Using FCA to Analyse how Students Learn to Program*

Uta Priss

Zentrum für erfolgreiches Lehren und Lernen
Ostfalia University of Applied Sciences
Wolfenbüttel, Germany
www.upriss.org.uk

Abstract. In computer science and mathematics education, universities often observe high failure rates among students if they are taught in a traditional, lecture-centric manner. Interactive engagement teaching methods are more successful but in order to develop suitable teaching materials, lecturers must be aware of potential conceptual difficulties of a domain in advance, for example, by analysing the data of student-submitted work from previous sessions. In computer science education, the data collected from computer-based assessment tools provides a possible source for analysing conceptual difficulties students encounter. The data can be analysed with data mining techniques and in particular with FCA as discussed in this paper.

1 Introduction

High failure rates among students in introductory computer science and mathematics courses at universities indicate that these topics can be difficult to teach and learn. Researchers in Physics Education Research (for example, Hestenes et al. (1992)) developed standardised "concept test inventories" in order to measure how much students actually understand and discovered that, although students might be able to pass exams, they often do not acquire sufficient understanding of underlying concepts when traditional lecturing methods are employed. A number of "interactive engagement" methods (Hake, 1998) have been developed which improve student learning as evidenced by higher scores on the standardised concept tests. Interactive engagement teaching involves activities such as problem-based learning or peer instruction which engage the students actively with the subject matter. The idea is that students are required to mentally construct conceptually difficult ideas themselves instead of completed ideas being presented in a traditional lecture. The underlying teaching philosophy is usually called "constructivist model" of learning (e.g., Ben-Ari, 1998), but it should be emphasised that this refers to the students' mental constructions of what they are learning and not to a philosophical constructivist view which denies the existence of an external world. Keeler and Priss (2013) argue that an appropriate philosophical foundation for interactive engagement teaching is Peircean pragmatism.

One challenge for interactive engagement teaching is that conceptual development in a learner's mind can be a slow process. Thus, if students are expected to acquire

* Published in Cellier & Distel (eds.), Proc of ICFCA'2013, Springer, LNAI 7880.

a solid level of understanding of what they are learning instead of just memorising facts and formulas which they reproduce in exams, fewer materials can be covered in this manner. Because the amount of materials covered in a course is often externally determined, lecturers need to carefully select which materials should be taught using interactive engagement and which materials can be taught at a faster pace, using more traditional methods. Ideally, conceptually difficult materials and foundational concepts should be taught using interactive engagement methods whereas more factual and algorithmic topics and materials that build on or extend foundational concepts can be taught in more traditional ways. Therefore lecturers need to know which are the foundational but difficult concepts in a domain and what exactly are the difficulties encountered by typical students with these concepts. McDermott (2001) argues that the conceptual difficulties students encounter are not individually determined but all students encounter similar difficulties at similar stages of their development - although some students are capable of overcoming such difficulties by themselves whereas other students need help. It follows that although it can be a labour-intensive process to develop interactive engagement teaching materials, such resources should be highly reusable once they have been developed. A number of frameworks (providing standards and software) for reusing and exchanging materials already exist, for example, the open educational resources (OER) movement[1]. Thus, identifying which concepts of a domain are particularly difficult to learn and then developing interactive engagement learning materials for these topics is a useful endeavour.

In order to reduce the cost of developing interactive engagement teaching materials, it is essential to have tools that support the detection and analysis of conceptual difficulties in a subject domain. We have previously argued that Formal Concept Analysis[2] (FCA) is suitable for modelling conceptual difficulties in learning processes (Priss, Riegler and Jensen, 2012). In that paper constructions of formal contexts using existing analyses of misconceptions in a domain and using data from student interviews were presented. The idea of building concept lattices from assessment data was mentioned in that paper but not further investigated. Therefore this paper examines the usefulness of assessment data produced by a computer-based assessment tool for identifying conceptual difficulties of a domain and for analysing typical learning strategies employed by students in that domain.

It might be of interest to compare FCA-based analyses with other data mining tools but that would be a topic for an entirely different paper. Our goal is not to investigate whether or not FCA is better than other tools but simply to study how FCA can be used in this case. Our reasons for choosing FCA are that there are command-line tools available (such as http://fcastone.sourceforge.net/) which can be easily integrated with the learning environment we are using and which we have previously successfully used for similar data analysis tasks.

The next section describes further details about data collected from computer-based assessment software. Section 3 provides a brief overview of the tools available specif-

[1] http://en.wikipedia.org/wiki/Open_educational_resources

[2] Because this conference is dedicated to FCA, this paper does not provide an introduction to FCA. Information about FCA can be found, for example, on-line (http://www.upriss.org.uk/fca/) and in the main FCA textbook by Ganter & Wille (1999).

ically for analysing Java code. Section 4 examines the types of changes students make when they resubmit their code to the on-line tool. Section 5 describes how the data can be modelled and analysed with FCA. Section 6 provides an overview of some related work. The paper finishes with a conclusion.

2 Computer-based assessment software

E-learning is a complex field with many different types of software. The focus of this paper is on what Priss, Jensen & Rod call "computer-based assessment" (CBA) software (2012a and 2012b). CBA tools are predominantly used for teaching computer programming and work by analysing and executing student-submitted source code. Following the approach by Uri and Dubinsky (1995) who advocate teaching abstract mathematical concepts using computer programming, CBA tools could also be used in teaching mathematics in a similar manner. CBA tools are smaller in scope than course management systems or virtual learning environments which provide a whole range of tools for teaching and managing a course including the storage and retrieval of exercises. CBA tools are more similar to on-line compilers or code "pastebins" which usually contain one area where code can be submitted, one area for the display of the result of the code and sometimes a third area for additional feedback. But in addition to such code assessment facilities, CBA tools also contain a small course management component which allows lecturers to select and group exercises and to view the students' submissions and results. Over the course of a semester, CBA tools accumulate a substantial amount of data on how the students approach a particular exercise, the number of attempts and the kinds of errors students make and also whether students work independently or copy from each other. The data can be analysed with data mining methods.

It should be noted that there are roughly two types of uses for CBA tools which have quite different requirements. On the one hand, CBA tools can be used as "submission systems" which have not much more functionality than submission systems used for the reviewing of academic papers. They allow students to submit documents (code or other) for assignments by deadlines which are marked by one or more tutors. Some automated tests are run on submission, for example, programs are compiled and unit tests are executed. Even submission systems for academic papers sometimes compile documents, for example, in the case of LaTeX documents. When CBA tools are used as submission systems, students usually use a different tool (such as an integrated development environment (IDE)) for developing their code which they upload after it has been carefully executed and checked within the IDE. Code that is submitted in that manner usually compiles but might still fail some of the other tests.

On the other hand, CBA tools can also be used as development environments themselves. This is particularly useful for teaching first-time programmers. The CBA tool can even be configured so that students only write code snippets which are then automatically completed to a larger program by the tool. In that manner students can focus on learning individual constructs of a programming language (such as the use of variables, control structures or string operations) without having to learn how to use an IDE at the same time. In particular for programming languages such as Java which are quite verbose and require a fair amount of declarations this approach is appealing.

The data collected by both types of CBA tools (as submission system versus as development environment) is quite different. In the first case, the student submitted code tends to not have any compilation errors and tends to be more complex. Students usually only resubmit the code a small number of times - depending on how much feedback the system provides and how many resubmissions are allowed. Since students can check the code within the IDE before they submit it, the only interesting additional tests performed by the CBA tool are those which evaluate how well the submitted code conforms to the specification given to the students or those which check cases that the students might have overlooked. Apart from such tests, the main function of the CBA tool is to make it easier for lecturers to manage, view and mark the submissions.

In the second case, if a CBA tool is used as a development environment, the complete history of how the code was developed is stored by the system. Using data mining techniques, this data can be explored to determine what kinds of misconceptions students have about programming as evidenced by the errors they make and the strategies they use to overcome the errors. The collected data is ideally suited for determining the conceptual difficulties involved in computer programming. Once difficulties are identified, the feedback provided by the CBA tool to the students can be optimised and lecturers can use that information for planning their next lectures.

From a student perspective there could be a danger that data mining techniques reveal too much detail about how learners are thinking. In non-e-learning environments, students can decide at what point they are ready to submit a piece of coursework and show it to the teacher. Students who are dissatisfied with their work might refuse to submit a piece of coursework at all. Like any other adaptive learning environment, a CBA tool used as a development environment monitors every step a student takes. Usually the feedback is produced automatically and the amount of data collected by such a tool means that it is not feasible for a lecturer to look at which steps each individual student took. Using data mining tools or FCA, however, it is possible to analyse the data in more detail and to alert lecturers to the problems of individual students. Students should be informed about how the data is analysed and potentially have an option to opt out of certain types of analyses.

At the moment we are still in the process of investigating what kind of information can be extracted from the data. But in the future, it needs to be carefully determined how the data is aggregated and presented in a manner that suits both lecturers and students. The data for this paper was collected during the initial testing phase of a CBA tool. The programming exercises were not part of marked assignments and the students volunteered to participate.

3 Tools for analysing Java code

The data from a CBA tool needs to be preprocessed in order to extract formal contexts for exploration with FCA. This section discusses which tools are suitable for preprocessing data collected from Java programming exercises. Automated code analysis is a complex problem domain which is still being actively researched and developed (Binkley, 2007). Furthermore, it appears that only a subset of the methods that are currently being researched are already implemented and available for use - at least if, as for our investigation, only freely available tools are considered. CBA tools usually use a number of static and dynamic testing and analysis methods in order to evaluate the student submitted code. Even though all programming languages can be evaluated with similar means, the feedback provided by tools for different languages is quite different. Thus tools for preprocessing depend on the programming language used in the exercises.

In the case of Java, analyses can focus on the source code or on the compiled byte code. Using standard testing tools, it is fairly straightforward to extract information about classes, methods and variables and to compile basic run-time metrics, such as how much CPU-time is used by one execution. Unfortunately, many software testing tools are intended to be used only by the programmers themselves, run in an interactive mode and require breakpoints or special code segments to be inserted into the code which is not feasible for CBA tools. Preprocessing tools that are more suited for use with CBA tools are those which are build for reverse engineering or unit and blackbox tests that evaluate whether a piece of software conforms to a set of requirements. The disadvantage of unit and blackbox tests is that possible problems must be anticipated in advance. Each test provides a yes/no answer for one particular problem and misses anything that does not exactly fit the test. Profiling or reverse engineering tools are most promising for our purposes. FCA can then be used to aggregate and visualise the results from such tools. Unfortunately, such tools are often quite complex to install and use. We evaluated a number of freely available tools[3] none of which seem to provide general analyses of the data that would show the difficulties students have with a particular exercise - unless the tests performed with a tool are configured in advance to look for particular problems.

Because the analyses made with such tools were disappointing, we wrote some basic scripts for comparing subsequent submissions for each individual student using Unix "diff" and regular expressions. Although we have not completely abandoned the idea of using profiling or tracing tools in the future, for now our basic scripts for analysing and classifying the changes between subsequent submissions seem very promising. Every change shows how a student perceived a certain problem with the code and attempted to solve it. Figure 1 presents the output of using Unix "diff" to compare two subsequent submissions of a student's code. Only the lines that were changed are shown. The two lines starting with ">" belong to the first version, the other two lines to the second version of the student's code. The student realised that the unit test used by the CBA tool requires the string "Hello World" to be returned instead of being printed to the standard output. The student's first attempt was basically a correctly functioning Java program but the requirements of the exercise were not followed. If several students make this initial mistake, the lecturer should check the description of the exercise to see whether this requirement is indeed clearly stated.

Some types of changes show that students are having conceptual difficulties. For example, if the code changes completely between two attempts, either the student started over from scratch, or, more likely, the student gave up on an initial attempt and obtained new code either from another student or the web. Another example of students struggling is if subsequent code submissions contain reversals of changes made before. In

[3] Standard command-line tools jcf-dump and javap, http://pmd.sf.net, http://www.doxygen.org, http://checkstyle.sf.net, http://sf.net/projects/javacalltracer, http://sf.net/projects/findbugs.

```
<       public String greeting()
>       public void greeting()
<               return "Hello World";
>               System.out.print("Hello World");
```

Fig. 1. Unix "diff" comparison of a student's subsequent code submissions

that case the student is using trial and error. Other types of changes can be classified and further analysed as described in the next section.

4 Classifying the types of changes between subsequent submissions of code

In order to identify conceptual difficulties students are having when they are learning to program it is essential to distinguish superficial problems which students can overcome quickly by themselves from deeper, conceptual problems. Examples of superficial problems are typos, syntax errors and specific requirements of an exercise which are overlooked by a student. Students may need help from an instructor or tutor if they encounter conceptual difficulties. By classifying the types of changes students are making to their code, the CBA tool can filter superficial from deeper problems.

The CBA tool we are using (described in more detail by Priss, Jensen and Rod (2012a and 2012b)) is currently deployed in a number of introductory Java courses. A typical course contains 10 - 30 programming exercises. Because an individual student might submit up to 10 different attempts for each exercise and a class might consist of 20 - 100 students, it is not feasible to manually compare subsequent submissions of individual students. Although the CBA tool automatically performs tests on the submitted code, the tool expects a lecturer or team of tutors to have at least a brief look at the final solutions submitted by each student because the automated marking is not always reliable. The CBA tool provides some basic statistics about the exercises such as which student completed which exercise with how many attempts. It calculates the difficulty of each exercise based on how many students achieve full points for the exercise. But that information is not sufficient to show exactly what kinds of problems the students encounter. It is our intention to improve the reports generated by the CBA tool for the lecturers. For example, the reports could highlight not just which exercises are difficult but also what kinds of problems students are having with a difficult exercise. Furthermore the reports could identify students whose problem solving approaches do or do not follow certain patterns and thus might need extra help. FCA can then be used to provide a visual summary of the data that is already provided by the CBA tool. As mentioned before, our goal is to analyse what benefit FCA can have for CBA tools and not to compare FCA to other technologies for this purpose.

As a first step we implemented a script which compares subsequent submissions of each student and attempts to classify the kinds of changes implemented by the student. Table 1 contains a summary of the types of changes that are currently detected by our script. All other changes are tagged as "unclassified" and need to be looked at by a person. The last column indicates what patterns are searched for by the script via regular expressions. The table and its classifications were created in an iterative manner. We identified some changes that could be easily classified. Then we adjusted the script to detect those changes and to print a list of everything that could not yet be classified. This process was continued until we could not see any further easily detectable patterns of changes. In this manner we were able to automatically classify about 60% of the types of changes made by the students. The accuracy of the automated classification was verified using a square root sample. It should be noted that we are not suggesting to use our script as an additional source of points for marking. The script cannot determine why a student made a change or whether the change was appropriate to be made. The sole purpose of the script is to detect patterns which highlight problems with the CBA tool itself (the first row in Table 1), problems with an exercise (for example if the instructions given in an exercise are unclear) or problems which particular students are having, in particular if these represent conceptual difficulties. Although in some cases it might be possible to automatically generate additional feedback for the students, in most cases a lecturer will need to look into the problems highlighted by the system and then take appropriate action. Table 1 shows that some of the changes are superficial and caused by the fact that there are differences between the CBA tool and other development environments and by the fact that automatic evaluation is much less forgiving than manual marking. We have added the superficial problems caused by the CBA tool to the bug tracker list of the tool so that they will be removed in future versions of the tool.

It is debatable whether one should insist on literal adherence to requirements. For example, if a student is asked to produce the output "Hello World", then "hello world!" would not usually be acceptable. On the one hand, one can write tests that allow for some variability of the student code with respect to the requirements. On the other hand, students can also be expected to learn to read requirements carefully since this may be necessary in their future jobs as well. Most students will learn to avoid superficial errors and to literally follow requirements quickly. Students with certain disabilities or dyslexia could be disadvantaged by the rigidity of the automatic assessment. Presumably such students require extra support anyway which hopefully is provided by their universities. For example, extra tutors could either help such students with the use of the CBA tool or manually assess the students' work.

The most interesting, currently detected types of changes in Table 1 which point to potential misconceptions are those that affect the program flow (Boolean logic and control structures) and those that affect constructs that are difficult in Java (such as the specifics of variable declarations using "public", "static" and so on). With further use of the system we hope to extend the table and provide a more extensive classification.

5 Analysing the data with FCA

This section discusses how FCA can be applied to data collected from a CBA tool. Using FCA, a lecturer can generate concept lattices to obtain a visual representation of the data produced by the scripts discussed in the previous section. We have not yet shown the lattices to actual users of the system but we believe that contrary to the usual argument that users might find it difficult to read the lattices this may not be a problem

Table 1. Types of changes between subsequent code submissions by individual students

Type of change/error	**Explanation**	**Changes in code**
Superficial, required by CBA tool	Differences between the CBA tool and other development environments require certain adjustments to the code. Problems can occur when copying and pasting non-ASCII characters (e.g. German umlauts). Affects students who have not read the help file!	package statements, use of main method, control characters, umlaut
Requirements not followed	Because of automatic marking, requirements must be implemented literally and exactly. For example, printed output must have exact capitalisation, use of whitespace, etc.	print statements, class and method names, string content, method signatures
Not all cases considered	The CBA tool usually performs a number of unit tests which supply different values to the code. Code is checked for exceptional input, such as empty input, null values, minimal or maximal values.	initial values of variables, numerical value change; if, for, while, etc
Boolean logic or control structure error	Students sometimes make mistakes with using Boolean logic expressions and with start and end conditions of loops and with nesting of loops.	and, or, not; if, for, while, etc
Java variable declarations	Students find this difficult.	void, static, public, etc.
Student is struggling	Many attempts, seemingly random changes, reversals of changes.	number of attempts, reversed changes, many lines changed
Comments and formatting	The CBA tool can be configured to check for the existence of comments and adherence to formatting styles.	comments, whitespace
Other type of change	Were lines added, deleted or changed? Was all of the code changed? Is this a case of plagiarism because identical to another student's code?	added or deleted lines, many lines changed, plagiarism
Unclassified	Our tool cannot yet classify the change.	

with our user group because of their background in computer science. Figure 2 shows an example of an analysis of a single student's submissions for four exercises which are the objects "1", "2", "3" and "4". Explanations for the attributes can be found in Table 1. The student in Figure 2 can be identified as a "heavily struggling student". The student struggled most with the first exercise. Quite possibly this was the very first attempt of the student to write Java code. Exercises 1 and 2 show that the student has problems with Java basics such as variable declarations. Apparently the student had no problems with exercise 3 because only the distribution of whitespace was changed for that exercise. Unfortunately exercise 4 has the attribute of "many lines changed". This usually means that the student made a first attempt and then submitted a completely different second attempt. Sadly, code that is changed in its entirety in one step often indicates that the code was copied from some other source (other student or the internet). Thus this is certainly a case where the lecturer should have a conversation with the student to see how the student can be helped.

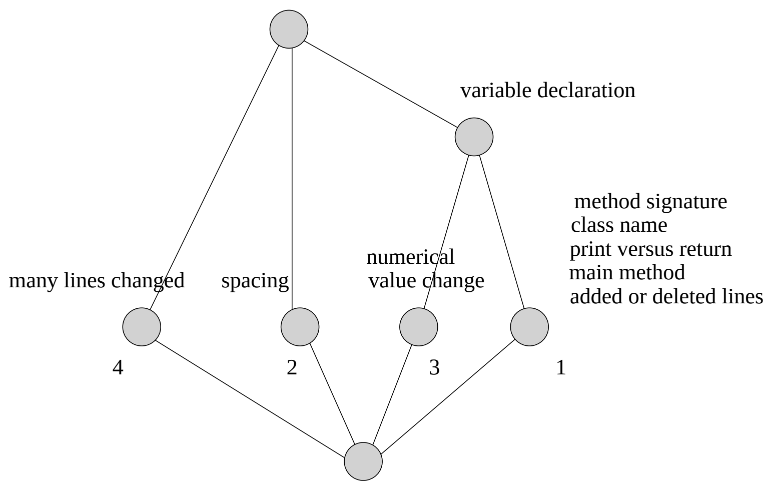

Fig. 2. Four exercises of a single student

Figure 3 shows a representation of all students for a single exercise. In this case only six students used the CBA tool for this exercise. Many of the attributes are of the "Superficial, required by CBA tool" type according to Table 1. The reason for this is because this data was obtained when we first started testing the CBA tool and we had not yet compiled a sufficiently detailed help file which we could give to the students. One student ("D") seems to already have a good understanding of Java. The student just had problems because he or she initially used an editor which encodes German umlauts in an incompatible encoding. The student solved the problem by removing umlauts from the code. (This is a short-coming of the CBA tool which will be fixed in the future.) Student A's problems are also of a superficial nature. A change in string content indicates that the student initially did not follow the requirements of the exercise correctly but the student resolved that quickly. Student E is also probably coping quite well. The student used an editor which is not suitable for writing code because it produces invisible control characters. The student changed the code completely presumably because he or she could not find the problem but from a programming viewpoint this is a superficial problem. Students B, C and F were having more problems with the exercise because they made changes to if and while statements. Closer inspection of the code changes of B and C showed that their while statement was the same. But because they only collaborated on some lines of their code not all of it, this appears to be not a case of plagiarism but a normal (and encourageable) occurrence of students discussing problems in a computer lab and then individually typing them in.

The formal contexts and lattices were generated directly from the data without further editing of the data using the FcaStone[4] software. It is feasible to create a small interface which allows to select courses, exercises or students and then to directly produce concept lattices. Alternatively, one can conduct analyses of the data with FCA and then compile the results into reports for the lecturers. Thus FCA would serve a role as an expert interface to the data but would be hidden from end users.

[4] http://fcastone.sourceforge.net/

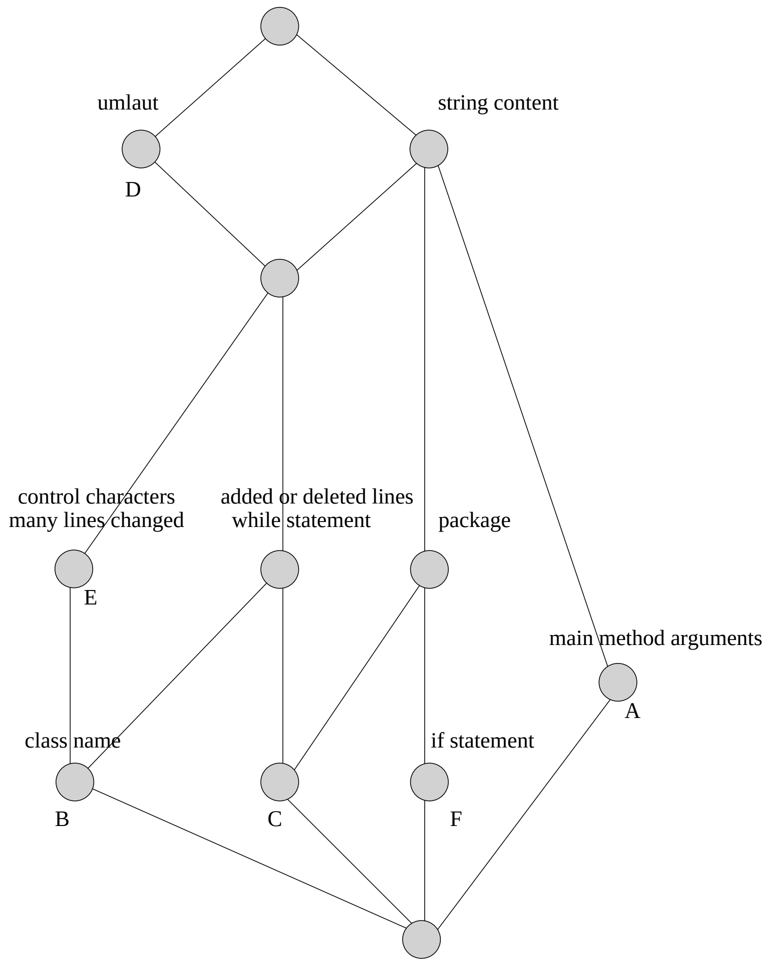

Fig. 3. Six students' submissions for one exercise

6 Related work

Other researchers have studied how to track student interaction with e-learning environments. But as Novak et al. (2012) observe most of the tracking research does not focus on source code analysis. Furthermore as far was we know there has been no research so far on using FCA for such purposes. The system suggested by Novak et al. uses mostly temporal and other quantitative variables (at what times students work on code; how long they work on an exercise; how many lines they added at each time and so on). They are not attempting to analyse the content of the students' work apart from showing which part of the students' work relates to which lecture based on the Java class names used by the students.

Spacco et al. (2005) take snapshots of the student code submissions from a CVS repository. Most of their analysis focuses again on quantitative attributes (the size of the code, the number of warnings) and uses tools similar to the CBA tool we are using (bug finders and unit tests). They are also analysing code changes using Unix "diff" which they discuss in detail. In order to compile their data into a relational database, they develop an algorithm to track individual lines of code across different code submissions. Their database associates lines of code with warnings and exceptions, test results and temporal attributes. While this is a very interesting approach, it is not yet clear how an instructor would actually use the database. As far as we know their software is not available for download. Therefore we were not able to evaluate it and compare it with our software.

While there is a large amount of literature on source code analysis in general (Binkley, 2007), it does not appear that there is much work on the specific question that we are interested in: how can one identify conceptual difficulties students are having by analysing their code submissions?

7 Conclusion

This paper discusses methods for analysing data collected from programming exercises presented in CBA tools. The goal is to identify conceptual problems students are having with the exercises. Initially it was attempted to combine standard software engineering tools with FCA software for analysing the data. Unfortunately, we did not find tools at least among freely available software that produced data which seemed to allow detection of conceptual problems. This is because such software mostly relies on configuration files that contain rules for searching for problems that are anticipated in advance. Otherwise, such tools produce general metrics or provide interactive modes for problem analysis but, in our opinion, none of these methods are suitable for detecting previously unknown problems. On the other hand, analysing the changes the students make when they resubmit their code to the CBA tool does seem promising because it reflects what stages students go through when they are developing code and how they are attempting to solve problems. We implemented some scripts which represent basic heuristics for classifying the kinds of changes made. Although it is not possibly to classify previously unknown problems in this manner, it is possible to eliminate large amounts of the data which represent superficial changes, to identify some changes which are symptomatic of certain problems and to reduce the amount of the remaining “unclassified” problems sufficiently so that they can be analysed manually. FCA concept lattices are a suitable tool in this context for visualising the data.

Acknowledgements

This work has been partially funded by the German Federal Ministry of Education and Research (BMBF) under grant number 01PL11066H. The sole responsibility for the content of this paper lies with the author. I would also like to thank the other members of the eCULT team, Nils Jensen, Oliver Rod and Stefan Bisitz, without whom using the CBA software would not have been possible.

References

1. Ben-Ari, Mordechai (1998). *Constructivism in computer science education.* SIGCSE Bull, 30, 1, p. 257-261
2. Binkley, D. (2007). *Source Code Analysis: A Road Map.* Future of Software Engineering, FOSE ’07, p. 104-119.
3. Ganter, Bernhard; Wille, Rudolf (1999). *Formal Concept Analysis. Mathematical Foundations.* Berlin-Heidelberg-New York: Springer.
4. Leron, Uri; Dubinsky, Ed; (1995). *An Abstract Algebra Story.* American Mathematical Monthly, 102, 3, p. 227-242.
5. Hake, Richard R. (1998) *Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses.* American Journal of Physics, 66, 1, p. 64-74.
6. Hestenes, D.; Wells, M.; Swackhamer, G. (1992) *Force Concept Inventory.* Phys. Teach., 30, p. 141-158.
7. Keeler, Mary; Priss, Uta (2013). *Toward a Peircean Theory of Human Learning: Revealing the Misconception of Belief Revision.* In: Pfeiffer et al (eds.), Proceedings of the 20th International Conference on Conceptual Structures, ICCS’13, Springer Verlag, LNAI 7735, p. 193-209.
8. McDermott, Lillian Christie (2001). *Oersted Medal Lecture 2001: “Physics Education Research-The Key to Student Learning”.* American Journal of Physics, 69, 11, p. 1127-1137.
9. Novak, M.; Binas, M.; Michalko, M.; Jakab, F. (2012). *Student’s progress tracking on programming assignments.* IEEE 10th International Conference on Emerging eLearning Technologies & Applications (ICETA), p. 279-282.
10. Priss, Uta; Riegler, Peter; Jensen, Nils (2012). *Using FCA for Modelling Conceptual Difficulties in Learning Processes.* In: Domenach; Ignatov; Poelmans (eds.), Contributions to the 10th International Conference on Formal Concept Analysis (ICFCA 2012), p. 161-173.
11. Priss, Uta; Jensen, Nils; Rod, Oliver (2012a). *Software for E-Assessment of Programming Exercises.* In: Goltz et al. (eds.), Informatik 2012, Proceedings of the 42. Jahrestagung der Gesellschaft für Informatik, GI-Edition, Lecture Notes in Informatics, P-208, p. 1786-1791.
12. Priss, Uta; Jensen, Nils; Rod, Oliver (2012b). *Software for Formative Assessment of Programming Exercises.* In: Urban; Müsebeck (eds.), elearning Baltics 2012, Proceedings of the 5th International eLBa Science Conference, Fraunhofer, p. 63-72.
13. Spacco, J.; Strecker, J.; Hovemeyer, D.; Pugh, W. (2005). *Software repository mining with Marmoset: an automated programming project snapshot and testing system.* ACM SIGSOFT Software Engineering Notes, 30, 4, p. 1-5.