6th Formal Methods Teaching Workshop, FMTea 2024, Milan, Italy, September 10, 2024
LNCS vol. 14939, pp. 25–42.

Introducing GitHub Classroom into a Formal Methods Module

Soaibuzzaman and Jan Oliver Ringert

Bauhaus-University Weimar, Germany

Abstract. We have developed an MSc-level module on Formal Methods for Software Engineering with exercises on applying SAT solvers, SMT solvers, Alloy, and nuXmv. In the first iteration of the module, assignments were submitted as documents and archive files. Here, we report on our experience of moving the exercises to GitHub Classroom and automating the feedback process through test cases. The main challenges we encountered were related to supporting free-response tasks and designing test cases that allow for multiple solutions, provide incremental feedback, and do not encode a solution. We present our setup of exercise repositories, test cases, and feedback report generation. We detail our approach in addressing the challenges of migrating from worksheets to GitHub Classroom and report on survey-based student feedback.

Keywords: teaching, formal methods, GitHub classroom, feedback

1 Introduction

We have developed an MSc-level module on Formal Methods for Software Engineering with exercises on applying SAT solvers [20,4], SMT solvers [23,3], Alloy [15], and nuXmv [5]. As well recognized by others, practical examples and exercises are important when teaching Formal Methods [6,7,8,26,19]. Our module contains five chapters, where each technical chapter (2-5) is supported by a specification and an implementation exercise (see Sect. 2). In specification exercises, students learn to use specification languages for smaller examples. In implementation exercises, students automate the programmatic translation of domain problems to solvers.

In the first iteration of the module, exercises were submitted as documents and archive files. Here, we report our experience moving the exercises to GitHub Classroom (GHC). Our main goals were to improve the student learning experience [28,32,10] by using autograding and providing immediate feedback [22] and to shorten grading times and teacher resources.

We present our setups of exercise repositories, test cases, and feedback report generation in Sect. 4.2. Our main challenges were supporting free-response tasks and designing test cases that allow for multiple solutions, provide incremental feedback, and do not encode a solution (see Sect. 4.3). We detail our approach to addressing the challenges of migrating from worksheets to GHC. We report

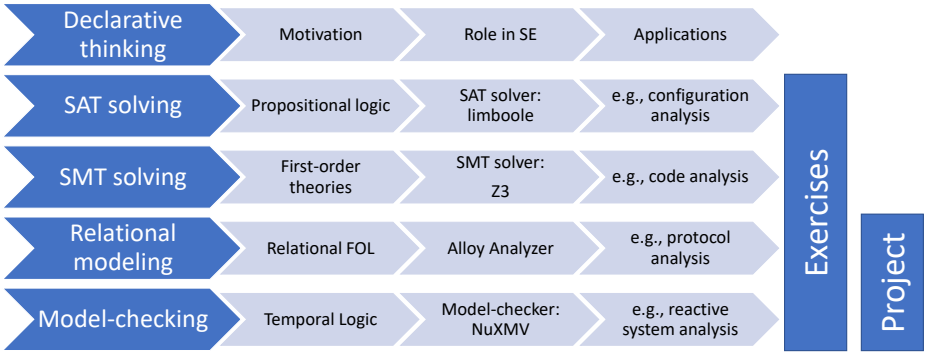

Fig. 1. Outline of Formal Methods for Software Engineering divided into five chapters

on survey-based student feedback collected after each exercise in the year of the migration to GHC in Sect. 5.

2 Module Overview

We have developed the module Formal Methods for Software Engineering as an MSc-level module of 6 ECTS (roughly 180 hours of student workload in one semester). The module is offered to students from three MSc degree programs at Bauhaus-University Weimar: Computer Science for Digital Media, Human-Computer Interaction, and Digital Engineering (most engineering and computer science backgrounds with mandatory programming experience and a total of 36 ECTS in computer science subjects). Our module has no formal prerequisites, but we expect that students have a BSc-level background in mathematics and basic programming knowledge. Additionally, Software Engineering and Object-Oriented Programming is recommended for Digital Engineering students from non-computer-science backgrounds.

An outline of the module is shown in Fig. 1 with five main chapters spread over 15 weeks. The first chapter *Declarative thinking* motivates the use of specifications for SE. Each following chapter introduces a logic, a language, a tool, and applications, e.g., the second chapter *SAT solving* introduces propositional logic, the syntax of limboole [4], and limboole; the final chapter *Model-checking* introduces Linear Temporal Logic [25], the SMV language, and the nuXmv tool [5].

The module's content and intended learning outcomes focus on selecting and applying specification formalisms and tools to address software engineering challenges rather than on the tools' underlying algorithms and decision procedures. The module is assessed 100% based on a project. Passing all seven exercises is mandatory before submitting a project. Two extensions or resubmissions (out of seven) are granted for failed exercises. Students work in pairs or alone.

2.1 Exercise Structure

Chapters 2-5 from Fig. 1 were each supported by a specification exercise (one week to complete) and an implementation exercise (two weeks to complete).

Specification exercises require manually writing specifications and encoding problems, e.g., solving a math puzzle using SMT as shown in Fig. 2 (Ex. 3, Task 2) or determining semantic equivalence of two LTL formulas using the nuXmv [5] model-checker. The goal of these exercises was to learn the features of the new language and to be able to express puzzles and problems in each formalism.

Implementation exercises require the automation of the encoding and the solving tasks as a program, e.g., finding dead features in a feature model [21] (SAT solving) or configuring a PC system from components with different prices, budget limitations, and configuration constraints (SMT solving).

We chose Java as a common language known to most students. The generation of input for the solver was based on String manipulations for SAT solving and APIs for SMT (JavaSMT [3]) and Alloy [15]. We did not provide an implementation exercise for the *Model-checking* chapter, as the students focused on their projects. The goal of these exercises was to learn the generalization of problems and the automation of systematically encoding and solving them.

We provide an overview of exercises and present some excerpts of the worksheets in Fig. 2. All exercise materials are available for inspection, modification, and reuse (Apache 2.0 license) from https://github.com/fm4se/exercises/:

1. SAT *spec*: formulas, checking conclusions, verifying Role-Based-Access
2. SAT *impl*: Feature Model analysis [21], dead features, product preservation
3. SMT *spec*: Agatha puzzle [24, P.55], math puzzle, PC configuration
4. SMT *impl*: PC configuration from CSV-files, budget and purpose
5. Alloy *spec*: domain model, Agatha puzzle [24, P.55], Trash can [19]
6. Alloy *impl*: Analysis of Alloy modules: dead signatures, minimal scopes
7. nuXmv *spec*: LTL equivalence, counterexamples, chess knight moves

3 Related Work

3.1 GH in Classroom

Numerous studies have evaluated the use of GitHub and GitHub Classroom for teaching programming courses. Hsing and Gennarelli [14] conducted a study exploring the advantages of incorporating GitHub into the classroom. Their research revealed that students who received feedback through GitHub achieved better learning outcomes and developed the necessary skills for collaborative work. Their survey included 7,530 students and 300 educators. Furthermore, a group of instructors from the University of Auckland [32] shared their experience using GitHub Classroom. They implemented Git and related systems in various courses with varying class sizes, experience levels, and contexts. Their findings indicated that while introducing GitHub to the classroom increased the

Ex. 3, Task 2
Encode the Puzzle:
(☁ * 🍼) + 🌼 = 110

(🌼 - 🍼) + ❤ = 17

(🍼 * ☁) - 🍼 = 90

(🌼 - ☁) - ❤ = ?
Note: use the same names as in the template to encode the puzzle.

Ex. 3, Task 3
Create a reusable encoding of the selection of PC components (see below table) and the calculated price of a PC.

- The selection must satisfy the listed constraints.
- Each component may only be selected once, i.e., you cannot include two CPUs.

The encoding should be reusable in the following sense:

- A user states a purpose and budget and the encoding produces models that represent PC configurations matching the requirement, if possible.
- Different purposes add further constraints listed below.

Use the following template to start with. It already contains the encoding of the variables budget and purpose. Keep these names.
Constraints:

- Each computer needs basic components: CPU, Motherboard, RAM, Storage
- ...

Purpose:

- Office use requires optical drive
- ...

Ex. 5, Task 1
Create an Alloy model for a scenario of your choice. The senario must make sense, i.e., not a sig A ... sig B example, and it needs to be different from the examples in the lecture.

- Declare at least 4 signatures each with at least 2 fields.
- Use inheritance between signatures at least once.
- Define at least 2 facts and 2 predicates.
- Add two run commands to your model, one unsatisfiable and one that returns at least 2 instances.

Ex. 7, Task 4
Create an SMV module to encode the moves of a knight on an 8x8 chess board. The knight always starts at coordinate (0, 1).
Knights can move as illustrated below ...

Fig. 2. Excerpts from exercise worksheets. The complete worksheets and task descriptions are available from https://github.com/fm4se/exercises/.

teaching workload, it provided significant value to the courses for students and instructors.

Angulo and Aktunc [2] have shared their insights on utilizing GitHub as an effective teaching tool for programming courses. They observed that GH offers advantages to educators by providing a collaborative classroom environment. Glassey [9] surveyed eight tools designed to ease the technical challenges of Git and GH in an educational setting, analyzing their general, technical, and pedagogical aspects. In a separate case study [10], Haaranen and Lehtinen shared their extensive experience teaching web software development using Git from both the instructors' and learners' perspectives. Kertész [16] found that students in an Operating Systems class preferred GHC's collaborative platform. They reported that the benefits of collaboration include learning from mistakes, receiving faster assistance from peers, mastering a common development platform, and improving decision-making through diverse solutions.

GitHub Classroom is also utilized to teach formal methods. To illustrate, Divasón and Romero [7] utilized GHC to submit their students' exercises in formal verification teaching, while Rozier [26] employed GHC to teach applied formal methods. In both instances, all homework was distributed and collected through GitHub Classroom. However, the authors provided no further details regarding their experience with the platform.

3.2 Autograding & Feedback

Automated grading of exercises has been a common feature in computer science courses for a long time [13,22]. Messer et al. [22] conducted a systematic review and report a focus mainly on OOP-languages and the correctness aspect. They report limitations in the quality of feedback and imposing limits on showing creativity. Haldeman et al. [12] proposed a methodology for extending autograders to provide meaningful feedback by collecting and analyzing exercise submissions and generating hints that can be used in future semesters. Later, they proposed a framework called CSF^2 [11], which provides formative feedback on programming exercises. Although these systems and frameworks are related, they primarily focus on programming languages and may be challenging to adapt in the context of formal methods.

3.3 Automatic Exercise Generation

The automatic generation of programming exercises and the concept of autograding have piqued interest. Sovietov [29] developed a general scheme for generating programming exercises in a Python language course. This scheme can produce intricate exercises that demand complex solutions from students. Tiam-Lee and Sumi [30] presented a method for generating customized, entry-level programming exercises, while Sarsa et al. [27] utilized large language models to generate programming exercises automatically. The content generated was assessed qualitatively and quantitatively, mostly novel and sensible. Tscherter [31] proposed

the Exorciser system for generating automatic exercises for a theory of computation class, e.g., regular languages and Markov algorithms. Exorciser provides feedback and visualization. Ábrahám et al. [1] discussed challenges in designing and generating automatic exercises for satisfiability checking and setting quality criteria for exercise generation.

4 Migration of Exercises to GitHub Classroom

GitHub Classroom (GHC) is an educational tool that helps teachers use Git and GitHub in their classes. Instructors create an assignment linked to a template repository, generating an invitation URL for students [32,2]. When a student accepts, an individual repository is created in the same GitHub organization. All worksheets are available as GitHub template projects from https://github.com/fm4se/exercises (Apache 2.0 license).

4.1 GHC Setup and Exercises

To transition from our traditional worksheets to GHC, we have implemented a task repository (template for replicated student repositories) for each worksheet. In our case task repositories are Java projects using the Gradle Build Tool[1]. Each repository's `README.md` (the *worksheet*) contains a comprehensive task breakdown and additional resources like videos and examples.

We have leveraged GitHub Actions to create an automated testing system and Continuous Integration (CI) workflow for evaluating student submissions. Our tailored workflow installs required dependencies and binaries, e.g., limboole [4], Z3 [23], nuXmv [5], related to each exercise. For some exercises, the workflow generates unique tasks (as in Fig. 2 (Ex. 3, Task 2)). We have employed a Community Action with a point bar[2] to display the exercise completion rate.

Further GHC features and limitations We allowed students to complete the exercises in pairs. GHC enables group assignments, and students are responsible for joining their own groups. However, joining the wrong group can make it challenging for the instructor to locate the correct repository for grading purposes [32]. To avoid this confusion, we provide administration rights to the students for adding their team members to the repositories.

GHC enables linking learning management systems (LMS) to import student rosters [2]. We did not link the systems to avoid data protection concerns and instead asked students to submit their repository links on our LMS (Moodle).

The free GitHub plan only offers a monthly allocation of 2,000 GitHub Action minutes per organization. This may limit the execution of test cases and report generation. Our setup did not get anywhere near these limits.

[1] Gradle Build Tool https://gradle.org/
[2] Available at https://github.com/marketplace/actions/points-bar

Task 4: Knight Moves

Test	Status	Reason
start (0,1)	☑ Passed	-
next (2,2)	✗ Failed	⚠ RuntimeException
reach (7,7)	✗ Failed	Starting at (0,1) the knight can reach (7,7)

Fig. 3. Excerpt of a feedback report of Ex. 7, Task 4 from Fig. 2

4.2 Tests and Automated Feedback

Providing feedback is essential for students to comprehend the quality, depth, and relevance expected of their work [28]. It offers specific information regarding their learning growth, which is a great motivator. Moreover, it encourages students to reflect on their work.

Automated feedback offers immediate evaluation of student work, empowering them to improve before the final assessment. Traditional submit-and-mark exercises have limitations, including long wait times, no opportunity for improvement after submission, and less relevant feedback. Descriptive feedback helps students understand what went wrong and why, improving their work.

Safeguarding test integrity Task completion is measured by test cases (as shown in Table 1). Providing these as part of the task repository exposes them to tampering, e.g., a student could modify test cases to always pass. To address this concern, we isolated the test cases within a secure cloud-based environment. During the GitHub Actions evaluation, we retrieved these test cases, integrated them into the student code, and executed the tests.[3]

Generating feedback report When running JUnit tests with Gradle, an XML report is generated with detailed information about the test results. However, this XML report is not very intuitive to read for students. As a solution, we have parsed this XML report to create a comprehensive and easily understandable feedback report in a markdown format for students. An excerpt of the feedback report for Ex.7, Task 4 from Fig. 2 is presented in Fig. 3.

Local feedback and report generation GHC generates a feedback report for every push to the repository. However, students may want to generate and inspect feedback locally. We used Gradle and JUnit to execute the test cases, and a Python script to create the feedback report. Given a local installation and the required binaries of the FM tools, students may run the test cases in their IDE or by using the command `./gradlew test`. Given a Python installation they may further generate the feedback report via `./gradlew generateReport`.

[3] Recently, GHC introduced *protected file paths*, which flag changes in essential files, including tests, providing an out-of-the-box solution to detect tampering.

```
@Test
void checkNumberOfFieldsPerSignature() {
  Module world = getModule(Tasks.task_1);  //parse with Alloy APIs
  for (Sig s : world.getAllSigs()) {  //iterate over signatures
    assertTrue(s.getFields().size() >= 2,  //check min num. fields
      "Number of fields is less than 2 in signature " + s.label);
  }
}
```

Listing 1. Example of using Alloy API for checking one criterion of the free-response task Ex. 5, Task 1 from Fig. 2: "... signatures each with at least 2 fields"

Managing feedback report Automated feedback is pushed to a separate branch of each repository to avoid the task of merging changes from the primary branch. Fig. 3 presents an excerpt of the automated feedback report for Ex.7, Task 4 from Fig. 2. The report comprises three columns: test, status, and reason for failures. The first test checks if the knight starts at (0,1) (as required by the task) and passes. The next test assesses whether the knight can move to (2,2) in the next step, but it fails due to a runtime error (this could be due to parsing errors). Lastly, the third test fails because the knight should be able to reach (7,7) starting from (0,1), which is not correctly implemented.

4.3 Migration Challenges

We reused most tasks from our worksheets with minor adjustments. We now list some challenges for moving from worksheet-based to GHC exercises. We indicate for each challenge whether it relates to specification tasks (*spec*), implementation tasks (*impl*), or to provide meaningful feedback (*feedback*).

Free-response questions (challenges: *spec*, *feedback*) Our worksheets included many free-response questions, e.g., "Provide a satisfiable formula with at least 3 three variables and at least 3 different operators" or Ex. 5, Task 1 from Fig. 2, to evaluate higher-order thinking [18] and suppress plagiarism. To evaluate answers in GHC, we had to implement ad-hoc parsers or use APIs where available, e.g., for parsing and inspecting Alloy modules as in Listing 1. Both are not trivial and add overhead when setting exercises.

Problem encoding (challenges: *spec*, *impl*) Many tasks require coming up with suitable domain concept formalizations, e.g., PC component types in SMT (as in Fig. 2 (Ex. 3, Task 3)) can be encoded as integer constants or as datatypes (with their own pros/cons). We believe that this challenge/freedom is very important, but it makes an automated assessment of submissions challenging. We developed some minimal specification interfaces, e.g., variables for the budget and purpose of PCs, to construct and assess scenarios independent of the chosen encoding. The test case in Listing 2 only uses this specification interface (variables `purpose` and `budget`) to check a possible PC configuration. In general, finding a balance of what variables to provide is non-trivial and specific to each task.

```
@Test
void testPurposeOffice1(){
  assertTrue(Z3Utils.isSatForConstraints(code,  //check possible PC config
    "(assert (= purpose office))\n(assert (= budget 283))"),
    "For office purpose, 283 Euro is enough.");  //feedback if failed
}
```

Listing 2. Example of using a specification interface (variables `purpose` and `budget`) for checking a possible PC configuration of Ex. 3, Task 3 from Fig. 2

```
@Test
public void testTask1a() throws IOException, InterruptedException {
  String testSpec = code + "\nLTLSPEC\n!(F(knight[0]=7&knight[1]=7));";
  String res = NuxmvExecutor.runNuxmv(testSpec);  //run nuXmv with spec
  assertTrue(res.contains(  //check nuXmv output for expected result
    "!( F (knight[0] = 7 & knight[1] = 7))  is false"),
    "Starting at (0,1) the knight can reach (7,7)"); //feedback if failed
}
```

Listing 3. Example of verifying additional LTL specifications on possible knight moves to check the user-defined transition relation of Ex. 7, Task 4 from Fig. 2

Solutions in test cases (challenges: *spec*, *feedback*) One way to test a correct encoding of constraints is by checking semantic entailment or equivalence with a solution constraint, e.g., employed in Alloy4Fun [19]. To improve feedback, we decided to publish test cases for inspection to our students. A simple check of constraint equivalence was no longer possible as it would reveal the solution constraints (as in Fig. 2 (Ex. 3, Task 3)). Instead, we constructed multiple satisfiable and contradicting constraints (e.g., Listing 2 or Listing 3) for analysis and feedback generation.

Scenario encoding For effective analysis, we had to develop individual methods to encode and check scenarios (test cases of specifications) in SAT, Alloy, SMT, and nuXmv, e.g., to check transition relations of an SMV module (as in Fig. 2 (Ex. 7, Task 4)), our tests add LTLSPEC constraints and check these (see Listing 3). This requires creative extensions of specifications with constraints and the interpretation of the generated output of tools, as not all provide APIs.

Submission format (challenges: *spec*) A further challenge was to find a canonical format for collecting and submitting specification documents. Previously, students submitted PDFs, but specifications from these are not easily extracted. We used the interactive online editor Formal Methods Playground[4] (FMP). While separate, external storage of specifications makes the submission less self-contained, the students were used to FMP from lectures and links embedded in lecture slides. We also provided templates to get started, where necessary.

[4] Available from https://play.formal-methods.net

False positives (challenges: *impl*, *feedback*) Some test cases pass when no constraints are generated; thus, some scores might go down when providing correct but partial solutions. This risks confusion or discouragement of students. We handled these cases by adding explicit explanations on the worksheets.

Task dependencies (challenges: *impl*, *feedback*) Our implementation exercises were mini-projects where later tasks might depend on earlier ones, e.g., finding dead features in a Feature Model requires correct encoding of Feature Models. When grading manually, one could trace failures of later tasks to errors in earlier ones and award partial marks. In our setup, this was not possible. We indicated dependencies on the worksheets to make students aware of the grading limitation.

Order and number of test cases (challenges: *feedback*) Although not relevant from a correctness point of view, we suspect that the order of test cases has an impact on delivering feedback. Typically, students solve exercises incrementally, and we ordered test cases to provide early, positive feedback, e.g., assessing adequate numbers of signatures in an Alloy module before analyzing its predicates. Our implementation tasks already contained (limited) JUnit tests in the first iteration to confirm that students were going in the right direction. We significantly extended the number of test cases and assertions for feedback generation and correctness checking. Still, some students asked for covering additional edge cases (see Sect. 5).

Long-running test cases and exceptions (challenges: *feedback*) Runtime errors, memory leaks, and exceptions thrown during testing may fail or interrupt test cases. We provide feedback on the type of exception, but understanding why the exception occurred without analyzing the code is challenging. Heavy computation, like infinite loops, further complicates the feedback process. We addressed long-running test cases by introducing timeouts, and we later (from Ex. 4) provided all test cases for local analysis of exceptions.

Additional Challenges (challenges: *spec*, *impl*, *feedback*) (1) We were not able to automatically generate meaningful feedback for text-based questions, e.g., "Explain the provided nuXmv counter-example". (2) For limboole and nuXmv, where we did not have parsers nor APIs, our analyses depend on output of the tools and are thus fragile to changes, e.g., tool updates. (3) Some students exploited the limited number of test cases, e.g., the SMT encoding of PC components in one submission handles some edge cases of budget values by returning an empty result.

4.4 On Test Creation Efforts

Some of the challenges listed above similarly apply to general programming exercises regardless of the inclusion of Formal Methods, e.g., *False positives* or

Exercises	No. of Test Cases	No. of Assertions	Published Test Cases
1	31	56	0
2	23	52	4
3	21	48	0
4	9	15	9
5	31	30	31
6	15	26	15
7	10	33	10

Table 1. Number of test cases, assertions (different feedback texts), and test cases published to students per exercise; after exercise 3, all test cases were published

Task dependencies. However, others are clearly specific, e.g., *Problem encoding* or *Scenario encoding*.

One challenge we found during the later inspection of submissions is related to the intended use of FM tools in implementation exercises (Ex. 2, 4, and 6). One pair of students solved most analysis tasks by a translation to SMT; however, they did not find a suitable encoding for one edge case and handled it on the Java level (exploiting a limited number of test cases). While handling parts of the problem in Java and others in SMT might be elegant, this was not the intended learning outcome here. In general, our setup of the exercises would make it very difficult to assess whether solutions are computed in Java or SMT.

Table 1 summarizes the total number of test cases and assertions for each exercise and the number of test cases made available to students. Each test case offers feedback through a varied set of assertions (often, later assertions are not meaningful without earlier ones). The median number of test cases is 21, while the median number of assertions is 33.

For exercises 1 and 3, no test cases were provided to students. Similarly, exercise 2 only had 4 test cases out of 23 available to students. However, after reviewing responses in student surveys, we provided all the test cases for exercises 4 to 7 to students. This potentially reduced their difficulties and helped them work better, as reflected in Figures 5 and 7. Obviously, without providing test cases, we would have avoided challenge *Solutions in test cases*. However, we believe that the more transparent feedback outweighs the necessary adjustments.

One concern regarding the up-front availability of all test cases is that students could tweak their specifications and code just enough to make all test cases pass rather than develop complete solutions to the given tasks. An alternative is to split the test cases into two sets, one for feedback and one for marking. We leave a deeper evaluation of this to future work.

We notice that the preparation of exercises for GHC involves a significant investment of resources. Designing appropriate tasks and worksheets to cover taught materials and intended learning outcomes remained the same with or without GHC. Designing reasonable feedback and grading through test cases can be very challenging and difficult to get right, as all possible reasonable solution attempts by students should be taken into account. For us, this investment was paid back by reducing the time to inspect and grade homework submissions

manually (only necessary in edge cases). However, low-quality or inadequate test cases could diminish this return on investment.

Finally, collusion and students passing on solutions over the years harm the reuse of the exercises for future iterations. We tried to mitigate this by including free-response questions and individually generated tasks in the specification exercises. For the implementation exercises, we employed plagiarism-checking tools.

4.5 Benefits and alternatives to GHC

As discussed in Sect. 4.1 we did not use all features of GHC. Still, for our case, it provided several benefits: automated replication of task repositories for students, automated execution of test cases and report generation, user management of student accounts, and available infrastructure. However, our exercise materials are not tied to GHC. The setup may be replicated on any other infrastructure that provides repositories and continuous integration.

5 Surveys and Evaluation

Over the past two years (WS22-23 and WS23-24), we have been conducting the Formal Methods for Software Engineering module. In the first iteration, a total of 86 students enrolled (24 groups submitted the first exercise), while in the second iteration, 44 students enrolled (11 groups submitted the first exercise)[5]. Figure 4 illustrates the number of groups per exercise. We observed that almost half of the groups dropped out in both semesters over all the exercises submitted. In WS22-23, submissions went from 24 groups to only 13 groups for the last exercise. Similarly, in WS23-24, we received 11 submissions for the first exercise but only 5 for the last exercise.

5.1 Surveys in WS23-24

We now report data from voluntary paper-based surveys conducted among students on the days after the submission of exercises in WS23-24. We conducted these surveys to gain insight into the effectiveness of implementing GitHub Classroom with automated feedback. The surveys were designed to evaluate the ease of using GitHub for submitting exercises, the quality of automated feedback, and whether it contributed to a perceived overall improvement of student work or additional workload. Additionally, we gathered suggestions from students to refine the submission process and enhance the quality of the exercises.

Figure 5 illustrates the declared level of difficulty experienced by students while using GitHub to work on and submit exercises. Notably, most students, including those from the Digital Engineering program (mixed background from

[5] Enrolling is an informal act for obtaining access to teaching materials and the numbers of initial submissions are more reliable indicators for participation.

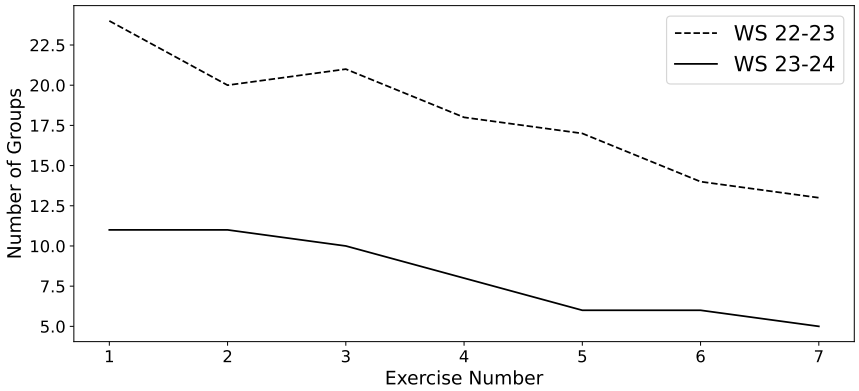

Fig. 4. Submission per exercise for the Winter semesters of 2022-23 and 2023-24

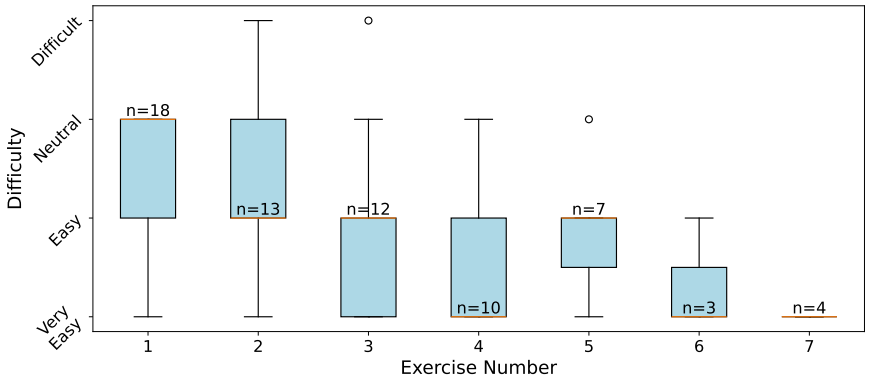

Fig. 5. The difficulty of using GitHub for completing the exercises

engineering disciplines), found using GitHub to be easy. Additionally, we observed that the students grew more comfortable with the platform over time. In the initial two exercises, the median difficulty level reported by students was "neutral" to "easy", whereas for the subsequent exercises, the median level ranged from "easy" to "very easy". It should be highlighted that the number of students participating in the exercises and surveys decreased over time.

Figure 6 shows the perceived quality of the automated feedback for the exercises. At first, we relied solely on the feedback given by GitHub Classroom through the GitHub Actions logs for the initial exercise. Unfortunately, but not surprisingly, the students found interpreting and comprehending the feedback challenging. As a result, we began automatically extracting feedback from test reports and aggregating it in tabular format starting from exercise 3. We observed that the quality of the feedback remained "good" for the first five exercises and even improved for the final two.

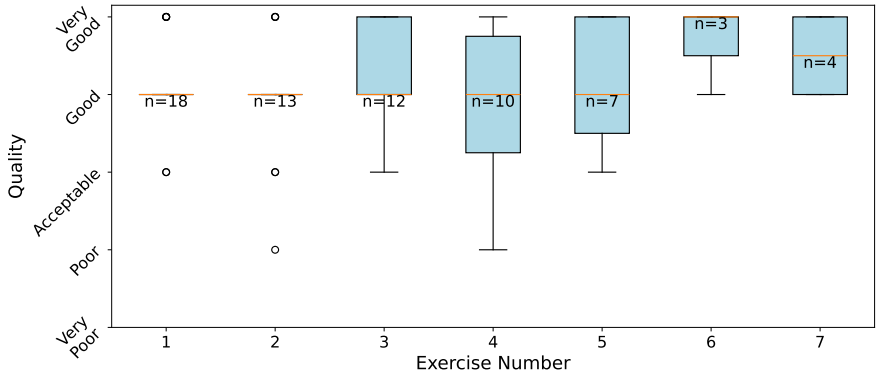

Fig. 6. The quality of the automated feedback for the exercises

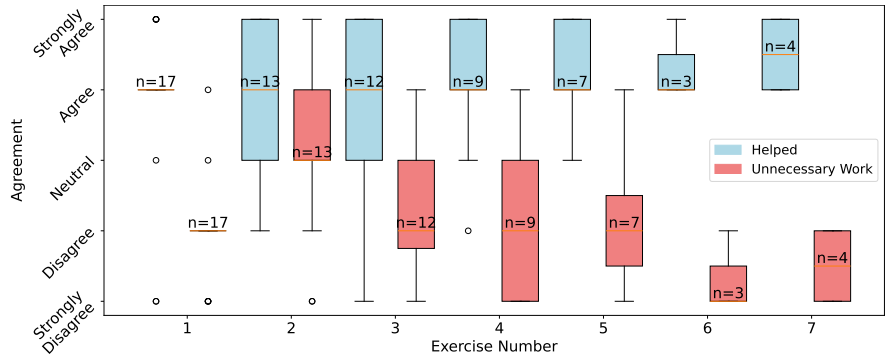

Fig. 7. Agreement on automated feedback being helpful (blue) or unnecessarily causing more work (red)

The data presented in Fig. 7 (blue) illustrates how students agree with the usefulness of automated feedback in improving their work. At the same time, Fig. 7 (red) highlights their level of agreement on whether generated feedback caused a higher workload. The majority of students believe that automated feedback has been beneficial in enhancing their work across all exercises, as reflected by the median. However, there were some instances of disagreement among students in exercises 2 and 3 (test cases not available to students), with 75% remaining neutral or strongly agreeing. Nonetheless, automated feedback was found to be helpful in improving their work in subsequent exercises. In terms of additional workload, students generally disagreed that feedback added to their workload, except for exercise 2, where the median was different.

In addition to the Likert scale questions above, our survey also includes open-ended prompts asking about challenges and suggestions. We analyzed the textual responses and grouped them into two categories: automated feedback and GH/GHC. These categories align with the themes presented in Figures 5-7.

Automated Feedback Students expressed appreciation for automated feedback, citing benefits like improved work quality, bug resolution, and better understanding of different scenarios:

- *The automated feedback was very important to evaluate in our cases it was very nice to know what improvements can be made in the code further.*
- *The feedback really helps with the process of completing and understanding the tasks. If a problem is encountered, the feedback helps in identifying the topic of concept that needs to be revised for completion.*
- *Continuous feedback on each statement helped me compare and understand the assignment better.*

However, they also identified areas for improvement, such as wanting more detailed feedback and the need to check all corner cases:

- *The automated tests didn't test for multiple components of the some category, which should not be possible.*
- *Provide more information on why the test case has failed and also the exact errors.*
- *Maybe include test cases or in this case the LTLSPECS in the playground template for easy access.*

GH/GHC Overall, students had a positive experience using GitHub and GitHub Classroom despite facing some challenges including group submissions and not having all the test cases locally (communicated verbally in class), which were provided later for exercises 4-7:

- *The assignment submissions on GitHub really helped me know what is going wrong with any work looking at the feedback.*
- *There can be better infrastructure assignments for group submissions of the assignments.*

5.2 Threats to Validity

Likert scales are well-known to be subject to various forms of response bias [17], thus, one has to be careful with conclusions solely based on these. Another potential bias in our data is the decline in the number of answers (n in Figures 5-7). While all figures show a positive trend, we cannot make conclusions about whether the quality of worksheets improved, the students became more experienced with GHC or the feedback, or whether simply the more experienced or confident students remained participating.

6 Conclusion

We have presented a brief overview of our module, Formal Methods for Software Engineering, and its two exercise types (specification and implementation). We

identified challenges in migrating from traditional submissions to GitHub Classroom (GHC) with automated grading and feedback report creation. Student surveys indicate low difficulty of using GHC, good quality and helpfulness of automated feedback, and low overhead for students.

References

1. Ábrahám, E., Nalbach, J., Promies, V.: Automated exercise generation for satisfiability checking. In: FMTea 2023. LNCS, vol. 13962, pp. 1–16. Springer (2023). https://doi.org/10.1007/978-3-031-27534-0_1
2. Angulo, M.A., Aktunc, O.: Using GitHub as a Teaching Tool for Programming Courses. In: 2018 Gulf Southwest Section Conference Proceedings. p. 31594. ASEE Conferences. https://doi.org/10.18260/1-2-370.620-31594, http://peer.asee.org/31594
3. Baier, D., Beyer, D., Friedberger, K.: JavaSMT 3: Interacting with SMT solvers in java. In: CAV, 2021, Proceedings, Part II. LNCS, vol. 12760, pp. 195–208. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_9
4. Biere, A.: Limboole. https://fmv.jku.at/limboole/ (2012), accessed 04/2024
5. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: CAV. LNCS, vol. 8559, pp. 334–342. Springer (2014)
6. Davies, J., Simpson, A., Martin, A.P.: Teaching formal methods in context. In: TFM 2004, Proceedings. LNCS, vol. 3294, pp. 185–202. Springer (2004). https://doi.org/10.1007/978-3-540-30472-2_12
7. Divasón, J., Romero, A.: Using krakatoa for teaching formal verification of java programs. In: FMTea 2019. LNCS, vol. 11758, pp. 37–51. Springer (2019). https://doi.org/10.1007/978-3-030-32441-4_3
8. Dubois, C., Prevosto, V., Burel, G.: Teaching formal methods to future engineers. In: FMTea 2019. LNCS, vol. 11758, pp. 69–80. Springer (2019). https://doi.org/10.1007/978-3-030-32441-4_5
9. Glassey, R.: Adopting Git/Github within Teaching: A Survey of Tool Support. In: CompEd '19. pp. 143–149. ACM. https://doi.org/10.1145/3300115.3309518
10. Haaranen, L., Lehtinen, T.: Teaching Git on the Side: Version Control System as a Course Platform. In: ITICSE 2015. pp. 87–92. ACM. https://doi.org/10.1145/2729094.2742608
11. Haldeman, G., Babeş-Vroman, M., Tjang, A., Nguyen, T.D.: CSF: Formative Feedback in Autograding **21**(3), 1–30. https://doi.org/10.1145/3445983
12. Haldeman, G., Tjang, A., Babeş-Vroman, M., Bartos, S., Shah, J., Yucht, D., Nguyen, T.D.: Providing Meaningful Feedback for Autograding of Programming Assignments. In: SIGCSE '18. pp. 278–283. ACM. https://doi.org/10.1145/3159450.3159502
13. Hollingsworth, J.: Automatic graders for programming classes. Commun. ACM **3**(10), 528–529 (1960). https://doi.org/10.1145/367415.367422
14. Hsing, C., Gennarelli, V.: Using GitHub in the Classroom Predicts Student Learning Outcomes and Classroom Experiences: Findings from a Survey of Students and Teachers. In: SIGCSE '19. pp. 672–678. SIGCSE '19, ACM. https://doi.org/10.1145/3287324.3287460
15. Jackson, D.: Alloy: a language and tool for exploring software designs. Commun. ACM **62**(9), 66–76 (2019). https://doi.org/10.1145/3338843, https://doi.org/10.1145/3338843

16. Kertész, C.Z.: Using GitHub in the classroom - a collaborative learning experience. In: SIITME 2015. pp. 381–386. https://doi.org/10.1109/SIITME.2015.7342358
17. Liu, M., Harbaugh, A.G., Harring, J.R., Hancock, G.R.: The effect of extreme response and non-extreme response styles on testing measurement invariance. Frontiers in psychology **8**, 227387 (2017)
18. Luxton-Reilly, A., Denny, P., Plimmer, B., Bertinshaw, D.J.: Supporting student-generated free-response questions. In: ITiCSE 2011. pp. 153–157. ACM (2011). https://doi.org/10.1145/1999747.1999792
19. Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A.C.R., Ramalho, M.S., Silva, D.C.: Experiences on teaching alloy with an automated assessment platform. Sci. Comput. Program. **211**, 102690 (2021). https://doi.org/10.1016/J.SCICO.2021.102690
20. Marques-Silva, J., Malik, S.: Propositional SAT solving. In: Handbook of Model Checking, pp. 247–275. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_9
21. Mendonça, M., Wasowski, A., Czarnecki, K.: Sat-based analysis of feature models is easy. In: Muthig, D., McGregor, J.D. (eds.) SPLC 2009. ACM International Conference Proceeding Series, vol. 446, pp. 231–240. ACM (2009), https://dl.acm.org/citation.cfm?id=1753267
22. Messer, M., Brown, N.C.C., Kölling, M., Shi, M.: Automated grading and feedback tools for programming education: A systematic review. ACM Trans. Comput. Educ. **24**(1), 10:1–10:43 (2024). https://doi.org/10.1145/3636515
23. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
24. Pelletier, F.J.: Seventy-five problems for testing automatic theorem provers. J. Autom. Reason. **2**(2), 191–216 (1986). https://doi.org/10.1007/BF02432151
25. Pnueli, A.: The temporal logic of programs. In: SFCS 1977. pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
26. Rozier, K.Y.: On teaching applied formal methods in aerospace engineering. In: FMTea 2019. LNCS, vol. 11758, pp. 111–131. Springer (2019). https://doi.org/10.1007/978-3-030-32441-4_8
27. Sarsa, S., Denny, P., Hellas, A., Leinonen, J.: Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In: ICER '22 - Volume 1. pp. 27–43. ACM. https://doi.org/10.1145/3501385.3543957
28. Shute, V.J.: Focus on formative feedback. Review of educational research **78**(1), 153–189 (2008)
29. Sovietov, P.: Automatic Generation of Programming Exercises. In: TELE 2021. pp. 111–114. IEEE. https://doi.org/10.1109/TELE52840.2021.9482762
30. Tiam-Lee, T.J.Z., Sumi, K.: Procedural generation of programming exercises with guides based on the student's emotion. In: SMC 2018. pp. 1465–1470. IEEE (2018). https://doi.org/10.1109/SMC.2018.00255
31. Tscherter, V.: Exorciser: Automatic generation and interactive grading of structured excercises in the theory of computation. Ph.D. thesis, ETH Zurich, Zürich, Switzerland (2004). https://doi.org/10.3929/ETHZ-A-004830877
32. Tu, Y.C., Terragni, V., Tempero, E., Shakil, A., Meads, A., Giacaman, N., Fowler, A., Blincoe, K.: GitHub in the Classroom: Lessons Learnt. In: ACE 2022. pp. 163–172. ACM. https://doi.org/10.1145/3511861.3511879