :: check(2,m_1,1) goto q1;
       :: goto end_q;
       fi;
q1:    if
       :: check(3,m_0,1) goto q1;
       :: check(4,m_1,1) goto q0;
       :: goto end_q;
       fi;
end_q:  end = 1;
end_KO: skip;
}
never  {    /* p U q */
T0_init:
        do
        :: atomic { ((q)) -> assert(!((q))) }
        :: ((p)) -> goto T0_init
        od;
accept_all:
        skip
}
```

Slika 7.6: Primjer traženja negativnog protuprimjera s `never` tvrdnjom (*never.pml*)

U prikazanom modelu su zadana ograničenja za riječi 0 0 0, 0 0 1, 1 0 i za praznu riječ ε. Pretpostavit će se kako su to sve regularne riječi traženog DFA, a sve druge riječi od tri i

manje znakova automat treba odbiti. Treba naglasiti kako se ova ograničenja za regularne riječi računaju automatski računaju prilikom obrade rezultata potrage za pozitivnim protuprimjerima s `notrace` tvrdnjom. Izvođenjem verifikacije ovoga modela (*spin -a never.pml*, *gcc -o pan.exe pan.c* i *pan.exe -a*) verifikator opet dojavi novu grešku – to je riječ 0 koja završava u regularnom završnom stanju. To je sada novi negativni protuprimjer kojeg *Učitelj* može dojaviti *Učeniku* kako bi on pokušao u nekoj idućoj iteraciji konačno pogoditi traženi DFA.

Algoritam funkcije *answer_equivalence_query()*

Prikazat će se algoritam funkcije *answer_equivalence_query()*, glavne funkcije u skripti *teacher.py* s kojom *Učitelj* automatski odgovara na upite o ekvivalenciji:

Algoritam 6 Algoritam funkcije *answer_equivalence_query()* iz skripte *teacher.py*

odgovor: $answer \leftarrow$ ””
potencijalni pozitivni protuprimjer: $potential\_answer\_positive \leftarrow$ ””
potencijalni negativni protuprimjer: $potential\_answer\_negative \leftarrow$ ””
potential_answer_positive = find_positive_counterexample()
Ako je $potential\_answer\_positive = \varepsilon$:
  $answer = potential\_answer\_positive$
Inače ako je $potential\_answer\_positive \neq$ ””:
  Ako je broj prijelaza u modelu > 400:
    $answer = potential\_answer\_positive$
  Inače:
    $length\_negative = len(potential\_answer\_positive) - 1$
    $potential\_answer\_negative = find\_negative\_counterexample(length\_negative)$
      Ako $potential\_answer\_negative! =$ ””:
        $answer = potential\_answer\_negative$
      Inače:
        $answer = potential\_answer\_positive$
Inače:
  Ako je broj prijelaza u modelu ≤ 400:
    $length\_negative = max\_word\_length$
    $potential\_answer\_negative = find\_negative\_counterexample(length\_negative)$
  Ako $potential\_answer\_negative! =$ ””:
    $answer = potential\_answer\_negative$
  Inače:
    Ako je $counterexample\_length\_difference > 0$:
      $answer =$ ””
    Inače:
      $answer = find\_negative\_notrace()$
Ako $answer = potential\_answer\_positive$:
  dodaj *answer* u listu pokrivenih riječi
Ako $answer =$ ””:
  $answer =$ ”*yes*”
vrati *Učeniku* odgovor *answer*

Dakle, na početku se izvodi funkcija *find_positive_counterexample()* kojom se traže pozitivni protuprimjeri pomoću `notrace` tvrdnje, a ta funkcija će vratiti najkraći pozitivni protuprimjer. Ako je protuprimjer prazna riječ ε onda se odmah stavlja u listu pokrivenih riječi i vraća *Učeniku*. Inače ako je nađen neki pozitivni protuprimjer pokušat će se naći neki kraći negativni protuprimjer s funkcijom *potential_answer_negative(length_negative)*, ali samo ako je broj prijelaza u modelu manji od 400. Ova granica je odabrana kako bi se ubrzalo izvođenje algoritma, a traženje negativnog protuprimjera u velikim modelima može trajati predugo. Ako je potom nađen takav kraći negativni protuprimjer onda se on uzima kao protuprimjer koji se vraća *Učeniku*. Inače se *Učeniku* vrati pozitivni protuprimjer.

Ako nije nađen pozitivni protuprimjer (sve riječi u datoteci *word_list* su pokrivene) traži se opet neki negativni protuprimjer. Ako je opet broj prijelaza u modelu manji ili jednak 400 onda se traži negativni protuprimjer iste duljine kao i najduža regularna riječ. Ako je takav negativni protuprimjer nađen vraća se *Učeniku*. A ako nema takvog negativnog protuprimjera još se treba provjeriti prihvaća li predloženi model i neki duži protuprimjer ako je to na početku zabranjeno. Sada se poziva funkcija *find_negative_notrace()* za traženje negativnog protuprimjera uz pomoć `notrace` tvrdnje. Ako je takva riječ pronađena vraća se kao negativan protuprimjer *Učeniku*.

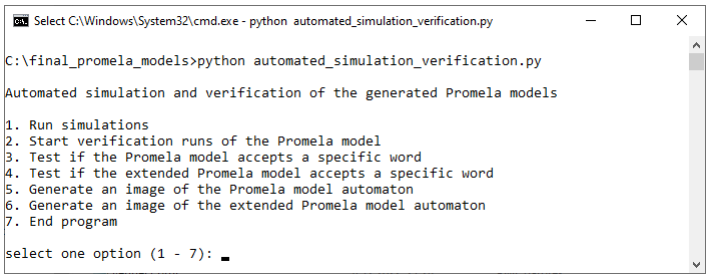
A ako na kraju nakon svih pokušaja pronalazaka pozitivnih i negativnih protuprimjera nije pronađen niti jedan *Učeniku* se vraća odgovor "yes" kao potvrda da je pogodio traženi DFA.

Na kraju treba naglasiti kao skripta *teacher.py* koristi različite optimizacijske opcije alata *Spin* kojima se može ubrzati izvođenje implementiranog *L** algoritma. Kod prevođenja svih verifikatora u izvršni oblik koristi se zastavica "-DBITSTATE" kojom *Spin* koristi poseban algoritam za stvaranje efikasne tablice raspršenog adresiranja. Nadalje, kod traženja svih pozitivnih protuprimjera se kod prevođenja verifikatora postavlja i zastavica "-DPRINTF" s kojom se prilikom verifikacije ispisuju korisničke kontrolne `printf` naredbe s kojima će se odmah pročitati koje se sve riječi pojavljuju u modelu. Na kraju se kod traženja negativnog protuprimjera s funkcijom *potential_answer_negative(length_negative)* koristi se i zastavica "-I" kod pokretanja verifikatora *pan.exe -a -I* s kojom će verifikator tražiti približno najkraći put do greške. Isto tako se kod prevođenja verifikatora u izvršni oblik koristi alat *tcc* kojim se pokazao kao značajno brži od standardnog *gcc* prevodioca [119].

7.3 Simulacija i verifikacija modela procesa provjere znanja

Na kraju treba naglasiti kao skripta *teacher.py* na kraju procesa učenja traženog DFA generira i skup simulacijskih i verifikacijskih modela kojima se može na detaljan način ispitati pronađeni DFA u obliku *Promela* modela. Kako bi se olakšao proces verifikacije i simulacije izrađena je i pomoćna *Python* skripta *automated_simulation_verification.py*. Funkcija pruža korisnicima jednostavan način za pokretanje verifikacije svih verifikacijskih modela koji koriste različite

LTL formule ili `assert` tvrdnje kako bi se osiguralo da prihvaćaju sve generirane nizove pitanja. Na slici 7.7 je prikazan glavni izbornik ove skripte.



Slika 7.7: Glavni izbornik skripte za simulacije i verifikacije pronađenih modela u *Promeli*

Isto tako omogućeno je i izvođenje simulacija simulacijskih modela kojima se na slučajan način izvode pronađeni *Promela* model. Dodatno je omogućeno izvođenje i spremanje veće broja uzastopnih simulacija čime se lako može stvoriti veliki skup scenarija izvođenja formativne provjere znanja.

Na kraju pomoćna skripta omogućava automatsko generiranje slike pronađenog modela u *Promeli* u različitim grafičkim formatima. Pritom se pokreće verifikator s naredbom *pan.exe -D* čime se automatski stvara graf procesnog modela u već spomenutom *.dot* formatu.

7.3.1 Automatsko proširenje modela procesa provjere znanja

Kako bi se što vjernije opisao proces provjere znanja na kraju se pronađeni DFA može automatski proširiti s dodatnim prijelazima koji definiraju ponašanje sustava za provjeru znanja u slučaju pogrešnog odgovora, odnosno ponavljanje pitanja uz povratak na prethodno pitanje. Zato se kroz skriptu *teacher.py* automatski dodaju povratni prijelazi za svaki prijelaz definiran u osnovnom pronađenom DFA. Dobiveni prošireni konačni automat se potom automatski transformira u simulacijski i verifikacijski procesni model u jeziku *Promela* te se automatska simulacija i verifikacija nadograđenog modela vrši opet alatom *Spin*. Kao što se može vidjeti na slici 7.7 kroz pomoćnu skriptu *automated_simulation_verification.py* može se verificirati nadograđeni modela procesa provjere znanja, a nakon odabira prve opcije bit će ponuđeno izvođenje simulacija tog nadograđenog modela.

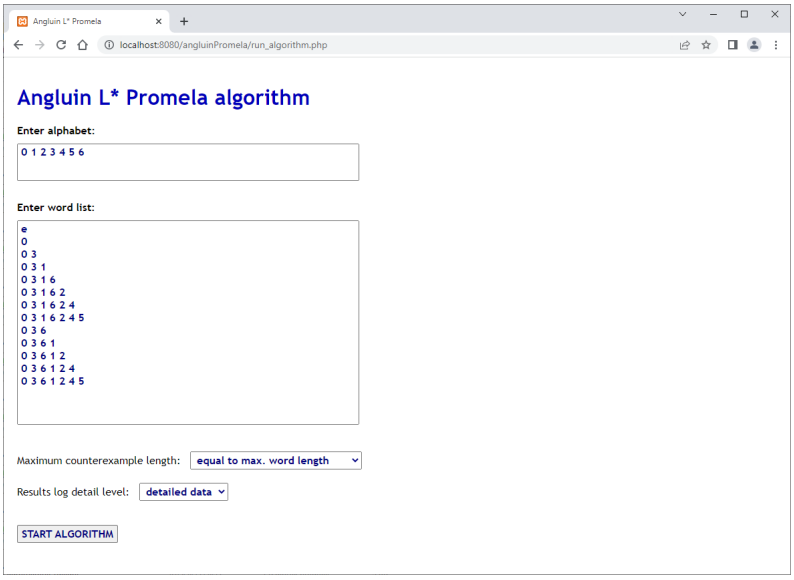
7.4 Rezultati i rasprava

Na početku treba napomenuti kako je već ranije u potpoglavlju 3.4.2 prikazan primjer izvođenja predloženog algoritma za otkrivanje deterministčkog konačnog automata koji prihvaća neki zadani regularni jezik. A u ovom potpoglavlju će se prikazati rezultati verifikacije metode za automatiziranu provjeru znanja. Ovdje će se radi jednostavnosti prvo prikazati svi rezultati na

jednostavnijem primjeru kraće provjere znanja, a potom će se prikazati i rezultati simulacije i verifikacije procesa formativne provjere znanja na primjeru velikog skupa od 888 automatski generiranih potpunih i djelomičnih nizova pitanja iz poglavlja 6.

7.4.1 Rezultati verifikacije i simulacije kraće provjere znanja

Za početak će se prikazati rezultati stvaranja DFA iz manjeg skupa generiranih nizova pitanja primjenom predloženog nadograđenog algoritma *L**. Npr. neka je zadana abeceda {0, 1, 2, 3, 4, 5, 6} gdje svaki znak odgovara jedinstvenom identifikatoru ispitnog pitanja, a nakon provođenja metode FCA i topološkog sortiranja dobiveni su sljedeći dozvoljeni nizovi pitanja: ε (prazna riječ), 0, 0 3, 0 3 1, 0 3 1 6, 0 3 1 6 2, 0 3 1 6 2 4, 0 3 1 6 2 4 5, 0 3 6, 0 3 6 1, 0 3 6 1 2, 0 3 6 1 2 4 i 0 3 6 1 2 4 5. Ovaj primjer se koristio u prethodnom vlastitom istraživanju [89] gdje se L* algoritam provodio ručno uz pomoć okruženja *libalf*. Na slici 7.8 je prikaz web sučelja s upisanim ulaznim podacima.

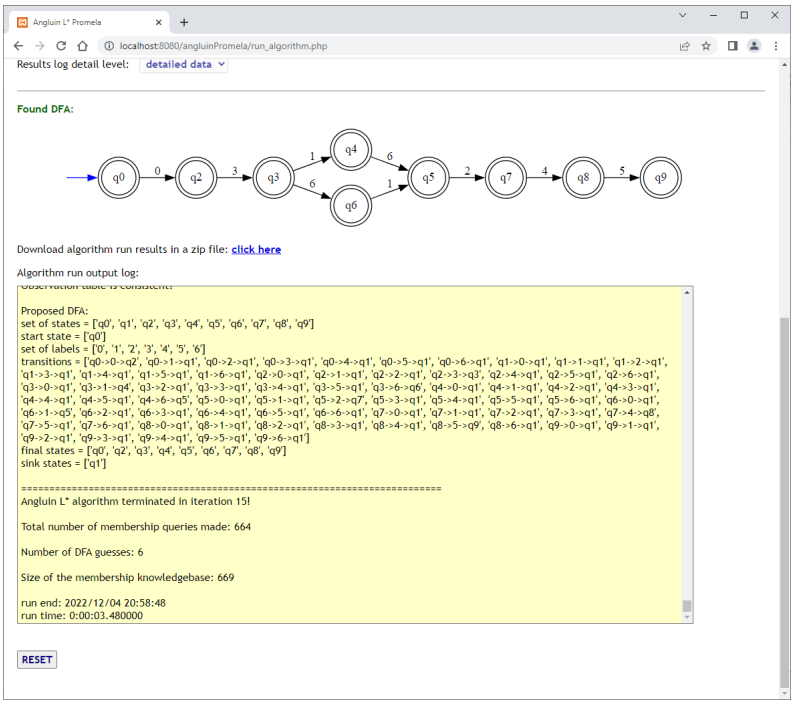


Slika 7.8: Web sučelje s upisanim ulaznim podacima

Na slici 7.9 je prikaz dobivenih rezultata kroz web sučelje. Pronađeni DFA je jednak onome koji dobiven u [89] kroz dugotrajni dijalog s *L** algoritmom u okruženju *libalf*. U ovom slučaju traženi DFA je pogođen iz 6. pokušaja, a pritom je Učenik postavio i 664 različita upita o pripadnosti pojedinih riječi traženom DFA. Uz to je Učenik u svoju bazu znanja samostalno ispravno odlučio još o statusu redom pet dobivenih protuprimjera u cijelom postupku učenja DFA pa se baza znanja sastoji od ukupno 669 elemenata, a traženi DFA je generiran nakon ukupno 3,48 s. Iz detaljnog prikaza ispisa algoritma može se iščitati i potpuni zapis traženog DFA s neregularnim završnim stanjem q_1, a koji je puno složeniji od odabranog sažetog prikaza grafa automata na slici 7.9. Isto tako, može se primijetiti kako među stanjima sažetog DFA nema neregularnog stanja q_1 koje je ostalo skriveno. Isto tako iz prikaza traženog DFA može

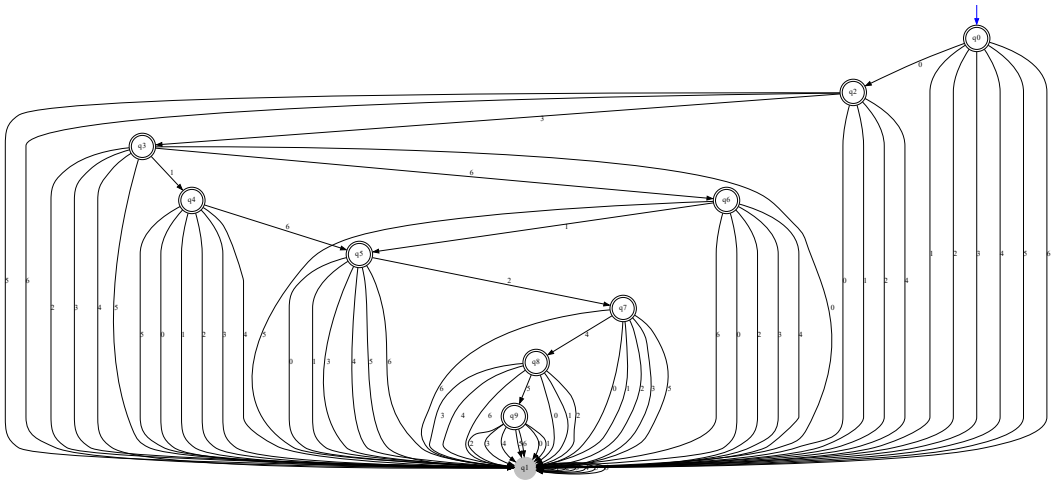
se lako vidjeti kako su u odabranoj kraćoj provjeri znanja moguća dva različita potpuna puta postavljanja pitanja: 0 3 6 1 2 4 5 i 0 3 1 6 2 4 5.

Treba naglasiti kako se u web aplikaciji može putem poveznice preuzeti *zip* datoteka s potpuni zapisom izvođenja algoritma i sa svim međurezultatima te s pomoćnom *Python* skriptom za izvođenje simulacija i verifikacija konačnih *Promela* modela.



Slika 7.9: Web sučelje s prikazom rezultata izvođenja algoritma

Na slici 7.10 je potpuni prikaz pronađenog DFA koji uključuje sivo osjenčano neregularno završno q_1. Može se isto tako vidjeti kako su završna stanja označena s dvostrukim koncentričnim krugovima, a inicijalno stanje je naznačeno s ulaznom plavom strelicom bez oznake.



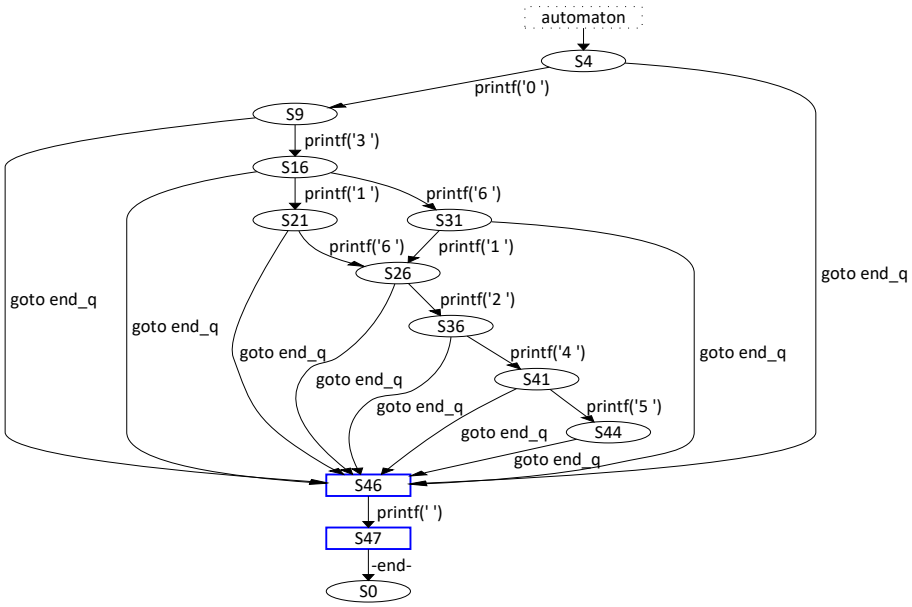
Slika 7.10: Pronađeni deterministički konačni automat – potpuni prikaz s neregularnim završnim stanjem

Na slici 7.11 je automatski dobiveni simulacijski procesni model u *Promeli* za pronađeni DFA prikazan u sažetom obliku na slici 7.9.

```
active proctype automaton () {
q0:     if
        :: printf("0 "); goto q2;
        :: goto end_q;
        fi;
q2:     if
        :: printf("3 "); goto q3;
        :: goto end_q;
        fi;
q3:     if
        :: printf("1 "); goto q4;
        :: printf("6 "); goto q6;
        :: goto end_q;
        fi;
q4:     if
        :: printf("6 "); goto q5;
        :: goto end_q;
        fi;
q5:     if
        :: printf("2 "); goto q7;
        :: goto end_q;
        fi;
q6:     if
        :: printf("1 "); goto q5;
        :: goto end_q;
        fi;
q7:     if
        :: printf("4 "); goto q8;
        :: goto end_q;
        fi;
q8:     if
        :: printf("5 "); goto q9;
        :: goto end_q;
        fi;
q9:     if
        :: goto end_q;
        fi;
end_q:  printf("\n");
}
```

Slika 7.11: Simulacijski *Promela* model za DFA na slici 7.9

Na idućoj slici 7.12 je prikaz simulacijskog procesnog modela u *Promeli* sa slike 7.11 u obliku konačnog automata koji je generiran automatski uz pomoćne *Python* skripte za simulaciju i verifikaciju pronađenih modela u *Promeli*.



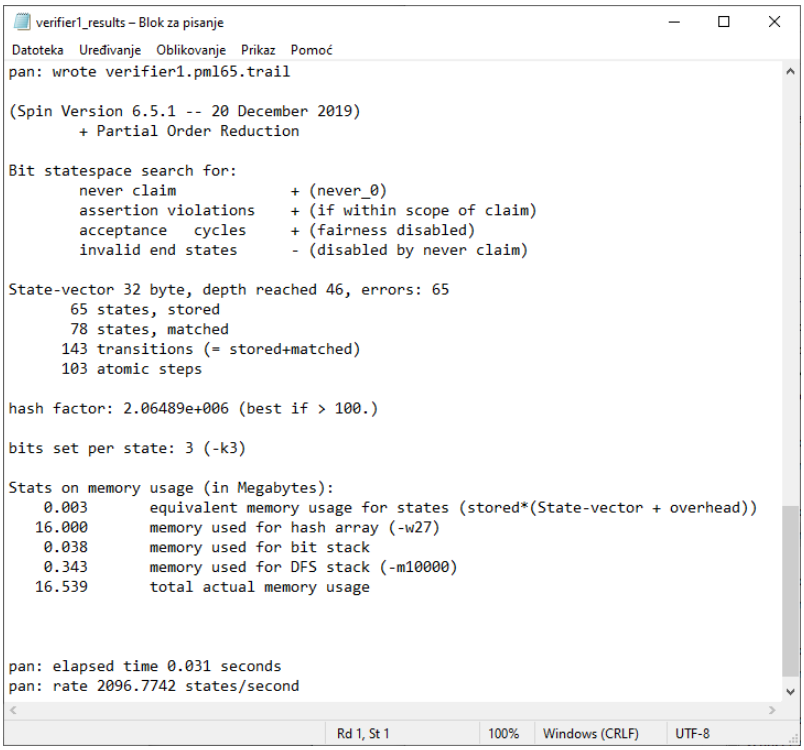
Slika 7.12: Konačni automat simulacijskog *Promela* modela sa slike 7.11

U nastavku će se prikazati jedan od 7 različitih verifikatora koji se automatski generiraju nakon otkrivanja traženog DFA. Kod prvog se zadaje svojstvo `assert` tvrdnjom, u sljedeća četiri se zadaju svojstva kao LTL formule u obliku `never` tvrdnji, a u zadnja dva verifikatora se koristi `notrace` tvrdnja. Prvih pet verifikatora traži sve riječi koje u traženom DFA dolaze u završno stanje, a šesti i sedmi verifikator ispituje da li DFA odnosno nadograđeni DFA prihvaća neku zadanu riječ. Na slici je prikazan drugi verifikator (*verifier1.pml*) u kojemu je zadano svojstvo LTL formulom $\Diamond(p \wedge q)$: napslijetku će vrijediti p (dužina riječi najviše 7 elemenata) i q (dosegnuto je završno stanje).

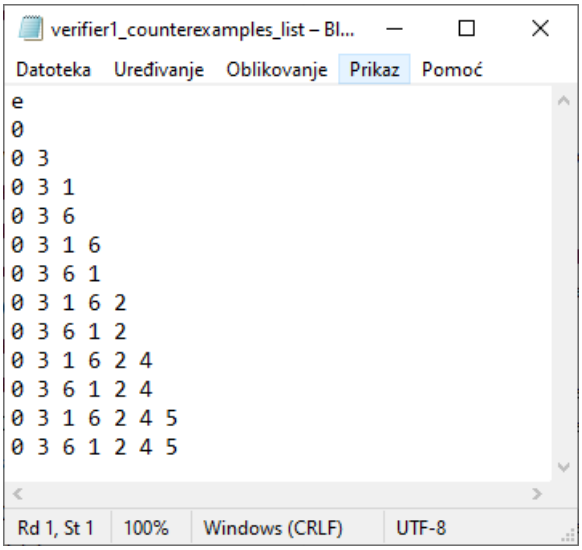
```
#define p (length <= 7)
#define q (end == 1)
inline check(x,y) {
        atomic {
                current = current + (x+3)*(x+7);
                if
                :: length >= 7 -> goto end_KO;
                :: else -> labels[length] = y;
                fi;
                length = length + 1;
        }
}
mtype {m_0,m_1,m_2,m_3,m_4,m_5,m_6}
bit end = 0;
int current;
int length;
mtype labels[7];
int i;
active proctype automaton () {
q0:     if
        :: check(1,m_0); goto q2;
        :: goto end_q;
        fi;
q2:     if
        :: check(2,m_3); goto q3;
        :: goto end_q;
        fi;
q3:     if
        :: check(3,m_1); goto q4;
        :: check(4,m_6); goto q6;
        :: goto end_q;
        fi;
q4:     if
        :: check(5,m_6); goto q5;
        :: goto end_q;
        fi;
q5:     if
        :: check(6,m_2); goto q7;
        :: goto end_q;
        fi;
q6:     if
        :: check(7,m_1); goto q5;
        :: goto end_q;
        fi;
q7:     if
        :: check(8,m_4); goto q8;
        :: goto end_q;
        fi;
q8:     if
        :: check(9,m_5); goto q9;
        :: goto end_q;
        fi;
q9:     if
        :: goto end_q;
        fi;
end_q:  atomic {
                printf("START ");
                for (i : 0 .. length-1) {
                        printm(labels[i]);
                        printf(" ");}
                printf("END\n");
                end = 1;
        }
end_KO: skip;
}
never  {    /* <>(p && q) */
T0_init:
        do
        :: atomic { ((p && q)) -> assert(!((p && q))) }
        :: (1) -> goto T0_init
        od;
accept_all:
        skip
}
```

Slika 7.13: Verifikacijski *Promela* model za DFA na slici 7.9

Rezultati izvođenja verifikacije modela sa slike 7.13 prikazani su na slici 7.14. Verifikacija je automatski pokrenuta kroz pomoćnu *Python* skriptu sa sljedećim nizom naredbi *spin -a verifier1.pml*, *tcc -o pan.exe pan.c -DBITSTATE -DPRINTF* te konačno *pan.exe -e -a -n -c5000*. Vidi se kako je detektirano 65 grešaka, ali sve se one odnose na zadanih 13 riječi (svaka riječ je detektirana u pet navrata) i zaključuje se da ovaj DFA ne prihvaća niti jednu drugu riječ. Pritom su iskorišteni minimalni memorijski resursi (zbog korištenja zastavice "-DBITSTATE") i verifikacija je odrađena za 0,031 s. Popis svih 13 protuprimjera prikazana je na slici 7.15.

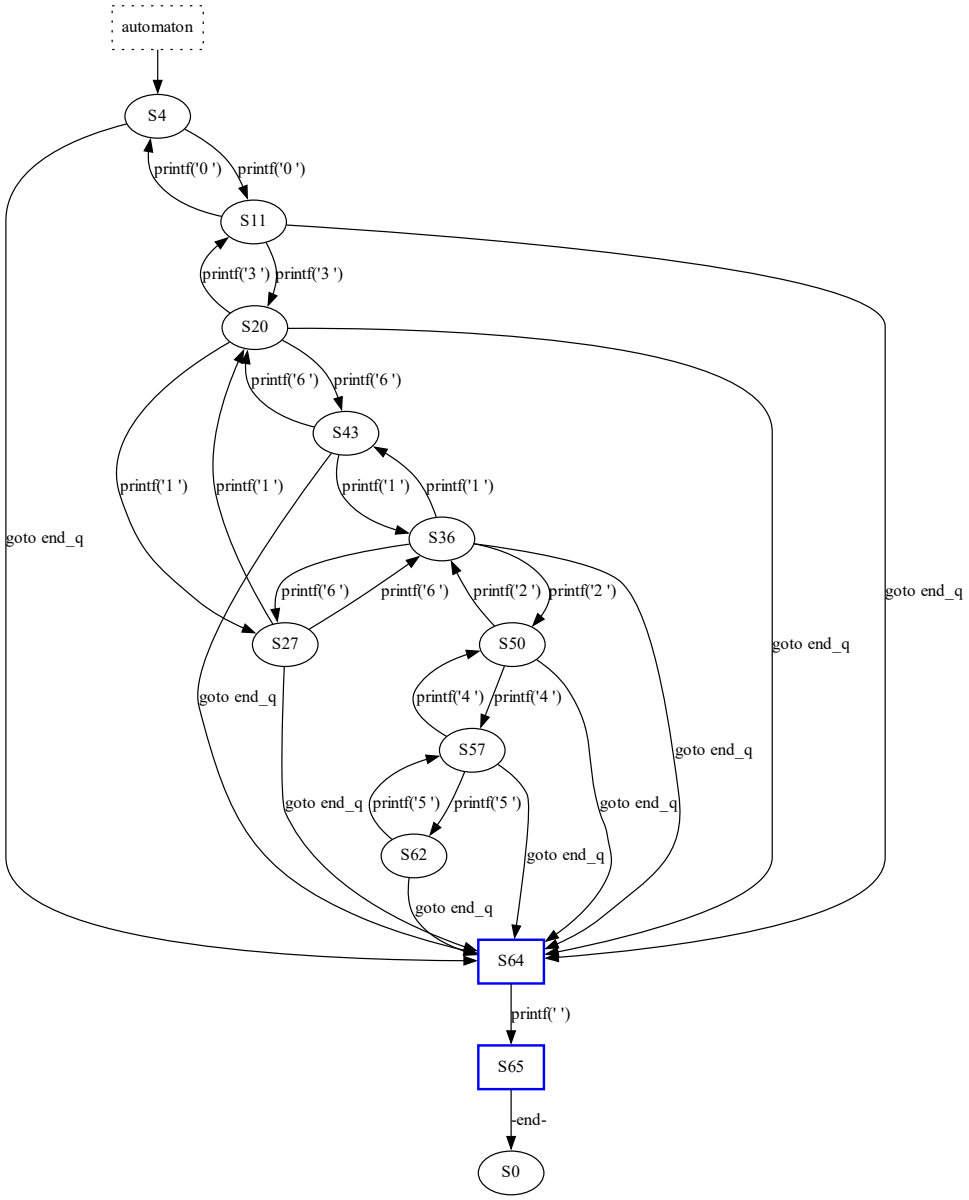


Slika 7.14: Rezultati verifikacije modela sa slike 7.13



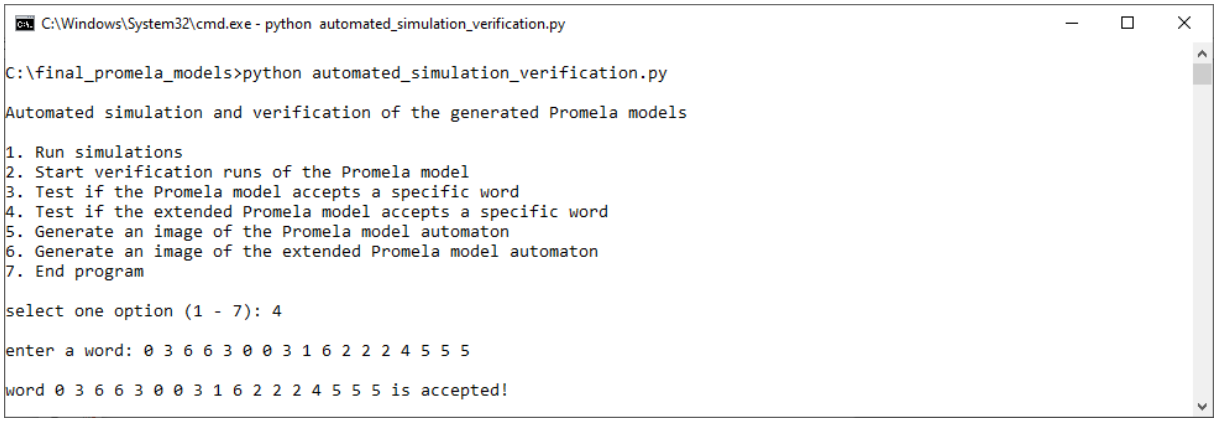
Slika 7.15: Popis riječi koje prihvaća pronađeni DFA sa slike 7.13

Na kraju će se prikazati rezultati simulacija i verifikacije nadograđenog konačnog automata iz otkrivenog DFA, a koji vjernije može opisati ponašanje procesa provjere znanja u sustavu za e-učenja. Za svaki prijelaz iz nekog stanja A u stanje B ($A \xrightarrow{x} B$) dodaje se povratna grana, odnosno prijelaz $B \xrightarrow{x} A$. Nadograđeni simulacijski model će tako bolje opisati ponašanje procesa provjere znanja jer se nakon netočnog odgovora studentu će nova inačica istog ili sličnog pitanja te se vraća na prethodno stanje u modelu. Na slici 7.16 je automatski dobiveni grafički prikaz nadograđenog simulacijskog procesnog modela u *Promeli* kojeg se može usporediti s grafom izvornog simulacijskog modela sa slike 7.12.



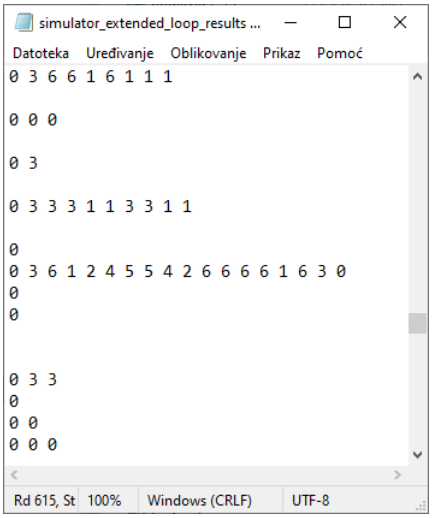
Slika 7.16: Konačni automat nakon nadogradnje automata sa slike 7.12

Skripta *automated_simulation_verification.py* pruža jednostavno sučelje putem koje korisnik može ispitati prihvaćaju li generirani modeli u *Promeli* neku proizvoljno zadanu riječ. U ovom slučaju koristi se oblik verifikatora s *notrace* tvrdnjom kao npr. na slici 7.5. Nakon što skripta utvrdi kako se riječ sastoji samo od dozvoljenih simbola alfabeta automatski se pripremi verifikator tako što se generira odgovarajući `notrace` tvrdnja za zadanu riječ te se pokreće verifikacija i potom obrađuje rezultat i vraća odgovor korisniku. Primjer jednoga takvog upita i odgovora je na slici 7.17. Korisnik želi ispitati prihvaća li nadograđeni model u *Promeli* prema slici 7.16 riječ 0 3 6 6 3 0 0 3 1 6 2 2 2 4 5 5 5, a skripta mu je vratila potvrdan odgovor. Pažljivim praćenjem ponašanja nadograđenog simulacijskog modela na slici 7.16 može se i ručno provjeriti da se ova riječ prihvaća.



Slika 7.17: Provjera prihvaćanja zadanog niza pitanja u nadograđenom *Promela* modelu prema slici 7.16

Naposljetku, prikazat će se rezultat simulacija nadograđenog modela u *Promeli* sa slike. Opet se koristi automatizirana skripta *automated_simulation_verification.py* za pokretanje 1000 slučajnih simulacija čiji se rezultati spremaju u datoteku *simulator_extended_loop_results.txt*. Mali uzorak iz te datoteke je prikazan na slici 7.18.

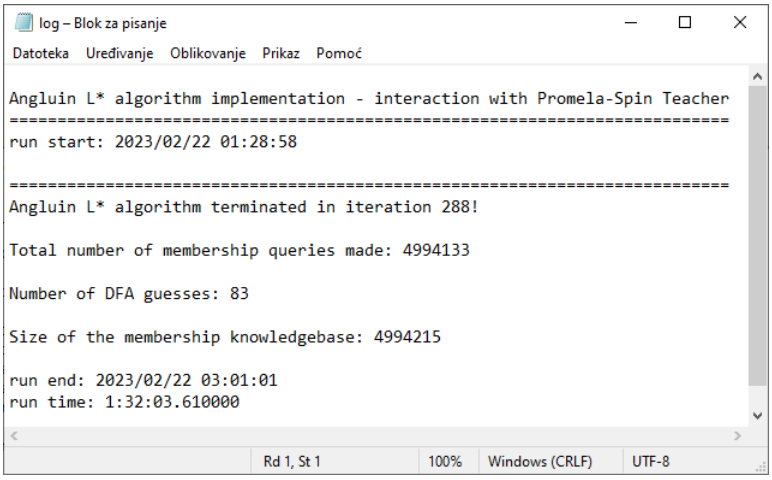


Slika 7.18: Uzorak datoteke s rezultatima 1000 slučajnih simulacija nadograđenog modela sa slike 7.16

7.4.2 Rezultati verifikacije i simulacije formativne provjere znanja

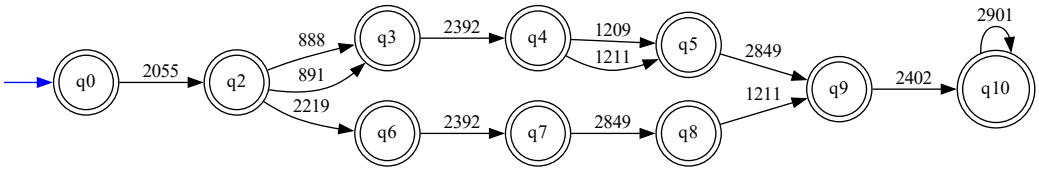
Nakon pregleda svih rezultata verifikacija i simulacija na jednostavnijim primjerima u ovom potpoglavlju će se predstaviti verifikacija metode za automatiziranu provjeru znanja nad rezultatima na primjeru duže formativne provjere znanja i to one automatski generirane formativne provjere znanja (*OE - provjera 1*) iz poglavlja 6. Kao što je ranije objašnjeno u tom slučaju su se automatski generiralo 888 različitih potpunih i djelomičnih nizova pitanja. Među njima se nalazi 16 različitih potpunih putova, svaki sa 103 pitanja. Ukupno se u generiranim nizovima pitanja nalazi ukupno 117 različitih osnovnih varijanti pitanja. Skripta *prepare_sequences.py* je generirala dvije datoteke koje će se iskoristiti kao ulaz u predloženi postupak verifikacije. To je datoteka *alphabet.txt* sa svim elementima abecede (117 jedinstvenih identifikatora pitanja) i datoteka *word_list.txt* sa svih 888 riječi (svi djelomični i potpuni nizovi pitanja kao i prazna riječ ε).

Sažetak rezultata učenja DFA koji prihvaća svih 888 mogućih riječi ili nizova pitanja prikazan je na slici 7.19. Kao što se može primijetiti točan DFA je pronađen u 83. pokušaju, a baza znanja sadrži podatke o pripadnosti za ukupno čak 4 994 215 različitih riječi. Sam proces učenja DFA s nadograđenim L^* algoritmom je trajao dosta dugo (1 sat i 32 minute), a pritom se najviše vremena trošilo na upite o pripadnosti, kao i na izgradnju ogromnih opservacijskih tablica za koje se učestalo trebalo provjeravati jesu li zatvorene i konzistentne. Isto tako dosta vremena se trošilo na izgradnju *.dot* datoteka trenutnih grafova, a od toga najviše na zapis velikih grafova potpunih DFA automata. Koliko se primijetilo samo izvođenje alata za provjeru modela *Spin* je bilo relativno brzo u svim iteracijama nadograđenog L^* algoritma i memorijski nije bilo zahtjevno. S druge strane, skripta *learner.py* je memorijski bila vrlo zahtjevna zbog izgradnje ogromne baze znanja te zbog pripreme i provjere opservacijskih tablica. Radi jednostavnosti u ovom slučaju nije bio odabran opširni zapis tijeka izvođenja algoritma jer bi se pritom stvorila ogromna dnevnička tekstualna datoteka.

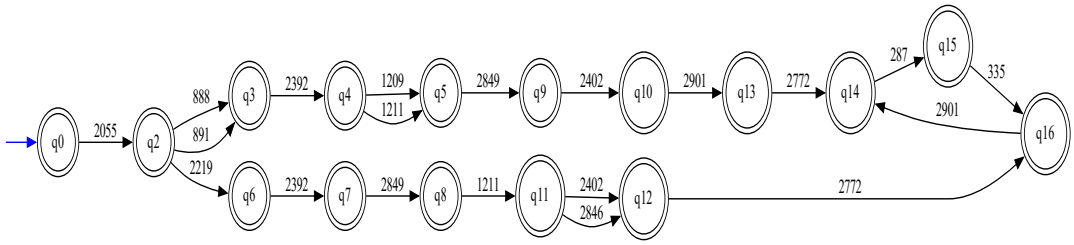


Slika 7.19: Sažetak rezultata učenja DFA za formativnu provjeru s 888 nizova pitanja

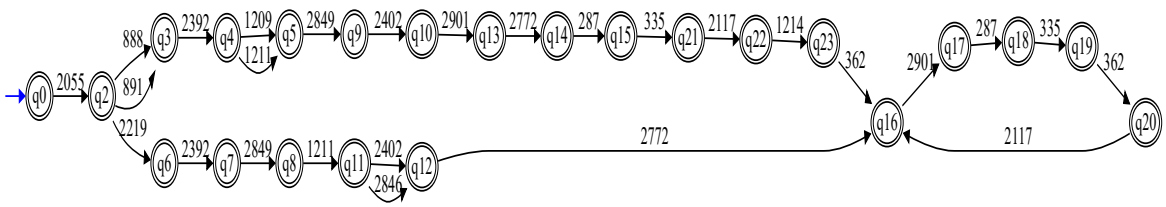
Na slikama 7.20, 7.21 i 7.21 prikazane su *Učenikove* pretpostavke o traženom DFA, i to redom 5., 8. i 10. pretpostavka. U svim navedenim pretpostavkama bilo je detektirano neregularno završno stanje pa je ono sakriveno prije pretvaranja DFA u *Promela* model koji se verificirao u potrazi za protuprimjerima. Isto tako vidljivo je kako s novim podacima o pitanjima u nizu raste složenost prikazanih konačnih automata.



Slika 7.20: 5. pretpostavka o traženom DFA

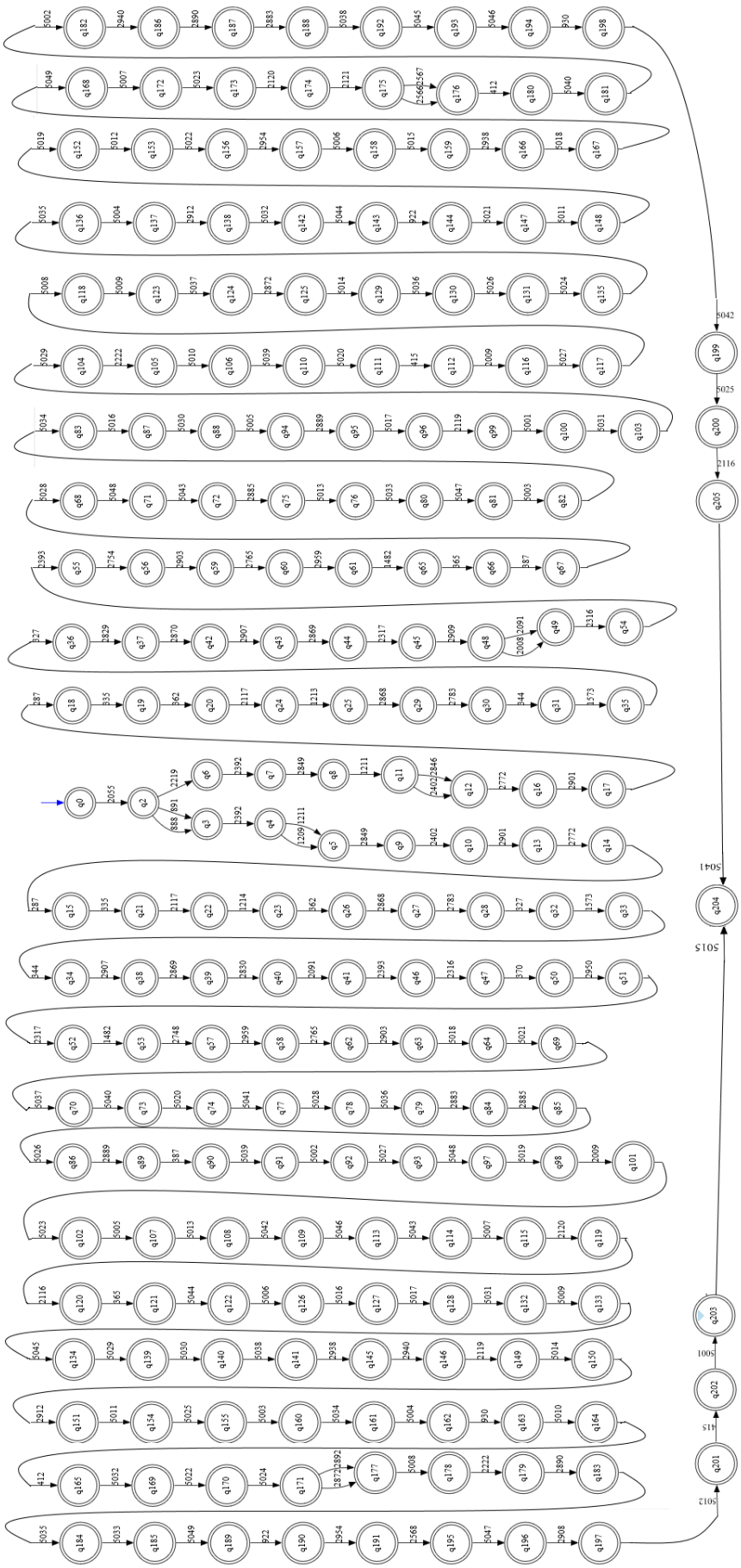


Slika 7.21: 8. pretpostavka o traženom DFA



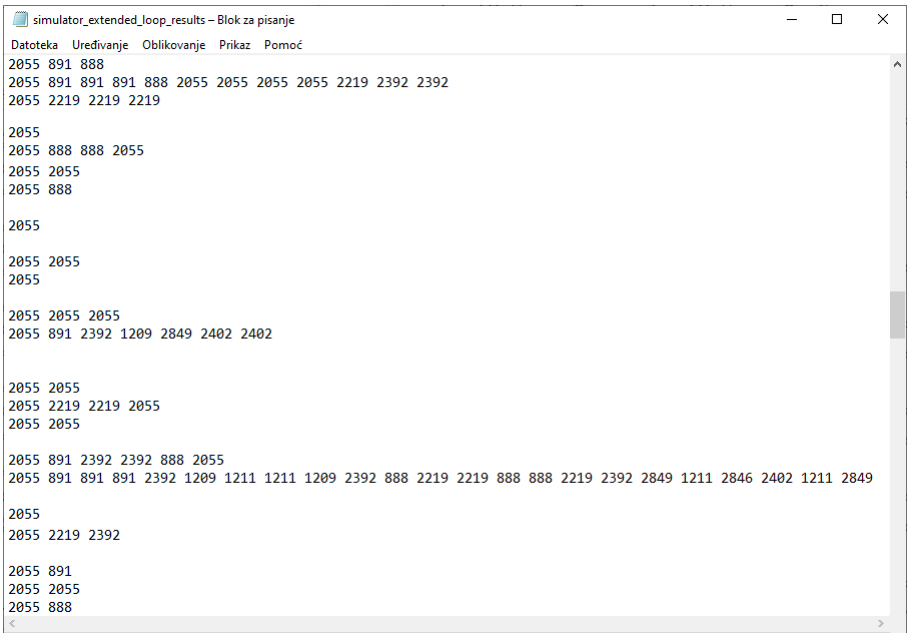
Slika 7.22: 10. pretpostavka o traženom DFA

U 83. pokušaju *Učenik* je pogodio traženi DFA koji je prikazan na slici 7.23. Taj DFA automat ima 206 stanja i 211 prijelaza, a u njemu se mogu identificirati dva osnovna redoslijeda čvorova koji odgovaraju generiranim topološkim sortiranjima formalnih koncepata odnosno *EKP* točaka iz konceptualne rešetke iz sažetog skupa od 139 pitanja.



Slika 7.23: Traženi DFA formativne provjere znanja ispravno pogođen u 83. pokušaju

Nakon pronalaženja traženog DFA na slici 7.23 skripta *teacher.py* automatski priprema i simulacijske i verifikacijske modele u *Promeli* za detaljnu analizu pronađenog DFA formativne provjere znanja, ali i nadograđenog DFA s povratnim vezama u slučaju krivog odgovora. Primjerice, korištenjem pomoćne skripte *automated_simulation_verification* izvedeno je i spremljeno 1000 slučajnih simulacija nadograđenog DFA formativne provjere znanja. Rezultati simulacija su prikazani na slici 7.4.2, a pružaju nastavnicima i ekspertima za oblikovanje sustava za e-učenje korisni izvor podataka o mogućim putevima izvođenja formativne provjere znanja.

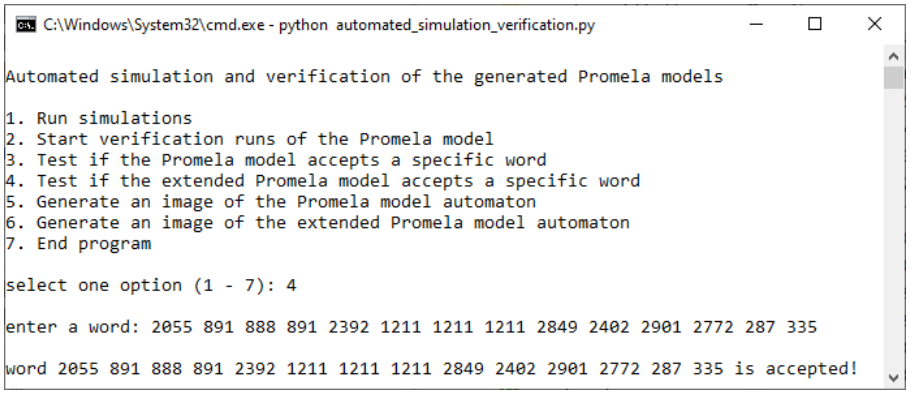


Slika 7.24: Slučajne simulacije nadograđenog *Promela* modela formativne provjere znanja

Vidi se kako su generirani nizovi pitanja ili prazni (ε) ili počinju s pitanjem s identifikacijskom oznakom 2055. Kao što se vidi iz pronađenog DFA na slici upravo s tim pitanjem počinju sva izvođenja ove formativne provjere znanja. Može se primijetiti kako je u nadograđenom DFA moguće prijeći iz jednog osnovnog puta rješavanja u drugi – u stanju q_2 se ti osnovni putevi razdvajaju, ali kod vraćanja na prethodno pitanje u automatu se može prijeći iz jednog puta u drugi. Treba naglasiti kako u trenutnom modulu za formativnu provjeru znanja to nije moguće nego će se formativna provjera uvijek rješavati po početno slučajno odabranom putu. Ali, u budućem radu će se razmotriti i implementacija ove značajke DFA u skriptu za vođenje formativne provjere znanja.

Dodatno će se provjeriti niz pitanja iz zapisa jednog pokušaja rješavanja ove formativne provjere znanja koji je prikazan na slici 6.27. Identifikacijske oznake pitanja koja su se postavljala na tom putu rješavanja pronašle su se u tablici *formative_tests_data*, a onda se taj niz pitanja (2055, 891, 888, 891, 2392, 1211, 1211, 1211, 2849, 2402, 2901, 2772, 287, 335) krenuo ispitivati sa skriptom *automated_simulation_verification*. Pritom je odabrana 4. opcija testiranja riječi u nadograđenom *Promela* modelu, čime se pokrenula verifikacija modela s `notrace` tvrd-

njom koju je skripta automatski generirala prema zadanom nizu pitanja. Kao što se vidi na slici 7.25 ovaj niz pitanja je dozvoljen, odnosno on predstavlja riječ koju deterministički konačni automat ove formativne provjere znanja prihvaća.



Slika 7.25: Provjera prihvaćanja zadanog niza pitanja u nadograđenom *Promela* modelu formativne provjere znanja

Na kraju se izvršila verifikacija modela formativne provjere znanja sa slike 7.23 kako bi se otkrili svi nizovi pitanja koje prihvaća deterministički konačni automat ovoga modela. Pritom se koristila pomoćna skripta za automatsku simulaciju i verifikaciju prikazana na slici 7.25, a odabrala se opcija 2 iz njenog početnog izbornika. Nakon otkrivanja DFA automatski se generirao predložak verifikacijskog modela u *Promeli* u kojem su ovom slučaju definirani izrazi p kao `length <= 103` (dužina riječi je manja ili jednaka 103) i q kao `end == 1` (dosegnuto je završno stanje). Iz tog predloška automatski je stvoreno pet različitih verifikacijskih modela u *Promeli* – svaki s drukčije zadanim svojstvom. U prvom modelu to je dodatni *Promela* proces sa zadanom tvrdnjom `assert(!(p && q))`, a u preostalim modelima to je `never` blok koji odgovara redom formulama LTL logike: $\Diamond(p \wedge q)$, $\Diamond\Box(p \wedge q)$, $\Box\Diamond(p \wedge q)$ i $p\,U\,q$. Pomoćna skripta automatski pokreće verifikaciju svih pet modela s alatom *Spin* i automatski obrađuje sve dobivene rezultate. Kroz verifikaciju svih pet verifikacijskih modela potvrđeno je kako model formativne provjere na slici 7.23 prihvaća svih 888 parcijalnih i potpunih nizova pitanja. Za primjer će se prikazati rezultati verifikacije provedeni s petim verifikacijskim modelom u *Promeli* (*verifier4.pml*). U tom slučaju je svojstvo zadano kao `never` blok koji odgovara formuli LTL logike $p\,U\,q$. Pomoćna skripta je automatski izvela sljedeći niz naredbi: *spin -a verifier4.pml*, *tcc -o pan.exe -DBITSTATE -DPRINTF pan.c* i *pan.exe -e -a -n -c5000 > verifier4_results.txt* s kojima se iz *Promela* modela *verifier4.pml* stvorio verifikator u jeziku C (*pan.c*) pa je potom pretvoren u izvršnu datoteku verifikatora (*pan.exe*) i pokrenut. Iz spremljenih rezultata se vidi kako je prilikom verifikacije generirano 4440 protuprimjera, verifikacija je trajala 7,16 s, a za nju je utrošeno ukupno samo 16,832 MB radne memorije. Daljnjom automatskom obradom svih protuprimjera otkriveno je da se odnose na svih 888 nizova pitanja (po pet protuprimjera za svaki niz pitanja) odnosno riječi koje DFA prikazan na slici 7.23 prihvaća.

Poglavlje 8

Zaključak

U ovom radu predložena je metoda za automatiziranu pripremu i vođenje formativnih provjera znanja u sustavima za e-učenje. Glavni preduvjet za primjenu predložene metode je definiranje skupa atributa koji opisuje zadano nastavno gradivo te priprema odabranog skupa ispitnih pitanja označenih s takvim atributima. Nakon toga se gradi početni formalni kontekst, odnosno binarna matrica u kojoj redci označavaju odabrana ispitna pitanja, a stupci označavaju definirane atribute, a iz sadržaja te binarne matrice se iščitava koje atribute ima svako ispitno pitanje. Nakon toga se provodi predložena metoda kombinatornog testiranja koja na automatiziran način identificira sažeti podskup svih odabranih ispitnih pitanja koja među svojim atributima pokrivaju sve dozvoljene parove ili trojke atributa iz formalnog konteksta. Pritom se u procesu kombinatornog testiranja generiraju testni slučajevi, odnosno opisi ispitnih pitanja tako da je svaka dozvoljena n-torka definiranih atributa željene veličine $n = t$ pokrivena barem s jednim testnim slučajem. Iz pronađenog sažetog podskupa ispitnih pitanja gradi se odgovarajući završni formalni kontekst, a primjenom metode formalne analize koncepata se iz njega automatski gradi odgovarajuća konceptualna rešetka koja se prikazuje kao parcijalno uređeni skup. Konceptualna rešetka predstavlja ontologiju zadanog nastavnog gradiva opisanog s označenim pitanjima, a sastoji se od međusobno povezanih formalnih koncepata u odnosu *natkoncept-potkoncept*. Svaki formalni koncept predstavlja par skupova ispitnih pitanja i njihovih dijeljenih atributa. U kontekstu ovoga rada predložen je naziv elementarna točka znanja (*EKP* točka) za formalni koncept jer svaka *EKP* točka sadrži neki skup pitanja i zajednički skup pojmova koji se s tim pitanjima ispituje. Primjenom topološkog sortiranja nad konceptualnom rešetkom dobije se potpuno uređeni skup formalnih koncepata u kojem su očuvani svi odnosi *natkoncept-potkoncept*. Iz generiranih linearnih poredaka formalnih koncepata ili *EKP* točaka poredanih od onih općenitijih prema onim specifičnijim mogu izdvojiti odgovarajući nizovi pitanja. A potom se generirani nizovi pitanja spremaju kao scenariji izvođenja formativne provjere znanja. Predložen je i postupak za izvođenje formativne provjere znanja koja vodi studente kroz odabrane nizove pitanja tako što odabire iduće pitanje na temelju ocjene odgovora na prethodno pitanje. Kao

pomoć u rješavanju formativne provjere znanja predloženo je korištenje interaktivnih nastavnih materijala. Na kraju je provedena verifikacija predložene metode za automatiziranu provjeru znanja primjenom formalne metode provjere modela. Pritom je predložen postupak za otkrivanje modela procesa provjere znanja u obliku konačnog automata primjenom nadograđenog *L** algoritma Dane Angluin i korištenjem alata za provjeru modela *Spin*. Generirani modeli procesa provjere znanja se automatski prevode u odgovarajuće procesne modele u jeziku *Promeli*, a potom se mogu simulirati i verificirati s alatom *Spin*.

U radu su izloženi svi rezultati implementirane metode za automatiziranu pripremu i vođenje formativnih provjera znanja. Prikazani su rezultati kombinatornog testiranja odabranog označenog skupa ispitnih pitanja koji se nakon postupka kombinatornog testiranja povećao s 473 na 522 pitanja. Potom je pronađen sažeti podskup od 139 pitanja koji pokriva sve dozvoljene parove atributa iz definiranog skupa atributa i iz njega je automatski izgrađen završni formalni kontekst. Nakon toga je automatski izgrađena odgovarajuća konceptualna rešetka s 1037 *EKP* točaka i 3413 usmjerenih grana korištenjem alata za metodu formalne analize koncepata. Automatskim postupkom topološkog sortiranja dobivene konceptualne rešetke generiralo se ukupno 888 potpunih i djelomičnih nizova pitanja. Iz generiranih 16 potpunih redoslijeda od 103 pitanja su automatski spremljeni scenariji izvođenja formativne provjere znanja u bazu podataka vlastitog sustava za e-učenje. Prikazani su zapisi o izvođenju formativne provjere znanja kao i izbor interaktivnih nastavni materijala koji se koriste za pomoć kod krivih odgovora. Na kraju je na temelju svih 888 potpunih i djelomičnih nizova pitanja automatski izgrađen odgovarajući model procesa formativne provjere znanja u obliku konačnog automata. A nakon automatskog prevođenja otkrivenog modela u procesni model u *Promeli* uspješno su izvedene simulacije modela kao i verifikacija modela korištenjem alata *Spin*.

Pokazalo se kako predložena metoda za automatiziranu pripremu i vođenje formativnih provjera znanja može pomoći nastavnicima i drugim domenskim ekspertima u izboru pitanja i kod sastavljanja novih pitanja, a studentima ovakav sustav može pružiti dodatnu pomoć u procesu usvajanja i vježbanja novog gradiva. Rezultati dobiveni sa simulacijama modela procesa formativne provjere znanja mogu pomoći ekspertima za oblikovanje sustava za e-učenje za dodatno testiranje web aplikacije za izvođenje provjeru znanja. A u budućem radu će se rezultati simulacija uspoređivati tehnikama dubinske analize procesa sa stvarnim podacima iz dnevničkih zapisa sustava za e-učenje koji će se dobiti prilikom rješavanja provjere znanja.

Literatura

[1]H. Rodrigues, F. Almeida, V. Figueiredo and S. L. Lopes, "Tracking e-learning through published papers: A systematic review," in *Computers & Education*, vol. 136, pp. 87–98, 2019.

[2]M. Gusev and G. Armenski, "E-assessment systems and online learning with adaptive testing," in *E-Learning Paradigms and Applications - Agent-based Approach* (M. Ivanovic and L. C. Jain, eds.), *Studies in Computational Intelligence*, vol. 528, pp. 229–249, Springer, 2014.

[3]A. Grubisic, S. Stankov, and B. Zitko, "Adaptive courseware: A literature review," *J. UCS*, vol. 21, no. 9, pp. 1168–1209, 2015.

[4]B. Zitko, S. Stankov, M. Rosic, and A. Grubisic, "Dynamic test generation over ontology-based knowledge representation in authoring shell," *Expert Syst. Appl.*, vol. 36, no. 4, pp. 8185–8196, 2009.

[5]S. Gulwani, "Example-Based Learning in Computer-Aided STEM Education," *Commun. ACM, vol. 57*, pp. 70–80, August 2014.

[6]C. Alvin, S. Gulwani, R. Majumdar and S. Mukhopadhyay, "Synthesis of geometry proof problems," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, July 27-31, 2014, Québec City, Québec, Canada. (C. E. Brodley and P. Stone, eds.), AAAI Press, 2014, pp. 245–252.

[7]R. Singh, S. Gulwani, and S. K. Rajamani, "Automatically generating algebra problems," in *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, July 22-26, 2012, Toronto, Ontario, Canada. (J. Hoffmann and B. Selman, eds.), AAAI Press, 2012.

[8]R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, Seattle, WA, USA, June 16-19, 2013 (H. Boehm and C. Flanagan, eds.), ACM, 2013, pp. 15–26.

[9]R. Wille, “Restructuring lattice theory: an approach based on hierarchies of concepts”, in: *Ordered Sets* (I. Rival, ed.), Reidel, Dordrecht-Boston, pp. 445–470, 1982.

[10]B. Ganter and R. Wille, *Formal Concept Analysis – Mathematical Foundations*, Springer-Verlag, Berlin, 1999.

[11]B. Ganter, G. Stumme and R. Wille (Eds.), *Formal Concept Analysis - Foundations and Applications*, Springer-Verlag, Berlin Heidelberg. 2005.

[12]S.O. Kuznetsov, S. Schmidt (Eds.), *Formal Concept Analysis, Fifth International Conference, ICFCA 2007*, Springer, 2007.

[13]S. Ferré, S. Rudolph (Eds.), *Formal Concept Analysis, Seventh International Conference, ICFCA 2009*, Springer, 2009.

[14]B. Ganter, R. Godin (Eds.), *Formal Concept Analysis, Third International Conference, ICFCA 2005*, Springer, 2005.

[15]M. A. Bedek, M. D. Kickmeier-Rust and D. Albert, “Formal concept analysis for modelling students in a technology-enhanced learning setting,” in *Proceedings of the 5th Workshop on Awareness and Reflection in Technology Enhanced Learning In conjunction with the 10th European Conference on Technology Enhanced Learning: Design for Teaching and Learning in a Networked World, ARTEL@EC-TEL 2015*, Toledo, Spain, September 15, 2015. (M. Kravcik et al., eds.), vol. 1465 of CEUR Workshop Proceedings, pp. 27–33, CEUR-WS.org, 2015.

[16]G. Beydoun, “Using formal concept analysis towards cooperative e-learning,” in *Knowledge Acquisition: Approaches, Algorithms and Applications, Pacific Rim Knowledge Acquisition Workshop, PKAW 2008*, Hanoi, Vietnam, December 15-16, 2008, Revised Selected Papers (D. Richards and B. H. Kang, eds.), vol. 5465 of *Lecture Notes in Computer Science*, pp. 109–117, Springer, 2008.

[17]G. Beydoun, “Formal concept analysis for an e-learning semantic web,” *Expert Syst. Appl.*, vol. 36, no. 8, pp. 10952–10961, 2009.

[18]U. Priss, “Using FCA to analyse how students learn to program,” in *Formal Concept Analysis, 11th International Conference, ICFCA 2013*, Dresden, Germany, May 21-24, 2013. Proceedings (P. Cellier et al., eds.), vol. 7880 of *Lecture Notes in Computer Science*, pp. 216–227, Springer, 2013.

[19]A. K. Sarmah, S. M. Hazarika and S. K. Sinha, “Formal concept analysis: current trends and directions,” *Artif. Intell. Rev.*, vol. 44, no. 1, pp. 47–86, 2015.

[20]P. K. Singh, C. Aswani Kumar and A. Gani, “A comprehensive survey on formal concept analysis, its research trends and applications,” *International Journal of Applied Mathematics and Computer Science*, vol. 26, no. 2, pp. 495–516, Jun. 2016.

[21]M. Clark, Y. Kim, U. Kruschwitz et al., “Automatically structuring domain knowledge from text: An overview of current research,” *Inf. Process. Manage.*, vol. 48, no. 3, pp. 552–568, 2012.

[22]L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu and J. Zhao, “Deepct: Tomographic combinatorial testing for deep learning systems,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2019, pp. 614–618.

[23]L. S. Ghandehari, Y. Lei, R. Kacker, D. R. R. Kuhn, D. Kung and T. Xie, “A combinatorial testing-based approach to fault localization,” *IEEE Transactions on Software Engineering*, 2018.

[24]C. Nie and H. Leung, “A survey of combinatorial testing,” *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, pp. 1–29, 2011.

[25]S. K. Khalsa and Y. Labiche, “An orchestrated survey of available algorithms and tools for combinatorial testing,” in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, IEEE, 2014, pp. 323–334.

[26]D. R. Kuhn, D. R. Wallace and A. Gallo, “Software Fault Interactions and Implications for Software Testing,” *IEEE Transactions on Software Engineering*, 30(6): 418–421, 2004.

[27]Q. N. Naveed, M. R. N. M. Qureshi, A. Shaikh, A. O. Alsayed, S. Sanober and K. Mohiuddin, “Evaluating and ranking cloud-based e-learning critical success factors (csfs) using combinatorial approach,” *IEEE Access*, vol. 7, pp. 157145–157157, 2019.

[28]D. I. Borissova and D. Keremedchiev, “Generation of e-learning tests with different degree of complexity by combinatorial optimization,” *Journal of e-Learning and Knowledge Society*, vol. 16, no. 2, pp. 17–24, 2020.

[29]Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[30]F. Škopljanac-Ma čina, I. Zakarija and B. Blašković, “Exam questions consistency checking,” in *Proceedings of 38th International Convention on Information and Communication Technology, Electronics and Microelectronics MIPRO 2015*, IEEE, 2015, pp. 921–924.

[31]G. J. Holzmann, *The SPIN model checker: primer and reference manual*, Reading: Addison-Wesley, 2004.

[32]G. J. Holzmann, "Software model checking with SPIN," *Advances in Computers*, vol. 65, pp. 78–109, 2005.

[33]F. Ricca and P. Tonella, "Analysis and testing of Web applications," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, Toronto, Ontario, Canada, 2001, pp. 25–34.

[34]M. Schur, A. Roth and A. Zeller, "Mining Workflow Models from Web Applications," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1184–1201, 1 Dec. 2015.

[35]K. Homma, S. Izumi, Y. Abe, K. Takahashi and A. Togashi,"Using the Model Checker Spin for Web Application Design," in *2010 10th IEEE/IPSJ International Symposium on Applications and the Internet*, Seoul, 2010, pp. 137–140.

[36]F. Škopljanac-Ma čina, B. Blašković and I. Zakarija, "Automated analysis of e-learning web applications," in *MIPRO 2019, 42nd International Convention, Proceedings*, IEEE, 2019, pp. 957–962.

[37]D. R. Stinson, *Combinatorial Designs: Constructions and Analysis.* New York: Springer-Verlag, 2004.

[38]K. Michael and K. W. Miller, "Big data: New opportunities and new challenges [guest editors' introduction]," *IEEE Computer*, vol. 46, no. 6, pp. 22–24, 2013.

[39]S. Sagiroglu and D. Sinanc, "Big data: A review," in *2013 International Conference on Collaboration Technologies and Systems, CTS 2013*, San Diego, CA, USA, May 20-24, 2013 (G. C. Fox and W. W. Smari, eds.), IEEE, 2013, pp. 42–47.

[40]W. M. Van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*, Berlin Heidelberg, Springer, 2011.

[41]W. M. P. van der Aalst, *Data science in action*, New York, NY: Springer, 2016.

[42]I. Zakarija, F. Škopljanac-Ma čina and B. Blašković, "Automated simulation and verification of process models discovered by process mining," *Áutomatika*, vol. 61, no. 2, pp. 312–324, 2019.

[43]I. Zakarija, F. Škopljanac-Ma čina and B. Blašković, "Discovering Process Model from Incomplete Log using Process Mining," in *Proceedings of ELMAR-2015 57h International Symposium, ELMAR 2015*, Zadar, Croatia, 28-30 September 2015, (M. Muštra, D. Tralić, B. Zovko-Cihlar, eds.), LotusGRAF, 2015, pp. 117–120.

[44]W.M.P. van der Aalst, S. Guo and P. Gorissen, "Comparative Process Mining in Education: An Approach Based on Process Cubes," in: *Data-Driven Process Discovery and Analysis, SIMPDA 2013* (P. Ceravolo, R. Accorsi et al., eds.), *Lecture Notes in Business Information Processing*, vol 203. Springer, Berlin, Heidelberg, 2013.

[45]M. Pechenizkiy, N. Trcka, E. Vasilyeva, W.M.P. van der Aalst and P.M.E. De Bra, "Process mining online assessment data," in *Educational Data Mining 2009: 2nd International Conference on Educational Data Mining: proceedings, EDM '09*, Cordoba, Spain. July 1-3, 2009, (T. Barnes, M. Desmarais et al., eds.), International Working Group on Educational Data Mining, 2009, pp. 279–288.

[46]P. Mukala, J.C.A.M. Buijs, M. Leemans and W.M.P. van der Aalst, "Learning Analytics on Coursera Event Data: A Process Mining Approach," in *Proceedings of the 5th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2015), Vienna, Austria, December 9-11, 2015* (P. Ceravolo and S. Rinderle-Ma, eds.), 2015, pp. 18–32.

[47]R. Studer, R. Benjamins and D. Fensel. "Knowledge engineering: Principles and methods," *Data & Knowledge Engineering*, 25(1–2):161–198, 1998.

[48]J. F. Sowa, *Knowledge Representation: Logical, Philosophical and Computational Foundations*, Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1999.

[49]J. F. Sowa, "Ontology, metadata, and semiotics," in *Conceptual Structures: Logical, Linguistic, and Computational Issues, 8th International Conference on Conceptual Structures, ICCS 2000*, Darmstadt, Germany, August 14-18, 2000, Proceedings (B. Ganter and G. W. Mineau, eds.), vol. 1867 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 55–81.

[50]R. Belohlavek, "Introduction to Formal Concept Analysis", https://phoenix.inf.upol.cz/esf/ucebni/formal.pdf, Olomuc, 2008.

[51]F. Škopljanac-Ma čina, B. Blašković, "Formal Concept Analysis – Overview and Applications," *Procedia Engineering*, 69: 1258–1267, 2014.

[52]D. Žubrni ć, *Diskretna matematika*, Element, Zagreb, 2001.

[53] *Concept Explorer 1.3*. (2009). [Online]. Available: www.sourceforge.net/projects/conexp

[54]B. Ganter, "Algorithmen zur Formalen Begriffsanalyse," in *Beiträge zur Begriffsanalyse*, (B. Ganter, R. Wille et al., eds.), B. I. Wissenschaftsverlag, pp. 241–254, 1987.

[55]R. Agrawal, T. Imielinski and A. Swami, “Mining association rules between sets of items in large databases.” in *Proc. SIGMOD Conf.*, 1993, pp. 207–216.

[56]W.W. Armstrong, “Dependency structures of data base relationships,” in *Information Processing 74: Proceedings of IFIP Congress 74*, (J.L. Rosenfeld and H. Freeman, eds.), 580-583, North Holland, 1974.

[57]M. Baranovi ć and S. Zakošek, “Baze podataka (PDS, nastavni materijali),” https://www.fer.unizg.hr/_download/repository/BP_doktorski_2015.pdf, Zagreb, 2015.

[58]S.O. Kuznetsov, “On the intractability of computing the Duquenne-Guigues base,” *Journal of Universal Computer Science*,10(8):927–933, 2004.

[59]V. Duquenne and J.L. Guigues, “Famille minimale d’implications informatives resultant d’un tableau de donnes binaires,” *Math. et Sci. Hum.*, 24(95):5–18, 1986.

[60] *Lattice Miner Platform 2.0*. (2017). [Online]. Available: https://sourceforge.net/projects/lattice-miner/

[61]B. Ganter and R. Wille, *Formale Begriffsanalyse – Mathematische Grundlagen*, Springer-Verlag, Berlin, 1996.

[62]T. S. Blyth, *Lattices and Ordered Algebraic Structures*, Springer-Verlag, London, 2005.

[63] *Wolfram Cloud*. (2022). [Online]. Available: https://www.wolframcloud.com/

[64]G. Stumme, R. Taouil, Y. Bastide and L. Lakhal,“Conceptual Clustering with Iceberg Concept Lattices.” in *The meeting of the Proc. GI-Fachgruppentreffen Maschinelles Lernen (FGML’01)*, Universität Dortmund 763, 2001.

[65]G. Stumme, R. Taouil, Y. Bastide, N. Pasquier and L. Lakhal, “Fast computation of concept lattices using data mining techniques,” in *Proc. 7th Intl. Workshop on Knowledge Representation Meets Databases*, Berlin, 2000, pp. 21–22.

[66]R. Wille, “The Basic Theorem of triadic concept analysis,” *Order*, 12, pp. 149–158, 1995.

[67]L. Wei, T. Qian, Q. Wan et al, “A research summary about triadic concept analysis,” *Int. J. Mach. Learn. & Cyber.*, 9, pp. 699–712, 2018.

[68]J. Poelmans, D.I. Ignatov, S. O. Kuznetsov, G. Dedene, “Fuzzy and rough formal concept analysis: a survey,” *International Journal of General Systems*, 43:2, pp. 105–134, 2014.

[69]K.E. Wolff, “Temporal concept analysis explained by examples,” *CDUD’11 – Concept Discovery in Unstructured Data*, 104, 2011.

[70]D. R. Kuhn, R. N. Kacker and Y. Lei, *Introduction to combinatorial testing*. Chapman & Hall/CRC, 2013.

[71]R. Kuhn, Y. Lei and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, 2008.

[72]R. D. Kuhn, R. N. Kacker and Y. Lei, "Combinatorial Testing," in *Encyclopedia of Software Engineering* (P. A. Laplante et al., eds.). Boca Raton: CRC Press, 2010.

[73] *AllPairs*. (2022). [Online]. Available: sourceforge.net/projects/allpairs

[74]N.J.A. Sloane, *A Library of Orthogonal Arrays*. (2023). [Online]. Available: http://neilsloane.com/oadir/

[75] *Tcases*. (2022). [Online]. Available: github.com/cornutum/tcases

[76] *Jenny*. (2022). [Online]. Available: burtleburtle.net/bob/math/jenny.html

[77] *Pict*. (2022). [Online]. Available: github.com/microsoft/pict

[78] *ACTS*. (2022). [Online]. Available: csrc.nist.gov/acts

[79]J.A. Bondy, U.S.R. Murty, *Graph Theory*, Springer, New York, 2008.

[80]J.A. Bondy, U.S.R. Murty, *Graph Theory with Applications*, Elsevier, New York, 1976.

[81]S.S. Skiena, "Section 15.2: Topological Sorting," in *The Algorithm Design Manual (2nd ed.)*, Springer-Verlag, London, pp. 481–483, 2008.

[82]T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Section 22.4: Topological sort," in: *Introduction to Algorithms, 2nd ed.*, MIT Press and McGraw-Hill, pp. 549–552, 2001.

[83]A.B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, 5(11): 558–562, 1962.

[84] *tsort*. (2022). [Online]. Available: https://en.wikipedia.org/wiki/Tsort

[85]R. Keller, "Formal Verification of Parallel Programs," *Commun. ACM*, 19, pp. 371–384, 1976.

[86]J. E. Hopcroft, R. Motwani and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation, 2nd ed.*, Addison Wesley, 2001.

[87] *SPIN model checker*. (2022). [Online]. Available: http://spinroot.com

[88]D. Angluin, “Learning regular sets from queries and counterexamples,” *Information and Computation*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[89]B. Blaškovi ć, F. Škopljanac-Mačina and I. Zakarija, “Discovering e-learning process models from counterexamples,” in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, IEEE, 2018, pp. 0593–0598.

[90] *libalf – The Automata Learning Framework*. (2011). [Online]. Available: http://libalf.informatik.rwth-aachen.de/

[91]R.L. Rivest, R.E. Schapire, “Inference of Finite Automata Using Homing Sequences,” *Information and Computation*, 103(2), 299–347, 1993.

[92]M. Shahbaz, R. Groz, “Inferring Mealy machines,” in: *FM 2009 Proceedings, LNCS vol. 5850*, 2009, pp. 207–222.

[93]G. H. Mealy, “A method for synthesizing sequential circuits,” *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[94]B. Bollig, P. Habermehl, C. Kern and M. Leucker, “Angluin-Style Learning of NFA.” in: *International Joint Conference on Artificial Intelligence*, 2009.

[95]O. Maler, A. Pnueli, “On the Learnability of Infinitary Regular Sets,” *Information and Computation*, 118(2), 316–326, 1995.

[96]D. Angluin, D. Fisman, “Learning regular omega languages,” *Theoretical Computer Science*, 650, 57-72, 2016.

[97]C. Baier, J.P. Katoen, *Principles of Model Checking*, MIT Press, 2008.

[98] *A Guide to the Project Management Body of Knowledge (PMBOK Guide) – Fifth Edition*, Project Management Institute, 2013.

[99]A. Pnueli, “The temporal logic of programs,” in: *18th IEEE Symposium on Foundations of Computer Science (FOCS)*, IEEE Computer Society Press, 1977, pp. 46–67.

[100] *ltl2ba*. 2022. [Online]. Available: http://www.lsv.fr/~gastin/ltl2ba/index.php

[101]G.J. Holzmann, “An Analysis of Bitstate Hashing,” *Formal Methods in System Design*, 13(3): 289–307, 1998.

[102]F. Škopljanac-Ma čina, I. Zakarija and B. Blašković, “Towards Automated Assessment Generation in e-Learning Systems Using Combinatorial Testing and Formal Concept Analysis,” *IEEE access*, 9, pp. 52957-52976, 2021.

[103]J. Annapurna and A. K. Cherukuri, “Exploring Attributes with Domain Knowledge in Formal Concept Analysis,” *Journal of Computing and Information Technology*, vol. 21, no. 2, pp. 109–123, 2013.

[104] *SQLite*. (2023). [Online]. Available: https://www.sqlite.org

[105] *Sqlite3*. (2021). [Online]. Available: www.sqlite.org/cli.html

[106] *SQLiteStudio*. (2021). [Online]. Available: sqlitestudio.pl

[107]A. Pavi ć and F. Škopljanac-Mačina, “Computer in testing the Fundamentals of electrical engineering at the University of Zagreb Faculty of electrical engineering and computing,” in *MIPRO 2012 - 35th International Convention on Information and Communication Technology, Electronics and Microelectronics*, Opatija, 2012.

[108]F. Škopljanac-Ma čina, B. Blašković and Z. Skočir, “Using formal concept analysis for student assessment,” in *ELMAR, 2014 56th International Symposium*, Zadar, Croatia, LotusGRAF, 2014, pp. 285–288.

[109] *PowerShell*. (2023). [Online]. Available: https://learn.microsoft.com/hr-hr/powershell/

[110] *GnuWin*. (2023). [Online]. Available: https://sourceforge.net/projects/gnuwin32/

[111] *pyodbc*. (2023). Available: https://pypi.org/project/pyodbc/

[112] *jQuery*. (2022). [Online]. Available: https://jquery.com/

[113] *JSXGraph*. (2023). Available: https://jsxgraph.uni-bayreuth.de/wp/index.html

[114] *SVG*. (2022). [Online]. Available: https://www.w3.org/Graphics/SVG/

[115] *MathJax*. (2022). [Online]. Available: https://www.mathjax.org/

[116] *Highcharts*. (2023). Available: https://www.highcharts.com/

[117] *Graphviz*. (2023). Available: https://graphviz.org/

[118]B. Blaškovi ć, F. Škopljanac-Mačina, P. Knežević and N. Palić, “Modeling constraint satisfaction problem with model checker.” in *Proceedings of the 29th DAAAM International Symposium 2018*, Zadar, 2018. pp. 1–9.

[119] *Tiny C Complier*. (2023). Available: https://bellard.org/tcc/

Popis slika

Popis tablica

Životopis

Frano Škopljanac-Mačina rođen je 25. prosinca 1983. u Zagrebu, gdje je završio osnovnu školu 1998. godine te srednju školu (*Klasičnu gimnaziju*) 2002. godine. Na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu diplomirao je 20. listopada 2009. na smjeru Telekomunikacije i informatika s diplomskim radom pod naslovom "Korištenje jezika OWL-S za opisivanje web usluga".
Nakon završenog studija radi na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu, na Zavodu za osnove elektrotehnike i električka mjerenja. Od 2010. do 2015. radi kao stručni suradnik, a od 2015. do danas kao voditelj laboratorija. Održava i razvija Zavodski sustav za e-učenje te sudjeluje u izvođenju nastave iz laboratorijskih vježbi na više predmeta preddiplomskog i diplomskog studija (*Osnove elektrotehnike*, *Elektromagnetska polja*, *Informacija, logika i jezici* te *Formalne metode u oblikovanju sustava*). Njegov znanstveni i stručni interes usmjeren je na oblikovanje sustava za e-učenje, a pogotovo na korištenje formalnih metoda i tehnika automatizirane provjere znanja u izgradnji naprednih adaptivnih sustava za e-učenje.
Autor je i koautor devetnaest znanstvenih radova objavljenih u časopisima i zbornicima međunarodnih znanstvenih konferencija. Član je međunarodne strukovne udruge IEEE.

Popis objavljenih djela

Radovi u časopisima

1. Škopljanac-Ma čina, Frano; Zakarija, Ivona; Blašković, Bruno Towards Automated Assessment Generation in e-Learning Systems Using Combinatorial Testing and Formal Concept Analysis. // IEEE access, 9 (2021), 52957-52976.
2. Zakarija, Ivona; Škopljanac-Ma čina, Frano; Blašković, Bruno Automated simulation and verification of process models discovered by process mining. // Automatika : časopis za automatiku, mjerenje, elektroniku, računarstvo i komunikacije, 61 (2020), 2; 312-324.
3. Vrani ć, Mihaela; Pintar, Damir; Škopljanac- Mačina, Frano Improved Visualization of Frequent Itemset Relationships Using the Minimal Spanning Tree Algorithm. // Tehnički vjesnik : znanstveno-stručni časopis tehničkih fakulteta Sveučilišta u Osijeku, 26 (2019), 2; 331-338.

4.Pintar, Damir; Beguši ć, Domagoj; Škopljanac- Mačina, Frano; Vranić , Mihaela Automatic extraction of learning concepts from exam query repositories. // Journal of communications software and systems, 14 (2018), 4; 312-319.

5.Škopljanac-Ma čina, Frano; Blašković, Bruno Formal Concept Analysis – Overview and Applications. // Procedia Engineering, 69 (2014), 1258-1267.

Radovi objavljeni u zbornicima skupova s međunarodnom recenzijom

1.Škopljanac-Ma čina, Frano; Zakarija, Ivona; Blašković, Bruno Organizing online exams during the COVID-19 pandemic. // 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO) - proceedings / Skala, Karolj (ur.). Rijeka: Croatian Society for Information, Communication and Electronic Technology - MIPRO, 2021. str. 841-846.

2.Škopljanac-Ma čina, Frano; Blašković, Bruno; Zakarija, Ivona Automated analysis of e-learning web applications. // MIPRO 2019, 42nd International Convention, Proceedings / Skala, Karolj (ur.). Rijeka: MIPRO, 2019. str. 957-962.

3.Blaškovi ć, Bruno; Škopljanac-Mačina, Frano; Knežević, Petar; Palić , Niko Modeling constraint satisfaction problem with model checker. // Proceedings of the 29th DAAAM International Symposium 2018 Zadar, Hrvatska, 2018. str. 1-9.

4.Pintar, Damir; Beguši ć, Domagoj; Škopljanac- Mačina, Frano; Vranić, Mihaela Annotating Exam Questions Through Automatic Learning Concept Classification. // 2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2018) / Rožić, Nikola ; Lorenz, Pascal (ur.). Supetar, Hrvatska: Curran Associates, Inc, 2018. str. 123-129.

5.Blaškovi ć, Bruno; Škopljanac-Mačina, Frano; Zakarija, Ivona Discovering e-Learning Process Models from Counterexamples. // Proceedings of the 41st International Convention on Information and Communication Technology, Electronics and Microelectronics MIPRO 2018 / Skala, Karolj (ur.). Rijeka: Croatian Society for Information and Communication Technology, Electronics and Microelectronics - MIPRO, 2018. str. 0593-0598.

6.Škopljanac-Ma čina, Frano; Blašković, Bruno Assessment Process Synthesis for Adaptive E- learning Systems. // Abstract Book - Second International Workshop on Data Science / Lončarić, Sven ; Šmuc, Tomislav (ur.). Zagreb, Hrvatska: Centre of Research Excellence for Data Science and Cooperative Systems Research Unit for Data Science Croatia, 2017. str. 31-33.

7.Haznadar, Zijad; Merzi ć, Ajla; Škopljanac- Mačina, Frano Prilagodba elektroenergetskih sustava novim suvremenim uvjetima rada. // Proceedings - zbornik radova New Technologies - Nove tehnologije NT-2016 / Doleček, Vlatko ; Karabegović, Isak ; Pašić, Sead (ur.). Bihać, Bosna i Hercegovina: Society for robotics of Bosnia and Herzegovina, 2016.

str. 306-309.

8.Škopljanac-Ma čina, Frano; Blašković, Bruno; Pintar, Damir Automated Generation of Questions for Basic Electrical Engineering Education. // Annals of DAAAM and Proceedings of the International DAAAM Symposium Volume 27 / Katalinic, Branko (ur.). Mostar, Bosna i Hercegovina, 2016. str. 377-385.

9.Vrani ć, Mihaela; Pintar, Damir; Humski, Luka; Skočir, Zoran; Škopljanac-Mačina, Frano; Brstilo, Ivana; Đuho, Nika; Klasnić, Kristina; Mališa, Snježana; Žuković, Ana Marija University Social Network Benefits Analysis and Proposed Framework. // 24th International Conference on Software, Telecommunications and Computer Networks - SoftCOM 2016 / Rožić, Nikola ; Begušić, Dinko (ur.). Split: FESB, 2016.

10.Haznadar, Zijad; Trkulja, Bojan; Škopljanac- Ma čina, Frano Nuklearne elektrane s torijskim reaktorom. // Zbornik radova Nove Tehnologije "New Technologies - NT 2015" / Karabegović, Isak ; Doleček, Vlatko ; Pašić, Sead (ur.). Bihać: Society for Robotics of Bosnia and Herzegovina, 2015. str. 35-37.

11.Škopljanac-Ma čina, Frano; Zakarija, Ivona; Blašković, Bruno Exam questions consistency checking. // Proceedings of 38th International Convention on Information and Communication Technology, Electronics and Microelectronics MIPRO 2015 / Biljanović, Petar (ur.). Rijeka: GRAFIK, 2015. str. 921-924.

12.Zakarija, Ivona; Škopljanac-Ma čina, Frano; Blašković, Bruno Discovering Process Model from Incomplete Log using Process Mining. // Proceedings of ELMAR-2015 57h International Symposium / Muštra, Mario ; Tralić, Dijana ; Zovko-Cihlar, Branka (ur.). Zagreb: LotusGRAF, 2015. str. 117-120.

13.Škopljanac-Ma čina, Frano; Blašković, Bruno; Skočir, Zoran Using Formal Concept Analysis for student assessment. // Proceedings of ELMAR-2014 56th International Symposium / Dijana Tralić ; Mario Muštra ; Branka Zovko-Cihlar (ur.). Zagreb: LotusGRAF, 2014. str. 285-288.

14.Pavi ć, Armin; Škopljanac-Mačina, Frano Computer in Testing the Fundamentals of Electrical Engineering at the University of Zagreb Faculty of Electrical Engineering and Computing. // MIPRO 2012, 35th International Convention on Information and Communication Technology, Electronics and Microelectronics / Biljanović, Petar (ur.). Rijeka: MIPRO, 2012. str. 1205-1210.

Biography

Frano Škopljanac-Mačina was born on December 25, 1983 in Zagreb, where he finished primary school in 1998 and secondary school (*Classical Gymnasium*) in 2002. He graduated from University of Zagreb, Faculty of Electrical Engineering and Computing on October 20, 2009 in the field Telecommunications and Informatics with the thesis "Using OWL-S language for describing web services".

Since graduation he works at University of Zagreb, Faculty of Electrical Engineering and Computing, Department of Electrical Engineering Fundamentals and Measurements. From 2010 to 2015 he worked as an expert associate, and from 2015 he works as a laboratory manager. He maintains and develops Department's e-learning system and partakes in teaching laboratory classes on different undergraduate and graduate courses (*Fundamentals of Electrical Engineering*, *Electromagnetic Fields*, *Information, logic and languages* and *Formal Methods in System Design*). His scientific and professional interest is focused on e-learning systems design, especially using formal methods and automated assessments techniques in building advanced adaptive e-learning systems.

He authored or co-authored nineteen scientific papers published in scientific journals and in the proceedings of international scientific conferences. He is a member of international professional association IEEE.