Comparing Correctness-by-Construction with Post-Hoc Verification—A Qualitative User Study

Tobias Runge[1(✉)], Thomas Thüm[2(✉)], Loek Cleophas[3,4(✉)], Ina Schaefer[1(✉)],
and Bruce W. Watson[4,5(✉)]

[1] TU Braunschweig, Braunschweig, Germany
{tobias.runge,i.schaefer}@tu-bs.de
[2] University of Ulm, Ulm, Germany
thomas.thuem@uni-ulm.de
[3] TU Eindhoven, Eindhoven, The Netherlands
loek@fastar.org
[4] Stellenbosch University, Stellenbosch, South Africa
bruce@fastar.org
[5] Centre for Artificial Intelligence Research, Stellenbosch, South Africa

Abstract. Correctness-by-construction (CbC) is a refinement-based methodology to incrementally create formally correct programs. Programs are constructed using refinement rules which guarantee that the resulting implementation is correct with respect to a pre-/postcondition specification. In contrast, with post-hoc verification (PhV) a specification and a program are created, and afterwards verified that the program satisfies the specification. In the literature, both methods are discussed with specific advantages and disadvantages. By letting participants construct and verify programs using CbC and PhV in a controlled experiment, we analyzed the claims in the literature. We evaluated defects in intermediate code snapshots and discovered a trial-and-error construction process to alter code and specification. The participants appreciated the good feedback of CbC and state that CbC is better than PhV in helping to find defects. Nevertheless, some defects in the constructed programs with CbC indicate that the participants need more time to adapt the CbC process.

1 Introduction

Correctness-by-construction (CbC) [17,19,25,30] as proposed by Dijsktra is a method for the construction of formally correct programs. The programmer refines an abstract statement with pre-/postcondition specification to a concrete implementation, guided by the specification and refinement rules. It is claimed that programmers construct programs with low defect rates with CbC [20]. There are three reasons for this that need to be evaluated. First, the structured reasoning discipline which is enforced by the refinement rules reduces the possibility to introduce

E. Sekerinski et al. (Eds.): FM 2019 Workshops, LNCS 12233, pp. 388–405, 2020.
https://doi.org/10.1007/978-3-030-54997-8_25

defects. Second, defects in the code can be traced to their source through the refinement structure. Third, programmers and users gain trust in the program because a formal methodology was used to create the program [25]. We implemented the correctness-by-construction approach in a graphical IDE called CorC,[1] which support users during the construction and verification of programs.

With deductive post-hoc verification (PhV), we refer to techniques as used in the KeY community [4], which verify a program after its creation. A verifier checks whether the program satisfies its pre-/postcondition specification. PhV does not provide a strict guideline on how to construct the program; the programmer can freely implement the program. This can decrease the time taken to create a first (potentially faulty) version of a program, but can increase the program verification time because it is more likely that defects occur in the code [36]. In order to evaluate this claim, we consider the post-hoc verifier KeY [4] as an instance. KeY can verify Java programs annotated with pre-/postcondition specifications in the Java Modeling Language (JML).

As the title suggests, we compare correctness-by-construction with post-hoc verification. In a qualitative user study, participants use CorC and KeY to implement and verify an algorithm with each tool. By analyzing 347 intermediate code snapshots, we get better insights in the process used by participants to construct and verify algorithms. With a user experience questionnaire, we compare which advantages and disadvantages of the verification techniques and the tools have been experienced. Our contributions in this paper are the following.

- We give an overview of advantages and disadvantages of CbC and PhV.
- We designed and performed a user study to compare both approaches. We analyze the defects in code and specification of each intermediate snapshot for both tools.
- We discuss our insights and compare CbC with PhV based on our user study.

2 Verification Techniques

In our user study, we evaluate the techniques PhV and CbC. Therefore, we first present and compare the foundations of both techniques. We also survey claims about their advantages and disadvantages as discussed in the research literature.

2.1 Post-hoc Verification

With post-hoc verification, we refer to a method which is used to verify whether a program satisfies a given specification. A programmer develops a program and a pre-/postcondition specification. Besides the pre-/postcondition specification, loop invariants can be defined to specify the behavior of loops in the code. The correctness of the program can be verified by using a deductive verification tool, such as KeY [4]. It translates the program and the specification to a dynamic logic formula (i.e., proof obligations). The program is executed symbolically, and

[1] see https://github.com/TUBS-ISF/CorC and [34] for explanation of the editor.

the formula is updated according to the new symbolic state. After the program is completely executed, it no longer appears in the formula, and the remaining first-order proof goal can be evaluated by theorem proving. The verification can be performed (semi-)automatically or interactively. We use automatic verification in this paper in order to be able to focus the user study on the construction of programs and specification. Most users in industry do not have a theoretical background to verify programs interactively.

2.2 Correctness-by-Construction

Correctness-by-construction in the classical Dijkstra-style [17,25] is a programming method which starts with a Hoare triple specification. This Hoare triple contains a precondition, an abstract statement (i.e., a statement that is a placeholder for concrete code), and a postcondition. The triple asserts total correctness. If the program is in a state where the precondition holds, its execution will terminate in a state where the postcondition holds. An abstract statement in a Hoare triple can be refined to a concrete program using refinement rules. The rules introduce new statements, such as loops or assignments. By refining the program, the pre-/postcondition specification is propagated through the constructed program, so that the refined statements are also surrounded by a pre- and a postcondition, forming more Hoare triples [17,25]. These refinement rules introduce proof obligations which have to be discharged to establish the correct application of the refinements rules. E.g., it has to be verified that by executing an assignment the corresponding postcondition is implied, or that a loop invariant holds after each iteration. The correctness of these proof obligations can be checked using verification tools [1,34]. We implemented tool support for the construction of programs following CbC [34]. The graphical editor CorC visualizes program refinements in a tree-like structure.

2.3 Contrasting Correctness-by-Construction and Post-hoc Verification

CbC and PhV are two different methods to create verified software. Nevertheless, they share commonalities. Both start with a pre-/postcondition specification and result in a program that satisfies this specification. The procedure to construct the program, however, is different. With CbC, the program is constructed stepwise by applying checkable refinement rules. With PhV, the program is constructed without a strict guideline (i.e., the programmer can freely develop the program and intermediate steps are not proven). Afterwards, the final program can be verified.

It is claimed that CbC can lead to well-structured code that can be verified more easily [25,36]. The additional time needed to construct the code is said to be amortized with a significantly reduced time to prove the code. When applying CbC, every refined statement leads to a provable side condition, where a theorem prover can check whether this condition is satisfied. If the check fails, the programmer can alter the refined statement to establish the proof. This is a potential advantage compared to PhV because problems in the verification process can be pinned to small parts of the program. In contrast, with PhV additional expertise or sophisticated tool support is necessary to infer the defect from open goals in the proof [33].

Programmers who use the CbC approach are bound to the stepwise refinement using rules. Therefore, after each refinement the program with all conditions can be reviewed by the programmers. They can continuously check the surrounding specification of every statement. This can raise awareness of defects in the program, resulting in fewer defects in comparison to PhV programming. The number of required iterations to get to a correct program with CbC may also be reduced because defects are detected early, even before a prover is used [36].

An open question is whether the experience of developers is crucial for the development of correct code. Using PhV, programmers can implement algorithms as they normally do and verify whether the program is correct afterwards. Using CbC, the programmer needs an understanding of the refinement rules to construct programs. Whether this barrier noticeably increases the time of the construction process, or whether the CbC method does not have a negative influence needs to be evaluated.

These claims are established in the literature but need to be evaluated in a user study. We analyze defects in intermediate and final programs and interpret the answers of a questionnaire to provide evidence for the claims.

3 Design of a User Study

To qualitatively evaluate CbC and PhV, we performed a user study with the two tools, CorC and KeY. We decided explicitly for a controlled experiment to monitor all participants in parallel during the tasks and to collect all programming results. We selected CorC because it is a new tool that supports the CbC method in a graphical user interface and which has been taught to the participants. KeY, which is a major tool for the automatic verification of Java programs, is used to get good comparability as CorC uses KeY as back-end for the verification. Therefore, we have a comparable expressiveness with both tools.

We provide the participants a pre-/postcondition specification for an algorithm, and they developed code to satisfy this specification. The algorithms can be implemented in under ten lines of code. We decided explicitly for this size, so the whole experiment could be done in 90 min because it is complicated to motivate people to do longer experiments. We also excluded the process of writing an adequate pre-/postcondition specification because this has to be done for both techniques and highly influences what needs to be implemented and verified. The same starting point reduces the divergence, so that we can analyze the results on the same basis. We want to qualitatively analyze how the participants develop and verify code. Therefore, we took intermediate snapshots of the code every time the code was verified and analyzed the defects created during the development process. We checked a total of 347 versions of programs, something which is not feasible with larger programs and more participants. The user experience with the tools was measured qualitatively by a questionnaire in order to find improvement potentials. The material of the user study is published on GitHub.[2]

Objective. We surveyed in Sect. 2.3 whether CbC can have a positive impact on programming and verifying code. Hence, we want to evaluate whether a positive impact can be detected (i.e, programmers appreciate that defects could be more easily detected with CbC). We consider three research questions to evaluate the methodologies (RQ1–2) and the tools (RQ3) qualitatively.

RQ1: What errors do participants make with CbC or PhV?
RQ2: What is the process of participants to create programs with CbC or PhV?
RQ3: Do participants prefer CorC or KeY?

Participants. Our participants were students of a software quality course at TU Braunschweig, Germany. We decided for these students because they were taught the fundamentals of software verification, and they got an introduction to both tools. They have experience in verifying methods with both tools although the specific algorithms of this experiment were new to them. We had ten participants which were divided into two groups randomly. The programming experience that was measured with an initial questionnaire [18] was 2.189 for group A and 1.791 for group B.[3] The experience of individuals ranged between 1.609 and 2.777. With a Mann-Whitney test, we calculated no significant difference between both groups (p-value = 0.1514). Most of the students have several years of programming experience in industry, and therefore, can be compared to junior developers. Six participants had three to seven years experience as programmer in industry, two were new programmers in larger projects, and only two never programmed in larger projects.

The participants voluntarily attended in the experiment. They knew that they took part in an experiment and that this experiment did not affect the grade of the course. Every participant was paid € 10 to create an incentive for them. Participants who solved one or both exercises also had the chance to win € 50 (i.e., one of them was randomly selected). This lottery should increase the motivation to solve the exercises by creating a realistic pressure to succeed.

Material. In our experiment, the participants had to implement and verify two algorithms. For every participant, we prepared a computer with an Eclipse installation that supports CorC and KeY, and contained a workspace with the two exercises. We also provided a cheat sheet containing the syntax of KeY and CorC to help the participants. In order for us to properly analyze the experiment, participants took the programming experience questionnaire before the exercises

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

[3] The calculation is explained in the work by Feigenspan et al. [18]. They derived with stepwise regression testing that the experience in comparison to classmates with factor 0.441 summed up with the logical programming experience with factor 0.286 is the best indicator for programming experience.

and a user experience questionnaire afterwards. The user experience questionnaire is a combination of open questions (OQ 1–4) and the User Experience Questionnaire[4] (UEQ).

OQ1: What was better in CorC/KeY?
OQ2: How did you proceed with the task in CorC/KeY?
OQ3: Which tool would you use for verification, and why?
OQ4: Which tool better supports avoiding or fixing defects, and why?

UEQ is an established questionnaire which measures six properties of a product (e.g., attractiveness) by asking the user to rate the product with 26 items. Each item describes the product positively and negatively, and the user must evaluate which and to what extent one of the descriptions fits. Additionally, the workspaces were saved to analyze the created code and specifications.

Tasks. We used the Latin square design to arrange the participants. Group A used CorC for a `maximum element` algorithm, and KeY for `modulo`. Group B did the exercises in the same order, but each one with the other tool. We switched the order of the tools to address learning and ordering effects. We believe that an order between tools is worse than an order between exercises because we want to get insights in the usability of the tools. Additionally, the order between exercises was not varied because a split into four groups was not manageable. For each exercise, we provided a pre-/postcondition, and a task description in which we explained the purpose of the algorithm, so that the partcipants understood what the implementation should achieve.

The algorithm `maximum element` finds the index of the maximum element in an array. The array is assumed to be non-empty to simplify the algorithm, so that an index of the array should always be returned. The algorithm `modulo` gets two integers a and b as input and computes the two values factor and remainder for the equation $factor * b + remainder = a$. For the construction of the algorithm, the division and modulo operations are prohibited. Both algorithms are similar in size and cyclomatic complexity.

The tasks were designed such that a small, manageable subset of Java is sufficient to implement the algorithms. `Assignments`, `If-Then-Else`, and `While` were the only necessary statements. We excluded method calls because they complicate the verification for these two algorithms unnecessarily.

Variables. In our experiment, the tool is an independent variable, with the two treatments CorC and KeY. To check the correctness of the code in KeY, we reran the proof for the solution of every participant. In CorC, we checked that all nodes in the refinement hierarchy are proven. If a solution was not proven, we checked whether the code is correct with KeY and, if necessary, adjusted the specification, such as a loop invariant, to close the proof. If the code was also incorrect, we checked how many defects were in the code by adjusting the code. To evaluate the programming and verification process, we analyzed the intermediate snapshots. Here, the changes and defects were also counted in

[4] https://www.ueq-online.org/.

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

#Defects	KeY		CorC	
	Code	Specification	Code	Specification
Verified	2		3	
No defects	8	2	4	3
Minor defects	1	4	3	2
Major defects	1	3	1	2
Incomplete	0	1	2	3

terms of changed lines. For example, if an incorrect assignment was fixed by a participant, we count one change in the program and reduce the number of defects by one. The time needed for every exercise was measured manually. If a participant solved a task, the time was noted. After 30 min, we interrupted the participants when they were not finished.

Deviations. The participants assigned themselves randomly to a group by selecting one computer. We missed that the participants per groups were unequal. Group A had six participants, and group B had only four. This unequal distribution changed which exercise was done with which tool. Since we used the Latin square design, the influence should not be significant because we still had ten results for each treatment.

4 Results and Discussion

In this section, we present the results of our evaluation. We analyzed the data of the created programs and the answers of the questionnaire. The comparably small number of participants reduces the generalizability of the results, but allows us to evaluate the process of the participants in detail by analyzing all 347 intermediate code snapshots. This gives us anecdotal evidence to qualitatively discuss advantages and disadvantages of CbC and PhV.

4.1 Defects in Implementation

To answer the first research question, RQ1, what errors do participants make with CbC or PhV, we analyze defects in the program and the specification.

There are ten implementations with each tool. The defects in the code are shown in Table 1 in column two and four, numbered left-to-right from one. With KeY, eight programs were correct and two of them were verified. In one case, only a loop guard was slightly incorrect (e.g., two variables were compared with less than, but less than or equal was correct). Only one program contained major defects. We classified a program to have major defects, if we could not correct

Table 2. Initial and final defects in the programs of participants

Row	Initial defects	Final defects	KeY	CorC
1	0	0	6	1
2	1	0	1	1
3	2	0	1	0
4	3	0	0	1
5	4	0	0	1
6	1	1	1	0
7	2	1	0	2
8	3	1	0	1
9	>5	>5	1	1
10	Incomplete		0	2

the program with at most five changes. With CorC, four programs were correct and three of them could be verified. In three programs, a minor defect occurred, one program had numerous mistakes, but also two programs were incomplete.

In the case of intermediate specifications which needed to be provided, for both tools the results were worse. In Table 1, the defects in intermediate and loop invariant specifications are shown in column three and five. Only in two cases for KeY and three cases for CorC no defects occurred. In KeY, four specifications contained minor defects, such as a missing boundary for a control variable or an incorrect comparison of two variables. Three programs had major defects in the specification. For example, it was not properly specified which elements of the array were already examined in the `maximum element` algorithm. One participant did not create an invariant. In the case of CorC, two minor and two major defects occurred, but also three algorithms had incomplete specifications. Two of these three incomplete specifications could be explained as incomplete programs. In the third case, the algorithm was created but not specified.

To analyze the defects in more detail, we counted the defects during the programming task. In Table 2, the defects in the initial (i.e., programs at the first verification attempt) and final programs are shown. One difference between programming in KeY and CorC is that the participants in KeY started the first verification after the program was completely constructed. In CorC, some users started earlier, with incomplete programs because they could verify Hoare triples for parts of the programs that were already completely concretized. With KeY, six participants created a program without any defects (Row 1). In two cases (rows 2 and 3), one or two defects were found. One participant started with one defect, but could not find the defect (Row 6). The participant also had three defects in an intermediate result, but never found the incorrect loop guard condition. One program had more than five defects in the beginning and the end (Row 9). With CorC, only one program had no defects in the beginning (Row 1). Three participants started with one to four defects and fixed the defects (rows 2, 4, and 5). One participant who started with two defects and ended with one (Row 7), had a correct intermediate result, but inserted one defect in the final version. One participant had a result which could not be fixed easily (Row 9).

Two programs were incomplete in CorC (Row 10). Their developers started with the first refinements, but could not finalize the program in the CorC editor.

The construction of algorithms with KeY was mostly the same. The participants created a correct or nearly correct algorithm. Afterwards, a loop invariant was constructed and the program was verified. Astonishingly, no participant could verify the program on the first try even though the program was correct because the loop invariants were incorrect or too weak (e.g., for `modulo` the special case that the input parameters could be equal was not handled). The approach of the participants to get the program to a verifiable state was different. Some participants mostly changed the invariant and verified the program again. Others changed the loop and the invariant. A correct program was changed up to ten times to another correct solution, but no sufficient invariant for KeY to verify the program was found. Some participants also changed whether the loop variable was increased or decreased several times.

With CorC, the most common approach was to create the program with all refinements and specify the intermediate conditions or loop invariants in parallel. Often the program was completely refined before the first verifier call. If the verification was not possible, missing parts such as the initialization of control variables were added, assignment or conditions were changed. In three cases, the initial defects were found, but in one case, a correct intermediate program was changed to an incorrect program. The participant with the incorrect result started with a program where he forgot to decrease the control variable in the loop. Afterwards, the participant decreased the variable correctly, but the loop invariant was wrong, so the statements could not be verified. So, the program was changed again to decrease the control variable at another place in the program. In the process, the participant introduced an incorrect execution path where the variable is not decreased. Two other participants started with a loop, but forgot the initialization of necessary variables. This mistake was recognized during the exercise.

In summary, both tools in some cases lead to correct and verified programs. Small defects occurred with both tools, but in CorC, we observed incomplete programs. If the program could not be verified, participants mostly changed the loop guards, the loop body, or loop invariants. The changes in the code are fewer with CorC than with KeY. If a program could not be verified, the problem was in most cases an insufficient loop invariant or a wrong loop guard. With PhV, most participants created correct code in the first place. As shown in Table 2, only three defects were found in the process in total. With CbC, the users started mostly with a defective program and found twelve defects in total. This higher number of found defects may be explained with better tool support in CorC, but also with the higher number of existing defects. With PhV, only four defects existed by excluding the completely wrong program. Thus 75% of the defects were found. For CbC, there are 15 defects in total, so 80% have been found.

RQ1. Comparing the defects in code, participants made similar errors with both techniques (e.g., incorrect loop guard), but they made fewer and mostly minor errors with PhV. This could be explained with the familiar environment of standard Java with JML. The two incomplete programs in CorC can be explained by problems interacting with the tool. Thus in total, more correct programs were created with PhV than with CbC. That more programs were verified with CbC anyway is interesting. One explanation could be that programs with CbC were less changed. The participants might have thought more about the program instead of changing the program by trial-and-error. Due to the similar correction rates of defects for both tools, we cannot confirm a negative influence of CbC in the programming process, but we should further investigate why more defect programs with CbC exist.

4.2 Analysis of Programming Procedure

From the intermediate snapshots, we can evaluate the programming procedure by analyzing the changes and defects in code and intermediate specification, and missing program or specification parts. We analyzed 20 solutions containing between 9 and 39 snapshots. We excluded the incomplete and entirely incorrect cases because we could not count wrong or missing parts with the same scale as for the other cases. In the following, three typical results are shown.

In Fig. 1, we show the graph of a participant solving the `maximum element` task in CorC. The participant started the verification process with two missing lines of code and two missing intermediate specification lines. The participant also had two defects in the intermediate specification. Overall, 25 steps were taken by the participant to achieve the correct solution. In the first 13 steps, the program and the specification were changed, but no defects were fixed. In Step 14, the invariant of the program was corrected. The special case that there can be more than one maximum element in the array was included in the invariant. The next steps were used to verify the program, until the participant realized that the initialization of variables was missing. After this fix in Step 21, the program was verified.

In Fig. 2, the process to construct the `maximum element` algorithm in KeY is shown. The participant started with a correct program where the invariant was missing. After introducing the invariant with a defect, the participant changed the code and the invariant during the whole task without finding a sufficient invariant. The program was changed to iterate the array from forward to backward and vice versa several times. The main reason that the program could not be verified was that the invariant did not specify which elements of the array were already visited. There were similar cases with KeY where also only the invariant was wrong. The code and intermediate specifications were changed by most participants during their development process. There were two participants who mostly changed the invariant instead of the code.

In Fig. 3, we show a graph of a user developing the `modulo` algorithm in CorC. The participant started with one defect in the code, an incorrect loop guard, and two missing specification parts, the invariant and an intermediate condition. In the first steps, the participant tried to verify the whole program without changing it. Then, the missing specifications were added, but both were wrong. In the invariant, the comparison operator was the wrong way around, and the intermediate condition was too weak (i.e., it was not specified that the correct factor was found). The specification was changed until step twelve, then the participant tried to verify the program again. As this did not lead to a

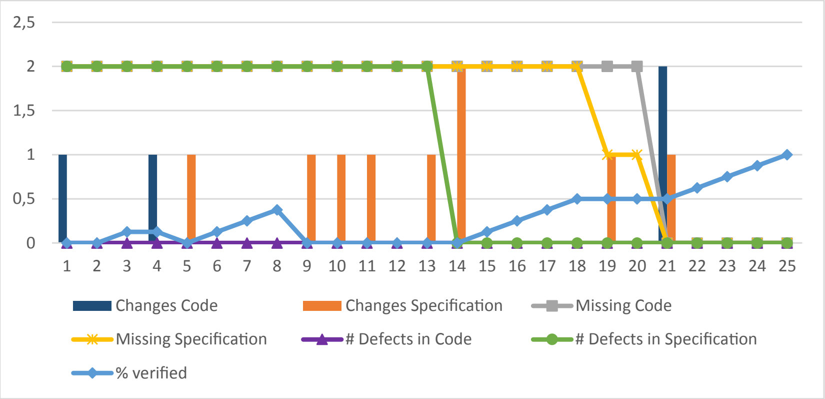

Fig. 1. Process to construct `maximum element` algorithm in CorC

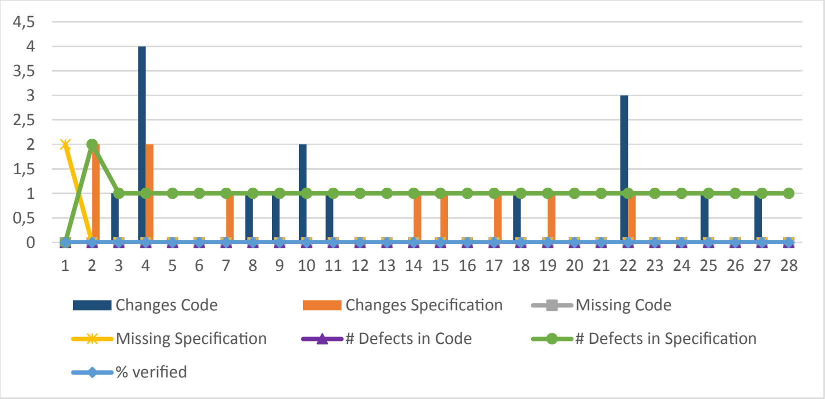

Fig. 2. Process to construct `maximum element` algorithm in KeY

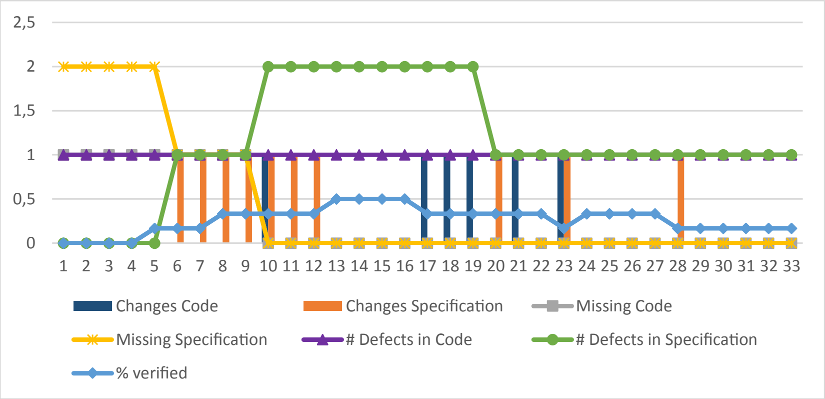

Fig. 3. Process to construct `modulo` algorithm in CorC

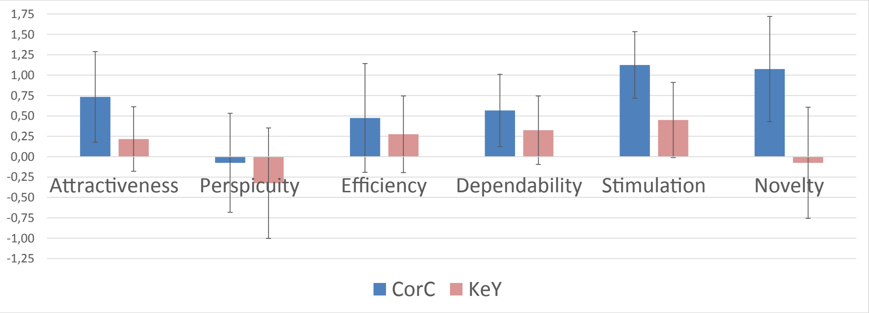

Fig. 4. Results of the user experience questionnaire

solution, the code and specification were changed again. The wrong comparison in the invariant was found, but the other two problems remained until the end.

RQ2. With the detailed analysis of all 347 program snapshots, we can discuss the programming process of the participants. We saw that correct programs were changed several times if they could not be verified, and surprisingly remained correct. The participants did not realize that only the intermediate specification was insufficient. They need better tool support to pinpoint the defects in code or specification. We also noticed a non-monotonic construction process for both techniques. By monotonic, we mean that a program is specified, constructed, and then verified to be correct. An example for the non-monotonic construction process with CbC and PhV are the trial-and-error changes in specification and code. For example with PhV, the users changed to iterate the array from forward to backward in several cases. With CbC, the users verified a correct part of the program, but changed it if they could not verify the complete program. In comparison to PhV, participants using the CbC approach changed the code less. Furthermore, a correct specification may favor the finding of mistakes in the program. Often defects were found after a correct loop invariant was introduced. In our evaluation, all programs with correct specifications had no defects.

4.3 User Experience

The results of the UEQ are presented in Fig. 4. The answers of the participants were evaluated according to the six measurements: attractiveness, perspicuity, efficiency, dependability, stimulation, and novelty. The scale is between +3 and −3 for each item. Overall, the average answers of the participants are higher for CorC. For perspicuity both tools got a negative mean value. KeY also has a negative result for novelty. We measured a significant difference with the T-test.[5] Stimulation (p = 0.0457) and novelty (p = 0.0274) are significantly different.

[5] Statistical hypothesis test to compare two independent samples which are normally distributed.

For the open questions, we clustered the answers to analyze whether the participants had similar experiences. The results are shown in the following.

OQ1. *What was better in CorC/KeY?* Five to six participants valued the clarity of CorC. They also valued the good feedback of CorC to spot the defects in the program because the program is split into individually provable statements. On the negative side, the unfamiliar syntax and the handling of the tool were mentioned. In the case of KeY, the well-known Java and JML syntax was appreciated by nearly all participants. Two participants also valued the clarity of KeY. One participant disliked the bad error messages of KeY. Another one mentioned that KeY gives more information about the problem, but this follows from the design of the experiment. CorC uses KeY as back-end for verification, but we suppressed the KeY editor on purpose in CorC because the verification problems for the implemented algorithms should be small enough to be verified automatically [34]. In the normal configuration, CorC can deliver the same information by opening the proof in KeY. In summary, the known syntax in KeY was an advantage, but the participants appreciated the better potential in CorC to find the defects because the program was decomposed into provable statements.

OQ2. *How did you proceed with the task in CorC/KeY?* In KeY, all participants created the code first, then they created the loop invariant and verified the program. One participant emphasized that the program was inferred from the postcondition. In CorC, the common case was to construct the code stepwise. Two participants explicitly mentioned that they created the program in CorC first, then specified the program. Two others started with the specification in CorC. In contrast to KeY, the participants wrote specifications only in CorC before or during the construction of the code.

OQ3. *Which tool (CorC/KeY) would you use for verification, and why?* Five participants decided to use CorC for verification. They appreciated the clarity. Two participants mentioned the support to verify and debug individual statements. One participant highlighted the reflective coding process that is encouraged by CorC. Four participants decided to use KeY. They liked the familiar environment and syntax. As in the first question, one participant mentioned that KeY gives more information. There is no clear trend towards one tool.

OQ4. *Which tool (CorC/KeY) better supports avoiding or fixing defects, and why?* Most participants decided for CorC to avoid or fix defects. They appreciated that defects are assigned to individual statements, therefore, it was easier to understand the problem. One participant mentioned that the stepwise construction helped to create correct programs. For both tools, some participants indicated that defects were detected and only correct code could be verified. Although nearly the same number of participants would use KeY or CorC for verification, most participants wanted to use CorC to find or fix defects in the coding process. That defects were associated to specific statements was well received by the participants.

RQ3. The third research question, whether participants prefer CorC or KeY, can be answered with the results of the questionnaire. The participants preferred

KeY because of the familiar syntax, and CorC for the better feedback if there were defects in the code. This leads to a balanced vote on which tool the participants would use for verification. Interestingly, the participants voted in favor of CorC when it comes to finding and fixing defects. This should be further investigated; what keeps participants from using CorC even though they mention that it helped better to find defects. With the answers of the participants and the analysis of the snapshots, we can also confirm how the participants worked on the tasks. In KeY, the program was developed, and afterwards the specification was constructed. So, the code was mostly correct in the first place. In CorC, they had different approaches. They interleaved coding and specification or started with the specification. This results in starting the verification earlier with incomplete or incorrect programs. Surprisingly, nobody complained about the additional specification effort in CorC.

4.4 Threats to Validity

In our experiment, we had only 10 participants. This reduces the generalizability of the results, but allowed us to analyze all 347 versions of program snapshots in detail. The participants were all students of a software quality course. We could ensure that all students had the required theoretical and practical precognition. They are no experts in verifying software, but smaller tasks, such as those of our experiment, were solved before by the participants in class. Most students also have part-time jobs in companies, so the results are generalizable to junior developers. The motivation of the students is doubtful, but the lottery gave an incentive to accomplish the tasks. Another limitation for the experiment was the limited time. Most participants have accomplished to write correct code, but only five out of twenty algorithms were also verified. With more time it is possible that more algorithms would have been verified. We only used two small size exercises in our experiment, and therefore, cannot generalize the results to bigger problems. The results of the experiment also depend on our introduction of the tools—though we tried to introduce both tools equally without bias to the students.

5 Related Work

In the literature, tool support for verification was previously evaluated, but PhV was not compared with CbC.

Spec# is an extension of the programming language C# to annotate code with pre-/postconditions and verify the code using the theorem prover Boogie [10,11]. Barnett et al. [11] explained their lessons learned of constructing this verification framework. In contrast, we focus on how users solve programming and specification tasks. Petiot et al. [33] examined how programmers could be supported when a proof is not closed. They implemented tool support that categorizes the failure and gives counter examples to improve the user feedback. This idea is complementary to the CbC method by pinpointing the failure to a small Hoare triple, which was appreciated by the participants in this study. Johnson et al. [23] interviewed developers about the use of static analysis tools. They came to the same result as we did that good error reporting is crucial for developers. Hentschel et al. [21] studied the influence of formal methods to improve code reviews. They detected a positive impact of using the symbolic execution debugger (SED) to locate errors in already existing programs. This setup is different to our evaluation where the participants had to program actively. The KeY tool [12,13] was already evaluated to get insight into how participants use the tool interactively. In contrast, we wanted to evaluate the automatic part of KeY because we think that most users do not have a theoretical background to verify a program interactively.

Besides CorC and KeY, there are other programming languages and tools using specification for program verification. For example Eiffel [28,29] with the verifier AutoProof [24,35], SPARK [9], Whiley [32], OpenJML [15], Frama-C [16], VCC [14], Dafny [26,27], VeriFast [22], and VerCors [5]. These languages and verification tools can be used to compare CbC with post-hoc verification. As we only used a subset of the Java language in our experiment (comparable to a simple while language), the difference to other programming languages is minimal, and we expect similar results for those tools as with KeY.

A related CbC approach is the Event-B framework [1]. Here, automata-based systems are specified, and can be refined to concrete implementations. The Rodin platform [3] implements the Event-B method. For the predecessor of Event-B, namely the B method, Atelier B [2] is used to prove correctness. The main difference to CorC is the different abstraction level. CorC uses source code with specification rather than automata-based systems. The CbC approaches of Back [8] and Morgan [30] are related to CbC by Dijkstra, and it would be interesting to evaluate these approaches in comparison to our CbC tool in a future study. For example, ArcAngel [31] could be used as an implementation of Morgan's refinement calculus. Back et al. [6,7] build the invariant based programming tool SOCOS. They start explicitly with the specification of not only pre-/postconditions but also invariants before the coding process. In their experiment, they discovered that good tool support is needed and that invariants are found iteratively by refining an incomplete and partly wrong invariant; an insight which we can confirm.

6 Conclusion and Future Work

We compared correctness-by-construction and post-hoc verification by using the tools CorC and KeY. Participants could create and verify programs, but the majority failed to create invariants that were strong enough. When a program could not be verified, trial-and-error was the most popular strategy to fix the program. Regarding user experience, KeY and CorC were both considered useful to verify software, but the good feedback of CorC was explicitly highlighted. Nevertheless, the defects in the programs with CorC indicate that the participants need more time to get used to CorC.

We evaluated the user study qualitatively to get insights in how users create verified programs. For future work, we could repeat the experiment with more participants to get quantitative data about defects in the programs. Furthermore, our insights about the trial-and-error programming process could be used to improve the usability of both tools.

Acknowledgment. We would like to thank Alexander Knüppel and Domenik Eichhorn for their help with the user study. The hints and suggestions of Alexander helped to construct the final version of the study. Thanks to Domenik for setting up the tools.

References

1. Abrial, J.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
3. 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. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
4. 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
5. 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
6. Back, R.-J.: Invariant based programming: basic approach and teaching experiences. Formal Aspects Comput. **21**(3), 227–244 (2009)
7. 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
8. Back, R.-J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (2012)
9. Barnes, J.G.P.: High Integrity Software: The Spark Approach to Safety and Security. Pearson Education, London (2003)
10. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM **54**(6), 81–91 (2011)
11. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3
12. 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

13. Beckert, B., Grebing, S., Böhl, F.: How to put usability into focus: using focus groups to evaluate the usability of interactive theorem provers. Electron. Proc. Theor. Comput. Sci. **167**, 4–13 (2014)
14. 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
15. 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
16. 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
17. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Upper Saddle River (1976)
18. Feigenspan, J., Kästner, C., Liebig, J., Apel, S., Hanenberg, S.: Measuring programming experience. In: 2012 IEEE 20th International Conference on Program Comprehension (ICPC), pp. 73–82. IEEE (2012)
19. Gries, D.: The Science of Programming. Springer, Heidelberg (1987)
20. Hall, A., Chapman, R.: Correctness by construction: developing a commercial secure system. IEEE Softw. **19**(1), 18–25 (2002)
21. 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
22. 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
23. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 2013 International Conference on Software Engineering, pp. 672–681. IEEE Press (2013)
24. Khazeev, M., Rivera, V., Mazzara, M., Johard, L.: Initial steps towards assessing the usability of a verification tool. In: Ciancarini, P., Litvinov, S., Messina, A., Sillitti, A., Succi, G. (eds.) SEDA 2016. AISC, vol. 717, pp. 31–40. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-70578-1_4
25. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27919-5
26. Leino, K.R.M.: Specification and verification of object-oriented software. Eng. Methods Tools Softw. Saf. Secur. **22**, 231–266 (2009)
27. 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
28. Meyer, B.: Eiffel*: a language and environment for software engineering. J. Syst. Softw. **8**(3), 199–246 (1988)
29. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992)
30. Morgan, C.: Programming from Specifications, 2nd edn. Prentice Hall, Upper Saddle River (1994)
31. Oliveira, M.V.M., Cavalcanti, A., Woodcock, J.: ArcAngel: a tactic language for refinement. Formal Aspects Comput. **15**(1), 28–47 (2003)

32. 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
33. 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
34. 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
35. 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
36. 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