Received March 5, 2021, accepted March 24, 2021, date of publication April 1, 2021, date of current version April 12, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3070510

Towards Automated Assessment Generation in e-Learning Systems Using Combinatorial Testing and Formal Concept Analysis

FRANO ŠKOPLJANAC-MAČINA[1], (Member, IEEE), IVONA ZAKARIJA[2], (Member, IEEE), AND BRUNO BLAŠKOVIĆ[1], (Member, IEEE)
[1]Faculty of Electrical Engineering and Computing, University of Zagreb, 10000 Zagreb, Croatia
[2]Department of Electrical Engineering and Computing, University of Dubrovnik, 20000 Dubrovnik, Croatia

Corresponding author: Frano Škopljanac-Mačina (frano.skopljanac-macina@fer.hr)

This work was supported by the European Regional Development Fund through the Operational Programme Competitiveness and Cohesion 2014–2020, through the Project System for Real-Time Monitoring and Control of Distributed Processes, Anomaly Detection, Early Warning, and Forensic Transaction Analysis–PCC under Grant KK.01.2.1.02.0097.

ABSTRACT In this paper, we research the use of software combinatorial testing techniques and the Formal Concept Analysis method for preparing sets of questions for student assessment in e-learning systems. Utilizing these techniques and methods, we ensure that the selected questions optimally cover the course material and that each question combines multiple topics. Therefore, in this paper we introduce our method for preparing student assessments that performs automated combinatorial testing and selection of questions, as well as automated generation of appropriate sequences of questions. The input for our method is a set of questions labelled with attributes or features. This set of questions is pre-processed using the Formal Concept Analysis method, and then the combinatorial testing of question features is performed, which generates a concise list of test-cases covering all pairs or triples of question features. Correspondingly, our method helps in identifying and selecting a subset of questions that covers all generated test-cases. Afterwards, the Formal Concept Analysis method automatically generates suitable sequences of selected questions for formative student assessments in e-learning systems. In this paper we implemented the proposed combinatorial testing method, and also demonstrated the feasibility of the proposed method on a use-case from an actual e-learning system.

INDEX TERMS Combinatorial testing, pairwise testing, formal concept analysis, e-learning systems, automated test generation, process mining.

I. INTRODUCTION

A central part of an e-learning system is its knowledge base, usually a database that contains various questions used for quizzes and assessments, or also practice exercises that the student solves along different learning paths in adaptive e-learning systems. One of the main issues in developing and maintaining an e-learning system is the effective management and organization of the knowledge stored in its database. This is especially important when teachers add new questions to the shared repository of questions. The e-learning system should provide them with a simple and reliable way for retrieving questions from the shared repository, so that the existing questions can be easily found and inspected before adding any new questions. A keyword search of the question texts will not always return all results, because valuable additional information can be conveyed through the question figures, as is often the case in STEM courses. One formal way of organizing domain knowledge is by manually building its ontology. Afterwards, questions can be added as new objects to the ontology. This can be time-consuming and laborious, even when organizing domain knowledge of smaller introductory courses. Moreover, this approach requires that the teachers are well acquainted with the querying and modifying of the ontology. Recently, there are also advanced deep learning techniques for automated recognition of the question figures, such as convolutional neural networks that analyze and classify images based on an initial training set

The associate editor coordinating the review of this manuscript and approving it for publication was Luca Cassano.

of manually labelled images. Again, this approach requires a lot of work to efficiently incorporate into existing e-learning systems.

Therefore, in this paper we propose a simple way of describing each question in the shared questions' repository that can facilitate reliable identification and selection of questions. First, we need to compile a concise set of labels that describes the course material. Then, we require that each question (including its text and figure) is described by a subset of those labels. Furthermore, we use machine learning method called Formal Concept Analysis (FCA) to automatically generate a corresponding formal description of the questions' set and to store it in a relational database. Afterwards, we can use standard SQL commands to query questions based on their labels. Moreover, we propose using combinatorial testing techniques to identify a subset of questions that combine multiple topics, e.g. we can find an almost minimal subset of questions that covers all meaningful pairs of labels at least once. Proposed combinatorial testing procedure can also recommend new questions, by providing a labelled description of questions that could to be added to the questions' set, so that it covers the course material more thoroughly. After a suitable subset of questions is identified using the combinatorial testing method, we again use the FCA method, this time to automatically create a basic ontology of the course material covered by the selected subset of questions. Finally, that ontology is used to generate sequences of questions that can be used primarily for formative student assessments. Generated sequences of questions are ordered by their generality, from the more general ones (described by fewer labels) to the more specific ones (described by more labels).

By monitoring the formative assessment process teachers can closely follow the progress of each student at the course. They could detect early warning signs that the students did not fully understand certain topics, or that they are losing interest in the course. In such cases teachers can offer students help and motivation for successful completion of the course.

The focus of this paper will be on the initial preparation of the questions' descriptions using the FCA method and on the proposed combinatorial testing method for identifying and selecting questions that combine knowledge from multiple topics in the course material.

The rest of the paper is organized as follows: Section II offers an overview of the related research concerning combinatorial testing and the FCA method, especially their applications in the e-learning domain. Afterwards, Sections III and IV provide brief theoretical introduction to the combinatorial testing and the FCA method, respectively. In Section V we propose our method for automated assessment generation in e-learning systems. Next, in Section VI we focus in detail on the combinatorial testing module that is used for selecting questions for student assessments. Section VII provides a use case, where our proposed method is applied to the set of labelled questions from our e-learning system database. Subsequently, in Section VIII we present and discuss the results of the combinatorial testing experiments. Finally, in Section IX we conclude our paper and discuss areas for future work.

II. RELATED RESEARCH

Combinatorial testing [1] can detect failures in the software that occur when multiple components interact, and it is designed to detect complex faults with relatively small number of tests. Since pairwise testing is computationally tractable and effective it is the most common approach to combinatorial testing. In pairwise testing, all feasible pairs of parameter values are covered by at least one test, and there are algorithms and tools to generate arrays with the value pairs. By extending combinatorial testing with model checking approaches it is possible to automatically generate test oracles by converting covering arrays into executable tests [2].

In study [3] a set of combinatorial testing criteria specialized for deep learning systems is proposed, as well as the guided test generation technique. It is demonstrated that combinatorial testing provides a promising approach for testing deep learning systems.

Since the combinatorial testing was proved as an effective strategy for software testing, in [4] a fault localization approach, called BEN, is presented. BEN produces a ranking of statements in terms of their probability of being faulty using the results of combinatorial testing.

Authors in [5] give an overview of the combinatorial testing research field. Basic concepts and notations of combinatorial testing are presented, the research in this field is classified into categories, and important issues, methods, and applications of combinatorial testing are identified.

The available algorithms and tools for generating a combinatorial test suite are identified and categorized in [6]. The results may be useful when searching for an appropriate combinatorial algorithm or tool.

The research [7] provides a systematic survey for cloud-based e-learning Critical Success Factors (CSFs) of teaching-learning process. Moreover, it utilizes the combinatorial approach for evaluation and prioritization of the various dimensions and CSFs of cloud-based e-learning. It is demonstrated that the combinatorial approach of AHP-GDM[1] and FAHP[2] methodology could be prosperous in classifying the CSFs in various grades of influence. The influence of such dimensions and factors could be useful in planning the strategy and resources for improving the knowledge transfer through cloud-based e-learning.

An algorithm for generation of questions for tests with different complexity levels using mathematical combinatorial optimization model is proposed in [8]. Furthermore, in a case study authors experimentally demonstrated application of the proposed algorithm, as well as that it could be applied in diverse learning contents.

[1]AHP-GDM: Analytic Hierarchy Process Group Decision Making
[2]FAHP: Fuzzy Analytic Hierarchy Process

In [9] model learning is considered as an effective method for building black-box state machine models of complex software systems. Applications of model learning for testing software systems in several domains are presented on use cases. Special attention was paid to Angluin's L* algorithm for learning deterministic finite-state automata (DFA) [10]. We also used Angluin's L* algorithm in our previous work [11] to generate DFAs that simulate student assessment based on the sequences of questions extracted from the ontology of questions built using FCA.

FCA is a machine learning method for data representation and for automated building of ontologies in a form of concept lattices. It is the applied in diverse fields, from software engineering, data mining to linguistics [12]. Authors in [13] conducted a systematic survey of over 350 recent research papers involving FCA. They also gave a thorough overview of the mathematical foundations of FCA and also presented various extensions of FCA. One of the identified research issues is the handling of large sets of input data and complex and impractical concept lattices. In [14] authors use FCA to represent behavior of industrial processes. A concept lattice reduction method is proposed that successfully preserves minimum representation of the process data. An interactive process or building concept lattices called attribute exploration is discussed in [15]. Authors used existing domain experts' knowledge during the attribute exploration and demonstrated the process on two use-cases form healthcare domain. In e-learning domain FCA was used to model and analyze student performance, and to help students in learning new concepts [16]. Also, FCA was used to identify how students search through the course material [17], [18], and to detect conceptual difficulties students encounter when learning new lessons [19].

Model checking [20] based method for process analysis is presented in [21]. This method can be utilized for the detection and analysis of expected and unexpected patterns in the software system behavior. Furthermore, proposed process mining [22] method can be used to analyze the usage of e-learning systems, especially students' activity during the formative assessments that students solve incrementally over a longer time period. This allows instructors and teachers to monitor students' progress and detect any anomalies that can be early warning signs that a student could fail the course.

The study in [23] identifies the main factors that influence the usage and acceptance of e-learning systems. Accordingly, it provides valuable insights into e-learning system usability that may be beneficial to universities worldwide.

III. COMBINATORIAL TESTING

Combinatorial testing is a software testing technique that aims to reduce the number of tests that need to be performed, while ensuring that most of the software system faults are detected. Kuhn *et al.* in [24] analyzed software system faults and observed that almost all system failures are triggered through interaction of maximum 4 to 6 different parameters. Also, they concluded that in various domains up to 90% of software system faults are caused by a single parameter value or by an interaction of two different parameters. Therefore, combinatorial testing technique focuses on the interactions between the software system input parameters. The goal of this technique is to generate an almost minimal number of test-cases, so that each possible parameter value tuple of predefined size is covered by at least one test-case. If the size of parameter value tuples is set to 2, then the combinatorial testing procedure will generate test-cases that cover all pairs of parameter values at least once. This variant of combinatorial testing is the most prevalent, and it is also known as the *pairwise testing*.

```
0:      [0, 0, 0]
1:      [1, 1, 0]
2:      [1, 0, 1]
3:      [0, 1, 1]
```

FIGURE 1. An example of a trivial orthogonal array $OA_1(4, 2, 3, 2)$.

Combinatorial testing was developed from the work in statistical field called Design of Experiments, that is used to improve industrial or agricultural production and medical treatments by measuring how combinations of input factors affect the response variable. Mathematically, combinatorial testing is based on the notion of orthogonal arrays and covering arrays [25], [26].

Definition 1: An orthogonal array, $OA_\lambda(N, t, k, v)$ is an $N \times k$ array, where N is the number of rows (tests), k is the number of columns (parameters), v is the number of values for each parameter and t is the interaction coverage strength. In every $N \times t$ subarray of the orthogonal array $OA_\lambda(N, t, k, v)$ each tuple of size t occurs exactly λ times.

If t is set to 2 and λ is set to 1, then the resulting orthogonal array will contain every pair of parameter values exactly once. Fig. 1 shows an example of a such simple orthogonal array ($t = 2$, $\lambda = 1$), with three parameters ($k = 3$) and two parameter values ($v = 2$, values: 0 and 1). This orthogonal array was generated using the combinatorial testing program *AllPairs* [27]. The orthogonal array $OA_1(4, 2, 3, 2)$ has four rows, and it can be easily checked that for any two selected columns (1-2, 2-3, 1-3) all four pairs of possible parameter values (00, 01, 10, 11) appear exactly once.

Sometimes it is very hard or even impossible to construct an orthogonal array that satisfies this strict requirement (e.g. $\lambda = 1$). Therefore, an extension of orthogonal arrays was developed called covering arrays that loosened this provision.

Definition 2: A covering array, $CA(N, t, k, v)$ is an $N \times k$ array, where N is the number of rows (tests), k is the number of columns (parameters), v is the number of values for each parameter and t is the interaction coverage strength. In every $N \times t$ subarray of the covering array $CA(N, t, k, v)$ each tuple of size t occurs at least once.

It can be seen that the pairwise testing ($t = 2$) conforms to the definition of the covering array. Moreover, observe that every orthogonal array is also a covering array, but not vice versa. Further extensions of orthogonal arrays and covering arrays allow mixed parameter values, i.e. every parameter can have a different number of parameter values. Lastly, it must be noted that not all parameter values combinations are always valid. Therefore, in orthogonal arrays there can be test-cases (rows) that are not feasible or not meaningful. Nevertheless, a further extension of covering arrays introduces constraints on parameter values, so that unwanted parameter value combinations do not appear in the resulting covering array. In this paper, our combinatorial testing method is based on mixed values covering arrays with and without parameter value constraints.

We analyzed a comprehensive list of available combinatorial testing tools.[3] We wanted to find a free and convenient command-line tool that supports parameter value constraints. However, most of the available tools are commercial or web-based. Some of the command-line tools we identified were the before mentioned *AllPairs*, *Pict* from *Microsoft* [28], *Tcases* [29], *Jenny* [30], and *ACTS* from *NIST* [31] that has both a GUI and a command-line interface. The *AllPairs* program does not support parameter value contraints, so it is not suitable for our research. On the other hand, *ACTS*, *Pict* and *Tcases* allow complex constraints declarations. However, they are all quite advanced tools that require special care with input preparation, as well as with output processing. Therefore, we opted for the combinatorial testing tool *Jenny* as the most feasible tool for our purpose. It has an intuitive user interface, supports basic constraints declarations, and its testing generation efficiency is comparable to other established tools.[4]

Jenny is a free, public domain command-line program written in C for generating regression tests that contain various parameter value (*feature*) combinations from defined parameters (*feature dimensions*). Instead of an exhaustive search for every feature combination, *Jenny* generates an almost minimal number of test-cases that cover each possible tuple of features (usually pairs or triples of features). Before running the *Jenny* program, we need to set following program parameters: define the size of the feature tuples (flag ‘‘-n’’, default size is 2) and list number of features in each dimension. Optionally, we can declare forbidden combinations of features (flag ‘‘-w’’) that should not appear together in generated test-cases.

An example of a *Jenny* program run can be seen in Fig. 2. The size of the feature tuples is set to 2 and there are three dimensions declared, with 2, 3 and 4 features, respectively. Additionally, there is one forbidden feature combination (*1a*, *2a*): first dimension's (named 1) first feature (named a) cannot appear in a test-case alongside second dimension's (named 2) first feature (named a).

After running the program with these parameters, it returns a list of 12 test-cases that cover all possible pairs of features excluding the pair (*1a*, *2a*). For example, we can easily check that the pair (*1a*, *3d*) is covered by the 8th test case: (*1a*, *2c*, *3d*). It should be noted that the *Jenny* program generated only 12 test-cases and by performing exhaustive search for all

[3]https://jaccz.github.io/pairwise/tools.html

[4]https://jaccz.github.io/pairwise/efficiency.html

```
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
```

FIGURE 2. An example run of the program *Jenny*.

feature combinations there would be $2 \cdot 3 \cdot 4 = 24$ test-cases, or 20 test-cases if we exclude the forbidden combination (*1a*, *2a*). Observe that the output of the *Jenny* program in Fig. 2 is a covering array with constraints and not an orthogonal array, because it does not have one combination – excluded forbidden combination (*1a*, *2a*), and e.g. a pair (*1a*, *2c*) appears in two test-cases and a pair (*1b*, *2b*) appears in only one test-case.

IV. FORMAL CONCEPT ANALYSIS

Formal Concept Analysis (FCA) is an unsupervised machine learning method for data analysis and data representation [32], [33]. From the prepared input data, the FCA method automatically identifies formal concepts (groups of objects with shared attributes) and creates their hierarchy that is visualized as a directed acyclic graph called a concept lattice. The concept lattice can be considered as a simple ontology of the objects and their attributes from the input domain. The FCA method was developed by Rudolf Wille in early 1980s, and it is built on the mathematical foundations of lattice theory and set theory [34], [35]. To use the FCA method we first need to prepare the input data from our domain of interest. We create a list of objects from that domain, and define a set of attributes that will be used to describe those objects. Afterwards, we need to prepare input data in a form of a two-dimensional binary matrix of objects (matrix rows) and attributes (matrix columns). If an object has a certain attribute, we need to place a mark ‘‘*X*’’ (or number 1) on the intersection of that object's row and that attribute's column. This input data matrix is called a formal context.

Definition 3: A formal context is a triple $\langle X, Y, I\rangle$, where X is a set of objects, Y is a set of attributes, and I is a binary relation between the elements of sets X and Y.

FCA automatically analyses formal context to detect two types of output data. Firstly, FCA builds a concept lattice, a directed acyclic graph (line diagram) with all identified formal concepts. Secondly, FCA generates a list of attribute implications, i.e. a list of association rules between the attributes that are inferred from the concept lattice. FCA uses concept forming operators to find groups of objects that share attributes and vice versa.

Definition 4: Concept forming operators over the formal context $\langle X, Y, I\rangle$ are functions $\uparrow$: $2^X \rightarrow 2^Y$ and

TABLE 1. A formal context example.

	resistance	*DC_current*	*DC_voltage*	*current_source*	*voltage_source*
Question1	X			X	
Question2	X	X	X	X	
Question3				X	X
Question4	X	X	X		
Question5	X	X	X		X
Question6	X			X	X

$\downarrow$: $2^Y \to 2^X$. For each subset of objects $A \subseteq X$ and for each subset of attributes $B \subseteq Y$ they are defined as $A \uparrow = \{y \in Y \mid \text{for each } x \in A : \langle x, y\rangle \in I\}$ and $B \downarrow = \{x \in X \mid \text{for each } y \in B : \langle x, y\rangle \in I\}$, respectively.

Therefore, $A \uparrow$ finds a set of attributes that are shared by all objects from A. Similarly, with $B \downarrow$ we get a set of objects that have all attributes form B. Formal concepts are maximal segments of the formal context that include all objects that share same attributes.

Definition 5: A formal concept in a formal context $\langle X, Y, I\rangle$ is a pair $\langle A, B\rangle$, where $A \subseteq X$, $B \subseteq Y$, $A \uparrow = B$ and $B \downarrow = A$.

A hierarchical operator $\leq$ is used to denote a super-concept – sub-concept relationship between the formal concepts. E.g., $\langle A_1, B_1\rangle \leq \langle A_2, B_2\rangle$ means that the formal concept $\langle A_2, B_2\rangle$ is a super-concept of the formal concept $\langle A_1, B_1\rangle$. This is true if and only if the set of objects A_1 is a subset of the set of objects A_2 ($A_1 \subseteq A_2$) and the set of attributes B_2 is a subset of the set of attributes B_1 ($B_2 \subseteq B_1$).

Definition 6: For a given formal context $\langle X, Y, I\rangle$, a collection of all its formal concepts is defined as: $B(X, Y, I) = \left\{\langle A, B\rangle \in 2^X \times 2^Y \mid A \uparrow = B, B \downarrow = A\right\}$. A concept lattice $\langle B(X, Y, I), \leq\rangle$ of a formal context $\langle X, Y, I\rangle$ includes a collection of all formal concepts $B(X, Y, I)$ and a hierarchical operator $\leq$.

Concept lattice is a partially ordered set in which any two formal concepts have a shared supremum (least common super-concept) and a shared infimum (greatest common sub-concept) [36].

As an example of the FCA method, Table 1 shows a simple formal context that contains 6 objects (each row denotes a question from the course *Fundamentals of Electrical Engineering*) and 5 attributes (each column denotes a question description). A marking "*X*" in a cell of the formal context states that the question in that cell's row has the attribute from its column.

We used one of available FCA tools, *ConExp 1.3* [39] to automatically produce a concept lattice for the formal context in Table 1. The resulting concept lattice is shown in Fig. 3, and it contains 12 formal concepts. Top formal concept includes a set of all questions and an empty set of their shared attributes (observe from Table 1 that no attribute is shared by all objects). Bottom formal concept includes a set of all attributes and an empty set of objects that have

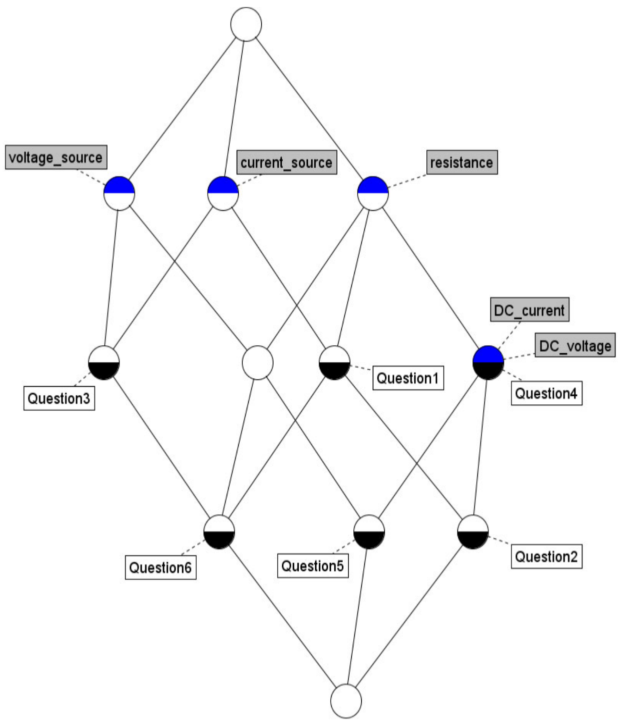


FIGURE 3. Concept lattice for the formal context in Table 1.

all attributes (again, from Table 1 we see there are no such objects).

Note that in Fig. 3 the edges are not drawn as directed arrows, instead they are implicitly directed from top to bottom (from super-concepts to sub-concepts).

Observe that each sub-concept has all attributes from its super-concepts, e.g. rightmost formal concept in Fig. 3 that contains *Question4* has attributes *DC_current*, *DC_voltage*, and also *resistance* (from its immediate super-concept). This can be stated as an association rule $\{resistance\} \to \{DC\_current, DC\_voltage\}$. This association rule has a support of 50% (it is true for 3/6 objects in the formal context) and a confidence of 60% (it is true for 3/5 objects that have the attribute *resistance*). An attribute implication is a stricter form of association rule that has a confidence of 100%. E.g., in this formal context one attribute implication is $\{DC\_current\} \Rightarrow \{DC\_voltage, resistance\}$. It has a support of 3/6 (it is true for 3/6 objects in the formal context) and a confidence of 100% (it is true for all 3/3 objects that have the attribute *DC_current*).

V. OVERVIEW OF THE PROPOSED METHOD FOR AUTOMATED ASSESSMENT GENERATION

In this section we introduce our method for automated generation of assessments in e-learning systems. Our method first identifies an appropriate set of labelled questions, and then it prepares sequences of questions that will be used primarily for formative student assessments in e-learning systems. Proposed method utilizes combinatorial testing techniques and the FCA method. An overview of our proposed method is given in Fig. 4.

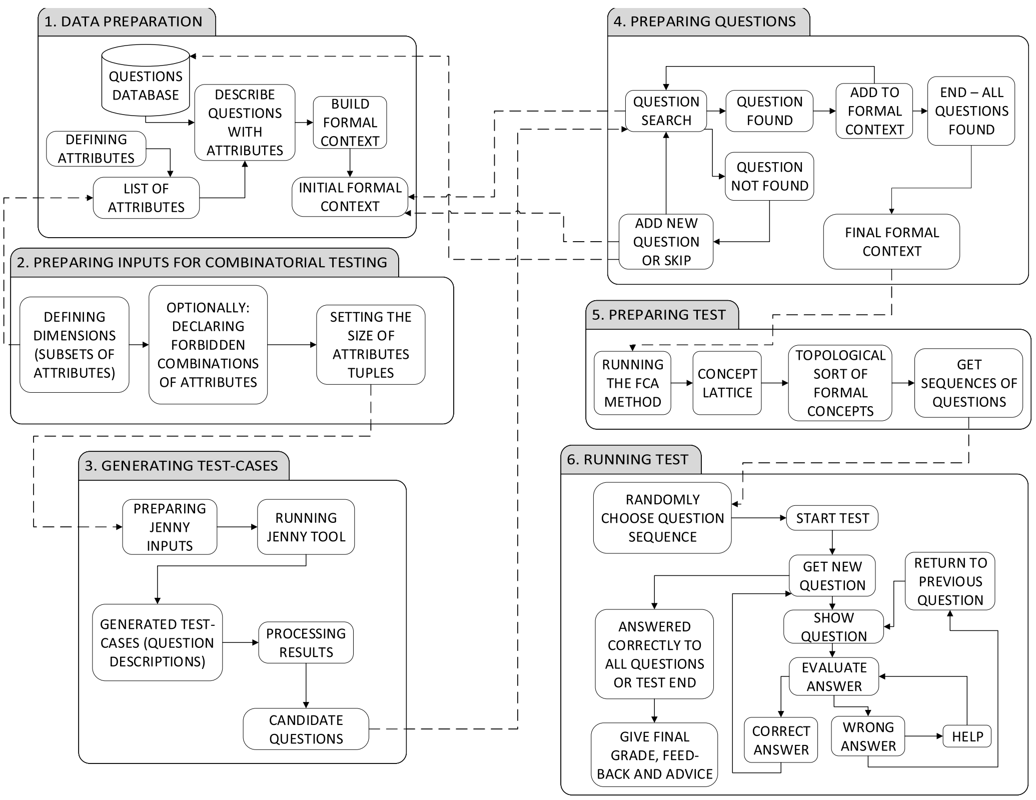


FIGURE 4. **An overview of the proposed method for automated assessment generation.**

Our method requires that the questions in the e-learning system's database are labelled with subsets of predefined attributes that adequately describe the course material. Therefore, if this requirement is not already met, in Step 1 of our method we first need to define a set of attributes that will be used to describe the questions. Afterwards, we extract the questions from the e-learning system's database and label each question with a subset of predefined attributes. This input data is used to build an initial formal context of the database questions, which is the first step of the FCA method as described in Section IV.

In the Step 2 and Step 3 we perform the combinatorial testing of the questions in the initial formal context. Attributes from the formal context can be considered as questions' features. Before starting with the combinatorial testing, we need to organize features in dimensions. Also, we can declare which feature combinations are not allowed, and we must set the size of the feature tuples that need to be completely covered by generated test-cases. Combinatorial testing is done using the *Jenny* tool, described in Section III. We consider each test-case generated by the *Jenny* tool as a question description.

In Step 4 we need to search formal context for questions that match each generated test-case. If a test-case is not completely matched by any existing questions, we should consider implementing it as a new question and adding it to the database and the initial formal context. Next, we add all found questions to the final formal context which will be used for student assessment.

In Step 5 the final formal context is automatically transformed into a concept lattice using the FCA method. We apply topological sorting algorithm to the formal concepts in the concept lattice, and by that we get possible sequences of questions ready for use in the formative assessment module of an e-learning system. The questions are sorted from the more general ones (questions in formal concepts near the top of the concept lattice) to the more specific ones (questions in formal concepts near the bottom of the concept lattice).

Step 6 is implemented by the assessment module of an e-learning system. Initially, system randomly chooses a sequence of questions and starts the test. After the question is answered the system evaluates the answer. If the answer was correct the next question from the sequence is opened. Otherwise, system offers help, and if student answers incorrectly again the system will give another variant of the question or return a step back to the previous (easier) question. After all questions are answered correctly or the assessment was terminated the system gives final evaluation and generates feed-back and advice for further learning and exercising. This assessment module is primarily suited for formative assessments because each student gets a longer list of questions that can cover entire course material. Students are not required to complete the assessment in one session, they can take it in sequential sessions during the course.

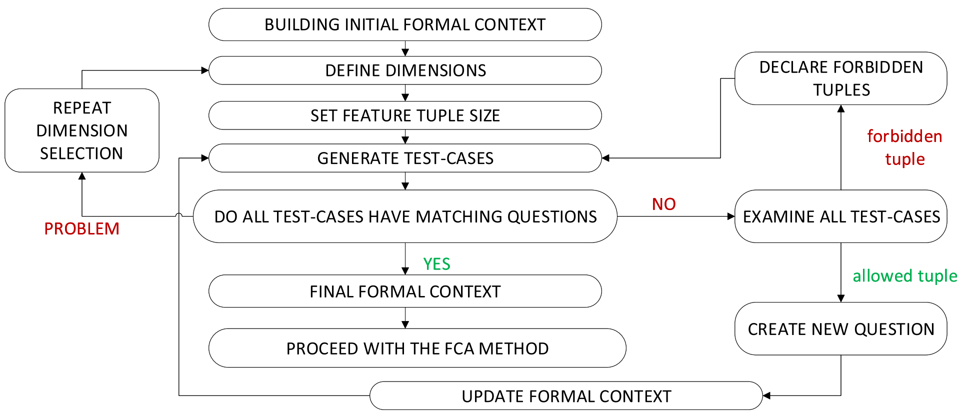


FIGURE 5. An overview of the combinatorial testing method.

A. COMBINATORIAL TESTING METHOD

In this paper we will concentrate on the initial Steps 1 – 4 from Fig. 4 where we need to describe the questions with attributes and build the formal context before running the combinatorial testing procedure that searches for most appropriate questions. Therefore, we present in Fig. 5 a more detailed overview of our combinatorial testing method.

From the Fig. 5 we see that the combinatorial testing method starts by building the initial formal context and ends when each test-case is fully covered by at least one of the questions in the final formal context. Before running the combinatorial testing process, we need to define dimensions (subsets of features) and set the size of the feature tuples (usually 2 or 3) that should be covered by the test-cases. Some of the generated test-cases could be already covered by the questions in the formal context. Others, that are not found in the formal context should be thoroughly examined. If the feature combination of a test-case is interesting and covered by the course material it should be implemented as a new question (or a set of new questions) and added to the initial formal context. Otherwise, we must declare that such feature combination is not allowed and run the combinatorial testing procedure again. Finally, when all the test-cases are fully covered we generate the final formal context containing only those questions that match the generated test-cases. This final formal context will be used by the FCA method for building the concept lattice and preparing the student assessment (Steps 5 and 6 in Fig. 4).

It must be noted that the combinatorial testing results depend on the initial definition of dimensions, as will be seen in Sections VII and VIII. If the combinatorial testing procedure does not return meaningful test-cases, it is very likely that our definition of dimensions is flawed and they need to be redefined before running the testing procedure again.

We must comment on the differences of our approach to the attribute exploration discussed in [15]. Attribute exploration is a manual interactive process for building the formal context and its concept lattice. Initially, the set of attributes is declared and the set of objects is empty. Objects are then added to the formal context depending on the answers that the domain expert gives to the FCA system. Domain expert must validate or refute each attribute implication proposed by the FCA system. This task can become very complex and time-consuming for larger attribute sets (e.g. 50 attributes). It can be especially hard to refute suggested attribute implications, because the domain expert must then provide a counterexample that should not contradict previously validated attribute implications. Furthermore, if the attribute exploration process ends successfully the domain expert must check all objects (candidate questions) in the formal context and compose all needed new questions strictly according to their often large attribute subsets.

Our proposed combinatorial testing method automatically finds a concize set of test-cases (candidate questions) that cover all pairs or triples of features from feature dimensions. Afterwards, it automatically checks if the candidate questions are already covered by the existing questions in the database. If that is the case they are automatically added to the final formal context. Only when a candidate question is not covered, the domain expert must decide whether it will be implemented as a new question or discarded. Furthermore, when composing a new question domain experts have a greater flexibility because they only need to cover the attributes (features) that appear in the test-case. Additionally, combinatorial testing approach can guarantee that all meaningful pairs or triples of attributes are covered by at least one question.

VI. COMBINATORIAL TESTING MODULE

This section details the combinatorial testing module that implements the combinatorial testing method introduced in Section V-A and shown in Fig. 5.

This module contains an SQLite database and three Python scripts for preparing, running, and analyzing combinatorial testing process. The SQLite database offers great flexibility when storing, retrieving and analyzing data using SQL queries without a need for implementing and manipulating complex Python data structures. Furthermore, our Phyton

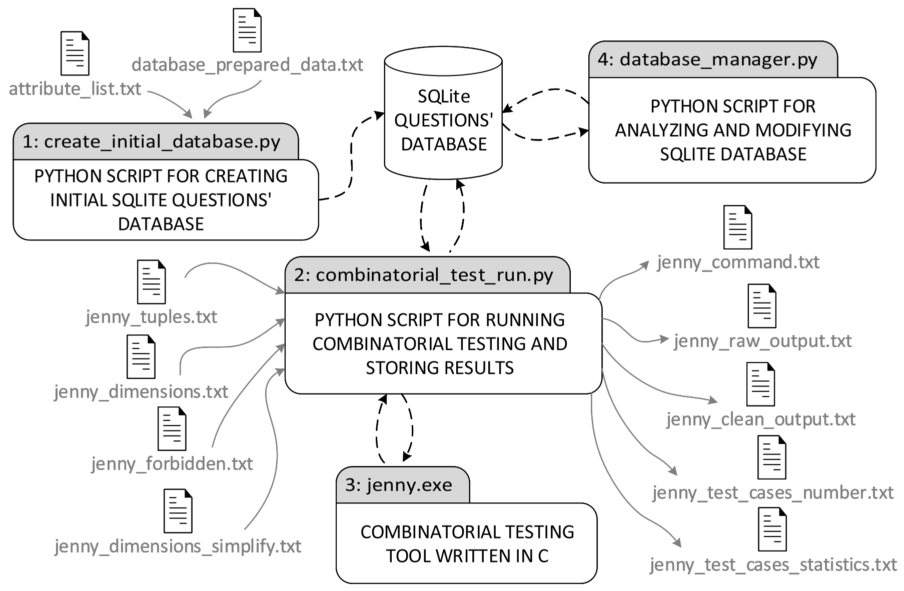

attribute_list.txt
database_prepared_data.txt
1: create_initial_database.py
PYTHON SCRIPT FOR CREATING INITIAL SQLITE QUESTIONS' DATABASE
SQLite QUESTIONS' DATABASE
4: database_manager.py
PYTHON SCRIPT FOR ANALYZING AND MODIFYING SQLITE DATABASE
2: combinatorial_test_run.py
PYTHON SCRIPT FOR RUNNING COMBINATORIAL TESTING AND STORING RESULTS
jenny_tuples.txt
jenny_dimensions.txt
jenny_forbidden.txt
jenny_dimensions_simplify.txt
jenny_command.txt
jenny_raw_output.txt
jenny_clean_output.txt
jenny_test_cases_number.txt
jenny_test_cases_statistics.txt
3: jenny.exe
COMBINATORIAL TESTING TOOL WRITTEN IN C

FIGURE 6. An overview of the combinatorial testing module.

scripts communicate with the *Jenny* program, described in Section III, that is used as a tool for generating test-cases. An overview of the combinatorial testing module is shown in Fig. 6.

Python script *create_initial_database.py* is used to automatically create main SQLite database that will contain questions' descriptions and testing results. This script creates following three tables: *questions*, *testcases* and *questions_testcases*. First of them, table *questions* will contain data from the formal context of the selected set of questions. It has an identifier – a primary key field *IDquestion* (integer type), followed by *n* fields (integer type) named after question attributes from the formal context. Each of these *n* fields can only have value 0 (if the question does not have that attribute) or value 1 (if the question has that attribute). Formal context attribute names are read from the prepared text file *attribute_list.txt* (every row contains one attribute name without white spaces) and a necessary CREATE TABLE command is automatically built and executed. Afterwards, script populates the *questions* table with the data from the formal context of the selected set of questions. Script reads the prepared formal context from a text file *database_prepared_data.txt* (each row is a space-separated list containing question ID and all question's attribute values given as 0 or 1) and creates and executes corresponding INSERT commands. The second table is *testcases* and it has one primary key field *IDtestcase* (integer type), which is a test-case identifier, and a text field *description* that will hold a list of all features of a test-case. The third table, *questions_testcases* has *IDquestion* field (integer type) and *IDtestcase* field (integer type) – question and test-case identifiers respectively, as well as *full_match* field (integer type), which states if the question completely matches the test-case (value 1) or if the question only partially matches the test-case (value 0). Tables *testcases* and *questions_testcases* are initially empty, and they will later hold data from the combinatorial test runs.

The central part of the combinatorial testing module is implemented in the Python script *combinatorial_test_run.py*. It acts as a wrapper for the combinatorial testing tool *Jenny* by preparing all the input parameters and running the *Jenny* tool as well as by decoding its output. Furthermore, this Python script analyses the generated test-cases and stores the experiment's results. Before running this script, users should prepare following input text files:

- *jenny_tuples.txt* (required) that contains only one number – the requested size of the feature tuples that need to be covered by the generated test-cases.
- *jenny_dimensions.txt* (required) that contains all the features grouped into different dimensions. Dimensions are disjoint subsets of the features set. Each row is a tab-separated list of features that represents one dimension.
- *jenny_forbidden.txt* (optional) that contains feature combinations that should not appear in any test-case. Each row is a space-separated list of forbidden feature combinations (usually feature pairs).

Additionally, we added an option of naming groups of features that can be useful when dealing with large feature sets. If this option is used all the names of the feature groups and their elements must be declared in a text file *jenny_dimensions_simplify.txt*. Each row contains data on one feature group: feature group name, hash sign and a space-separated list of features in the group. We will consider that a question covers a feature group if it has at least one attribute (feature) from the feature group. Feature group names can then be used when declaring feature dimensions or listing forbidden feature combinations.

```
function FIND_TEST_CASES()
    $test\_case\_counter \leftarrow 0$
    if feature groups names are declared in jenny_dimensions_simplify.txt then
        create dictionary $jenny\_simplified\_dict$ for storing feature groups elements
        for each $line$ in jenny_dimensions_simplify.txt do
            split $line$ at "#" and store results as $data[0]$ and $data[1]$
            add key-value pair {$data[0]$, "(" + $data[1]$.replace(" ","=1 OR ") + "=1)"} to $jenny\_simplified\_dict$
        end for
    end if
    run function that deletes old data from the tables testcases and questions_testcases
    for each $line$ (generated test-case) from decoded jenny program output jenny_output_clean.txt do
        $test\_case\_counter \leftarrow test\_case\_counter + 1$
        run function that stores the test-case in the table testcases
        if $jenny\_simplified\_dict$ exists then
            for each $key$ in $jenny\_simplified\_dict$ do
                find and replace $key$ in $line$ with the $value$ of the $key$
            end for
        end if
        find if the test-case is fully covered by any questions and store results in the table questions_testcases
        find if the test-case is partially covered by any questions and store results in the table questions_testcases
    end for
    run function that generates test-cases statistics
end function
```

FIGURE 7. Algorithm of the function *find_test_cases()*.

After reading the input text files, the script will automatically prepare the corresponding command-line inputs for the *Jenny* tool, execute it and decode its output by replacing the implicit feature names (as seen in Fig. 2) with the feature (or feature groups) names declared in the input text files. Script also stores the executed command-line call of the *Jenny* program, its raw and decoded output, as well as the counted number of generated test-cases in separate text files.

Then, script calls the function *find_test_cases()* that implements the main algorithm of our combinatorial testing method shown in Fig. 7. Initially, the function checks if the simplified feature groups are declared. If they are used, then a simple dictionary is created that stores key-values pairs: *feature group name*, SQL command segment with the all the N elements in the feature group: ($attribute1 = 1$ OR $attribute2 = 1 \cdots$ OR $attributeN = 1$).

Next, the function empties the tables *testcases* and *questions_testcases*, and then stores each generated test-case using an automatically built INSERT SQL command (line 13 in Fig. 7).

Afterwards, for each decoded test-case function automatically builds a more complex INSERT INTO SELECT command in SQL that determines if the test-case is covered fully or partially by the questions from the *questions* table, and stores the data on fully/partially covered questions to the *questions_testcases* table (lines 19 and 20 in Fig. 7). A test-case is covered fully if there are one or more questions that have attributes that match with the all the features in the test-case. To measure if a test-case is partially covered

```
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
```

FIGURE 8. SQL command template for finding and storing data on fully covered test-cases.

```
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
```

FIGURE 9. SQL command template for finding and storing data on partially covered test-cases.

we introduced a global variable *partially_covered_limit* that sets the lower limit of features that need to be covered by the questions. Consequently, we consider a test-case is covered partially if there are one or more questions whose attributes match with at least $n = partially\_covered\_limit$ of test-case attributes. In Fig. 8 and Fig. 9 we show SQL templates for dynamically building commands for finding and storing data on fully and partially covered test-cases, respectively.

Note that in SQLite there is no special *Boolean* datatype, and therefore *True* is equal to integer value 1 and *False* is equal to integer value 0. We used this characteristic of

SQLite in the SQL command in Fig. 9 to quickly identify all the questions that partially match a given test-case. If the *partially_cover_limit* is set to 1, then the WHERE clause from Fig. 9 is equal to ($test\_case\_feature\_1 = 1$ OR $test\_case\_feature\_2 = 1$ OR $\cdots$ OR $test\_case\_feature\_N = 1$).

Finally, the function performs an analysis of the results and stores them in the text file *jenny_test_cases_statistics.txt*. Script executes a group of six more complex SELECT commands that aggregate data about the generated test-cases (line 22 in Fig. 7).

First SQL command lists all the test-cases along with the number of questions that fully and partially match each test-case. As an example, this SQL command is formally described with relational algebra expressions (1), (2) and (3) over relations *testcases*(*<u>IDtestcase</u>*, *description*) and *questions_testcases*(*IDquestion*, *IDtestcase*, *full_match*). We used extended relational algebra notation from [38].

The queries declared in (1) and (2) are virtual relations (views) that use aggregate operation γ on a left outer join operation ⟕ between relations *testcases* and *questions_testcases* to count the number of fully and partially matched questions for each test-case, respectively.

Final query in (3) applies a projection operation Π on an inner join operation $\bowtie$ between relations *testcases* and views *Full* and *Partial* to display all test-cases with additional information from (1) and (2).

$$\begin{aligned} Full \leftarrow\ & \Pi_{x.IDtestcase \text{ as } ID,\, full\_count}(\\ & \quad {}_{x.IDtestcase}\gamma_{\text{count}(y.IDtestcase) \text{ as } full\_count}(\rho_x(testcases) \\ & \quad ⟕_{x.IDtestcase=y.IDtestcase \wedge y.full\_match=1} \\ & \quad \rho_y(questions\_testcases))) \qquad (1) \\ Partial \leftarrow\ & \Pi_{x.IDtestcase \text{ as } ID,\, partial\_count}(\\ & \quad {}_{x.IDtestcase}\gamma_{\text{count}(y.IDtestcase) \text{ as } partial\_count}(\rho_x(testcases) \\ & \quad ⟕_{x.IDtestcase=y.IDtestcase \wedge y.full\_match=0} \\ & \quad \rho_y(questions\_testcases))) \qquad (2) \\ & \Pi_{IDtestcase,\, description,\, full\_count,\, partial\_count}(\\ & \quad testcases \bowtie_{testcases.IDtestcase=Full.ID} Full \\ & \quad \bowtie_{Full.ID=Partial.ID} Partial) \qquad (3) \end{aligned}$$

Following two SQL commands separately list the test-cases that are fully matched and the test-cases that are only partially matched, respectively. Later, this can be useful for manual inspection of test-cases. The fourth SQL command calculates basic statistics of the generated test-cases – it lists the test-cases number, the number and the percentage of test cases that are fully matched, as well as only partially matched by the existing questions. Afterwards, the fifth SQL command lists all the questions that have been fully matched by the test-cases, together with the number of test-cases they correspond to. Lastly, the sixth SQL command prepares and lists formal context of the questions that have been fully matched by the test-cases. When each test-case is matched by at least one question the combinatorial testing process ends and the listed output formal context becomes the final formal context. The FCA method will automatically build its concept lattice, which will be then used to generate sequences of questions for the student assessment module (Steps 5 and 6 in Fig. 4).

Before the script ends it copies all the used input and output text files, together with the SQLite questions' database into a timestamp-named directory inside an experiments subfolder.

The combinatorial testing module contains another Python script (*database_manager.py*), which is used to manage the SQLite questions' database. This script provides a simple, menu-based interface to the SQLite database that facilitates viewing, querying and modifying its data. Control and communication with the database in all of our scripts was achieved using Python's *sqlite3* module.

Using this script users can manually run commands for creating, emptying or deleting previously described database tables, and get basic statistical information about the tables. Script also facilitates adding new question descriptions to the table *questions*, e.g. when composing new questions that implement previously uncovered generated test-cases.

Furthermore, script allows users to run arbitrary SQL commands. This is especially useful when analyzing combinatorial testing results, e.g. if there are no questions that fully match a given test-case, we can create a SELECT command that finds questions that are most similar to the given test-case. Also, users can modify question description data, e.g. if we determine that the question's attribute values were not correctly set.

Therefore, for most users with an intermediate knowledge of SQL this database management script provides a simple and useful alternative to more complex SQLite database management tools, such as command-line shell for SQLite *Sqlite3* [39] or a GUI desktop application *SQLiteStudio* [40].

VII. USE CASE OVERVIEW

As a use case for the combinatorial testing method we will test a set of 473 questions form the freshmen year course *Fundamentals of Electrical Engineering* taken from our e-learning system *WebOE*.[5] This set of 473 questions was manually labelled with attributes from a predefined set of 50 attributes (presented in Fig. 10) that covers the course material. The labelled question set was used in our previous research [41], and it represents a fraction of more than 3500 questions stored in our e-learning system.

Various question types are supported (e.g. multiple-choice, true/false, fill-in or short answer and computational), although the questions in the selected set are multiple-choice (with three to five given answers, only one of them is correct),

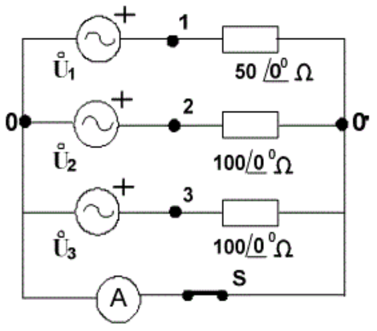
[5]https://osnove.tel.fer.hr – *WebOE* e-learning system used on the course *Fundamentals of Electrical Engineering*, developed at the Department of Electrical Engineering Fundamentals and Measurements of the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia

```
01: force_electric
02: field_electric
03: potential_electric
04: energy_electric
05: charge
06: capacity
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
```

FIGURE 10. **A predefined set of 50 attributes describing *Fundamentals of Electrical Engineering* course material.**

Question ID2909

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.



Answers:

A) 1200 W B) 2400 W C) 3600 W D) 6400 W E) 12800 W
(*correct answer D*)

Question labels (attributes):

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

FIGURE 11. **A example multiple-choice question from the questions' set.**

and computational, where students need to input one to three numerical solutions that are automatically evaluated. As an example, Fig. 11 shows one multiple-choice question from the set (question text is originally in Croatian) and the list of its attributes.

Although the text of the question presented in Fig. 11 seems quite short and straightforward, the solution of the stated problem is not trivial. Valuable additional information is included in the questions' figure, e.g. we see that the load is purely resistive, and it is connected in an unbalanced star configuration, and also that the ammeter is in the neutral line. To calculate the solution efficiently, we need to apply basic electrical laws (Ohm's law and Kirchhoff's current law), and need to know how to calculate current, voltage and power in AC circuits using phasor/vectors. We can see that this question is quite specific, and therefore it is labelled with an extensive list of attributes.

By testing the questions' set using combinatorial testing techniques we want to ensure that the questions cover all pairs or triples of attributes. Therefore, the questions will connect different learning concepts and will demand that the students learn the course material more thoroughly. To obtain best results, we also need to declare which combinations of attributes are not allowed. Some of them are not meaningful – e.g., pair (*phasors*, *non_sinusoidal_sources*), and others are too advanced or not covered by the course material – e.g., pair (*three_phase_sources*, *harmonics*). After automatically generating test-cases we need to check if they are already covered by the questions in the set, otherwise we should consider implementing uncovered test-cases as new questions. Additionally, we can use generated test-cases to check if the questions were labelled incorrectly. For example, if a generated test-case is meaningless, but is nevertheless covered by one or more questions, then those questions are probably labelled wrong.

Before running the combinatorial testing method, we need to prepare input text files as detailed in Section VI – list of attributes from Fig. 10 must be stored as *attribute_list.txt*, and the questions' formal context must be stored as *database_prepared_data.txt*. We extracted the formal context directly from our e-learning system database, and an excerpt of the *database_prepared_data.txt* input file is shown in Fig. 12.

```
2746 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 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
2747 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 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
2748 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 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
2749 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 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
2750 0 0 1 0 0 0 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 0 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
2751 0 0 1 0 0 0 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 0 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
2752 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 0 0 0
```

FIGURE 12. **An excerpt of the formal context from *database_prepared_data.txt*.**

```
DIMENSION 1:
force_electric, field_electric, potential_electric,
energy_electric, charge, capacity, capacitors,
inductance, inductive_coupling, force_magnetic,
field_magnetic, energy_magnetic, flow_magnetic,
induced_voltage, non_sinusoidal_sources, harmonics,
transients
DIMENSION 2:
Ohm, Kirchhoff, bridge, delta_star, superposition,
Thevenin, Norton, Millman
DIMENSION 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
DIMENSION 4:
DC_current, DC_voltage, DC_power, resistance,
AC_current, AC_voltage, AC_power, impedance, sine_wave,
vectors, phasors, frequency_dependence
```

FIGURE 13. **List of dimensions in Strategy 1.**

We should point out that the FCA method will find 3504 formal concepts from the formal context of the 473 questions. Therefore, it will generate a large and complex concept lattice containing 3504 vertices (formal concepts) and 15284 directed edges.

Afterwards, we need to define feature dimensions that will be used for combinatorial testing. As we stated earlier, each attribute will be considered as a distinct feature, and dimensions are disjoint subsets of features. The way in which dimensions are selected has a great influence on the number and quality of the test-cases. Therefore, we have prepared four different main strategies for defining dimensions:

- *Strategy 1* – features are grouped into four different dimensions. First dimension contains 17 features covering electrostatics, magnetism, and advanced circuits phenomena. Second dimension covers fundamental laws and theorems of electrical engineering, and it has 8 features. Third dimension is comprised of 13 features that cover ideal and real voltage and current sources, as well as three-phase systems. The last, fourth dimension has 12 features covering DC and AC electrical values, phasors, and frequency dependence. The list of dimensions is given in Fig. 13. There are at most $17 \cdot 13 \cdot 8 \cdot 12 = 21216$ different test-cases. This strategy has some drawbacks, as for each test-case only one feature per dimension is allowed, e.g. an interesting combination such as (*Thevenin*, *Millman*) will not be covered because they both belong to dimension 2.
- *Strategy 2* – rather than organizing multiple attributes in dimensions, in this approach we consider each of the 50 attributes in Fig. 10 as a separate dimension.

```
DIMENSION 1:
three_phase_star_balanced, inductive_coupling,
real_source, flow_magnetic, induced_voltage,
three_phase_sources, capacity, delta_star, sine_wave,
voltage_source, AC_power, frequency_dependence
DIMENSION 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
DIMENSION 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
DIMENSION 4:
field_magnetic, transients, potential_electric,
harmonics, bridge, sine_sources, Norton,
non_sinusoidal_sources, three_phase_neutral
```

FIGURE 14. **List of dimensions in Strategy 3.**

  Therefore, there are 50 dimensions, and for each dimension we defined only two feature values: *True* or *False*. Value *True* is denoted as *attribute_name_1* (then a test-case has that attribute), and value *False* is denoted as *attribute_name_0* (then a test-case does not have that attribute). In comparison to Strategy 1 all attribute combinations are now possible, but at the cost of enormous maximum number of test-cases $\left(2^{50}\right)$.
- *Strategy 3* – similarly to the Strategy 1, features are grouped into four dimensions, but now this partitioning is done randomly. Dimensions 1 – 4 are given in Fig. 14, and they have 12, 12, 17 and 9 features, respectively.
  Maximum number of all test-cases in Strategy 3 is $12 \cdot 12 \cdot 17 \cdot 9 = 22032$. This is comparable to the number of test-cases in Strategy 1. Again, there are drawbacks, e.g. a very interesting feature combination (*Ohm*, *DC_voltage*) will not be covered by any test-case (they both belong to dimension 2), but a feature combination (*Thevenin*, *Millman*), not covered in Strategy 1 will now be covered (now those features belong to dimensions 2 and 3, respectively).
- *Strategy 4* – in this approach we start with the dimensions defined in Strategy 1 (Fig. 13), but then we simplify them by grouping closely related features into feature groups. Feature groups are shown in Fig. 15 and simplified dimensions are given in Fig. 16.
  By taking this approach we have reduced the size of the dimensions, which now have 7, 8, 5 and 6 features, respectively. Because of that, the maximum number of all test-cases is significantly smaller, $7 \cdot 8 \cdot 5 \cdot 6 = 1680$.
  When checking if the question is covered by the generated test-case, we will consider that a feature group (from Fig. 15) is covered if the question has at least one of the attributes in that feature group. This gives us some flexibility in the questions' testing process, e.g. a test-case that contains a feature group *phasor_diagram* could be matched by a question with at least one of the following attributes: *sine_wave*, *vectors*, and *phasors*.

```
electrostatics:        magnetism:
force_electric         inductive_coupling
field_electric         force_magnetic
energy_electric        field_magnetic
charge                 energy_magnetic
capacitors             flow_magnetic
                       induced_voltage

DC_values:             sources:
DC_current             current_source
DC_voltage             voltage_source
resistance             real_source
                       ideal_source
                       sine_sources
                       non_sinusoidal_sources

AC_values:             phasor_diagrams:
AC_current             sine_wave
AC_voltage             vectors
impedance              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
```

FIGURE 15. Feature groups used in Strategy 4.

```
DIMENSION 1:
electrostatics, magnetism, potential_electric, capacity,
inductance, harmonics, transients

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

DIMENSION 3:
sources, three_phase_easier, three_phase_star,
three_phase_delta, three_phase_harder

DIMENSION 4:
DC_values, AC_values, DC_power, AC_power,
phasor_diagrams, frequency_dependence
```

FIGURE 16. List of dimensions in Strategy 4.

We have used each of the four outlined strategies to generate test-cases that cover all pairs of features (tuple size $t = 2$) and all triples of features (tuple size $t = 3$) from the defined dimensions, with and without declared forbidden feature combinations.

VIII. RESULTS AND DISCUSSION

In this section we will present results of the proposed combinatorial testing method applied to the initial set of labelled 473 questions taken from our e-learning system database. A test-cases list is generated in every iteration of the combinatorial testing process, and our Python script *combinatorial_test_run.py* automatically checks if each generated test-case is covered fully or partially by the questions in the database, as was detailed in Section VI. Finally, all meaningful uncovered test-cases should be implemented as new questions and added to the database.

TABLE 2. Combinatorial testing results using Strategy 1.

Test run	1	2	3	4
Test parameters	$t = 2$	$t = 2$ and 3 forbidden combinations	$t = 3$	$t = 3$ and 3 forbidden combinations
Number of test-cases	225	223	2670	2641
Test-cases fully covered	11 [4.89%]	17 [7.62%]	153 [5.73%]	163 [6.17%]
Test-cases partially covered	102 [45.33%]	109 [48.88%]	1280 [47.94%]	1282 [48.54%]
Questions fully covered by test-cases	124 [26.22%]	113 [23.89%]	234 [49.47%]	257 [54.33%]

First, we will give results of the initial combinatorial test runs using four different strategies as described in Section VII. Each of the strategies has been run four times, using a feature tuple size of $t = 2$ and $t = 3$, and with, as well as without a declared list of feature combinations that are not allowed in the generated test-cases. Also, in all test runs a test-case will be considered as partially covered if there are one or more questions whose attributes match at least 3 test-case features. Afterwards, we will choose the most appropriate strategy and complete the whole combinatorial testing process.

A. COMBINATORIAL TESTING RESULTS USING STRATEGY 1

In the first testing strategy we used four feature dimensions listed in Fig. 13, and the testing results are presented in Table 2. Each data column in the table corresponds to a test run with a different set of input parameters.

For the first test run, we set the feature tuple size to $t = 2$ and declared no forbidden feature combinations. Number of generated test-cases was 225. There are 102 test-cases (45.33%) that are partially covered by the questions, but only 11 test-cases (4.89%) are fully covered by at least one of the questions. Note that 124 of 473 questions (26.22%) are covered by at least one test-case, but this is mainly due to the test-case (*potential_electric*, *Ohm*, *sine_sources*, *impedance*) that is covered by 107 questions.

In the second test run, we set $t = 2$ and declared a short list of three forbidden feature combinations: (*DC_current*, *harmonics*), (*DC_current*, *induced_voltage*) and (*force_electric*, *Millman*). Subsequently, the number of generated test-cases was reduced to 223, and 7.62% of test-cases were fully covered by the questions.

In the third and fourth test runs we set the number of feature tuples to $t = 3$. In the third run we did not use forbidden feature combinations, and in the fourth run the above-mentioned short list of forbidden feature combinations was used. 2670 test-cases were generated in the third run, and 2641 test-cases were generated in the fourth run. The percentage of fully covered test-cases was very small in both test runs (5.73% and 6.17%, respectively). Because of the greater number of generated test-cases the ratio of questions

TABLE 3. Combinatorial testing results using Strategy 2.

Test run	1	2	3	4
Test parameters	$t = 2$	$t = 2$ and 85 forbidden combinations	$t = 3$	$t = 3$ and 85 forbidden combinations
Number of test-cases	14	17	36	46
Test-cases fully covered	0 [0.0%]	0 [0.0%]	0 [0.0%]	0 [0.0%]
Test-cases partially covered	14 [100.0%]	17 [100.0%]	36 [100.0%]	44 [95.65%]
Questions fully covered by test-cases	0 [0.0%]	0 [0.0%]	0 [0.0%]	0 [0.0%]

TABLE 4. Combinatorial testing results using Strategy 3.

Test run	1	2	3	4
Test parameters	$t = 2$	$t = 2$ and 4 forbidden combinations	$t = 3$	$t = 3$ and 4 forbidden combinations
Number of test-cases	213	209	2480	2443
Test-cases fully covered	13 [6.1%]	13 [6.22%]	161 [6.49%]	162 [6.63%]
Test-cases partially covered	88 [41.31%]	88 [42.11%]	1088 [43.87%]	1057 [43.27%]
Questions fully covered by test-cases	127 [26.85%]	164 [23.89%]	227 [47.99%]	234 [49.47%]

covered by test-cases increased to 49.47% and 54.33%, respectively.

As we can see from Table 2, this strategy initially produces many different test-cases, especially when we want to cover all feature triples. Also, in all four test runs initial test-case full coverage is very small. This could be countered by generating a more extensive list of forbidden feature combinations, but this is a very delicate and time-consuming job considering there are 50 features. Furthermore, as we noted in Section VII, this strategy will fail to cover some meaningful feature combinations if they are included in the same dimension.

B. COMBINATORIAL TESTING RESULTS USING STRATEGY 2

In the second testing strategy we used fifty feature dimensions, each corresponding to one of the features, and the testing results are presented in Table 3.

Using this strategy, we did not obtain good results. In all four test runs, using feature tuples sizes 2 and 3, without and with a list of 85 forbidden feature combinations, there were initially no fully covered test-cases. As there are 50 dimensions, each with only 2 feature values (*True* and *False*) combinatorial testing procedure will efficiently find a small number of test-cases that will cover all possible feature pairs (when $t = 2$) or feature triples (when $t = 3$) at least once. Unfortunately, almost all of the 14 to 46 test-cases generated in the four test runs represent feature combinations that are too complex, contradictory or not meaningful, even when using a list of 85 forbidden feature combinations. This can explain why none of the test-cases was fully covered by at least one existing question in the database. Furthermore, such test-cases that include numerous various features would be hard to implement as new questions. Again, one possible time-consuming solution to this problem is to devise a much longer list of forbidden feature combinations. Also, we could consider expanding feature tuple sizes to $t = 4$ or more, which would increase the number of generated test-cases, but also increase the probability that the test-cases would be more meaningful. Note that the almost all of the generated test-cases in four test runs were partially covered by the questions. However, this is not too surprising because each test-case has 50 features, and it was easy to find questions whose attributes match with at least 3 out of its 50 features.

C. COMBINATORIAL TESTING RESULTS USING STRATEGY 3

In the third testing strategy we used four randomly generated dimensions listed in Fig. 14, and the testing results are presented in Table 4.

This strategy produced quantitatively very similar results to the Strategy 1 where all four dimensions were manually set. Here, in second and fourth test run we used a small list of four forbidden feature combinations: (*real_source*, *field_electric*), (*induced_voltage*, *harmonics*), (*three_phase_delta_balanced*, *DC_current*) and (*three_phase_star_balanced*, *DC_current*), but the full coverage of the test-cases did not improve considerably. Most of the generated test-cases in test runs 1 and 2 were not meaningful, which was expected because the 50 features were randomly partitioned into four dimensions. However, in test runs 3 and 4 there was a significant number of useful test-cases, e.g. test-case no. 1700 in test run 4: (*real_source*, *current_source*, *DC_current*, *Norton*) that was not among the generated test-cases when using Strategy 1.

D. COMBINATORIAL TESTING RESULTS USING STRATEGY 4

In the fourth testing strategy we used four dimensions listed in Fig. 16, together with feature groups from Fig. 15, and the testing results are presented in Table 5.

In this approach, we used feature groups to reduce the size of the dimensions. As a result, combinatorial testing process generated considerably fewer test-cases. When covering all feature pairs ($t = 2$), there were 56 test-cases generated in the first test run and 66 test-cases in the second run (with forbidden feature combinations). Also, in coverage of all feature triples ($t = 3$), there were 352 test-cases in the third test run and 321 test-cases in the fourth test run (when forbidden feature combinations are declared). There were initially 30 forbidden feature combinations used in test runs 2 and 4, and they are listed in Fig. 17.

TABLE 5. Combinatorial testing results using Strategy 4.

Test run	1	2	3	4
Test parameters	$t = 2$	$t = 2$ and 30 forbidden combinations	$t = 3$	$t = 3$ and 30 forbidden combinations
Number of test-cases	56	66	352	321
Test-cases fully covered	9 [16.07%]	14 [21.21%]	55 [15.63%]	76 [23.68%]
Test-cases partially covered	40 [71.43%]	55 [83.33%]	223 [63.35%]	262 [81.62%]
Questions fully covered by test-cases	142 [26.85%]	80 [16.91%]	246 [52.01%]	253 [53.49%]

```
(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)
```

FIGURE 17. Initial list of forbidden feature combinations in Strategy 4.

Furthermore, in comparison to previous strategies a greater percentage of test-cases was initially fully covered by the questions, 16.07% in test run 1 and 15.63% in test run 3, and when using forbidden feature combinations 21.21% for test run 2 and 23.68% for test run 4. Note, that when we required feature coverage of all pairs $56 - 9 = 47$ test-cases were not fully covered in test run 1, and $66 - 14 = 52$ test-cases were not fully covered in test run 2. These uncovered test-cases, if meaningful, should be implemented as new questions, and added to the initial set of 473 questions. In test runs 3 and 4, which required coverage of all feature triples there were considerably more test-cases not fully covered (297 and 245, respectively). However, these numbers are still manageable compared to results of third and fourth test runs in strategies 1 and 3.

E. DISCUSSION

From the results of initial combinatorial test runs using all four presented strategies we can conclude that the number of generated test-cases varies substantially according to the size of the feature tuples that need to be covered, and depending on the number and sizes of the feature dimensions. To maximize the potential benefit that the combinatorial testing provides, special care should be given when selecting feature dimensions, and also when determining feature combinations that are not allowed. Listing all the forbidden feature combinations is a hard task, but the combinatorial testing itself can help us to identify them in the generated test-cases. Moreover, note that all combinations of features placed in same dimension are implicitly forbidden, e.g. *DC_values* and *frequency_dependence* from dimension 4 in Fig. 16.

By examining the results of the combinatorial testing runs we found that large subsets of questions are covered by a few generated test-cases. For example, in the third test run from Strategy 4 ($t = 3$, no forbidden combinations used) there were 246 questions covered by at least one test-case, and the dominant test-cases were (*potential_electric*, *Ohm*, *sources*, *DC_values*) – covered by 207/246 questions, and test-case (*potential_electric*, *Ohm*, *sources*, *AC_values*) – covered by 139/246 questions. This is expected, as the focus of our freshman year course *Fundamentals of Electrical Engineering* is on the basic DC and AC circuit analysis. On the contrary, only 1 of 246 questions was covered by a test-case (*inductance*, *Ohm*, *three_phase_delta*, *AC_values*). Therefore, teachers could consider devising additional questions about inductive delta-connected loads in three phase systems.

After completing initial test runs using all four presented strategies, we have decided to use Strategy 4 to demonstrate the complete combinatorial testing process until all test-cases are covered by the questions. As we can see from the results, Strategy 4 has the best initial test-case coverage, and a manageable number of uncovered test-cases. Furthermore, the usage of simplified dimensions with feature groups from Fig. 15 offers teachers and instructors more flexibility in preparing new questions based on uncovered generated test-cases.

In future research we will also consider using other combinatorial testing tools to compare results and performance with the *Jenny* program used in these experiments. Also, proposed combinatorial testing method is currently implemented only as a command-line tool. Therefore, our aim is to provide a Web interface for the implemented combinatorial testing method that could greatly help teachers and instructors in applying the proposed method on their courses.

F. COMPLETE COMBINATORIAL TESTING PROCEDURE USING STRATEGY 4

For this demonstration, we used the input data prepared for Strategy 4, with feature groups from Fig. 15 and four

TABLE 6. Complete combinatorial testing process using Strategy 4 ($t = 2$).

Test run	1	2	3	4	5
Number of questions in the database	473	473	473	473	522
Number of forbidden combinations	30	38	39	42	42
Number of test-cases	66	71	71	72	72
Test-cases fully covered	14 [21.21%]	16 [22.54%]	16 [22.54%]	23 [31.94%]	72 [100.0%]
Test-cases partially covered	55 [83.33%]	65 [91.55%]	65 [91.55%]	65 [90.28%]	72 [100.0%]
Questions fully covered by test-cases	80	56	70	90	139

feature dimensions from Fig. 16. A coverage of all pairs of features between the feature dimension is required, so the size of the feature tuples was set to $t = 2$. The initial list of 30 forbidden feature combinations from Fig. 17 was also used.

After each test run all of the generated test-cases were inspected for meaningless feature combinations or those too advanced for our course. If such feature combinations were identified, they were added to the list of the forbidden feature combinations and the test was run again. When all of the uncovered test-cases are meaningful, we treat them as prepared suggestions for new questions. Teachers implement new questions according to the uncovered generated test-cases and add them to the questions' set. Also, the formal context of the questions' set is updated with the labelled description of new questions. The combinatorial testing process continues until each of the generated test-cases is covered by at least one question. The complete results of this combinatorial testing procedure are presented in Table 6.

Note, that the first test run is equal to the initial test run 2 from Table 5. In second and third row of the Table 6 we give the current size of the questions' set and the current number of feature combinations that are not allowed. Furthermore, it can be seen that we successively increased the number of forbidden feature combinations from 30 in the initial first test run to 42 in the fourth and fifth test run. All forbidden feature combinations additionally added to the initial list in Fig. 17 in test runs 2, 3 and 4 can be seen in Fig. 18.

As the number of forbidden feature combinations increased, the percentage of test-cases covered fully and partially by the questions also grew. However, after eliminating all unwanted feature combinations in test run 4 only 23/72 generated test-cases were fully covered by the questions. Therefore, after test run 4, we needed to create at least 49 new questions that implement the remaining 49 meaningful test-cases, which were not fully covered by the existing questions. Using the script *database_manager.py* we added these 49 new questions to the initial questions set, thereby increasing its size to 522. Afterwards, the combinatorial

```
ADDED IN TEST RUN 2:
(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)

ADDED IN TEST RUN 3:
(harmonics, DC_power)

ADDED IN TEST RUN 4:
(harmonics, frequency_dependence)
(magnetism, frequency_dependence)
(magnetism, DC_power)
```

FIGURE 18. Additional forbidden feature combinations added to the list in Fig. 17.

testing process successfully ended in the fifth test run because each of the 72 generated test-cases was fully covered by at least one question.

Test-cases' overview in Fig. 19 shows that majority of test-cases are partially covered by a substantial number of questions (i.e. match 3/4 test-case features), however there are only a few test-cases that are fully covered by a considerable number of questions (i.e. match all 4 test-case features).

A detailed list of all the test-cases generated in the fifth test run is presented in Table 7, including test-case ID, test-case features, and numbers of questions that fully and partially cover each test-case. Test-cases that were fully covered in the final – fifth test run after adding 49 new questions to the database are marked with an asterisk.

As an example, we show one of the new 49 questions in Fig. 20. This question covers the test-case ID13 (*electrostatics*, *bridge*, *sources*, *DC_values*), and here we can see the flexibility of the feature groups listed in Fig. 15. Feature group *electrostatics* is covered by attributes *capacitors*, *charge* and *energy_electric*; feature group *sources* is covered by attributes *ideal_source* and *voltage_source*, and feature group *DC _values* is covered by attribute *DC_voltage*. Note that the question can be labelled with additional attributes, e.g. *capacity* and *delta_star*, provided that all the features from the test-case are covered.

After the combinatorial testing procedure was completed, we see that in the final set of 522 questions there is a subset of 139 questions covered by all the 72 generated test-cases. Most of the generated test-cases are covered by 1 to 4 questions, but there are some exceptions, e.g. test case ID67 (*inductance*, *Ohm*, *sources*, *frequency_dependence*) is covered by most questions, 32/139. Our combinatorial testing method also automatically prepares the formal context of the resulting subset of 139 questions that covered all of the generated test-cases. Afterwards, this formal context is processed using the FCA method according to the Steps 5 and 6 of our method for automated assessment generation (Fig. 4). First, a concept lattice is automatically built from the formal context, and then our method identifies suitable sequences of

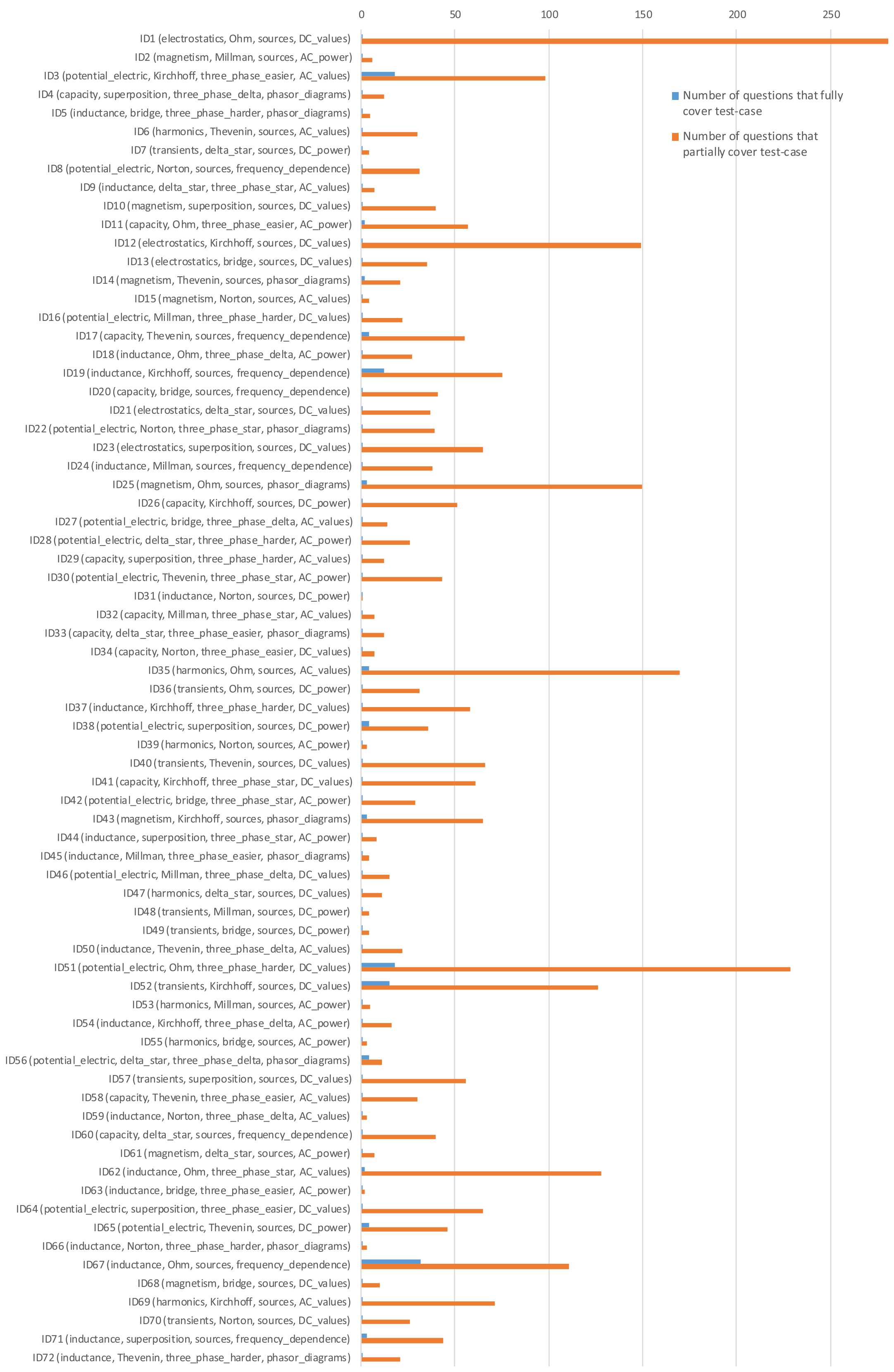


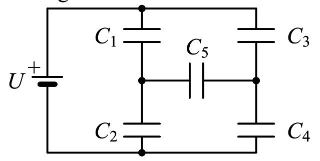
FIGURE 19. **Overview of test-cases in the final, fifth test run.**

TABLE 7. List of test-cases generated in the final, fifth test run.

ID	Test-case features	Questions fully covered	Questions partially covered
1	(*electrostatics*, *Ohm*, *sources*, *DC_values*)	1	281
2*	(*magnetism*, *Millman*, *sources*, *AC_power*)	1	6
3	(*potential_electric*, *Kirchhoff*, *three_phase_easier*, *AC_values*)	18	98
4*	(*capacity*, *superposition*, *three_phase_delta*, *phasor_diagrams*)	1	12
5*	(*inductance*, *bridge*, *three_phase_harder*, *phasor_diagrams*)	1	5
6*	(*harmonics*, *Thevenin*, *sources*, *AC_values*)	1	30
7*	(*transients*, *delta_star*, *sources*, *DC_power*)	1	4
8*	(*potential_electric*, *Norton*, *sources*, *frequency_dependence*)	1	31
9*	(*inductance*, *delta_star*, *three_phase_star*, *AC_values*)	1	7
10	(*magnetism*, *superposition*, *sources*, *DC_values*)	1	40
11	(*capacity*, *Ohm*, *three_phase_easier*, *AC_power*)	2	57
12*	(*electrostatics*, *Kirchhoff*, *sources*, *DC_values*)	1	149
13*	(*electrostatics*, *bridge*, *sources*, *DC_values*)	1	35
14	(*magnetism*, *Thevenin*, *sources*, *phasor_diagrams*)	2	21
15*	(*magnetism*, *Norton*, *sources*, *AC_values*)	1	4
16	(*potential_electric*, *Millman*, *three_phase_harder*, *DC_values*)	1	22
17	(*capacity*, *Thevenin*, *sources*, *frequency_dependence*)	4	55
18	(*inductance*, *Ohm*, *three_phase_delta*, *AC_power*)	1	27
19	(*inductance*, *Kirchhoff*, *sources*, *frequency_dependence*)	12	75
20*	(*capacity*, *bridge*, *sources*, *frequency_dependence*)	1	41
21*	(*electrostatics*, *delta_star*, *sources*, *DC_values*)	1	37
22*	(*potential_electric*, *Norton*, *three_phase_star*, *phasor_diagrams*)	1	39
23*	(*electrostatics*, *superposition*, *sources*, *DC_values*)	1	65
24*	(*inductance*, *Millman*, *sources*, *frequency_dependence*)	1	38
25	(*magnetism*, *Ohm*, *sources*, *phasor_diagrams*)	3	150
26*	(*capacity*, *Kirchhoff*, *sources*, *DC_power*)	1	51
27*	(*potential_electric*, *bridge*, *three_phase_delta*, *AC_values*)	1	14
28*	(*potential_electric*, *delta_star*, *three_phase_harder*, *AC_power*)	1	26
29*	(*capacity*, *superposition*, *three_phase_harder*, *AC_values*)	1	12
30*	(*potential_electric*, *Thevenin*, *three_phase_star*, *AC_power*)	1	43
31*	(*inductance*, *Norton*, *sources*, *DC_power*)	1	1
32	(*capacity*, *Millman*, *three_phase_star*, *AC_values*)	1	7
33*	(*capacity*, *delta_star*, *three_phase_easier*, *phasor_diagrams*)	1	12
34*	(*capacity*, *Norton*, *three_phase_easier*, *DC_values*)	1	7
35	(*harmonics*, *Ohm*, *sources*, *AC_values*)	4	170
36*	(*transients*, *Ohm*, *sources*, *DC_power*)	1	31
37*	(*inductance*, *Kirchhoff*, *three_phase_harder*, *DC_values*)	1	58
38	(*potential_electric*, *superposition*, *sources*, *DC_power*)	4	36
39*	(*harmonics*, *Norton*, *sources*, *AC_power*)	1	3
40*	(*transients*, *Thevenin*, *sources*, *DC_values*)	1	66
41*	(*capacity*, *Kirchhoff*, *three_phase_star*, *DC_values*)	1	61
42*	(*potential_electric*, *bridge*, *three_phase_star*, *AC_power*)	1	29
43	(*magnetism*, *Kirchhoff*, *sources*, *phasor_diagrams*)	3	65
44*	(*inductance*, *superposition*, *three_phase_star*, *AC_power*)	1	8
45	(*inductance*, *Millman*, *three_phase_easier*, *phasor_diagrams*)	1	4
46*	(*potential_electric*, *Millman*, *three_phase_delta*, *DC_values*)	1	15
47*	(*harmonics*, *delta_star*, *sources*, *DC_values*)	1	11
48*	(*transients*, *Millman*, *sources*, *DC_power*)	1	4
49*	(*transients*, *bridge*, *sources*, *DC_power*)	1	4
50*	(*inductance*, *Thevenin*, *three_phase_delta*, *AC_values*)	1	22
51	(*potential_electric*, *Ohm*, *three_phase_harder*, *DC_values*)	18	229
52	(*transients*, *Kirchhoff*, *sources*, *DC_values*)	15	126
53*	(*harmonics*, *Millman*, *sources*, *AC_power*)	1	5
54	(*inductance*, *Kirchhoff*, *three_phase_delta*, *AC_power*)	1	16
55*	(*harmonics*, *bridge*, *sources*, *AC_power*)	1	3
56	(*potential_electric*, *delta_star*, *three_phase_delta*, *phasor_diagrams*)	4	11
57*	(*transients*, *superposition*, *sources*, *DC_values*)	1	56
58*	(*capacity*, *Thevenin*, *three_phase_easier*, *AC_values*)	1	30
59*	(*inductance*, *Norton*, *three_phase_delta*, *AC_values*)	1	3
60*	(*capacity*, *delta_star*, *sources*, *frequency_dependence*)	1	40
61*	(*magnetism*, *delta_star*, *sources*, *AC_power*)	1	7
62	(*inductance*, *Ohm*, *three_phase_star*, *AC_values*)	2	128
63*	(*inductance*, *bridge*, *three_phase_easier*, *AC_power*)	1	2
64*	(*potential_electric*, *superposition*, *three_phase_easier*, *DC_values*)	1	65
65	(*potential_electric*, *Thevenin*, *sources*, *DC_power*)	4	46
66*	(*inductance*, *Norton*, *three_phase_harder*, *phasor_diagrams*)	1	3
67	(*inductance*, *Ohm*, *sources*, *frequency_dependence*)	32	111
68*	(*magnetism*, *bridge*, *sources*, *DC_values*)	1	10
69*	(*harmonics*, *Kirchhoff*, *sources*, *AC_values*)	1	71
70*	(*transients*, *Norton*, *sources*, *DC_values*)	1	26
71	(*inductance*, *superposition*, *sources*, *frequency_dependence*)	3	44
72*	(*inductance*, *Thevenin*, *three_phase_harder*, *phasor_diagrams*)	1	21

Question ID5009

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.



Answers:

A) 48 µJ B) 64 µJ C) 96 µJ D) 128 µJ E) 0 J

(*correct answer E*)

Question labels (attributes):

energy_electric, charge, capacitors, potential_electric, capacity, voltage_source, ideal_source, DC_voltage, Kirchhoff, bridge, delta_star

FIGURE 20. A new multiple-choice question implementing test-case ID13.

questions that can be used for formative student assessment in e-learning systems. Each sequence of questions starts with the most general questions (usually easier) and ends with the most specific questions (usually harder). It must be noted that from the final formal context of 139 questions the FCA method found 1024 formal concepts and built a large concept lattice, consisting of 1024 vertices (formal concepts) and 3371 directed edges. However, it is far more manageable than the concept lattice of all the 473 questions in the initial set (3504 vertices and 15284 directed edges, as stated earlier in Section VII).

IX. CONCLUSION AND FUTURE WORK

In this paper we proposed a method for automated generation of assessments in e-learning systems, by focusing on the questions that cover and combine multiple topics from the course material. We used the FCA method to formally describe each question in the database with a subset of attributes. Then, we applied combinatorial testing technique used in software testing as a tool for identifying and selecting an almost minimal set of questions that covers all the defined attributes and ensures that all meaningful pairs or triples of attributes are covered by at least one question. The implemented combinatorial testing method can also give educators the descriptions of new questions that could be created and added to the repository of questions. Lastly, we outlined how the FCA method can automatically generate sequences of questions sorted by their generality, which is primarily suitable for use in formative student assessments.

To demonstrate the usefulness of the implemented combinatorial testing method we applied it to the initial set of 473 labelled questions from our e-learning system. Our method suggested 49 new questions that cover additional pairs of attributes that were missing in the initial question set. Furthermore, the method found a subset of 139 questions, such that all meaningful pairs of attributes are covered by at least one question.

In future work we will use other established combinatorial testing tools to compare their results and performance with the *Jenny* combinatorial testing tool used in this research. Also, to facilitate the usage of the proposed combinatorial testing method we will aim to create a Web-based interface for the developed command-line tools. Furthermore, we will apply process mining and model checking techniques to analyze students' performance and activity when taking formative student assessments created using our method. Teachers and instructors could then closely monitor students' progress and timely detect any anomalies and warning signs that suggest a student might not pass the course.

REFERENCES

[1] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*. Boca Raton, FL, USA: CRC Press, 2013.

[2] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Prof.*, vol. 10, no. 3, pp. 19–23, May 2008.

[3] 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 *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2019, pp. 614–618.

[4] L. Sh. Ghandehari, Y. Lei, R. Kacker, R. Kuhn, T. Xie, and D. Kung, "A combinatorial testing-based approach to fault localization," *IEEE Trans. Softw. Eng.*, vol. 46, no. 6, pp. 616–645, Jun. 2020.

[5] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 1–29, Jan. 2011.

[6] S. K. Khalsa and Y. Labiche, "An orchestrated survey of available algorithms and tools for combinatorial testing," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*, Nov. 2014, pp. 323–334.

[7] Q. N. Naveed, M. R. N. Mohamed 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.

[8] D. I. Borissova and D. Keremedchiev, "Generation of E-learning tests with different degree of complexity by combinatorial optimization," *J. E-Learn. Knowl. Soc.*, vol. 16, no. 2, pp. 17–24, 2020.

[9] F. Vaandrager, "Model learning," *Commun. ACM*, vol. 60, no. 2, pp. 86–95, Jan. 2017.

[10] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[11] B. Blaškovic, F. Škopljanac-Mačina, and I. Zakarija, "Discovering E-learning process models from counterexamples," in *Proc. 41st Int. Conv. Inf. Commun. Technol., Electron. Microelectron. (MIPRO)*, 2018, pp. 0593–0598.

[12] 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, Jun. 2015.

[13] P. K. Singh, C. Aswani Kumar, and A. Gani, "A comprehensive survey on formal concept analysis, its research trends and applications," *Int. J. Appl. Math. Comput. Sci.*, vol. 26, no. 2, pp. 495–516, Jun. 2016.

[14] S. M. Dias, L. E. Zárate, M. A. J. Song, N. J. Vieira, and C. A. Kumar, "Extraction of qualitative behavior rules for industrial processes from reduced concept lattice," *Intell. Data Anal.*, vol. 24, no. 3, pp. 643–663, May 2020.

[15] J. Annapurna and A. K. Cherukuri, "Exploring attributes with domain knowledge in formal concept analysis," *J. Comput. Inf. Technol.*, vol. 21, no. 2, pp. 109–123, 2013.

[16] M. A. Bedek, M. D. Kickmeier-Rust, and D. Albert, "Formal concept analysis for modelling students in a technology-enhanced learning setting," in *Proc. 5th Workshop Awareness Reflection Technol. Enhanced Learn. Conjunct*, Toledo, Spain, vol. 1465, M. Kravcik, Ed., Sep. 2015, pp. 27–33.

[17] G. Beydoun, "Using formal concept analysis towards cooperative E-learning," in *Knowledge Acquisition: Approaches, Algorithms and Applications* (Lecture Notes in Computer Science), vol. 5465, D. Richards and B. H. Kang, Eds. Hanoi, Vietnam: Springer, Dec. 2008, pp. 109–117.

[18] G. Beydoun, "Formal concept analysis for an e-learning semantic Web," *Expert Syst. Appl.*, vol. 36, no. 8, pp. 10952–10961, Oct. 2009.

[19] U. Priss, "Using FCA to analyse how students learn to program," in *Formal Concept Analysis* (Lecture Notes in Computer Science), vol. 7880, P. Cellier, Ed. Dresden, Germany: Springer, May 2013, pp. 216–227.

[20] G. J. Holzmann, *The SPIN Model Checker: Primer Reference Manual*. Reading, MA, USA: Addison-Wesley, 2004.

[21] I. Zakarija, F. Škopljanac-Macina, and B. Blaškovic, "Automated simulation and verification of process models discovered by process mining," *Automatika*, vol. 61, no. 2, pp. 312–324, Apr. 2020.

[22] W. M. P. van der Aalst, *Data Science in Action*. New York, NY, USA: Springer, 2016.

[23] M. A. Almaiah and I. Y. Alyoussef, "Analysis of the effect of course design, course content support, course assessment and instructor characteristics on the actual use of E-learning system," *IEEE Access*, vol. 7, pp. 171907–171922, 2019.

[24] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 418–421, Jun. 2004.

[25] R. D. Kuhn, R. N. Kacker, and Y. Lei, "Combinatorial testing," in *Encyclopedia of Software Engineering*, P. A. Laplante, Ed. Boca Raton, FL, USA: CRC Press, 2010.

[26] D. R. Stinson, *Combinatorial Designs: Constructions Analysis*. New York, NY, USA: Springer-Verlag, 2004.

[27] (2009). *AllPairs*. [Online]. Available: https://sourceforge.net/projects/allpairs

[28] (2021). *Pict*. [Online]. Available: https://github.com/microsoft/pict

[29] (2020). *Tcases*. [Online]. Available: https://github.com/cornutum/tcases

[30] (2005). *Jenny*. [Online]. Available: https://burtleburtle.net/bob/math/jenny.html

[31] (2016).*ACTS*. [Online]. Available: https://csrc.nist.gov/acts

[32] B. Ganter, G. Stumme, and R. Wille, Eds., *Formal Concept Analysis—Foundations and Applications*. Berlin, Germany: Springer-Verlag, 2005.

[33] R. Belohlavek. (2008). *Introduction to Formal Concept Analysis*. [Online]. Available: https://belohlavek.inf.upol.cz/vyuka/IntroFCA.pdf

[34] R. Wille, "Restructuring lattice theory: An approach based on hierarchies of concepts," *Ordered Sets*, I. Rival, Ed. Dordrecht, The Netherlands: Reidel, 1982, pp. 445–470.

[35] B. Ganter and R. Wille, *Formal Concept Analysis—Mathematical Foundations* Berlin, Germany: Springer-Verlag, 1999.

[36] T. S. Blyth, *Lattices and Ordered Algebraic Structures*. London, U.K.: Springer-Verlag, 2005.

[37] (2009). *Concept Explorer 1.3*. [Online]. Available: https://www.sourceforge.net/projects/conexp

[38] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. New York, NY, USA: McGraw-Hill, 2020.

[39] (2021). *Sqlite3*. [Online]. Available: https://www.sqlite.org/cli.html

[40] (2021). *SQLiteStudio*. [Online]. Available: https://sqlitestudio.pl

[41] F. Skopljanac-Macina, B. Blaskovic, and Z. Skocir, "Using formal concept analysis for student assessment," in *Proc. ELMAR*, Zadar, Croatia, Sep. 2014, pp. 285–288.

FRANO ŠKOPLJANAC-MAČINA (Member, IEEE) received the M.S. degree in electrical engineering from the Faculty of Electrical Engineering and Computing (FER), University of Zagreb, in 2009, where he is currently pursuing the Ph.D. degree in electrical engineering. He was an Affiliate with the Department of Electrical Engineering Fundamentals and Measurements, FER, from 2010 to 2015, where he has been the Laboratory Manager, since 2015. His research interests include formal methods and automated assessment techniques in designing advanced adaptive e-learning systems. He has been a member of the IEEE Education Society since 2012.

IVONA ZAKARIJA (Member, IEEE) received the B.Sc., M.Sc., and Ph.D. degrees from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 1993, 2011, and 2020, respectively. After graduation, she moved to Dubrovnik and started to work with "ITI Computers." In 1997, she changed employer and started to work with "Nivel" Company. In 2000, she was employed with the "Laus CC." Since 2006, she has been working with the University of Dubrovnik, where she is teaching several courses in computing. She has published 16 papers in scientific journals and in the proceedings of international scientific conferences. Her research interests include design and building of information systems, deep learning, artificial intelligence, data science, and process mining. She is also a member of the IEEE Computer Society and Big Data Community.

BRUNO BLAŠKOVIĆ (Member, IEEE) received the B.Sc., M.Sc., and Ph.D. degrees in electrical engineering from the Faculty of Electrical Engineering and Computing (FER), University of Zagreb, Zagreb, Croatia, in 1982, 1985, and 1996, respectively. In December 2003, he was promoted to an Associate Professor. Since December 1986, he has been working with the Department of Electrical Engineering Fundamentals and Measurements, FER. He published over 80 papers in journals and conference proceedings. His current research interests include formal methods, model checking, model transformations, business process modeling, protocol and software synthesis, software testing, e-learning, inteligent tutoring systems, satisfiability modulo theories (SMT), and network reliability. He serves on the editorial boards of scientific journals and the international program committee of numerous conferences. He is also a member of ACM.