TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 02/11 pp325–335
DOI: 10.26599/TST.2019.9010012
Volume 25, Number 3, June 2020

Towards Connecting Discrete Mathematics and Software Engineering

Tun Li*, Wanwei Liu, Juan Chen, Xiaoguang Mao, and Xinjun Mao

Abstract: To enhance training in software development, we argue that students of software engineering should be exposed to software development activities early in the curriculum. This entails meeting the challenge of engaging students in software development before they take the software engineering course. In this paper, we propose a method to connect courses in the software engineering curriculum by setting comprehensive development projects to students in prerequisite courses for software development. Using the Discrete Mathematics (DM) course as an example, we describe the implementation of the proposed method and teaching practices using several practical and comprehensive projects derived from topics in discrete mathematics. Detailed descriptions of the sample projects, their application, and training results are given. Results and lessons learned from applying these practices show that it is a promising way to connect courses in the software engineering curriculum.

Key words: Discrete Mathematics (DM); software engineering; proof checker; database management system; symbolic execution engine

1 Introduction

Software engineering has become a first-level discipline. Typically, software engineering education curricula focus on teaching students the principles of developing software systems using engineering methods and techniques. Such training requires much support and material from a variety of subjects, including programming, operating systems, database systems, principles of compilation, Discrete Mathematics (DM), practical projects of a certain scale, and principles of software engineering.

In typical software engineering curricula, some courses in the foundations of computer science are scheduled after software engineering courses. These circumstances are inefficient for the training of students in software engineering, and make it difficult to design practical projects of a suitable scale and complexity for software development training.

Furthermore, according to our teaching experience, most students feel that the majority of software engineering principles are too abstract to be learned in isolation. In contrast, students who have some experience with software development may have a strong potential to deeply understand and apply software engineering principles. Therefore, there is a challenge involved in keeping students involved with software development activities before taking the software engineering course.

The discipline of software engineering applies mathematical and computer science principles to the development and maintenance of software systems. The mathematical principles involved are primarily those of discrete mathematics, and especially logic. According to the ACM/IEEE Software Engineering Curricula 2014[1], software engineering programs require discrete mathematics at the core — "Logic and discrete mathematics should be taught in the context of their application to software engineering or computer science problems."

Typical software engineering education curricula do not coordinate the teaching of discrete mathematics and

• Tun Li, Wanwei Liu, Juan Chen, Xiaoguang Mao, and Xinjun Mao are with the School of Computer, National University of Defense Technology, and are also with Laboratory of Software Engineering for Complex Systems, Changsha 410073, China. E-mail: {tunli, wwliu, juanchen, xgmao, xjmao}@nudt.edu.cn.
∗ To whom correspondence should be addressed.
Manuscript received: 2019-03-15; accepted: 2019-04-04

software engineering. The two courses are usually both required in the university program, but an appropriate way to combine them is not clearly offered.

Discrete mathematics is often set as one of the prerequisites for a software engineering course. The topics in discrete mathematics are important to software development activities, especially analysis. In most curricula, the two core courses are scheduled in different semesters. As a result, the courses are not tightly connected in present teaching activities.

Presently, there are three general approaches to teaching the discrete mathematics course:

(1) Discrete mathematics as a traditional mathematics course;

(2) Discrete mathematics with software engineering applications; and

(3) Discrete mathematics with programming.

When teaching discrete mathematics as a traditional mathematics course, the focus is on the elegance of mathematical logic and proofs. In contrast, when it is taught with software engineering applications, the focus is on the applications of mathematical models and algorithms in software engineering. Finally, when taught with programming, there is a dual focus on mathematics and programming.

We find that the scale of the programming training cases used when teaching discrete mathematics with programming is too small for software engineering training purposes, containing less than one or two hundred lines of code. We argue that discrete mathematics should be not only taught with programming, but also connected tightly with software engineering.

Software development principles should be embedded in the curriculum of software engineering courses. Alongside programming practices, students should be exposed to software development practices early in the undergraduate curriculum and should gain experience in handling various problems in software development.

Additionally, prerequisite courses should provide practice cases for the software engineering course. There are two benefits to this. First, the projects undertaken in prerequisite courses will help students to easily understand the background of the software system to be developed and make students focus on the application of the principles of software engineering. Second, students will review previously learned topics from another viewpoint, which will enhance their understanding and application of the topics.

Discrete mathematics is a very suitable starting point to connect software engineering courses. The topics in discrete mathematics are the foundation of many other courses, such as principle of compilation, database systems, and operating systems. It is therefore natural to connect applications in various courses using the topics of discrete mathematics.

In this paper, we propose a method to connect discrete mathematics and software engineering courses using software development projects derived from various topics in discrete mathematics. The method is shown in Fig. 1.

The idea behind the proposed method is a try-this-before-we-explain-everything approach. Students are encouraged to practice software development activities before taking the software engineering course.

Our work makes two main contributions.

(1) We present a feasible and practical method for connecting discrete mathematics and software engineering.

(2) We provide some examples of novel and practical software development projects to illustrate our method. The projects have not been provided previously and serve as a guide to course connection practices.

The background of the work is introduced in

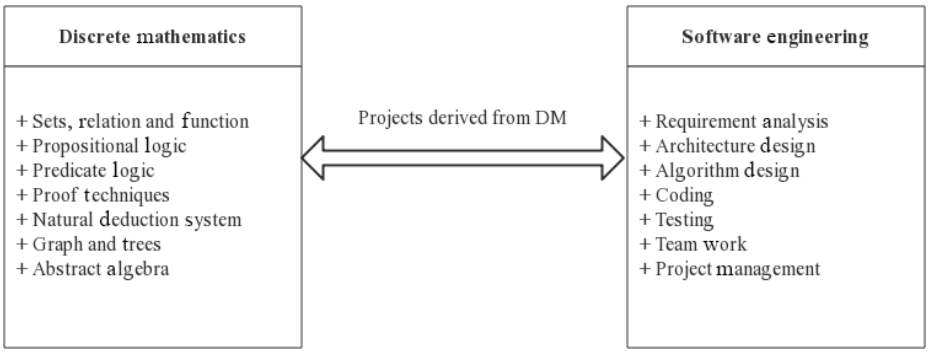


Fig. 1 Proposed method in this paper.

Section 2. Section 3 introduces four sample projects in detail. Lessons learned from course connection practice are presented in Section 4. Section 5 presents related work and its relationship with our method. Finally, we conclude the paper in Section 6.

2 Background

At our university, software engineering is a first-level discipline. The undergraduate curriculum consists of a set of modules, such as introduction to computing and programming, discrete mathematics, principles of compilation, operating systems, networking, software engineering, and so on. In this curriculum, the discrete mathematics course is scheduled in the third semester and the software engineering course is scheduled in the fourth semester. The topics taught in discrete mathematics are as follows:

- Sets, relations, and functions;
- Propositional logic and predicate logic;
- Proof techniques;
- Natural deduction system;
- Graphs and trees; and
- Abstract algebra.

The topics in the software engineering course are requirements analysis, object-oriented analysis and design methodology, software development process, software testing, and foundations of project management.

Before taking these two courses, students study courses on computational thinking and programming in Python in the first and the second semesters, respectively. The topics covered in two courses are

- Core concepts of computational thinking;
- Foundations of programming using Python;
- Applications of computing techniques:
  - Numerical computation;
  - Modeling and simulation;
  - Data storage and processing;
- Basic concepts in software engineering.

Each topic in the applications of computing techniques section consists of a number of sub-topics and concepts. For example, data storage and processing topics include accessing data stored in MySQL databases using Python with the PyMySQL[2] module. Modeling and simulation topics include the modeling and simulation of Von Neumann machines and Turing machines. Among the topics in basic concepts in software engineering is text parsing using compilation techniques, which is introduced using Python modules, such as *pyparsing*[3].

In association with these topics, students have not only been assigned some simple practical projects, such as the implementation of differential computation based on given formulas, but also some complicated projects, such as the implementation of an instruction simulator for a given instruction set. Details of project assignments are available at www.educoder.net (https://www.educoder.net/paths/13 and https://www.educoder.net/paths/121).

Besides course topics, students are exposed to some state-of-the-art teaching and learning paradigms, such as flip-classroom[4], collaborative assessments[5], and so on.

By the time the students take the discrete mathematics course, they have therefore already gained some programming skills and software development experience. Every student should be able to handle problems involving about one hundred lines of Python code.

Through being assigned complicated projects derived from discrete mathematics topics, students will gain experience in software development activities, such as requirement analysis, software architecture design, team work, object-oriented design, and testing[6].

The prior studies of students support our proposal for connecting discrete mathematics and software engineering. On the one hand, students have already gained the essential programming techniques that are to be used in developing the derived projects. On the other hand, the complexity of the derived projects will stimulate students to try their best to fulfill the development tasks.

However, students will use the methods they have at hand (which may or may not be genuine software engineering methods) to finish the projects. Therefore, students will have some exposure to software development before they study software engineering. This exposure will motivate them to seek systematical methods for software development, thus assisting students to gain deeper insight into software engineering techniques.

3 Sample Projects

In this paper, we provide four typical examples to illustrate our method and practices. In fact, there are various comprehensive projects that could be derived from the various topics in discrete mathematics. We provide our samples as motivation for teachers to

practice the proposed method using these and other projects.

The sample projects derived from the topics of the discrete mathematics course are

- A proof checker of natural deduction system;
- A simple database management system;
- A propositional logic proof system; and
- A simple symbolic execution engine.

As shown in Table 1, each project covers several topics in discrete mathematics and some topics in other courses, such as principles of compilation. All of the projects are to be implemented in Python. The implementation techniques used in each project do not go beyond the capabilities of students. However, some of the software development activities are new to students, such as requirement analysis. Other activities, such as testing, may not be new to students, but may have a new emphasis.

Consequently, we will introduce each project in detail from the viewpoint of motivation, requirement, implementation and software development activities.

3.1 Proof checker of natural deduction system

Natural deduction is a name for the class of proof systems composed of simple and self-evident inference rules. In a natural deduction system, a proof is constructed by building the syntactic deducibility relation between proof steps using the given inference rules.

Different natural deduction systems may have different inference rules. In our discrete mathematics course, we introduce a natural deduction system with 21 inference rules. For example, the rule of conjunction elimination is defined as follows,

$$\Gamma, A \wedge B \vdash A, B.$$

In this rule, Γ is a set of predicate logic formulas, and $\vdash$ is used to point out that the relation of deducibility holds between the premises and the conclusion of a rule instance. In what follows, such phrases are called sequents.

Students new to natural deduction, and even more advanced students, are often left stuck in the middle of a proof not knowing what to do next, and even when they have accomplished the task they can be still unsure whether the proof is valid.

3.1.1 Motivation

The project is motivated by some existing tools, including a natural proof editor, proof checker, and proof assistant with a proof-hint for each step[7–9]. In fact, constructing an automatic natural deduction proof generator is very hard for second-year undergraduate students. However, based on the distinct feature of natural deduction proof sequents — syntactically matching according to the given rules — building a proof checker is a highly suitable project. The checker will be used to determine whether the given proof sequent is correct according to the given premise, conclusion, and inference rules. Therefore, we derive a project from the natural deduction system topic.

Table 1 Overview of the sample projects.

Project	DM topics	Implementation techniques	Software development activities
Proof checker of natural deduction system	• Basic logic • Natural deduction system • Sets	• Compiler construction • Object-oriented programming • Recursion • Pattern matching of objects stored in a list	• Requirement analysis • Architecture design • Decomposition • Interface design • Testing
Simple database management system	• Relations • Sets	• Python sequential data structure • Object-orient programming • Domain specific language	• Requirement analysis • Decomposition • Testing
Propositional logic proof system	• Basic logic • Sets	• Compiler construction • Object-oriented programming • Recursion • Interaction and integration with third-party modules	• Requirement analysis • Decomposition • Composition • Testing
Simple symbolic execution engine	• Predicate logic • Graph • Sets • Semantic equivalence	• Compiler construction • Object-oriented programming • Recursion • Interaction and integration with third-party modules	• Requirement analysis • Architecture design • Decomposition • Composition • Testing

3.1.2 Requirements

The system to be implemented should have the following features.

• The system should support a predefined format for the user to enter a natural deduction proof sequent associated with the inference rules used in each step. The format is defined as shown in Fig. 2. Each line consists of five parts separated by Tabs: line no, premise, ⊢, conclusion, and inference rule used.

• The checker will notify users whether the proof sequent is valid. In case of an invalid proof, the user should be notified with some hint as to the incorrect parts of the proof that cause it to be invalid.

• At present, no graphical user interface is required and the user input is provided in a formatted text file.

3.1.3 Implementation

To implement the system, the following techniques are involved.

• Compiler construction: Lexical analysis and parsing techniques are required to parse the predicate logic formula and the proof sequent entered by the user. In this project, the *pyparsing* module is sufficient to construct a compiler for grammar in Backus-Naur Form (BNF).

• Object-oriented programming: Classes are defined and used to encapsulate formulas/sub-formulas and inference rules of a natural deduction system.

• Recursion: For formula defined in BNF format, it is naturally to traverse the parsing tree in a recursive manner.

• Matching of objects in a list: For formulas and inference rules stored as objects, pattern matching should work at an object level, dealing with objects stored in Python sequential data structures, such as in a List.

3.1.4 Software development activities

Implementing the project in a course context is a difficult task for a single student. Therefore, students are organized into teams of four to five students. During the implementation, the software development activities include:

• Requirements analysis to find all the functionalities to be implemented;

• Designing the architecture of the system and decomposing the whole system into sub-systems;

• Defining detailed data structures and interfaces for system composition;

• Cooperations among team members; and

• Testing of the checker by the team, other teams and teachers using corresponding homework from textbooks.

Figure 2 shows a sample proof sequent, where a user is aiming to construct a proof using the given inference rules to show that the premise will deduce the conclusion. The sequent is correct, and the checker will return a valid result.

However, if we change the "ore 3, 6" to "ore 4, 6", the proof sequent is invalid and, according to the requirements, notifications should be provided to help the user to locate the errors in the proof sequent. The output for this example is shown in Fig. 3 in Chinese, indicating that the seventh step is wrong due to the incorrect use of an inference rule.

```
Premise: A->C, B->C, A\/B
Conclusion: C

1.      A,A->C, B->C    |-      A       prem
2.      A,A->C, B->C    |-      A->C    prem
3.      A->C, B->C,A    |-      C       imple 1,2
4.      B,A->C, B->C    |-      B       prem
5.      B,A->C, B->C    |-      B->C    prem
6.      A->C, B->C,B    |-      C       imple 4,5
7.      A->C, B->C,A\/B |-      C       ore 3,6
```

Fig. 2 A sample proof sequent.

3.2 Simple database management system

In the discrete mathematics course, after introducing the traditional binary relation on a set, we extend the relation topic to cover n-ary relations on n sets. The n-ary relation and the operations defined on it are the theoretical basis for relational databases. Three operators are introduced on n-ary relation R and S with condition c.

• "Selection" is used to select a subset of R that satisfies the given condition c;

• "Projection" is used to define a new relation identical to R except that it has only the specified fields; and

• "Join" is used to define a new relation that has tuples with all fields from R and S constructed by matching R and S tuples, where matching here means identical values on sets of joined fields.

3.2.1 Motivation

We show students the application of formal mathematics topics, such as relations, to reveal the principles of database management. At the same time, it is useful

```
litundeMBP-2:proofChecker_Python3 tunli$ python test_pc.py
证明中的以下步骤有问题： 7
7. 使用了析取消去规则，但是：当前步骤前提（去掉最后一个公式）与2个被应用步骤的
前提（去掉最后一个公式）不相等
```

Fig. 3 Output of the checker.

for students to continue to be exposed to software development activities. Therefore, the project to be derived from this topic is to implement a simple database management system.

3.2.2 Requirements

The simple database management system should have following features.

- Users can create tables.
- The system has support for "projection", "selection", and "join" operations.
- A Domain-Specific Language (DSL)[10] is defined (to simplify the development task, we do not require the system to support SQL). The DSL should be implemented as objects in Python supporting the former two features.
- At present, no graphical user interface is required.

Figure 4 shows a sample result of running the system. In Fig. 4, ">>>" is the command prompt token of the Python IDLE consoler. "Rel", "add", and "display" are operators of the DSL that are used to create a table, add records to a table, and display the contents of a table, respectively.

3.2.3 Implementation

To implement the system, the following techniques are used.

- Object-oriented programming: Classes are defined and used to define and support the DSL for relational database operations. All operations involving database management and operations are then implemented as member functions or functions based on the defined classes.

```
>>> classRoom = Rel(("Place", "Seats", "BoardType", "Computer"))
>>> classRoom.display()
+-------+-------+-----------+----------+
| Place | Seats | BoardType | Computer |
+-------+-------+-----------+----------+
+-------+-------+-----------+----------+
>>> classRoom.add(Place="Fan171", Seats="80", BoardType="Chalk", Computer="Yes")
>>> classRoom.add(Place="Lws210", Seats="25", BoardType="No", Computer="Yes")
>>> classRoom.add(Place="Nek138", Seats="50", BoardType="Chalk", Computer="No")
>>> classRoom.add(Place="Agr212", Seats="200", BoardType="No", Computer="Yes")
>>> classRoom.display()
+--------+-------+-----------+----------+
| Place  | Seats | BoardType | Computer |
+--------+-------+-----------+----------+
| Lws210 | 25    | No        | Yes      |
| Fan171 | 80    | Chalk     | Yes      |
| Nek138 | 50    | Chalk     | No       |
| Agr212 | 200   | No        | Yes      |
+--------+-------+-----------+----------+
```

Fig. 4 Sample running output of the simple database management system.

- Python data types: Selecting a suitable data structure will help students to simplify the task. Taking the background of students into consideration, we recommend using the embedded data structures of Python, such as tuple and dictionary, rather than defining new data structures.
- Domain-specific language: A DSL is defined and implemented based on existing languages, (e.g., Python). Students are first required to analyze the operations involved in database management and operations. These common operations are then extracted and defined as member methods inside a class or functions outside of classes with necessary parameters and proper outputs.

3.2.4 Software development activities

According to the complexity of the project, students are organized in teams of at least two members. During the implementation, the software development activities include:

- Requirements analysis and development task decomposition according to functional requirements;
- Extraction of the supported database management operations and mapping them to corresponding functions;
- Assigning of sub-tasks to team members;
- Composing the necessary modules to form the system; and
- Testing of the database management system by students from other teams or by teachers.

3.3 Propositional logic decision system

In computer science, the satisfiability problem is concerned with deciding whether a given Well-Formed Formula (WFF) is valid, satisfiable or unsatisfiable. The given WFF may be expressed in propositional logic language or in predicate logic language.

3.3.1 Motivation

Satisfiability problems are solved manually as exercises in discrete mathematics courses. However, when there are too many propositions, a computer program is needed to find satisfiable solutions for the given formula. There are many solvers for this problem, such as minisat[11] and Z3[12]. Most of these tools provide an Application Programming Interface (API) so they can serve as the back end of other applications. Therefore, besides motivating students to automate theoretical problem solving through software development, they are also given an opportunity to experience the integration of third-party tools in developing software.

3.3.2 Requirements

The decision system should have the following features.

• The system can recognize formulas expressed in a predefined propositional language, which is given in its BNF format.

• Z3 is seamlessly integrated as the back-end solver.

• The system can translate the given formulas into a format that can be recognized by the Z3 solver.

• The decision result from Z3 is to be translated into a human-readable format.

3.3.3 Implementation

The following techniques are involved in implanting the system.

• Compiler construction: Lexical analysis and parsing techniques are required to parse propositional logic formula. The *pyparsing* module is sufficient to construct a compiler for grammar in BNF format. The major part of the compiler is the code generation phase, which will translate the parsed formula into the Z3-recognizable format.

• Object-oriented programming: Classes are defined and used to encapsulate formulas/sub-formulas with their code generation operations.

• Recursion: For formulas defined in BNF, it is natural to traverse the parsing tree in recursive manner. Especially during the code generation phase, translating the whole formula is implemented by translating its sub-formulas recursively.

• Interaction and integration with third-party modules: The generated Z3 input is fed to Z3 via its API, then the satisfiable solution results returned by Z3 are translated into human-readable format.

3.3.4 Software development activities

The project is less difficult in comparison to the former two projects. Therefore, students are organized in teams with at most three members. During the implementation, the software development activities include:

• Requirements analysis and development task decomposition according to functional requirements;

• Design of the system architecture, divided into several sub-modules;

• Assigning of sub-tasks to team members;

• Composing the necessary modules to form the system; and

• Testing of the decision system by students from other teams or by teachers using exercises from the textbook.

3.4 Simple symbolic execution engine

In the discrete mathematics course, one of the applications of the basic logic topic is a program correctness proof. By constructing a Hoare triple[13] using precondition, command, and postcondition, assertions expressed in predicate logic are checked to prove the correctness of the given programs. The command is extracted from program statements under proof, usually using symbolic execution techniques[14].

3.4.1 Motivation

To implement a complete program correctness prover is beyond the capabilities of second-year students. Therefore, we derived a project to implement just the core of a prover – a simple symbolic execution engine. The purpose of the project includes connecting the theory of the topic to its applications, integrating third-party modules, and experiencing software development activities.

3.4.2 Requirements

The engine should have following features.

• It is able parse programs written in a simple C-like programming language, supporting simple type definitions (e.g., integer, real, and boolean), assignments, and if-else conditional statements.

• It can translate the given programs into formulas in a Z3-recognizable format. Static Single Assignments (SSA) generation techniques[15] are required during translation. Figure 5 shows an example code snippet and its corresponding translated formulas using SSA generation techniques.

• It can parse and translate assertions expressed in first-order into formulas accepted by Z3. The assertions are used to express post-condition, pre-condition, or test generation constraints.

• The engine can solve the translated constraints by calling Z3 and show results from Z3 in a human-readable format.

$\text{int } x=0, y;$	$x_1 == 0$
$y = x + 1;$	$\wedge y_1 == x_1 + 1$
$x = 2 \times x;$	$\wedge x_2 == 2 \times x_1$
$\text{int } z = x \times y;$	$\wedge z_1 == x_2 \times y_1$

Fig. 5 A code snippet and its SSA constraints.

3.4.3 Implementation

The following techniques are involving in implementing the system.

• Compiler construction: Lexical analysis and parsing techniques are required to parse programs written in the predefined simple programming language. The *pycparser*[16] module is sufficient to construct a compiler for grammar in BNF. The major part of the compiler is the code generation phase, which will translate the parsed C-like statements into formula in both SSA and Z3 formations.

• Static single assignment: Students should study SSA techniques on their own. The difficult part of this is the indexing of variables and the automatic index increment mechanics when generating SSA.

• Object-oriented programming: Classes are defined and used to encapsulate the parsing result and SSA generation operations.

• Recursion: For statements defined in BNF, it is natural to traverse the parsing tree in a recursive manner. Especially during the SSA generation phase, translating a statement is implemented by translating its sub-statements recursively.

• Interaction and integration with third-party modules: The generated SSA should be fed to Z3 via its API, and the results of the Z3 solver should then be translated into a human-readable format.

3.4.4 Software development activities

The project is to be assigned to a team of at least three students. Suring the implementation, the software development activities include:

• Requirements analysis and development task decomposition according to functional requirements;

• Design of the system architecture, divided into several sub-modules;

• Assigning of sub-tasks to team members;

• Composing the necessary modules to form the system; and

• Testing of the engine by students from other teams or by teachers using some sample programs.

4 Practices and Lessons Learned

We have practiced this method to connect the two courses for three years. The projects are distributed at the beginning of the discrete mathematics course, when students are required to form teams and choose one project. The discrete mathematics course is taught in two 2-hour sessions per week over 13 weeks. It begins with the basic logic topics, followed by sets and relations topics. Therefore, students could begin on their projects after three to six weeks of course study, leaving them within eight to ten weeks for project development.

During the development process, teachers only provide advice on the implementation techniques, not on the engineering methods of how to develop software. The difficulties that students encounter in software development will assist them to develop a common sense to take in to their work on the general principles on software engineering.

To connect the discrete mathematics and software engineering courses, we ask students to review and refactor their work in the software engineering course in the following semester. This review and refactoring take place by comparing the actual project practices with the principles and engineering methods taught in the software engineering course.

The following questions are put to students to rethink the development of the selected projects.

• What mistakes have you made in requirement analysis? How and in what aspects could the requirement analysis be improved?

• If object-oriented design and programming was used in your project, can you enhance the quality of your code by refactoring?

• Why did your project fails in meeting the deadline? What could be done to accelerate your development process?

• Did team work hinder the development process? What could be done to improve team work?

• How did you carry out testing? What aspects of testing could be improved?

By answering these questions and rethinking the challenging aspects of their development project experience, students will gain a deeper insight into the engineering methods for software development.

We also conduct a survey of students regarding the projects derived from discrete mathematics, looking at the levels of engagement with projects, lines of code, and the strengths and weaknesses of the projects. The results, which help us to continually improve our project designs, are shown in Table 2.

Generally, the survey results show that students do obtain benefits from working on these projects. Three major benefits can be identified: students gain a deeper understanding of the foundations and applications of theoretical topics in discrete mathematics; students are motivated and trained by participating in software

Table 2 Evaluation of the sample projects by students.

Project	Lines of code	Engaged	Difficulty	Strength	Weakness
Natural deduction proof checker	About 2000	• Principle of compile • Software engineering	Hard	• A very nice mixture of theory and practice • A comprehensive project to experience software development activities	• It may be too hard for the second year students without systematical software development training. • Students need more help from instructors on compiler construction and pattern matching. • The development process is late than scheduled.
Simple database management system	About 600	• Database system • Software engineering	Medium	• A very nice mixture of theory and practice • A very comprehensive project to experience software development activities • It is very suitable for students at present stage	• The tough part is how to map theoretical relational operations to their implementation. • Instructors need to provide more examples on relational operations.
Basic logic proof system	About 1000	• Principle of compile • Software engineering	Medium	• A very nice mixture of theory and practice • A very nice project to experience software development activities • A very comprehensive project for experiencing integration of existing software systems	• The most tough part is the implementation of code generation phase, which needs more instructions from teachers. • To understand the usage of Z3 API spends too much time, because the documents are not so rich.
Simple symbolic execution engine	About 1500	• Principle of compile • Software engineering	Hard	• A very nice mixture of theory and practice • A very comprehensive project to experience software development activities	• The C-like language compiler may be too hard for the second year students. • Instructors need to provide much help on compiler construction and integration of third-party modules. • The actual development timetable is late than scheduled.

development projects appropriate to their background and capabilities; and students can learn much more from the software development process by comparing their own practices with the principles and methods learned in the software engineering course.

However, there are some lessons learned, from which we can improve our future practices on the design of projects.

• The software development abilities of students should be estimated properly. Some projects may be too difficult for students at this stage of their education. We should simplify the requirements of projects or provide more help during project development.

• For the two projects evaluated as hard, a pre-built implementation framework is needed to ease the development task and balance the workload between the four projects.

• There are overlaps between the three projects related to mathematical logic, which makes code copy between teams possible.

• Additional projects engaging with additional topics are needed to broaden the courses involved to operating systems, digital circuit design, etc.

5 Related Work

Cohoon and Knight[17] were the first to attempt to connect discrete mathematics and software engineering. They aimed to not only promote an understanding and appreciation of the discrete mathematical structures that are the foundation of software engineering theory, but also provide motivation and training in modern software development and analysis tools. The cornerstone of their connected courses is the use of problems that arise in software development to motivate the mathematics analysis.

However, our method and practices are in the reverse

direction to their work. We try to use the projects that derive from discrete mathematics to motivate the software development activities. Our object is to maintain software development training throughout the curriculum.

There have been some other practices used to connect computer science and discrete mathematics. Page[18] used a functional program to evaluate the question of whether studying mathematics improves programming skills. He built a software-oriented discrete mathematics course by reasoning software using mathematical methods. The three-year-long evaluation showed that studying mathematics may be effective in improving programming skills.

Flatland and Matthews[19] used an engaging problem, which could be used in both a discrete mathematics course and a programming course, to strengthen the links between computer science and mathematics. They selected a problem from a data structure course to be used as an analysis case in discrete mathematics. By comparing the mathematical analysis results and simulation results, programming and mathematics were connected.

The work in Refs. [20, 21] integrated mathematics and computer science by reinforcing mathematical topics through programming exercises and motivating applications. In Ref. [22], VanDrunen argued that a discrete mathematics course that introduces programming in the functional style provides an ideal context for the integration of discrete mathematics in computer science.

However, the scale of the examples used in these works is small, and cannot comprehensively train students in software development. In our work, we provide four sample projects, the implementation of which involves around one or two thousand lines of code. Further, we provide projects from discrete mathematics to be used as exposure to software engineering practices.

We believe that our method complements previous work and is an effective way to integrate mathematics and software development for the software engineering curriculum.

6 Conclusion

In this paper, we report on our experiences connecting discrete mathematics and software engineering using a try-this-before-we-explain-everything approach. The philosophy and consideration behind the practices and the design of several practical projects are introduced. Survey results show that our approach is effective in linking courses in a software engineering curriculum. The results also provide stimulus to continuously improve our method and to design more comprehensive projects to tightly connect more courses in the near future.

Acknowledgment

The work was supported in part by the National Key R&D Program of China (No. 2018YFB1004202).

References

[1] M. Ardis, D. Budgen, G. W. Hislop, J. Offutt, M. Sebern, and W. Visser, SE 2014: Curriculum guidelines for undergraduate degree programs in software engineering, *Computer*, vol. 48, no. 11, pp. 106–109, 2015.

[2] PyMySQL, https://github.com/PyMySQL, 2019.

[3] P. McGuire, *Getting Started with Pyparsing (1st ed.)*. Sebastopol, Canada: O'Reilly, 2007.

[4] B. A. T. Brown, Flipping the classroom, in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)*, Raleigh, NC, USA, 2012, p. 681.

[5] H. Yuan and P. Cao, Collaborative assessments in computer science education: A survey, *Tsinghua Science and Technology*, vol. 24, no. 4, pp. 435–445, 2019.

[6] T. Li, W. Liu, X. Guo, and J. Wang, Software testing without the oracle correctness assumption, *Frontiers of Computer Science*, DOI: 10.1007/s11704-019-8434-4.

[7] E. Björnsson, F. Johansson, J. Liu, H. Ly, J. Olsson, and A. Widbom, Proof editor for natural deduction in first-order logic– The evaluation of an educational aiding tool for students learning logic, Bachelor degree thesis, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden, 2017.

[8] K. Klement, Fitch-style natural deduction proof editor and checker, https://github.com/OpenLogicProject/fitch-checker, 2019.

[9] J. Jacky, FLiP: Logical framework in Python, http://staff.washington.edu/jon/flip/www/index.html, 2019.

[10] M. Fowler, *Domain Specific Languages (1st ed.)*. Upper Saddle River, NJ, USA: Addison-Wesley Professional, 2010.

[11] N. Eén and N. Sörensson, An extensible SAT-solver, in *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, Santa Margherita Ligure, Italy, 2003, pp. 502–518.

[12] L. M. Moura and N. Bjørner, Z3: An efficient SMT solver, in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*, Budapest, Hungary, 2008, pp. 337–340.

[13] C. A. R. Hoare, An axiomatic basis for computer programming, *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[14] R. Baldoni, E. Coppa, D. C. Delia, C. Demetrescu, and I. Finocchi, A survey of symbolic execution techniques, *ACM Computing Surveys*, vol. 51, no. 3, pp. 50:1–50:39, 2018.

[15] J. Aycock and R. N. Horspool, Simple generation of static single-assignment form, in *Proceedings of the 9th International Conference on Compiler Construction (CC'00)*, Berlin, Germany, 2000, pp. 110–124.

[16] E. Bendersky, pycparser module, https://github.com/eliben/pycparser, 2019.

[17] J. P. Cohoon and J. C. Knight, Connecting discrete mathematics and software engineering, in *Proceedings of the 36th IEEE Annual Conference on Frontiers in Education (FIE'06)*, San Diego, CA, USA, 2006, pp. 13–18.

[18] R. L. Page, Software is discrete mathematics, *SIGPLAN Notices*, vol. 38, no. 9, pp. 79–86, 2003.

[19] R. Y. Flatland and J. R. Matthews, Using modes of inquiry and engaging problems to link computer science and mathematics, in *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE'09)*, Chattanooga, TN, USA, 2009, pp. 387–391.

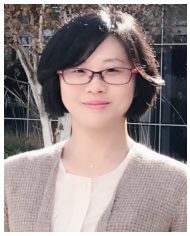
[20] K. McMaster, N. Anderson, and B. Rague, Discrete math with programming: Better together, in *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'07)*, Covington, KY, USA, 2007, pp. 100–104.

[21] R. Arnold, M. Langheinrich, and W. Hartmann, Infotraffic: Teaching important concepts of computer science and math through real-world examples, in *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'07)*, Covington, KY, USA, 2007, pp. 105–109.

[22] T. VanDrunen, Functional programming as a discrete mathematics topic, *ACM Inroads*, vol. 8, no. 2, pp. 51–58, 2017.

Tun Li is a professor in the School of Computer, National University of Defense Technology. He received the PhD degree from National University of Defense Technology in 2003. His research interests include electronic design automation and VLSI design verification. He taught several courses for undergraduate and graduate students, such as discrete mathematics, computer programming, advanced software engineering, etc.

Wanwei Liu is an associated professor in School of Computer, National University of Defense Technology. He received the bachelor degree and the PhD degree from National University of Defense Technology. His research interests include theoretical computer science, software/hardware verification, and software engineering.



Juan Chen received the PhD degree from National University of Defense Technology, China, in 2007. She is now an associate professor at National University of Defense Technology, China. Her research interests focus on supercomputer systems and energy-efficient software optimization method.

Xiaoguang Mao is a professor in National University of Defense Technology. He received the PhD degree from National University of Defense Technology in 1997. His research interests include software maintainability and software dependability. He taught several courses for undergraduate and graduate students, such as discrete mathematics, computer programming, software dependability, etc. He has published more than 100 papers in various conferences and journals.

Xinjun Mao is currently a professor in the School of Computer, National University of Defense Technology. His main research interests are in the area of software engineering for intelligent systems, self-adaptive and self-organized systems, and autonomous robot systems. He taught several courses for undergraduates, such as software engineering, software architecture and design, etc. He has published more than 100 papers in various conferences and journals.