Teaching Correctness-by-Construction and Post-hoc Verification – The Online Experience

Tobias Runge[1(✉)], Tabea Bordis[1(✉)], Thomas Thüm[2(✉)], and Ina Schaefer[1(✉)]

[1] TU Braunschweig, Brunswick, Germany
{tobias.runge,t.bordis,i.schaefer}@tu-bs.de
[2] University of Ulm, Ulm, Germany
thomas.thuem@uni-ulm.de

Abstract. Correctness of software is an important concern in many safety-critical areas like aviation and the automotive industry. In order to have skilled developers, teaching formal methods is crucial. In our software quality course, we teach students two techniques for correct software development, post-hoc verification and correctness-by-construction. Due to Covid, the last course was held online. We present our lessons learned of adapting the course to an online format on the basis of two user studies; one user study held in person in 2019 and one online user study held after the online course. For good online teaching, we suggest the use of accessible (web-)tools for active participation of the students to compensate the advantages of teaching in person.

1 Introduction

Development of correct software is a concern which is becoming increasingly important. In areas like aviation and the automotive industry, where human lives depend on it, software safety requirements are considerably higher. Therefore, formal methods should be taught to young software developers, so that they learn a reasonable approach to develop correct programs, instead of hacking programs into correctness. With this skill of correct software development, developers can avert major errors in software projects by specifying and verifying the safety-critical parts [35]. Additionally, a development process that includes verification can reduce overall development time because most software is correct from the start, which reduces maintenance time and effort. Besides the prevalent post-hoc verification (PhV) approach, where software is verified after implementation, correctness-by-construction (CbC) [23] is an approach where software is incrementally refined from a specification. In CbC, each refinement step guarantees the correctness of the whole programs. CbC expands the repertoire of programmers with a formal reasoning style that prevents errors in the first place.

Teaching formal methods, the correct specification and verification of programs, is the topic of the Master course Software Quality 2 at TU Braunschweig,

J. F. Ferreira et al. (Eds.): FMTea 2021, LNCS 13122, pp. 101–116, 2021.
https://doi.org/10.1007/978-3-030-91550-6_8

Germany. In the course, students learn the basics of deductive software verification and correctness-by-construction on the practical example of specifying and verifying Java code. In the corresponding exercises, the students solve tasks using corresponding tools. In contrast to previous years, we offered the course of last term online (due to the pandemic).

A difficulty in teaching formal methods is that courses are based on a lot of formal background which may discourage students. Therefore, we highlight the benefit to integrate practical experiences of practical tool usage in teaching that help students to consolidate the taught topics [17]. With tools, students receive immediately feedback if their found solutions are correct. They can also work on larger tasks that are not doable on pen-and-paper. A problem here is the effort to install various tools on different machines, especially when students use their own machines due to online teaching. Tools should be easy to install or web-based such that students enjoy active participation in lectures and exercises of formal method courses. Some good examples for tool support are KeY [3], Whiley [30], and Dafny [25].

Besides an experience report on our online course, we evaluate the learning success of the students with two user studies. We compare the results of an online user study with an earlier user study conducted in the same course, but in person [33]. In the qualitative user study, we evaluate how students solved tasks with two verification techniques post-hoc verification and correctness-by-construction. They used the tools KeY [3] (as instance of a PhV tool) and (Web)CorC [32] (as instance of a CbC tool). Therefore, our user study has four quadrants. We compare CbC with PhV and online with in person courses. With the data from these two studies, we share our lessons learned in transferring the course to an online format, we discuss the quality of the online course, and point out challenges and opportunities for improving online courses in the future. We also compare the web version WebCorC with the previous version CorC to determine important aspects of good tool support. Furthermore, we compare how well students interact with CbC and PhV by collecting their user feedback.

As a result, we confirmed the findings of the first user study. The students made fewer defects in the code with PhV than with CbC, but overall the results are worse than in the first user study. This indicates a worse learning outcome due to the online format. The qualitative questionnaire was answered in favor of CbC. The students liked the structured reasoning of CbC and rated the support provided by the CorC tool as more suitable for finding defects than KeY for PhV. For online teaching, we detect that easily accessible tool support is beneficial for participation in exercises. Additionally, courses on formal methods should be interactive to encourage student participation, thus we discuss how to improve online teaching.

2 Related Work

Teaching formal methods has been discussed by many researchers [11,14,17, 26]. They discuss their teaching experiences and evaluate the learning success

of students with respect to different tools and teaching strategies. In detail, Liu et al. [26] highlight the benefits of a good mix of pen-and-paper and tool supported exercises. On paper, students consolidate what they learned without being supported by a tool, and with tools they increase their productivity and learn to analyze defects in the specifications or programs. The interest of students also rises if they can solve exercises on tools and get positive feedback by verifying programs. Creuse et al. [14] mention that teaching by example is beneficial for an easier and more practical entry into formal methods. That the students demand immediate and good feedback on their specification or verification process and want to understand clearly occurring problems is identified by Catano [11]. We differ from this related work [11,14,17,26] that does not examine the aspects of online teaching. In this paper, we discuss new challenges regarding online teaching by comparing our course with a previous course held in person.

With respect to the user study, we compare it with related work that evaluated the usage of verification tools. Petiot et al. [31] evaluated the interaction of programmers with verification tools. The authors analyzed how programmers can be supported when they encounter an open proof goal. To improve user feedback, they categorize defects and calculate counter examples. In the work of Johnson et al. [22], developers were interviewed about their usage of static analysis tools. They also recognized that developers need good error reporting. The influence of formal methods in code reviews was studied by Hentschel et al. [20]. In their study, the symbolic execution debugger (SED) had a positive impact on the location of defect in programs. KeY was also evaluated to analyze how participants interact with the tool during the verification process [9,10]. Back [5] evaluated in an experiment that good tool support is necessary to develop correct software with a refinement based approach. Additionally, he discovered an iterative procedure to refine an incomplete or partially incorrect invariant to a final solution, an insight that we confirm in our first user study [33]. In our user study, we focus more on the usability of the tools during program constructions, how programmers utilize the tools and adapt their programming procedure. In our study, we used KeY as automatic verifier. Thus, we excluded the expertise on interactive proving from our study. We focused on the development of correct programs guided by a specification. In our study, the participants have to implement the programs by themselves.

3 Teaching Formal Methods – Software Quality 2

In this section, we describe the structure of the formal methods course Software Quality 2 at TU Braunschweig. We compare the previous course held in person with the current course in an online format. The goal of this course is to teach students deductive post-hoc verification and correctness-by-construction such that students are able to construct correct programs with these approaches. The course is named Software Quality 2, as we also offer a course Software Quality 1 that focuses on testing software.

In Person Course. The course in person is divided into two parts, 12 lectures with 11 corresponding exercises, each one lasting 90 min. The students attending this course are mostly Master students that had at least two courses in programming, and three courses in theoretical computer science. Normally, between 20 and 50 students attend the course. The lectures are a presentation on topics like design by contract, software model checking, sequent calculus, deductive verification, specifying programs with JML, verifying programs with KeY, and constructing correct programs with CbC. The presentations include some intermediate questions to the audience (e.g., to complete examples) and questions in the end to consolidate the lessons learned. We provide a video of each lecture, so students can prepare the exams by rewatching specific lessons. In the videos, the slides and the audio of the lecture are recorded.

The exercises are divided between pen-and-paper exercises for writing first-order logic and using the sequent calculus, and tool-supported exercises. For the tool-supported exercises, we use different tools: OpenJML [13] to show testing with JML annotated code, Java Pathfinder [19] for software model checking, KeY [3] for program verification, and CorC [32] for correct-by-construction software development. All these tools are pre-installed on machines at the university such that exercises are performed smoothly. The exercises are mostly interactive such that the students solve the tasks and present the solutions, and these solutions are discussed with the audience. This structure of the exercises should consolidate knowledge better than a frontal presentation of solutions.

The course exam is oral. We have a small group of students such that oral exams are feasible. In these exams, we can check whether students understood the topics of this course and can answer cross-cutting questions, and whether they can apply learned techniques to solve code specification and verification tasks. Oral exams are more time-consuming than written exams, but as a teacher one gets better feedback on whether students have understood the content.

Online Course. The setting for the online course is the same as for the previous courses in person: 12 lectures with 11 exercises with weekly meetings covering the same topics. The number of students slightly increased to around 60 students. We upload videos of the recorded lectures of the previous year. Additionally, we give a short recap of the topic followed by a discussion where students can ask questions in the weekly meeting. For the exercises, we upload exercise sheets with tasks that have to be prepared as homework. If the task includes the use of tools, we add an instruction for the installation and usage. In the weekly online session for the exercises, the students are asked to present their solutions which are discussed with the audience afterwards. Thereby, we use Google Docs documents that can be edited by everyone in the session to collect and store the correct answers for exam preparation.

To keep the interactive character of the exercise in the online course, we use the same tools in the exercises as we do for the course in person (i.e. OpenJML, Java Pathfinder, KeY, and CorC). Due to the format of an online session, we could not monitor whether students were actually actively participating. For the tools, we tried to find the easiest way with as few steps as possible for

the installation. However, with most of the tools we had some problems with the installation due to outdated documentations or the need of specific JDK or Eclipse versions such that finding a good solution was time-consuming.

The oral exams are held online in the video conference system provided by our university. The student has to attend with camera such that we can check that the right person is taking the exam and that no other persons are in their room. An advantage of taking the oral exam online is that we are able to include practical tasks using tools introduced in the exercises. To omit difficulties in the installation, we installed the tools on our computer and shared the screen. The students then have to explain what they would do and what result they expect.

4 Verification Techniques and Tool Support

We compare in our user study, how students solve tasks using post-hoc verification (PhV) and correctness-by-construction (CbC). We briefly introduce PhV and CbC in the following.

4.1 Post-hoc Verification

The post-hoc verification process [3], verifies the correctness of programs after the implementation. A prover checks that the implementation complies with the pre-/postcondition specification. PhV does not give a development guideline, such that programmers can freely implement the programs as long as the specification is in the end fulfilled. This free process decreases the time to construct a first (potentially faulty) version of a program, but can increase the time to construct a verified version, as it is more likely that defects occur in the code [35].

As an instance, KeY [3] verifies the correctness of Java programs that are specified with the Java modelling language (JML). Starting from a specified program, KeY symbolically executes the programs and closes the remaining proof obligations (semi-)automatically. As we are focusing on the programming and specification aspects in our user study, we use KeY as an automatic tool. This goes along with most programmers not having a theoretical background to verify programs interactively.

Besides KeY, there are a number of tools in the area of specification and program verification: the language Eiffel [27] with the verifier AutoProof [34], the languages SPARK [8], Dafny [25], and Whiley [30], and the tools OpenJML [13], Frama-C [15], VCC [12], VeriFast [21], and VerCors [4]. All languages and tools are candidates to be compared with the CbC methodology, but we decided for KeY because of the previous familiarity of our study participants. Since we used only a subset of the Java language without method calls or custom objects, the difference to other programming languages is minimal.

4.2 Correctness-by-Construction

Correctness-by-construction [16,23,28] is a methodology to incrementally construct correct programs. Starting with a pre-/postcondition specification and an

initially abstract program, refinement rules are applied to create an implementation that fulfills the specification. The correctness is guaranteed by the rules if specific side conditions for their applicability hold. Dijsktra [16] and Kourie and Watson [23] identified that the CbC process guides programmers to a correct implementation that has low defects rates and is of better structure than a program ad hoc hacked to correctness. A disadvantage of CbC is the fine-grained refinement process that programmers must adhere to. This complicates program construction for inexperienced programmers, but the fine-grained development with the explicit specification in each node raises awareness for defects in the mind of the programmer [35].

Besides the CbC approach proposed by Kourie and Watson [23], there are other refinement based approaches that guarantee the correctness of the program under development. In the Event-B framework [1], specified automata-based system are refined to concrete implementations. It is implemented in the Rodin platform [2]. In comparison to the CbC approach used here, the abstraction level is different. CbC uses specified source code instead of automata as main artifact. Morgan [28] and Back [7] proposed also related CbC approaches. Morgan's refinement calculus is implemented in the tool ArcAngel [29]. Back et al. [5,6] developed the tool SOCOS. In comparison to CbC, SOCOS starts with invariants additionally to a pre-/postcondition specification.

The tool CorC [32] implements the CbC process in a graphical and textual editor. CorC supports developers by offering refinement rules as proposed by Kourie and Watson [23] to develop programs and checking the correctness of each applied refinement with a program verifier KeY [3]. In CorC, a programmer builds stepwise a correct method by getting feedback directly when one refinement is not correct, for example, when the programmer specifies an invariant that is not satisfied at the beginning of the loop.

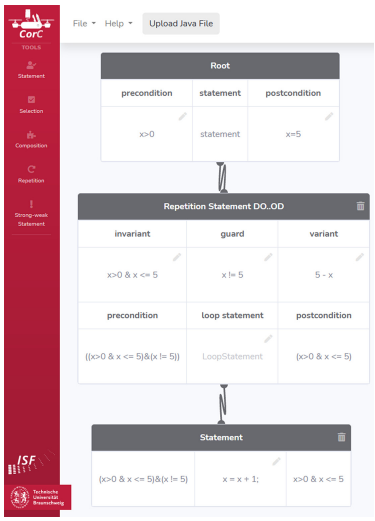

Fig. 1. Program construction in WebCorC

WebCorC[1] is an adaption of CorC [32]. Similar to CorC, we decided for the graphical editor in WebCorC because of the student feedback collected during the Software Quality courses. The graphical editor helps students learn the CbC approach by visualizing all important aspects of specifying and refining a program into a correct result. CorC is implemented in Java in the Eclipse framework. In WebCorC, we transferred the graphical editor using a client-server structure, reusing the logic of CorC on server side, but redeveloping the graphical editor as web-frontend. In comparison to CorC, the implementation of WebCorC has no detailed feedback in a console

[1] https://www.isf.cs.tu-bs.de/WebCorC/.

when a refinement cannot be proven. This feedback was added only after the user study.

In Fig. 1, we show a program construction in WebCorC. At the top, we specify in the first gray node the program under development. The precondition states that an integer `x` is greater than zero. In the postcondition, `x` should be equal to 5. We solve this problem by introducing a loop statement in the first refinement step, called repetition in WebCorC. We introduce a repetition statement for illustration purposes. Of course, an assignment directly solves the problem. For the repetition, we need additional specification: a loop guard, a loop invariant, and a variant. We continue the loop as long as `x` is not equal to 5. The loop body introduced in the next refinement step, the third node at the bottom, increments `x` by one. Both refinement steps are checked by WebCorC to be correct.

5 User Study Design

In this section, we describe the design of our user study that was conducted online after the end of the Software Quality 2 course. The goal of this evaluation is to compare the results of the students with the results of a previous study that was held in person. Therefore, we adopt the design of the previous study [33]. We want to get insights into the learning success of the students whether the online course and the use of WebCorC leads to noticeable differences in the outcome. To better compare both studies, we explain the commonalities and differences in the user study design in the following. Note that we used CorC in the first user study and WebCorC in the second user study. If we talk about both tools, we write (Web)CorC.

5.1 General User Study Design

The user study is designed such that students solve two programming tasks each with a different tool. We compare correctness-by-construction and post-hoc verification with the tools (Web)CorC and KeY. Starting with a pre-/postcondition specification, an algorithm should be implemented and verified.

Objective. We want to evaluate whether the CbC approach has a positive or a negative impact on the programming results. We consider the following two research questions to evaluate the verification approaches and tools.

RQ1: What kind of errors do participants make with CbC and PhV?
RQ2: To which degree do participants prefer CbC over PhV?

To evaluate the usability of CbC and PhV, we take the user experience questionnaire (UEQ), and ask the questions OQ1−OQ8:

OQ1: How do you rate your overall work with WebCorC from 1 (very bad) to 5 (very good)?
OQ2: What is your general process when solving tasks with WebCorC/KeY?

OQ3: Do you prefer a web-frontend over the Eclipse environment and why?
OQ4: Were there any specific obstacles during the task execution process?
OQ5: Is the construction of a program by modeling through a refinement structure helpful and why?
OQ6: Do you prefer WebCorC or KeY in general and why?
OQ7: Which of these two tools would you use for verification and why?
OQ8: Which tool better supports avoiding or fixing defects and why?

The UEQ [24] is a standardized test to measure six usability properties of a tool. A participant is asked to rate the tool with 26 items. Each item is a pair of adjectives that describe the tool, one negative and one positive adjective. The user can rate on a 7-point Likert-scale which of the adjectives and to what extent fits more. The range for the answers is between $+3/-3$.

General Design Decisions. The user study is limited to 90 min. To compare both tools, each participant should implement an algorithm with each tool. We set 30 min per task. Thus, the algorithms should be implementable and verifiable in this time frame. We decide for algorithms with a size of under ten lines, but including a loop to have participants writing loop invariants. We give the pre-/postcondition specification of the algorithms so that all participants have the same starting point. This reduces the divergence and lead to comparable programming results. Writing the pre-/postcondition would also cost too much time in this experiment. We decide for the tools (Web)CorC and KeY because the participants have experience with these tools which increases the expressiveness of this study. The material of the user study is published on GitHub.[2]

Tasks. Two algorithms must be correctly implemented and verified. The algorithm `minimum element` calculates the index of the minimum element in an array. The array is non-empty to omit the special case from the algorithm. The algorithm `modulo` calculates the remainder from two input integers; a dividend a and a divisor b. In the algorithm, division and modulo operators are prohibited. The algorithms are similar in size and cyclomatic complexity.

We design the tasks to be small enough to be doable in the time frame. We also design them such that both (Web)CorC and KeY can be used to implement them correctly. For both tasks, assignments, conditional statements and loops where sufficient.

The groups of participants are arranged with the Latin square design. Group A uses (Web)CorC for the first task, and PhV afterwards. Group B does the same tasks but the tools in different order. The tools are switched to address the possibility of learning effects by forcing a specific tool order.

Participants. The participants are students at TU Braunschweig, Germany attending the Software Quality course. These students were taught the fundamentals of software verification, and they learned to use the tools (Web)CorC and KeY. During the course they implement, specify, and verify methods with both tools. We analyzed the programming experience of the participants with a questionnaire [18]. To weaken the restraint against a group of students, we had

[2] https://github.com/Runge93/UserstudyCbCPhV.

several students with two to five years of programming experience in industry. Therefore, the participants can be compared with junior developers. The students freely attended the user study. We told them the goal of this experiment, and we offered a monetary payment for attendance. We raffled two times € 25.

Variables. We have the tools as independent variable in our user study, with the treatments CbC and PhV. The correctness of the task were checked with KeY using the automatic mode. For CbC, we checked the task by reverifying each refinement step. When some task was not verifiable, we manually checked for defects in the program or specification. These defects were counted by line.

5.2 Differences in the First and Second User Study

In the first user study, we have 10 participants in two groups who have no significant difference in the programming experience [33]. In the second user studies, we have 13 participants. With a programming experience questionnaire [18], we measure a similar experience in both groups. A value of 2.137 for group A and 2.550 for group B[3]. With a Mann-Whitney test, no significant difference between the two groups is measured.

In the first user study, we prepare machines at the university with CorC and KeY. They directly implement both tasks in the Eclipse IDE. Here, they can interact with KeY directly to get feedback about the verification status. In the second user study, we prepare a workspace where the participants can develop one of the algorithms with WebCorC. For the PhV process, they can use their preferred IDE. When they want to verify the algorithm, they upload it and get feedback about the success of the verification. This process can be repeated until a verified result is achieved. As the participants in the second study only upload files in the post-hoc verification tasks and do not interact with KeY directly, we abandon the UEQ for the tool KeY.

In the first user study, we monitored all participants in a controlled experiment in person. In the second user study, we adapt this by having an audio conference. Due to legal restrictions, we cannot use proctoring tools.

6 Results and Discussion

In this section, we show the results of our user study. We evaluate the implementations of each participant by looking at the final result. We focus on a qualitative evaluation of the programming procedure and results of CbC and PhV. Additionally, we evaluate the answers of our questionnaire (UEQ and OQ1−OQ8) to complete the discussion.

6.1 Defects in Implementation and Specification

To answer the first research question, we analyze defects in the code and the specification. By code, we refer to the implemented algorithm without the specification. By specification, we refer to auxiliary annotations such as loop invariants.

[3] The calculation is explained in the work by Feigenspan et al. [18].

Table 1. Defects in code and specification of the final programs of participants

#Defects	PhV 1st		CbC 1st		PhV 2nd		CbC 2nd	
	Code	Spec.	Code	Spec.	Code	Spec.	Code	Spec.
No defects	8	2	4	3	9	1	2	1
Minor defects	1	7	3	4	4	10	4	5
Major defects	1	0	1	0	0	0	0	0
Incomplete	0	1	2	3	0	2	7	7

We classify a program to have major defects, if the program cannot be corrected without rewriting the algorithm. Otherwise, we classify it to have minor defects. The same classification applies for defects in the specification.

Table 1 shows the defects the participants have in their final result when they finished a task in the first and second user study. When we compare CbC and PhV, the participants have fewer coding defects with PhV than with CbC in both user studies. The coding results for PhV and CbC are generally better in the first user study, we have more results without defects for both approaches. Across all participants, a typical defect is a loop guard with a wrong logical comparison operator. With CbC, a recurring problem is that participants forget to initialize variables correctly. In both studies participants have incomplete results in the program for CbC. In the second study, seven participants have not completed the program for CbC.

When we compare defects in the auxiliary specification, more participants have no defects with CbC in the first user study compared with the results for PhV. In the second user study, for each approach only one participant has no defects. In general, the specification results are better in the first user study. More participants have no defects with PhV and CbC. Typical defects with PhV are a missing variant or missing checks whether variables stay in a specific boundary (e.g., out of bounds checks in arrays). With CbC, a common specification defect is that the invariant does not hold initially or after the last loop iteration. However, the participants do not forget the variant when they specify a loop. In both user studies and with both approaches, we have incomplete specifications. Five incomplete results of the auxiliary specification are due to incomplete code.

6.2 User Experience

For the evaluation of the user experience, we show the results of the user experience questionnaire in Fig. 2. The blue results for CorC are from the first user study [33], the red results for WebCorC are from the second study. The answers of the participants are combined into six measurements: *attractiveness*, *perspicuity*, *efficiency*, *dependability*, *stimulation*, and *novelty*. Except for *efficiency*, the results are better in the first user study. The largest differences are in the scales of *stimulation* and *dependability*. The *stimulation* is rated lower because some participants rate WebCorC *demotivating*. Participants also rate WebCorC as

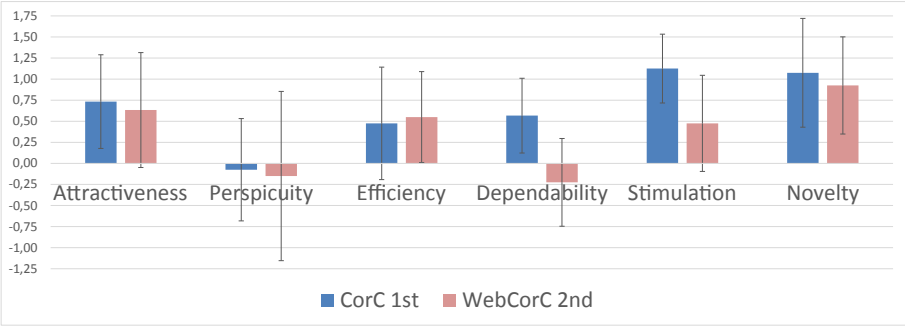

Fig. 2. Results of the user experience questionnaire

unpredictable which results in a negative score for *dependability*. The items *easy to learn/difficult to learn* and *complicated/easy* are answered differently resulting in a big variance for the *perspicuity* measurement in the second user study.

For question OQ1-OQ8, common answers of the participants are summarized in Table 2. Some participants dislike the limited feedback of WebCorC in comparison to CorC, but they prefer the web-frontend due to the easy accessibility. The general process of solving tasks is split between writing specification or the program first. When comparing WebCorC with KeY, the majority of participants prefer WebCorC to solve verification tasks because of the structured process. The participants in favor of KeY argue that they are more familiar with textual programming.

6.3 Discussion of the Research Questions

RQ1. When we compare the defects in code, the participants have similar defects in both approaches (e.g., incorrect loop guards or incomplete invariants), but they have fewer defects with PhV. A possible reason is the familiar environment of writing Java code in a textual editor. Overall, we have worse results in the second study. Regarding the complete results, we explain the difference between both studies with the better feedback of CorC in comparison to WebCorC such that students can find defects more easily. Another reason is that we monitored active participation in the exercises in person. For the online course, we cannot confirm this. It seems that the students were better prepared in the first user study. We noticed that considerably more participants in the second user study have not the necessary knowledge to construct programs with CbC. Some students may not have participated in the exercises and may not have familiarized themselves with (Web)CorC.

RQ2. We answer the second research question, whether participants prefer CbC or PhV. Participants like the familiar programming style with PhV, but the majority prefer (Web)CorC over KeY. The participants mention that CorC has better and fine-grained console feedback which helps detecting defects during program construction. In the previous study, the participants highlight the good

Table 2. Answers for the questions OQ1−OQ8

Question	Answer
OQ1	On average, the participants rate the work with WebCorC slightly worse (2.1/5)
OQ2	They think about the solution first. The group of participants is split between first writing code or specification
OQ3	CorC has more functionality. Web-frontend is easier to access and system independent
OQ4	Some participants are not experienced enough to interact with WebCorC
OQ5	Participants find defects in corner cases with WebCorC. They divide the problem into smaller blocks. CbC rules are too restrictive. Some are unfamiliar with graphical programming
OQ6	Six answers in favor of (Web)CorC. CorC has better feedback than WebCorC. Two participants prefer KeY because of the familiar programming style
OQ7	Six answers in favor of (Web)CorC. Two answers in favor of KeY
OQ8	Six answers in favor of (Web)CorC, mostly because of the better feedback for verification results. Two answers in favor of KeY, as KeY shows the whole proof tree

feedback for each refinement step, which is not implemented in WebCorC yet. With better feedback, they would prefer WebCorC over CorC due to the easier handling and installation. Surprisingly, nobody complains about the additional specification effort in CbC.

Compared to the first UEQ shown in Fig. 2, we get slightly worse results in the second study. The main reason is worse user feedback for WebCorC in comparison to CorC. This insight coincides with the answers of the open questions in both user studies. Thus, participants rate WebCorC more demotivating, unpredictable, and harder to learn because CorC is more advanced. Due to the online course, it was also harder to teach the tools to the students. Students asked fewer questions, therefore, problems were not discovered that also arose during the user study (e.g. the correct initialization of variables). In person, problems stand out more quickly and can be easily explained. Nevertheless, the participants in both studies prefer (Web)CorC over KeY. Considering that the students have more defects with (Web)CorC, the students seem to factor in that they like the CbC approach for correct software development. The main reason against (Web)CorC is that participants are more familiar with the programming style in KeY. A limitation that is likely due to the shorter time of working with the CbC process.

6.4 Threats to Validity

External Validity. The user studies had 10 and 13 participants. With this limited number of participants, the generalizability of the results is restricted, but we

were able to analyze the programming results of each participant in detail. The participants are all Computer Science students that learned verification in the Software Quality 2 course. Therefore, they are not experts in verification, but should be able to solve smaller examples as the ones asked in this study. Through their statements in the programming experience questionnaire, most students can be compared to junior developers. Furthermore, the small algorithms reduce the generalizability for larger algorithmic problems. Regarding the time frame of a course, a longer study was not feasible.

Internal Validity. The motivation of the participants and their effort of solving the tasks could not be monitored due to the online version of this user study. As the time was limited for each task, most solutions were not verified completely. With additional time, it would be possible for more algorithms to be verified.

7 Lessons Learned for Online Teaching

In this section, we conclude the paper by summarizing our lessons learned for online teaching. The first three findings are based on the results in the user studies. The last three also include our experiences from the online course.

Procedure of Software Development. By analyzing the questionnaire and the programming results of the user study, we notice that students are mostly hacking programs into correctness. By teaching the correctness-by-construction approach, we enable students to think of the specification and the corner cases first, before starting to program. This is well-received by the students, but the approach needs time to be adopted.

Accessibility of Tools. In the questionnaire, students highlight that tools should be easy to install for online teaching. If a tool needs many installation steps or has a high potential to fail on some machines, students will not actively participate in exercises. Students that are not able to solve the CbC tasks indicate missing knowledge in (Web)CorC. Also, tools should be freely available such that students are not excluded because they cannot afford the tool. Many tools that we use during our lectures are Eclipse plug-ins. For Eclipse plug-ins, the easiest way to install them is by using an update site. However, for some tools, the update sites are not accessible anymore or only work with specific versions of Eclipse or JDK. That has to be checked before a course.

Feedback of Tools. From the questionnaire, we know that tools should give detailed and fine-grained feedback if errors occur in the development process. Without feedback, the finding of defects during new tasks gets frustrating such that students tend to give up faster online. This confirms results in the literature [11,26].

Besides of the user studies, we also collect feedback during the courses. Good teaching is characterized by active participation of students [17,26]. As students are more quickly distracted online, we describe how to improve the online course such that students actively participate. Regarding the fact that we have more

programming defects in the second user study, we still have to improve the online course to be as good as the course in person.

Breakout Rooms. During the online exercises, we found that including tasks where students can work in small groups in breakout rooms increases the number of actively participating students. This holds, especially when the teacher is not constantly in the same room and the students can work together on a task, which has not been prepared in advance. Generally, breakout rooms also help students to connect with each other and build learning groups for exams which became more difficult during the pandemic.

Interactions in an Online Setting. In online courses, it is way more important that students are willing to follow the lecture and to participate in exercises. In the results of the user study, we encounter several students that indicate missing background knowledge for the tasks. To prevent this, we derive the following best practices for online teaching: Students should attend exercises with cameras which increases attention. Students should be integrated into lectures by asking questions. When videos and slides are provided in addition to a lecture, students can consolidate what has been learned. Exercises with voluntary tasks are working only for a minority of students. Other students will attend the exercises unprepared. So exercises need to be mandatory or could provide bonus points for the final exam.

Openness to Novel Approaches. Students are open minded for new techniques and tools as we can see from our experiences with (Web)CorC. As teachers, we have to ensure that new topics are introduced interactively and with examples. However, when the new technique is not introduced properly, students will not consider it for future tasks and fall back to old familiar approaches. We want to ensure that formal methods are not taught for the sake of the course, but be anchored in the mind of young computer scientists. So the introduction of the new techniques needs to be thorough, well illustrated using meaningful examples, and supported by accessible tools.

Acknowledgments. We thank Huu Cuong Nguyen and Malena Horstmann for their help in preparing and conducting the user study.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification-The KeY Book: From Theory to Practice, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6

4. Amighi, A., Blom, S., Darabi, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Verification of concurrent systems with VerCors. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 172–216. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07317-0_5
5. Back, R.J.: Invariant based programming: basic approach and teaching experiences. FAOC **21**(3), 227–244 (2009)
6. Back, R.-J., Eriksson, J., Myreen, M.: Testing and verifying invariant based programs in the SOCOS environment. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 61–78. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73770-4_4
7. Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (2012)
8. Barnes, J.G.P.: High Integrity Software: The Spark Approach to Safety and Security. Pearson Education (2003)
9. Beckert, B., Grebing, S., Böhl, F.: A usability evaluation of interactive theorem provers using focus groups. In: Canal, C., Idani, A. (eds.) SEFM 2014. LNCS, vol. 8938, pp. 3–19. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15201-1_1
10. Beckert, B., Grebing, S., Böhl, F.: How to put usability into focus: using focus groups to evaluate the usability of interactive theorem provers. EPTCS **167**, 4–13 (2014)
11. Cataño, N.: Teaching formal methods: lessons learnt from using Event-B. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 212–227. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_14
12. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
13. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35
14. Creuse, L., Dross, C., Garion, C., Hugues, J., Huguet, J.: Teaching deductive verification through FRAMA-C and SPARK for non computer scientists. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 23–36. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_2
15. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16
16. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Hoboken (1976)
17. Divasón, J., Romero, A.: Using Krakatoa for teaching formal verification of Java programs. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 37–51. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_3
18. Feigenspan, J., Kästner, C., Liebig, J., Apel, S., Hanenberg, S.: Measuring programming experience. In: ICPC, pp. 73–82. IEEE (2012)
19. Havelund, K., Pressburger, T.: Model checking Java programs using Java pathfinder. STTT **2**(4), 366–381 (2000)

20. Hentschel, M., Hähnle, R., Bubel, R.: Can formal methods improve the efficiency of code reviews? In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 3–19. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_1
21. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_21
22. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: ICSE, pp. 672–681. IEEE Press (2013)
23. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer, Heidelberg (2012)
24. Laugwitz, B., Held, T., Schrepp, M.: Construction and evaluation of a user experience questionnaire. In: Holzinger, A. (ed.) USAB 2008. LNCS, vol. 5298, pp. 63–76. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89350-9_6
25. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
26. Liu, S., Takahashi, K., Hayashi, T., Nakayama, T.: Teaching formal methods in the context of software engineering. ACM SIGCSE Bull. **41**(2), 17–23 (2009)
27. Meyer, B.: Eiffel: a language and environment for software engineering. JSS **8**(3), 199–246 (1988)
28. Morgan, C.: Programming from Specifications, 2nd edn. Prentice Hall, Hoboken (1994)
29. Oliveira, M.V.M., Cavalcanti, A., Woodcock, J.: ArcAngel: a tactic language for refinement. FAOC **15**(1), 28–47 (2003)
30. Pearce, D.J., Groves, L.: Whiley: a platform for research in software verification. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 238–248. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02654-1_13
31. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: Your proof fails? Testing helps to find the reason. In: Aichernig, B.K.K., Furia, C.A.A. (eds.) TAP 2016. LNCS, vol. 9762, pp. 130–150. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41135-4_8
32. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: Tool support for correctness-by-construction. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 25–42. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_2
33. Runge, T., Thüm, T., Cleophas, L., Schaefer, I., Watson, B.W., et al.: Comparing correctness-by-construction with post-hoc verification—a qualitative user study. In: Sekerinski, E. (ed.) FM 2019. LNCS, vol. 12233, pp. 388–405. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54997-8_25
34. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_53
35. Watson, B.W., Kourie, D.G., Schaefer, I., Cleophas, L.: Correctness-by-construction and post-hoc verification: a marriage of convenience? In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 730–748. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_52