Can Formal Methods Improve the Efficiency of Code Reviews?

Martin Hentschel, Reiner Hähnle, and Richard Bubel

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany
`hentschel|haehnle|bubel@cs.tu-darmstadt.de`

Abstract. Code reviews are a provenly effective technique to find defects in source code as well as to increase its quality. Industrial software production often relies on code reviews as a standard QA mechanism. Surprisingly, though, tool support for reviewing activities is rare. Existing systems help to keep track of the discussion during the review, but do not support the reviewing activity directly. In this paper we argue that such support can be provided by formal analysis tools. Specifically, we use symbolic execution to improve the program understanding subtask during a code review. Tool support is realized by an Eclipse extension called Symbolic Execution Debugger. It allows one to explore visually a symbolic execution tree for the program under inspection. For evaluation we carefully designed a controlled experiment. We provide statistical evidence that with the help of symbolic execution defects are identified in a more effective manner than with a merely code-based view. Our work suggests that there is huge potential for formal methods not only in the production of safety-critical systems, but for any kind of software and as part of a standard development process.

Keywords: Code Review, Symbolic Execution, Empirical Evaluation

1 Introduction

Writing and reading source code is the daily business of software developers. Whenever the behavior of a program is not well understood or a program does not behave as expected, then an interactive debugger becomes an important tool. In interactive debugging, first concrete input values must be found that bring program execution to a point of interest, for example, by setting suitable breakpoints. Now the developer interactively controls execution and studies each execution step until the program behavior is fully understood.

Suitable input values are sometimes provided by failed test cases or by bug reports. In general, however, it can be challenging to determine input conditions under which the code under inspection exhibits faulty behavior. Another limiting factor in conventional interactive debugging is the fact that only one particular execution path is inspected per debugging session. To inspect a different execution path, debugging needs to start over with different input values.

In addition to debugging and testing, source code can be studied in a (static) code review. Goals of a review include to find defects or to improve design and code quality. A review can be performed either by a team or by a single person. An important category of team review is an inspection [1,2] where people with different roles study the source code according to aspects defined by their role. Even a single developer can review source code as part of his or her personal software process (PSP) [3,4] to ensure that he or she is satisfied with the achieved quality. Checklists are often used to guide a review and to define the criteria under which the source code is reviewed.

Reviews can be performed on all kinds of documents without the need for tool support. However, we believe that symbolic execution [5,6,7,8] nurtures program understanding and that its use for bug finding is promising. To validate this claim we take an empirical approach that is standard in experimental software engineering [9]. We compare source code reviews with and without having a symbolic execution tree available. For tool support we use the Symbolic Execution Debugger (SED) [10], an Eclipse extension for interactive symbolic execution of Java into which any symbolic execution engine for Java can be integrated. In our work we use KeY [11] as the underlying symbolic execution engine.

The SED visualizes a symbolic execution tree representing all possible behaviors of a given program until a certain point. The SED's visual output is a tree where each node represents an execution step of the program under inspection. The user can interact with the visualization, for example, one can inspect and visualize the symbolic state at an execution point to help comprehension of program behavior. Standard symbolic execution explores only finite fragments of an execution which is a serious limitation in the presence of loops with symbolic bounds, method calls with unknown implementations, and recursive calls. The SED is able to process formal specifications in the form of method contracts and loop invariants during symbolic execution. This guarantees that the full program behavior is explored for *any* possible execution path of the program [12].

The purpose of our experiment is to evaluate the effectiveness and efficiency of a source code review with and without the SED. Traditional debugging is explicitly allowed in a direct code review (DCR) without the SED. The experiment is run from the research perspective to find out if the SED significantly improves the review quality. During the evaluation Java source code is shown to the participants and they are asked questions about it to measure their performance. Each code example realizes a small, but functionally complete program and is inspired by the literature or other interesting problems. The expected behavior is always described by comments and sometimes additionally by Java Modeling Language (JML) [13] specifications. The questions asked are a form of checklist used to review the source code. The experiment was performed with engineers at Bosch Engineering GmbH and was announced in public on the KeY website, also on the JML and KeY mailing lists. We summarize the scope of the experiment:

Analyze *code review with and without the SED*
for the purpose of *evaluation*
with respect to *effectiveness and efficiency*
from the point of view of the *researcher*
in the context of *Java developers in industry and research.*

The paper is organized as follows: Sect. 2 discusses related work. Then we describe the planning and setup of the experiment. The measured variables are determined in Sect. 3, the hypotheses to test in Sect. 4. Sect. 5 lays out the design of the experiment and Sect. 6 presents the instrumentation. We conclude experiment planning by discussing threats to validity in Sect. 7. The execution of the experiment is presented in Sect. 8. Collected data are analyzed in Sect. 9 before discussing the results in Sect. 10. We conclude the paper with Sect. 11.

2 Related Work

In [1,2] software inspection is introduced and its impact is confirmed by experience reports. The effectiveness of inspections has been confirmed in many case studies and experience reports, for example, [14,15].

Systems like Gerrit[1] organize the information that is accumulated during a review such as comments. A comparison of early computer support systems for software inspection is in [16]. For some roles assumed during a team review tool support exists. For instance, the moderator can be assisted by decision support facilities [17]. The defect detection step itself is targeted by [18]: they realize learning from the experience encoded in checklists and automatic scans of the source code for violations of checklist items.

To the best of our knowledge, no tool support targeting the interactive defect detection step in a source code review exists. In addition, the effectiveness of reviews is usually confirmed by experience reports whereas in this paper a controlled experiment is performed to draw conclusions with statistical relevance.

3 Variable Selection

First we need to determine and classify the variables of the experiment. We distinguish two kinds of variables: *independent variables* and *dependent variables*. The independent variables are those which can be varied or at least controlled by us and whose influence and effect on the outcome of the experiment we intend to study. Dependent variables are those that are measured during the course of the experiment and which we want to study. A value of an independent variable that was changed during the experiment is called *treatment*.

Table 1 lists the variables of our experiment. The independent variables which can be varied by us are M (with treatments SED and DCR) and S (with the six code examples to review). The subset of independent variables that are merely *controlled* are called $E_{Java|JML|SE|SED}$. Their values provide an answer to the question about a participant's experience level. The separation between less and more than two years is made to separate beginners from experienced users assuming that this is roughly the time needed to master Java, JML and symbolic execution well enough for the evaluation. As the SED is rather new, it is assumed that participants have not much experience with it. For this reason, the separation between beginners and experienced users is set to one year.

[1] www.gerritcodereview.com

	Name	Value domain	Description
Independent Variable	M	{SED, DCR}	The compared methods
	S	{`BankUtil`, `MathUtil`, `IntegerUtil`, `Stack`, `ValueSearch`, `ObservableArray`}	The reviewed source code example and related questions
Controlled Variable	E_{Java}	$\{none, < 2, \geq 2\}$	Years of experience with Java
	E_{JML}	$\{none, < 2, \geq 2\}$	Years of experience with JML
	E_{SE}	$\{none, < 2, \geq 2\}$	Years of experience with symbolic execution
	E_{SED}	$\{none, < 1, \geq 1\}$	Years of experience with SED
Dependent Variable	Q_{tm}	Integer	Number of correctly answered questions per treatment tm of M
	QS_{tm}	Real	Correctness score per treatment tm of M
	C_{tm}	Integer	Confidence score per treatment tm of M based on Q
	CS_{tm}	Real	Confidence score per treatment tm of M based on QS
	T_{tm}	Integer	Time needed to answer questions of a treatment tm of M in seconds

Table 1: Variables of the experiment

The dependent variables are used to quantify *efficiency* and *effectiveness* of the different methods. Efficiency is measured by the time T_{tm} spend to answer the questions using method $tm \in M$. Effectiveness is measured in the number of correctly answered questions and the confidence in the given answers. Questions are single or multiple choice questions to enable an automatic analysis. Each question lists a number of correct and wrong answers from which the participant has to choose. For a given method tm, a multiple choice question is answered correct (measured by Q_{tm}) if all and only the correct answers are selected.

A correctness score is used to give credit for partially correct answers. The correctness score $QS_{tm} = \sum_{q \in tm} qs(q)$ is the sum of the scores over all questions of treatment tm. For a single question q the score $qs(q)$ is defined as:

$$qs(q) = \begin{cases} \frac{\#corSelAnsw(q) - \#wrgSelAnsw(q)}{\#corSelAnswers} & \text{if } \#corSelAnsw(q) > \#wrgSelAnsw(q) \\ \frac{\#corSelAnsw(q) - \#wrgSelAnsw(q)}{\#wrgSelAnswers} & \text{if } \#corSelAnsw(q) \leq \#wrgSelAnsw(q) \end{cases}$$

Intuitively, the question score $qs(q)$ is the difference between the number of selected correct answers $\#corSelAnsw(q)$ and the number of selected wrong answers $\#wrgSelAnsw(q)$ of question q. But each question has a different number of correct (wrong) answers. To achieve comparability between questions, the difference between correct and wrong answers is divided by the total number of correct (wrong) answers.

For each question we asked the participants about the confidence in their answers. Available confidence levels are *sure (My answer is correct!)*, *educated guess (As far as I understood the content, my answer should be correct.)* and *unsure (I tried my best, but I don't believe that my answer is correct.)*.

	Correct answer of q or $qs(q) > 0$	Wrong answer of q or $qs(q) \leq 0$
Sure	2	-2
Educated Guess	1	-1
Unsure	-1	1

Table 2: Confidence ratings $c(q)$, $cs(q)$ of a single question q

For each question q (each question score $qs(q)$) a confidence rating $c(q)$ ($cs(q)$) is computed according to Table 2. A participant who is sure the answer is correct when it is actually correct (the confidence score is positive) obtains maximal points. If the answer is wrong, but the participant was sure that it is correct (the confidence score is positive), he or she gets the lowest possible rating. If the answer is based on an educated guess, which is weaker than certainty, the participant gets less (or loses less) points. If the participant is unsure and thinks the answer is wrong, and it is actually wrong (the confidence score is not positive), then still one score point is assigned, because the intuition was correct. If the participant thinks the answer is wrong but it is right (the confidence score is positive), he or she loses one score point for the same reason. Finally, the confidence score $C_{tm} = \sum_{q \in tm} c(q)$ is the sum of the confidence ratings over all questions answered for treatment tm. The confidence score based on question scores $CS_{tm} = \sum_{q \in tm} cs(q) \cdot qs(q)$ takes partially correct answers into account.

4 Hypothesis Formulation

From our experiment we want to gain statistical evidence that the SED increases effectiveness and efficiency of a code review. To this extent we formulate for each dependent variable (see Table 1) an alternative hypothesis $H_{1_Q} - H_{1_T}$ (see Table 3). As usual [9] the claims of these hypotheses are confirmed by ruling out each corresponding null hypothesis $H_{0_Q} - H_{0_T}$.

5 Choice of Experiment Design Type

An important design decision of the experiment is to ensure that participants benefit from their participation. To achieve this, each participant uses both methods resulting in a paired comparison design (see Table 4). In case participants are unfamiliar with the SED, this allows them to try it out and to decide whether it is helpful for their work.

We applied the general experiment design principles *randomization*, *blocking* and *balancing* [9] to avoid biases, to block out effects in which we are not interested in, and to simplify and strengthen hypothesis testing. We randomized the order of code examples presented to the participants to avoid that, for instance, differences in the level of difficulty can influence the result of the experiment.

Name	Hypothesis	Def. of $\mu_{V_{tm}}$ for dependent variable V, treatment tm
$H0_Q$	$\mu_{Q_{SED}} = \mu_{Q_{DCR}}$	with $\mu_{Q_{tm}} = \frac{Q_{tm}}{\#questionsOfTmnt} \in \{x \in \mathbb{Q} \vert 0 \leq x \leq 1\}$
$H0_{QS}$	$\mu_{QS_{SED}} = \mu_{QS_{DCR}}$	with $\mu_{QS_{tm}} = \frac{QS_{tm}}{\#questionsOfTmnt} \in \{x \in \mathbb{Q} \vert 0 \leq x \leq 1\}$
$H0_C$	$\mu_{C_{SED}} = \mu_{C_{DCR}}$	with $\mu_{C_{tm}} = \frac{C_{tm}}{\#questionsOfTmnt} \in \{x \in \mathbb{Q} \vert -2 \leq x \leq 2\}$
$H0_{CS}$	$\mu_{CS_{SED}} = \mu_{CS_{DCR}}$	with $\mu_{CS_{tm}} = \frac{CS_{tm}}{\#questionsOfTmnt} \in \{x \in \mathbb{Q} \vert -2 \leq x \leq 2\}$
$H0_T$	$\mu_{T_{SED}} = \mu_{T_{DCR}}$	with $\mu_{T_{tm}} = \frac{T_{tm}}{timeOfAllTmnts} \in \{x \in \mathbb{Q} \vert 0 \leq x \leq 1\}$
$H1_Q$	$\mu_{Q_{SED}} > \mu_{Q_{DCR}}$	
$H1_{QS}$	$\mu_{QS_{SED}} > \mu_{QS_{DCR}}$	
$H1_C$	$\mu_{C_{SED}} > \mu_{C_{DCR}}$	
$H1_{CS}$	$\mu_{CS_{SED}} > \mu_{CS_{DCR}}$	
$H1_T$	$\mu_{T_{SED}} < \mu_{T_{DCR}}$	

Table 3: Hypotheses

	Example 1	Example 2	Example 3	Example 4	Example 5	Example 6
SED	Subject$_n$	Subject$_n$	Subject$_n$	Subject$_{n+1}$	Subject$_{n+1}$	Subject$_{n+1}$
DCR	Subject$_{n+1}$	Subject$_{n+1}$	Subject$_{n+1}$	Subject$_n$	Subject$_n$	Subject$_n$

Table 4: Paired comparison design

The first three code examples are always to be reviewed using the same method and the next three code examples with the other one (recall that we have six code examples to review). The decision which method is used for the first three code examples is random. This avoids multiple switches between methods which could confuse the participant. Additionally, a participant who is not familiar with a reviewing method has more experience in the later tasks. The server used to collect evaluation results guarantees that all possible permutations of example orders will be evaluated equally often as well as all other constraints.

The performance of the participants may depend on their experience with Java which is used for blocking. Grouping the participants according to their experience level with Java allows us to interpret the results for the different groups separately. Balancing is automatically achieved by the chosen design, because each participant uses both methods and reviews all six code examples. Thus the number of participants is the same for each treatment. The number of participants who reviewed a source code example might be not balanced in case participants decided not to review all of them.

6 Instrumentation

We did not want to limit the group of participants to people familiar with symbolic execution, JML or the SED. The only hard requirement on the participants was basic knowledge in Java (or a similar language). To accommodate this decision, the evaluation had to be self-explanatory. We achieved this by showing three instructional videos: an introduction to the evaluation itself and one to each method. A brief textual introduction was given on how to read and write JML specifications (the JML specifications used in the evaluation do not use advanced concepts).

During the evaluation participants reviewed source code with and without using the SED. As the SED is available within Eclipse, the evaluation itself is implemented as an Eclipse wizard which is opened in an additional window so that the functionality of Eclipse is not impaired.

The evaluation setup consists of two phases during which information is collected and sent to the server. The first phase collects background knowledge on the participant and determines the order of code examples and the method assignment. The actual evaluation is performed in the second phase. A participant who cancels the evaluation during the second phase is asked to send partial results to the server. When that participant opens the evaluation wizard the next time, he or she is offered to recover the previous state to continue the already started evaluation. The evaluation workflow in detail is as follows:

1. Initialization Phase
   (a) *Terms of Use*: Terms of use need to be accepted.
   (b) *Background Knowledge*: Information about background knowledge is gathered (Java, JML, symbolic execution, SED).
   (c) *Extent*: Participant chooses between reviewing four or six code examples.
   (d) *Sending Data*: Data is sent and order of code examples is received.
2. Evaluation Phase
   (a) *Evaluation Instructions*: A video explaining how to answer questions.
   (b) *JML*: A textual documentation introducing the features of JML necessary for the evaluation.
   (c) *SED/DCR Instruction*: A video explaining needed features and best practices to review a code example with and without the SED (depending on order).
   (d) *Code Examples 1 and 2 (and 3)*: The first two/three code examples and the questions that test the understanding.
   (e) *The complementary SED/DCR Instruction*: The remaining video.
   (f) *Code Examples 4 and 5 (and 6)*: As above
   (g) *Feedback about SED and Evaluation*: The participant is asked to rate the usefulness of SED features (mentioned in the videos).
   (h) *Sending Data and Acknowledgment*: Data is sent and the successful completion of the evaluation is acknowledged.

We summarize the six code examples[2] to be reviewed and the defects the participants were supposed to identify:

`BankUtil` implements a stair-step table lookup, inspired by [19, page 427]. The source code does not adhere to its documentation because a wrong value is returned for an age above 35.

[2] Available at www.key-project.org/eclipse/SED/ReviewingCodeEvaluation/examples

`IntegerUtil` is inspired by [20, page 255]. The challenge is to find the mistake that in one case `y` is returned instead of `x`.

`MathUtil` can throw an uncaught `ArrayIndexOutOfBoundsException` caused by overflow. Such an overflow was present in Java's binary search implementation, see bug item JDK-5045582[3].

`ValueSearch` contains unreachable code, namely statement **`return false`**. The surrounding method `accept` is only called within the loop in method `search`. The loop guard already ensures that the then branch of `accept` is never taken. Such issues are difficult to detect by test case generation tools based on symbolic execution: when unrolling the loop further there might be a future loop iteration in which the then branch will be taken after all. Further, there is a defect in method `find`. The parameter `value` is never used. Finally, in case the `array` is **`null`**, an uncaught `NullPointerException` is thrown.

`ObservableArray` is inspired by [21, page 265]. The class constructor and the method `setArrayListeners` behave according to the documentation. But method **`set`** has several problems: (i) in case the index is outside the array bounds an uncaught `ArrayIndexOutOfBoundsException` is thrown, (ii) in case the element is not compatible to the component type of the array an `ArrayStoreException` is thrown, (iii) not all observers that exist at call time are informed about the change in case an observer changes the available observers during the event, and (iv) if an observer sets `arrayListeners` to **`null`**, an uncaught `NullPointerException` is thrown.

`Stack` is inspired by [21, page 24]. The first constructor which creates a new stack throws an uncaught `NegativeArraySizeException` if the maximal size is negative. The second constructor which clones a stack does not behave as documented, because the clone shares the same `elements` array and is thus not independent. Further, it throws a `NullPointerException` if the existing stack is **`null`**. Method `push` behaves as documented, but method `pop` does not remove the top element from the stack which violates the class invariant.

We designed questions and answers for each code example according to the following schema: The participant is asked (i) for each method/constructor to review whether the implementation behaves as documented, (ii) in case a class invariant is present to review whether it is preserved/established, (iii) for methods with a return value to choose which claims from a given list are valid and (iv) to determine which statements can be reached. In case a participant answers that the source code does not behave as documented, all applicable reasons why this is the case from a predefined list of potential causes had to be chosen. In case an undocumented exception is thrown, the participant is asked which one. When a participant answers that an invariant is violated, the invalid parts have to be identified. Entering free text with additional reasons is always possible.

For the code examples to be reviewed with the SED we asked participants to rate the helpfulness of the symbolic execution view. For DCR, we asked whether the conventional Java debugger was used and if so, how helpful debugging was.

[3] bugs.java.com/bugdatabase/view_bug.do?bug_id=5045582

In every review using the SED the fully explored symbolic execution tree is shown. Interactive symbolic execution is avoided to ensure that all participants review the source code under the same conditions. In DCR a main method with a call of the code to review is provided. Required values of parameters are missing and have to be provided by the participant. The code is there to help participants without or little Java experience to start a debugging session.

7 Validity Evaluation

In this section we discuss threats to the validity of our experiments and the drawn conclusions. For each threat we provide a mitigation strategy.

"*Conclusion validity* concerns the statistical analysis of results and the composition of subjects." [9, p. 185] The hypotheses of this experiment are tested with well known statistical techniques. Threats to conclusion validity are the low number of samples and the quality of the answers. Subjects may fake answers to compromise the experiment. However, several participants are people we know (colleagues, project partners, students, etc.), in addition, some were monitored during the evaluation. We consider the motivation of subjects to compromise the experiment to be very low.

"*Internal validity* concerns matters that may affect the independent variable with respect to causality, without the researcher's knowledge." [9, p. 185] Reviewing source code is time intensive and the participation time is up to 90 minutes; hence, participants may get tired or bored. There is a risk that participants lack motivation and thus answer questions not seriously. However, participation is voluntary and can be done at any time convenient for the subject.

Maturation is a threat to internal validity as each method is applied to three code examples and participants may learn how to use it, which is desired. We consider this non-critical as randomization is applied to the order of treatments. Participants have most likely no experience with the SED which is uncritical as the instrumentation introduces all relevant features and best practices of both review methods. There might be a threat that some participants are not willing to learn how to use the SED. However, SED is designed to support the reviewing process. A potential bias about the experience with the SED would only contribute against our claim that SED improves upon efficiency and effectiveness and not in its favor. Other threats are considered to be uncritical.

"*Construct validity* concerns generalisation of the experiment result to concept or theory behind the experiment." [9] A threat to construct validity is that the chosen code examples are not be representative. To mitigate the code examples were chosen from the standard literature or from widely discussed problems. Other threats to construct validity are considered uncritical. Even though a participant might guess the expected outcome from the general motivation of the experiment (a comparison between a code review with and without the SED) and that SED is a new tool, we consider it uncritical as the exact hypotheses and related measurements are unknown to them. In addition, the participants do not have any advantage or disadvantage from the outcome of the experiment.

“*External validity* concerns generalisation of the experiment result to other environments than the one in which the study is conducted.” [9, p. 185] A threat to external validity is that the source code is kept to a minimum—in some cases only one method. This is required to reduce the time participants need to review the source code and to read its documentation. Real Java code is much more complex. On the other hand, symbolic execution with specifications is modular, only a small part of the source code needs to be considered. The participants are selected randomly and their experience varies from none to expert. Consequently, the selection of participants is not a threat to external validity.

We state that there are threats to the validity of the experiment, and hence, the drawn conclusions are valid within the limitations of the threats.

8 Execution

The experiment started in September 2015 with the staff of the Software Engineering group at TU Darmstadt. It included students, PhD students and post-docs. Each participant was monitored during the evaluation to improve the instructions and answers. Questions and answers remained stable after the first participant. For this reason, the results of the first participant were excluded, but the others were kept.

The evaluation was then performed with engineers at Bosch Engineering GmbH. None of the engineers use Java in their daily business and the Java experience was none or less then a year. However, participants were interested in new methods and liked to try out the SED. In total, 11 engineers started the evaluation. One participant canceled the evaluation and three participants did not submit the answers. Additionally, one participant did not follow the instructions and used the SED for all code examples. Consequently, the results of six participants are considered as valid.

The evaluation went public in October 2015. It was announced on the KeY website, as well as the KeY and JML mailing lists. The evaluation is available as a preconfigured Eclipse product. Installation instructions and download links are available on the KeY website.[4] Main steps are to download and run the Eclipse product and to perform the evaluation. An installation is not required, the participants' system is unaffected. Until mid January 2016, 27 participants started the evaluation, of these 19 completed it. Twelve of the participants were monitored during the evaluation (with their approval).

The distribution of background knowledge of participants is shown in Fig. 1. The experience of participants with knowledge in JML and symbolic execution is fairly distributed. Most of the participants had more than two years of Java experience and none with the SED.

The relation between the Java and the SED experience is shown in Table 5. It shows that all participants with SED experience have at least two years of Java experience.

[4] www.key-project.org/eclipse/SED/ReviewingCode.html

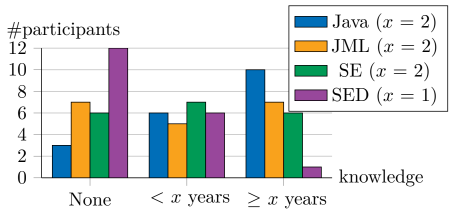
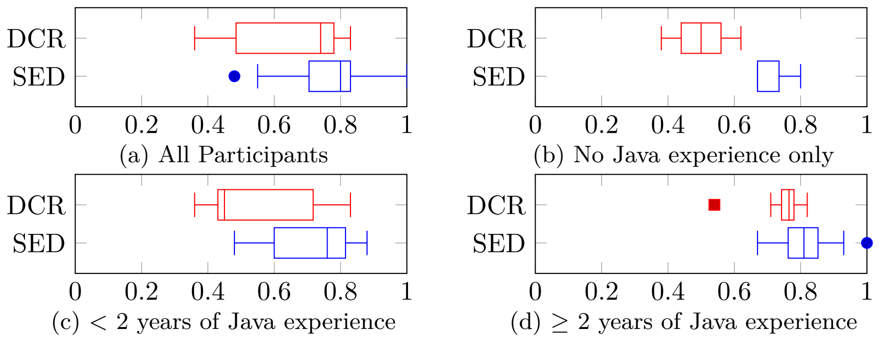


Fig. 1: Knowledge of participants

		SED		
		None	< 1 year	≥ 1 year
Java	None	3	0	0
	< 2 years	6	0	0
	≥ 2 years	3	6	1

Table 5: Java vs SED experience



(a) All Participants

(b) No Java experience only

(c) $<$ 2 years of Java experience

(d) $\geq$ 2 years of Java experience

Fig. 2: Correct Answers

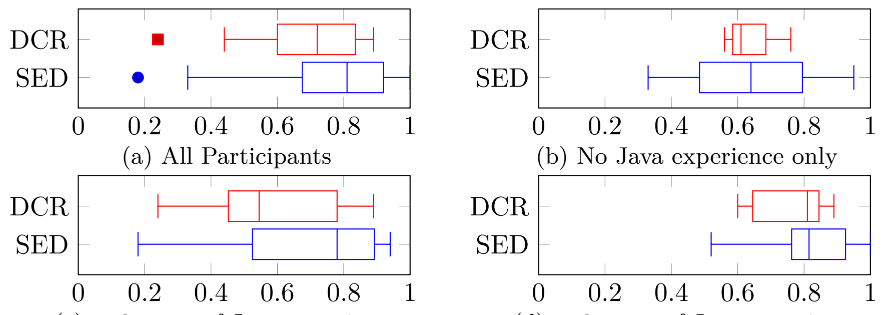
9 Analysis

Now we visualize the collected data to get a first impression about their distribution and to identify possible outliers, before we test our hypotheses. Interpretation and discussion of the results is done in Sect. 10.

To visualize data we use boxplots (Figs. 2–6). The middle vertical bar in the rectangle of a boxplot indicates the median of the data. The left border represents the lower quartile lq and the right border represents the upper quartile uq. The left and right whiskers indicate the theoretical bounds of the data assuming a normal distribution. Data points outside the whiskers are outliers. The left whisker is defined as $lq - 1.5\,(uq - lq)$ and the right whisker as $uq + 1.5\,(uq - lq)$. Additionally, whiskers are truncated to the nearest existing value within the bounds to avoid meaningless values. The constant 1.5 is chosen following [22].

The boxplots in Fig. 2 show the distribution of the correctly answered questions with the lower bound 0 meaning that no question was answered correctly and the upper bound 1 attained when all answers were correct. The boxplots in Fig. 2a show the distribution for all participants for the treatments SED and DCR, whereas Figs. 2b–2d show the distribution of correct answers broken down to different levels of Java experience. In all boxplots, the achieved correctness is better using SED.

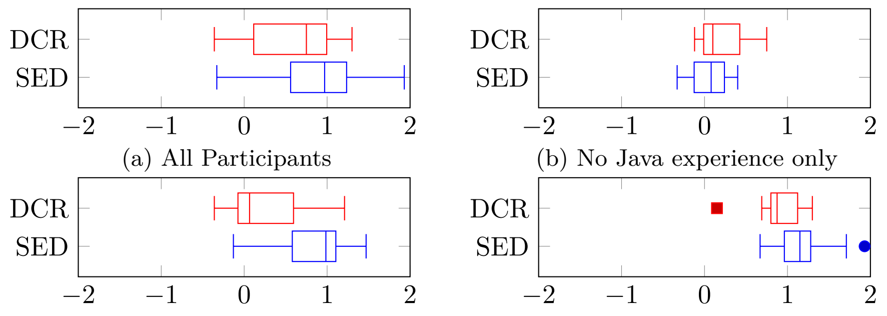
The distribution of the measured correctness scores (taking partially correct answers into account, see Sect. 3) is shown in Fig. 3. In each case the correctness



(a) All Participants (b) No Java experience only

(c) $<$ 2 years of Java experience (d) $\geq$ 2 years of Java experience

Fig. 3: Correctness Score



(a) All Participants (b) No Java experience only

(c) $<$ 2 years of Java experience (d) $\geq$ 2 years of Java experience

Fig. 4: Confidence Score

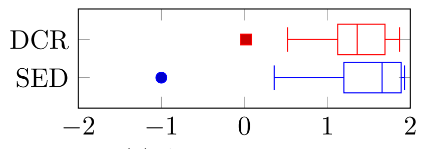
score is at least as high as in the distribution of correct answers (Fig. 2). Again, the achieved correctness is always better using SED. In the class of participants without Java experience, the achieved correctness using SED varies considerably.

The distributions of the confidence score are similar to those of the correctness score. As Figs. 4–5 show, the confidence is higher with SED. An exception is the class of participants without Java experience which achieved slightly better confidence without using the SED. In each case the confidence score increases (the boxplot "moves right") when taking partially correct answers into account.

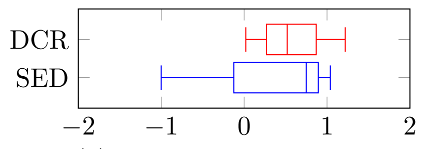
The measured time[5] is shown in Fig. 6. A value of 1 means that a participant spent all the time using one reviewing method. The opposite is 0 when a participant spent no time with a reviewing method. In all boxplots, the review time is less using SED.

The null hypotheses of Table 3 can be rejected without assuming a normal distribution using a one sided *Wilcoxon Signed Rank Test* or a one sided *Sign Test*, see [9]. As basis for the tests we used the results of *all* participants and did not test each experience level separately, because there are too few participants

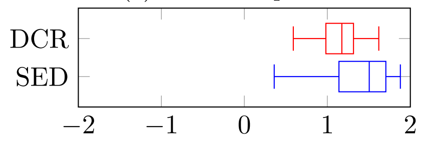
[5] The times measured for three of the participants were invalid and, therefore, excluded.



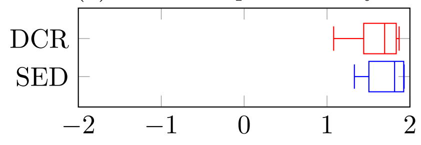
(a) All Participants



(b) No Java experience only

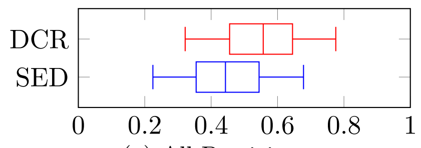


(c) $<$ 2 years of Java experience

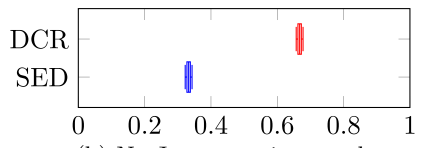


(d) $\geq$ 2 years of Java experience

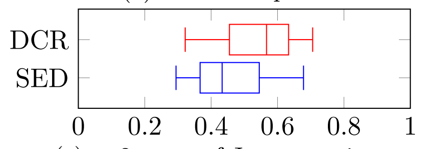
Fig. 5: Confidence Score of Partially Correct Answers



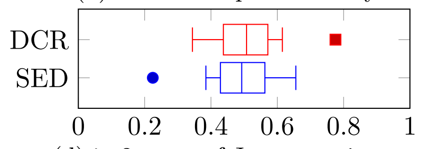
(a) All Participants



(b) No Java experience only



(c) $<$ 2 years of Java experience



(d) $\geq$ 2 years of Java experience

Fig. 6: Time

in each class to apply any test method. The significance level is set to 0.05 meaning that there is a 5% chance at which a hypotheses is wrongly rejected. The results of the tests are shown in Table 6.

All tests reject the correctness-related hypothesis H_{0_Q} and the Wilcoxon Signed Rank Test rejects in addition the confidence-related hypothesis H_{0_C}. Looking at the promising boxplots we expect to reject the other hypotheses as well once more people participate in the experiment.

10 Interpretation

10.1 Correctness of Answers

The analysis of our experiment permits to conclude that participants performed significantly better in code reviewing tasks when the SED is used. Also the participant's confidence in the given answers is higher when the SED is involved.

Hypothesis	Wilcoxon Signed Rank Test			Sign Test	
	W-value	p-value	rejected	p-value	rejected
H_{0_Q}	176,5	0,0002	true	0,0012	true
$H_{0_{QS}}$	119,5	0,1660	false	0,5000	false
H_{0_C}	151	0,0115	true	0,0835	false
$H_{0_{CS}}$	119	0,1762	false	0,1796	false
H_{0_T}	82	0,1147	false	0,1509	false

Table 6: Results from the One Sided Tests with $\alpha = 0.05$

To answer the question whether the SED performs universally better or only in a specific code reviewing situation, we look at how often an expected correct answer was given using each of the methods. We summarize now the results.[6]

Questions whether the implementation behaves as documented and whether a constructor establishes an invariant were answered more often correctly in a direct code review without using the SED, particularly, in the examples `Stack` and `ValueSearch`. In traditional debugging sessions the explored execution path shows only symptoms of a defect. This is also true for symbolic execution paths visualized by the SED. To locate the defect in the program logic causing the observed symptom (at runtime) the source code needs to be reviewed. This relation between debugging and reviews is also discussed in [3]. However, the SED shows source code and symbolic execution tree at the same time.

Questions about the reason for a misbehavior were only answered by participants when they had realized before that something is wrong. As participants using the SED often failed to identify a problem, it is not surprising that the correct reasons were also more often identified in a direct code review. Interestingly, participants identified more often that an exception is thrown and the correct type of the thrown exception using the SED. Thrown exceptions are explicitly visualized in the SED by special symbolic execution tree nodes.

In addition to the symbolic execution tree, the SED highlights the statements that were reached during symbolic execution. This seems to be helpful as the reachable statements were more often correctly identified using the SED.

Participants were also asked to identify valid claims about a method's return value from a given list of options. The SED visualizes symbolic values that can be potentially returned as part of the method return node. In the `MathUtil` and `ValueSearch` example the correct answers were more often selected using SED, whereas in the `BankUtil`, `Stack` and `IntegerUtil` example the direct code review achieved better results.

In total correct answers were more often identified in the classes of participants with Java experience using the SED. In the class without Java experience correct answers were more often selected in a direct code review. Taking into account that only three participants are in that class and that many questions

[6] Detailed results are available at www.key-project.org/eclipse/SED/ReviewingCodeEvaluation/results

are never answered with both methods, no meaningful conclusions can be drawn. Overall, a few more correct answers were identified in a direct code review.

We conclude that the SED helps to answer questions about aspects that are represented in the symbolic execution tree. According to the given answers, the SED seems not to increase the understanding of the program logic.

10.2 Perceived Usefulness of Features

For the SED, nearly all participants considered the symbolic execution tree view and the highlighting of source code reached during symbolic execution as helpful or very helpful. The variable view used to visualize the symbolic state of a symbolic execution tree node as well as the properties view showing additional information like path conditions were in many cases not used by the participants. But those who used them considered them mostly as (very) helpful.

Participants had the opportunity to use the Java debugger in a direct code review. But the source code required to do so was often not written and if it was, its helpfulness varies from very to somewhat helpful.

Several participants provided constructive suggestions for improvement: first, source code reached in the currently selected execution path as well as the corresponding symbolic execution nodes should be highlighted. Following further suggestions we plan to visualize additional information, including the path condition and selected memory locations within symbolic execution tree nodes.

Participants were also asked whether they prefer a code review with or without using the SED (as the SED is designed to support a review and not to replace it). Our evaluation shows, that the SED effectively helps to discover information about feasible execution paths. Studying a symbolic execution tree, however, is not sufficient to understand the program logic. It is, therefore, not surprising that roughly two thirds of the participants would consider the SED depending on the nature of the given source code.

11 Conclusion

We described an experiment comparing the effectiveness and efficiency of a code review with and without using the SED. The result provides statistically significant evidence for increased effectiveness when a symbolic execution tree view is used during code reviews.

Very few formal methods are evaluated with user studies. Usually, perceived advantages are simply claimed or, in the best case, validated against a case study. As far as we know, this is the first comparative experimental user case study of its kind. We strongly believe that more work like it is needed to convince industrial stakeholders about the value of formal methods. The advantages of case studies are obvious: effectivity claims get empricially substantiated which is a solid basis for decision makers. In addition, researchers obtain valuable feedback from the field and can make better usage of resources.

One must also clearly state that user studies mean a lot of work. Statistical techniques most researchers in formal methods are unfamiliar with have to be mastered. Designing and carrying out the actual study is very time consuming. To sustain further studies, participants must be reimbursed. But we think that the increased credibility of our claims and the insights we gained are well worth the effort.

Acknowledgment: We thank all participants of the evaluation for their valuable time and feedback.

References

1. Fagan, M.E.: Design and code inspections to reduce errors in program development. IBM Systems Journal **15**(3) (1976) 182–211
2. Fagan, M.E.: Advances in software inspections. IEEE Transactions on Software Engineering **12**(7) (1986) 744–751
3. Humphrey, W.S.: A Discipline for Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
4. Humphrey, W.S.: Introduction to the Personal Software Process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
5. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT—A formal system for testing and debugging programs by symbolic execution. ACM SIGPLAN Notices **10**(6) (June 1975) 234–245
6. Burstall, R.M.: Program proving as hand simulation with a little induction. In: Information Processing '74. Elsevier/North-Holland (1974) 308–312
7. Katz, S., Manna, Z.: Towards automatic debugging of programs. In: Proc. Intl. Conference on Reliable Software, Los Angeles, ACM Press (1975) 143–155
8. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7) (July 1976) 385–394
9. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B.: Experimentation in Software Engineering. Springer (2012)
10. Hentschel, M., Bubel, R., Hähnle, R.: Symbolic execution debugger (SED). In Bonakdarpour, B., Smolka, S.A., eds.: Runtime Verification, 14th Intl. Conference, RV, Toronto, Canada. Volume 8734 of LNCS., Springer (2014) 255–262
11. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. Number 4334 in Lecture Notes in Computer Science. Springer-Verlag (2007)
12. Hentschel, M., Hähnle, R., Bubel, R.: Visualizing unbounded symbolic execution. In Seidl, M., Tillmann, N., eds.: Proceedings of Testing and Proofs (TAP) 2014. LNCS, Springer (July 2014) 82–98
13. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML Reference Manual. (May 31, 2013) Draft Revision 2344.
14. Doolan, E.P.: Experience with Fagan's inspection method. Softw. Pract. Exper. **22**(2) (February 1992) 173–182
15. Russell, G.W.: Experience with inspection in ultralarge-scale development. IEEE Softw. **8**(1) (January 1991) 25–31
16. Macdonald, F., Miller, J.: A comparison of computer support systems for software inspection. Automated Software Engg. **6**(3) (July 1999) 291–313

17. Miller, J., Macdonald, F., Ferguson, J.: Assisting management decisions in the software inspection process. Inf. Technol. and Mgmt. **3**(1-2) (January 2002) 67–83
18. Nick, M., Denger, C., Willrich, T.: Experience-based support for code inspections. In Althoff, K., Dengel, A., Bergmann, R., Nick, M., Roth-Berghofer, T., eds.: Professional Knowledge Management, Third Biennial Conference, Kaiserslautern. Volume 3782 of LNCS., Springer (2005) 121–126
19. McConnell, S.: Code Complete, Second Edition. Microsoft Press, Redmond, WA, USA (2004)
20. Zeller, A.: Why programs fail—A guide to systematic debugging. 2nd edn. Elsevier (2006)
21. Bloch, J.: Effective Java. 2nd edn. Prentice Hall, Upper Saddle River, NJ, USA (2008)
22. Frigge, M., Hoaglin, D.C., Iglewicz, B.: Some Implementations of the Boxplot. The American Statistician **43**(1) (1989) 50–54