CARPATHIAN J. MATH.
Volume **38** (2022), No. 1,
Pages 159 - 168

Online version at https://www.carpathian.cunbm.utcluj.ro/
Print Edition: ISSN 1584 - 2851; Online Edition: ISSN 1843 - 4401
DOI: https://doi.org/10.37193/CJM.2022.01.13

Dedicated to the memory of Academician Mitrofan M. Choban (1942-2021)

Formal concept analysis model for static code analysis

SIMONA MOTOGNA, DIANA CRISTEA, DIANA ŞOTROPA and ARTHUR-JOZSEF MOLNAR

ABSTRACT. Tools that focus on static code analysis for early error detection are of utmost importance in software development, especially since the propagation of errors is strongly related to higher costs in the development process. Formal Concept Analysis is a prominent field of applied mathematics that uses conceptual landscapes to discover and represent maximal clusters of data. Its expressive visualization method makes it suitable for exploratory analyses in different fields. In this paper we present a Formal Concept Analysis framework for static code analysis that can serve as a model for quantitative and qualitative exploration and interpretation of such results.

1. INTRODUCTION

Progresses in programming languages, software development methodologies and associated tooling, such as integrated development environments (IDEs) have made it possible to produce larger and more complex software applications, developed over longer periods of time by larger teams. In these conditions, software development should also include practices to control the process and assure the desired quality. Code review has been acknowledged as a fundamental step in software development with clear benefits of early error detection and quality control.

Even as manual code review is still used, code review tools are introduced to complement existing human expertise, especially for large and complex applications. The introduction of static analysis techniques in these tools has significantly improved their capabilities.

The main benefit of static code analysis resides in the fact that it provides important information about the source code prior to its execution. Several formal methods have been applied to construct strategies such as abstract interpretation [5], data flow analysis [9], symbolic execution [10] and many others. Static analysis tools are regarded as of third generation, since the strategies started to be applied on the program abstract syntax tree [13]. Thus, they allow early detection of several issues including software defects, bad practices, code smells and security vulnerabilities.

Such tools are available for several programming languages, can be easily integrated in development processes and customized according to the programmers' needs. Some existing tools, such as SonarQube, Coverity or PMD can analyze code in several programming languages, while others are language specific, such as Checkstyle and Findbugs for Java, FxCop and NDepend for .NET and Pylint for Python. Their increasing use in industry has been reported as detecting on average 55% up to 60% of defects [14].

A deeper analysis of the issues detected by such tools can provide important benefits: (i) Help developers better understand how such issues are introduced and how they can

Received: 13.01.2021. In revised form: 23.03.2021. Accepted: 30.03.2021
2010 *Mathematics Subject Classification.* 68N30, 03G10, 06B15.
Key words and phrases. *formal concept analysis, static analysis, code review.*
Corresponding author: Simona Motogna; simona.motogna@ubbcluj.ro

be resolved; (ii) Improve application efficiency, maintainability, reliability and security; and (iii) Enhance code comprehension.

The purpose of our study is to use Formal Concept Analysis (FCA) techniques to investigate static analysis results, in order to perform an in depth analysis of detected issues. We applied this model for Pylint, a static analysis tool for the Python language, and conducted a case study targeting student assignments for the Fundamentals of Programming first year course at the Babeş-Bolyai University. We show how this model can be used for further investigation and analysis of the collected data.

2. Related work

In 2005, Tilley [20] published a survey with different applications of FCA to several problems from software engineering (SE). Their study showed that most applications were in software design and maintenance, solving problems related to re-engineerings and class identification, with an increase in the last years in application to testing. Since then, several other proposals have been made in this direction, such as detecting causal dependencies in execution traces [16], or combining FCA with information retrieval techniques in order to solve problems related to concept location, enhancing program comprehension. Another survey [6], published in 2011, extended to solving SE problems using RCA (Relational Concept Analysis) and presented such an application to learning model transformation patterns. As FCA is constructing conceptual hierarchy based on data, and the whole software development processes can collect data, it is definitely clear that this remains an unexploited domain of study. Our approach to apply FCA to source code static analysis is, to the best of our knowledge, the first such attempt of its kind.

Tools that support construction of contexts and concept line diagram rendering [1, 4, 7, 11, 12] have simplified the application of this mathematical framework to different domains in computer science but also to other fields such as Economics, Biology or Medicine.

Code review and static analysis results may offer important information to improve software comprehension, specifically for understanding changes [3] or can be successfully used in programming education [15]. Information provided by such tools are of immediate use for developers to correct bugs, improve several aspects of code such as efficiency or security [2], but qualitative analysis of these results can offer additional benefits, which we aim to identify using our FCA-based approach.

3. FCA Model

3.1. **Research objectives.** Pylint is a static analysis and code review tool assisting programmers with error detection, coding style and code smell detection. It reports all found issues in terms of messages that are classified as shown in Table 1.

Table 1. Pylint message types ([22])

Message Type	Abbrev	Usage
Information	I	Information message (does not contribute to analysis score)
Refactor	R	Usually associated with code smells
Convention	C	Coding standard violation
Warning	W	Stylistic problem, minor programming issue
Error	E	Important programming issue (probably a bug)
Fatal	F	Internal error which prevents further processing

www.pylint.org

Each message contains a unique code and a corresponding description, which can be used to automatically aggregate issues across Python modules or projects, and offers a clear diagnosis that helps programmers understand and address detected issues. Table 2 offers a snippet of possible messages, including their codes and descriptions, as defined in [18].

The analysis is performed on modules, and for each module a list with all detected issues is produced as output. This analysis is especially useful in the case of applications that consist of many modules, when it becomes difficult to assess all the issues and decide their individual importance and potential impact.

Our *main research objective* is to investigate how FCA can provide a mathematical foundation for the analysis of issues detected using Pylint. Then we examine how the FCA model can be used to investigate the distribution, frequency and correlations between Pylint messages.

3.2. **Basic notions in Formal Concept Analysis.** FCA is based on complete lattices theory and was developed at the end of the 1980's in order to restructure the lattice theory in a form suitable for applications in data analysis [8, 21]. The basic notions of FCA are those of a formal context and a formal concept.

Definition 3.1. A *formal context* $\mathbb{K} := (G, M, I)$ consists of two sets G (*objects*) and M (*attributes*) and the incidence relation I between the sets of objects and the attributes [8].

In order to express that an object g is in a relation I with an attribute m, we write $gIm \in I$ and we read "the object g has the attribute m". Next, we define concept forming operators as certain derivations in the power sets of G and M, respectively.

Definition 3.2. For $A \subseteq G$, we define $A' := \{m \in M \mid \forall\, g \in A, gIm\}$, and for $B \subseteq M$ we define $B' = \{g \in G \mid \forall\, m \in B, gIm\}$. A' represents the set of attributes common to the objects in A, and likewise, B' represents the set of objects with all attributes in B [8].

Definition 3.3. A *formal concept* of the context (G, M, I) is a maximal pair (A, B) with $A \subseteq G$ and $B \subseteq M$ with $A' = B$ and $B' = A$. A is called the *extent* and B the *intent* of the concept (A, B) [8].

A formal concept contains a set of objects with a common set of attributes. Intuitively, the maximality property of A refers to the fact that for every object that is not in A there exists an attribute in B that the object does not have. Similarly, the maximality of B means that for every attribute that is not in B, there is some object in A that does not have that attribute. For example in Figure 2(B) the highlighted concept contains 107 objects having the attributes Recommendation and Convention Messages in common. In this particular example we can see that the rest of the objects are not part of the extent because they do not have the attribute Recommendation Messages.

$\mathfrak{B}(G, M, I)$ denotes the set of all concepts of the context (G, M, I). The order relation $\leq$ over the formal context, i.e. the *subconcept-superconcept relation* is formalized by: $(A_1, B_1) \leq (A_2, B_2) \iff A_1 \subseteq A_2 \iff B_1 \supseteq B_2$. The basic theorem of FCA proves that the set of all concepts of a context $\mathbb{K}$ together with the order relation $\leq$ is a complete lattice and every complete lattice occurs as a concept lattice of a suitable chosen formal context.

Usually, the attributes are not used only to describe a property that an object may or may not have. When considering different attributes, such as grades or colors, we notice that multiple values may be allowed for the attributes. In that case the data sets can be interpreted as many-valued contexts.

Definition 3.4. A *many-valued* context is a tuple (G, M, W, I), where G, M are sets, $I \subseteq G \times M \times W$ is a ternary relation and for all $g \in G$ and $m \in M$ if $(g, m, w) \in I$ and

$(g, m, v) \in I$ then $w = v$, i.e. the value of the object g on the attribute m is uniquely determined. We write $m(g) = w$ instead of $(g, m, w) \in I$ and read "the attribute m has the value w for the object g". [8]

In order to assign concepts to a many-valued context, the common approach is to transform it into a one-valued context through conceptual scaling. A scale for an attribute m of a many-valued context is a formal context $S_m := (G_m, M_m, I_m)$ with $m(G) \subseteq G_m$. The set of scales can be used in order to navigate the conceptual structure of the many-valued context. There is a list of predetermined scales, such as nominally, ordinally, etc., but for more complex views particular scales can be defined to unveil patterns in the analyzed data. After the scales are defined, concept lattices are built and analyzed in order to visualize knowledge clusters and to understand connections between the data.

Definition 3.5. Let $(M, \leq)$ be an ordered set and $a \in M$. An *ideal* (or a *downset*) of $(M, \leq)$ is a subset $D \subseteq M$ such as $\forall d \in D$ and $m \in M$ with $m \leq d$ then $m \in D$ [8].

Definition 3.6. Let $(M, \leq)$ be an ordered set and $a \in M$. A *filter* (or an *upset*) of $(M, \leq)$ is a subset $D \subseteq M$ such as $\forall d \in D$ and $m \in M$ with $m \geq d$ then $m \in D$ [8].

When representing the concept lattice, reduced labeling is usually used to improve readability, such that each object and attribute is assigned to a single label. When using reduced labelling, the extent of a concept is equal to the set of objects in the principal ideal generated by that concept, meaning all the objects contained in the extents of the subconcepts of the corresponding concept. Similarly, the intent of the concept contains the attributes that can be found in the principal filter generated by the concept.

3.3. **Applying FCA to static analysis results.** In this section we present the stages, as depicted in Figure 1, through which FCA can be used to cluster and analyze results produced by static analysis tools. The first step is performed once and then all the other steps can be defined depending on the desired criteria. This assures a large applicability of the model, and even more, this approach can be applied for any data collection that satisfies the requirements for constructing the objects and attributes of the FCA model.

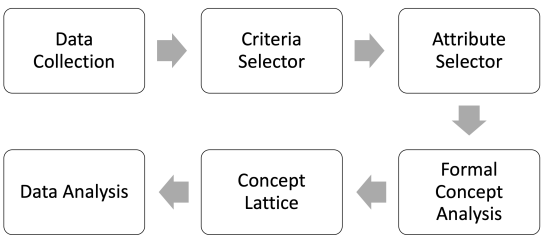


FIGURE 1. Stages of applying FCA to code review results

A. Data collection Our data set was built based on the introductory *Fundamentals of Programming* (FoP) first year course, taken by undergraduate students in computer science. Students are evaluated continuously during the semester using several assessment methods. These include a number of programming assignments ranging from introductory to the development of a turn-based board game, two practical tests taken during the semester as well as a final examination with both a written and practical component. Our study covers the 2019-2020 academic year. 225 students were registered for the first-semester course, which was not affected by the subsequent pandemic.

Assignments were submitted to a designated email address using the students' faculty email, which allowed staff to identify students. Tooling was employed to download and pre-process submissions. Data validity was ensured through automated checks, including checking that the correct source code was submitted, as well as a plagiarism check using Stanford's MOSS [19]. For the purposes of our work, an additional manual examination was carried out in order to ensure that all submissions were assigned to the correct assignment and to eliminate any third-party, or instructor-provided source code that could skew obtained results. Our case study covers five assignments and two coding tests handed in during the semester, as well as the practical session for the regular and retake examinations. This resulted in 1090 submissions, as not all students resolved given assignments, or they refrained from submitting non-working code for the examination.

B. Criteria selector The purpose of the performed analysis can be defined in two different ways, as exemplified in the following section: use the classification given by the Pylint tool, or define a specific goal, and then associate attributes corresponding to this goal (such as prevent errors).

C. Attribute selector Depending on the specified criterion, from all possible codes produced by the static analysis tool, we select a subset that will become the current set of attributes for building the FCA model.

D. Formal Concept Analysis For the qualitative approach the FCA framework contains the following elements:

objects: set of o_i, represented by students who handed in a specific assignment. Data uniformity is provided by the fact that all assignments are solving identical requirements.

attributes: set of a_i, represented by messages that correspond to the attribute selector, generated by Pylint for the chosen assignment.

the incidence relation: set of pairs (o_i, a_j) which represent the set of messages a_j found in the Pylint results for the student o_i and the selected assignment.

extent: all objects that share the attributes.

intent: all attributes that share the objects.

However, we aim to further increase the generality of our model. When switching to a quantitative approach, for instance, the set of attributes can be adjusted to include information about the messages as well as their number of occurrences.

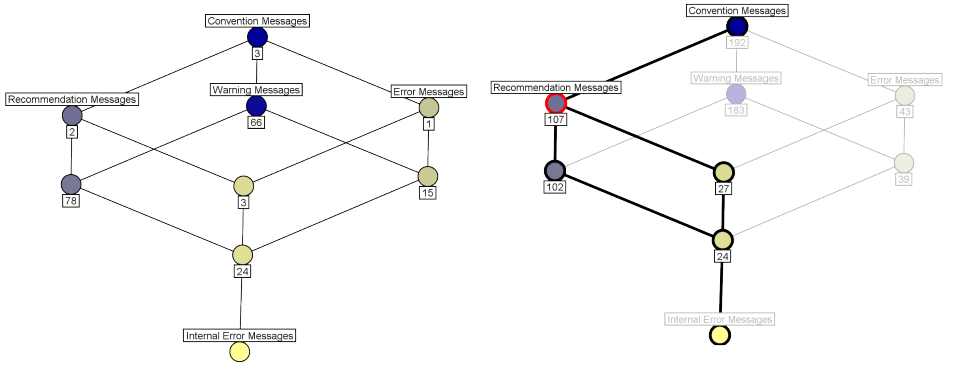
E. Concept Lattice As the main advantage of an FCA based analysis is its graphical representation of the data clusters, we built conceptual scales using the ToscanaJ system [4].

We have used Elba, which is the most advanced editor implemented in the ToscanaJ suite. Our main goal was to build more complex scales than the classic ones (i.e. nominal, ordinal, grid). Thus, we have generated some particular scales based on the selected criterion for each case. Afterwards, we used Toscana to visualise conceptual schemas and to display information filtered and clustered accordingly in the lattice diagram.

The data set can be interpreted as a many-valued context due to the fact that objects (i.e. students submitting an assignment) have attributes (i.e. codes) with specific values (i.e. how many times the code was present in the Pylint output for the given student/assignment pair). For this proof of concept we focus on a qualitative approach while analysing the data set, but due to the many-valued attribute, we are able to switch whenever it is necessary to a more quantitative analysis, in order to analyze not only whether a particular assignment encounters messages from one category, but also to see how many such messages it encounters.

---

https://sourceforge.net/projects/toscanaj/



(A) Show only exact matches

(B) Show all matches with highlight over a concept

FIGURE 2. Students' assignments for the mid-term test: An overview over Pylint general message types

In Figure 2(A), the criteria is defined as the correlation between the students' assignments and the general message types (attributes of the FCA model described in Table 1), with the purpose of highlighting the most common problems.

We represent the queried data in order diagrams consisting of nodes, edges and labels. Each node represents a formal concept, characterized by its extent and intent. Some nodes are connected by edges, which emphasize the concept order. When a node is selected, as shown in Figure 2(B), its filter and ideal are highlighted [4]. This feature enhances the readability of the subconcept-superconcept hierarchy.

As mentioned previously, reduced labeling is used to improve lattice readability. Labels below nodes represent objects and labels above nodes represent attributes. However, the tool allows for different visualizations of the diagram and the labels containing objects/attributes can be replaced with labels showing information regarding the size of the extent/intent or the number of objects/attributes that match only the corresponding intent/extent of the node (as shown in Figure 2). For this particular example we have 192 objects (i.e. student IDs) and 5 attributes (i.e. *Convention Messages*, *Warning Messages*, *Error Messages*, *Recommendation Messages* and *Internal Error Messages*) which are attached to one of the formal concepts and are written above the node. In order to read the components of a formal concept, i.e. a node of the lattice, one has to look at the object labels of its subconcepts and at the attribute labels of its superconcepts. For instance, the intent of the formal concept highlighted in Figure 2(B) contains attributes 'Recommendation Messages', and 'Convention Messages'. The top node represents the concept containing all the objects in its extent, while the bottom one represents the concept containing all the attributes in its intent.

Figure 2 illustrates the same concept lattice represented once with labels containing the extent size in Figure 2(B) and once with labels containing the number of objects that match only the attributes of the node intent (without matching additional attributes) in Figure 2(A). In order to understand the correlation between them, let us consider the highlighted concept in Figure 2(B) with extent size 107. Among these 107 students, as it can be seen in Figure 2(A), 2 of them had recommendation and convention messages, 78 had recommendation, convention and warning messages, 3 had recommendation, convention and error messages, and 24 had recommendation, convention, warning and error messages.

Node colors are employed to encode the size of the concept's extent, so that extreme values can be quickly determined by examining a node's color gradient, in descending order from blue to yellow [4]. The correlation between node colors and the extent size can be easily observed in Figure 2(B), where the lower label of the nodes contains exactly the size of the extent.

F. Data Analysis Based on the selected criterion and constructed concept lattice, data analysis can be performed, providing an in depth investigation of the dataset.

For example, considering Figure 2(A), we see that all assignments had convention messages, but none of them had internal error messages. Moreover, 3 of the 192 analyzed solutions only had convention messages, without messages from any other category. Another 24 solutions had messages from all four categories (convention, recommendation, warning and error).

4. Case studies

The available dataset and the constructed model can be used for several goals. In this section we describe two case studies, one for a goal corresponding to a check provided by a Pylint feature, and another for a user-defined goal.

4.1. **Design checker.** In this example, the criterion is selected as being "design checker", namely it will identify issues in code related to design problems. The set of attributes for this criterion is taken as defined in the Pylint documentation [18]. The FCA model was run for Assignment 4, for which 179 solutions were handed in. The corresponding concept lattice is presented in Figure 3.

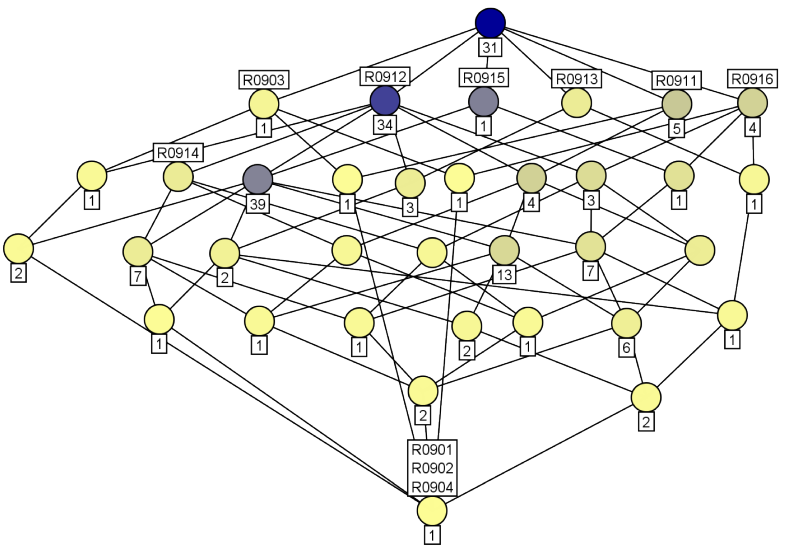


FIGURE 3. Design checker criterion for assignment 4

When analyzing the concept lattice for the design checker criterion we noticed that 31 students did not have any design problem for this assignment, while the recommendation message labeled R0912 is one of the most frequently encountered messages in our data. Basically, that means that source code cyclomatic complexity was too high, making code comprehension difficult. There are two other concepts that have a big extent size,

TABLE 2. Set of attributes for criterion "prevent errors"

Code	Description	Code	Description
R1707	Disallow trailing comma tuple	W0102	Dangerous default value %s as arg
W0106	Expr. "%s" assigned to nothing	W0109	Duplicate key %r in dictionary
W0123	Use of eval	W0126	Using a cond with potent. wrong func
W0128	Redeclared var %r in assign.	W0141	Used builtin function %s
W0231	_init_ from base class %r not called	W0232	Class has no _init_ method
W0233	_init_ from a non direct %r called	W0236	Method %r was expected to be %r
W0601	Global var %r undef at module level	W0602	Using global for %r but no assign.
W0621	Redef name %r from outer scope	W0631	Using possibly undef loop var %r
W0642	Invalid assign to %s in method	W0702	No exception type(s) specified
W0703	Catching too general exception %s	W1113	Keyword arg position in %s function
W1309	Using an f-string no interpolated var		

shown with a gray background. The concept labeled with R0915 emphasizes that students usually write functions or methods that have too many statements. The other gray concept has edges towards the nodes corresponding to the concepts mentioned above. Therefore, there are 39 students for whom both messages were generated (i.e R0912 and R0915). Thus, we may recommend to a large number of students to split the code into smaller functions in order to improve its understandability.

4.2. **Prevent errors.** The second example consists of a specific criterion that we defined, namely to identify the situations (corresponding to Pylint messages) that might result in future source code errors. Given the authors' experience, the list of attributes was constructed from all possible Pylint codes that identified situations that might generate errors. As a result, the set of attributes is (for complete corresponding description see [18]) presented in table 2.

In the next step, with the selected set of attributes, we construct the correspondent conceptual lattices for different data sets. Our case study takes into account data from assignment 4, resulting in the conceptual lattice from Figure 4, respectively data from assignment 10, with the corresponding conceptual lattice from Figure 5. This use case shows how the two models can be used to observe that the most common mistakes, namely: W0621, i.e. 'Redefining name from outer scope' [18], W0702, i.e. 'No exception type(s) specified' [18] and W0703, i.e. 'Catching too general exception' [18]. The results may be used to observe patterns of errors and their frequencies, or to study student progress during the semester, in order to determine whether programming skills have improved.

5. CONCLUSIONS AND FUTURE WORK

Static code analysis and code review are two processes that can significantly improve source code quality, which makes analysis and interpretation of their results of great interest in software development research. In this study, we proposed a model based on Formal Concept Analysis that can be applied for any such tool, and the objectives of the analysis may be specified depending on user requirements. We showed how such a model can be used to analyze the results produced by Pylint for an extensive set of student assignments, and described two case studies in detail.

Moreover, we showed that the essential characteristics of Formal Concept Analysis - concept hierarchy and clustering, provide a strong mathematical background for deep data analysis for this type of problem.

In conclusion, we believe that our approach can be successfully used as a mathematical foundation for analysis and interpretation of static analysis results. Our main goal is to

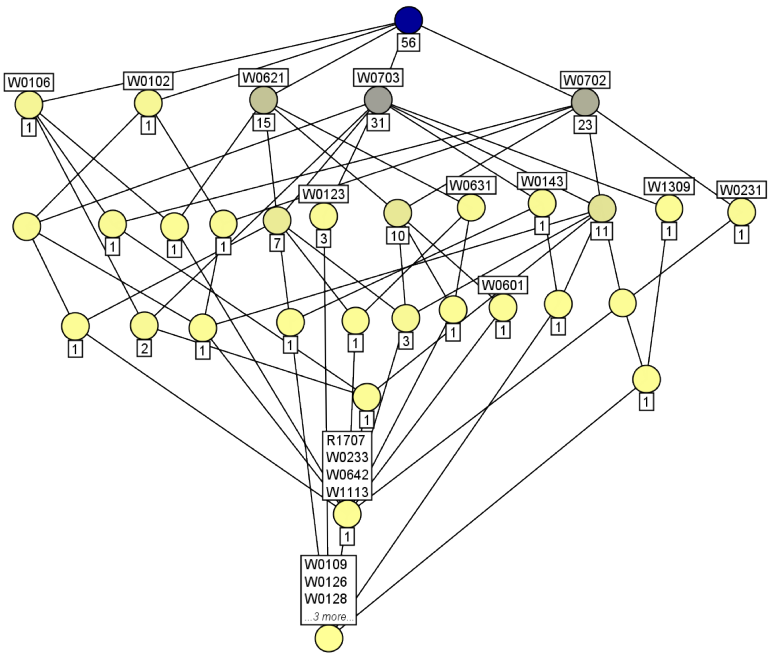


FIGURE 4. Prevent errors criterion for assignment 4

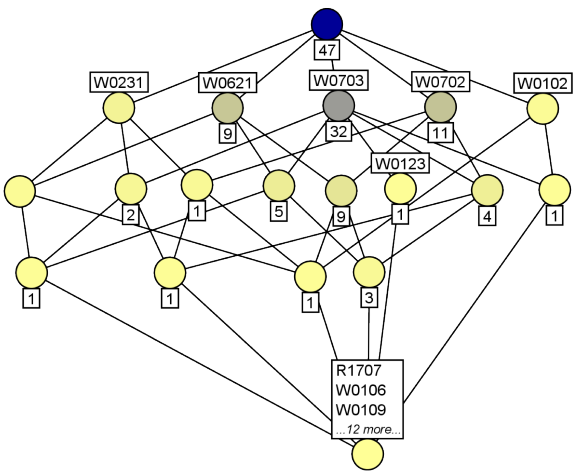


FIGURE 5. Prevent errors criterion for assignment 10

further extend the model to be able to deal with many-valued contexts in order to facilitate new analysis possibilities through knowledge discovery and data mining.

Future plans include several application of this model from different perspectives, such as analysis of software system quality based on static analysis tool results, or in case of software development education for student assessment, by detecting common errors, the misuse of programming artefacts, as well as to analyse student progress through the use of data covering several academic years.

REFERENCES

[1] Alam, M.; Nguyen, Le T. N.; Napoli, A. Steps Towards Interactive Formal Concept Analysis with LatViz. *FCA4AI@ECAI* 2016, vol. 1703, 51–62, CEUR-WS.org, 2016.

[2] Arusoaie, A.; Ciobâca S.; Craciun, V.; Gavrilut, D.; Lucanu, D. A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code, *Proc. SYNASC* 2017, 161–168.

[3] Barnett, M. et al. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. *IEEE/ACM 37th IEEE ICSE 1* 2015, 134–144.

[4] Becker, P.; Correia, J. H. The Toscana J Suite for Implementing Conceptual Information Systems. *Formal Concept Analysis, Foundations and Applications* 2005, 324–348.

[5] Cousot, P.; Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *ACM SIGACT-SIGPLAN (POPL '77), ACM* 1977, 238-–252.

[6] Dolques, X.; Huchard, M.; Nebut, C.; Saada, H. Formal and Relational Concept Analysis approaches in Software Engineering: an overview and an application to learn model transformation patterns in examples. *ICESE'11: First Virtual Workshop on Search-based Model-Driven Engineering* 2011.

[7] Dürrschnabel, D.; Hanika, T.; Stumme, G. DimDraw - A Novel Tool for Drawing Concept Lattices. *Suppl. Proc. of ICFCA 2019, CEUR Workshop Proc. vol. 2378, 60–64, CEUR-WS.org*, 2019.

[8] Ganter, B.; Wille, R. *Formal Concept Analysis - Mathematical Foundations*, Springer, 1999.

[9] Kildall, G. A. A Unified Approach to Global Program Optimization. *ACM SIGACT-SIGPLAN (POPL '73) ACM* 1973, 194–206.

[10] King, J. Symbolic execution and program testing. *Communications of the ACM* **19** (1976), no. 7, 385–394.

[11] Kis, L. L.; Sacarea, C.; Sotropa, D. Visualizing Conceptual Structures Using FCA Tools Bundle. *ICCS 2018, LNCS* **10872** 193–196, Springer, 2018.

[12] Kis, L. L., Sacarea, C.; Troanca, D. FCA Tools Bundle - A Tool that Enables Dyadic and Triadic Conceptual Navigation. *FCA4AI@ECAI 2016, vol. 1703, 42–50, CEUR-WS.org, 2016.*

[13] Lebanidze, E. The Need for Fourth Generation Static Analysis Tools for Security – From Bugs to Flaws. *Application Security Conference*, 2008.

[14] McConnell, S. *Code Complete, Second Edition*, Microsoft Press, 2004.

[15] Molnar, A.; Motogna, S.; Vlad, C. Using static analysis tools to assiststudent project evaluation. *EASEAI@ESEC/SIGSOFT FSE 2020, ACM.*

[16] Pfaltz, J. L., Using Concept Lattices to Uncover Causal Dependencies in Software. in: *Formal Concept Analysis* LNCS, **3874** Springer, 2006.

[17] Poshyvanyk, D.; Marcus A. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. *IEEE ICPC '07* 2007, 37–48.

[18] *Pylint documentation*, `https://pylint.pycqa.org/_/downloads/en/latest/pdf/`, acc. Jan 2021.

[19] Stanford's MOSS system, `https://theory.stanford.edu/~aiken/moss/`, acc. June 2020.

[20] Tilley T.; Cole, R.; Becker, P.; Eklund, P. A Survey of Formal Concept Analysis Support for Software Engineering Activities. *Formal Concept Analysis*, LNCS, **3626** Springer, 2005

[21] Wille, R. Restructuring lattice theory: an approach based on hierarchies of concepts. *Ordered sets*, Springer.

[22] Pylint message types, `http://pylint.pycqa.org/en/latest/user_guide/output.html`

DEPARTMENT OF COMPUTER SCIENCE
BABEŞ-BOLYAI UNIVERSITY
KOGALNICEANU 1, 400084, CLUJ-NAPOCA, ROMANIA
Email address: `simona.motogna@ubbcluj.ro`
Email address: `diana.cristea@ubbcluj.ro`
Email address: `diana.halita@ubbcluj.ro`
Email address: `arthur.molnar@ubbcluj.ro`