Tool support for teaching Hoare logic[⋆]

Tadeusz Sznuk and Aleksy Schubert

Institute of Informatics, University of Warsaw
ul. Banacha 2, 02–097 Warsaw, Poland
[tsznuk,alx]@mimuw.edu.pl

Abstract. Currently, software verification is perceived as an overly difficult and hard to understand task. This image can be changed through effective instruction of prospect programmers during their studies. Teaching Hoare logic is a process that can be made more appealing to students if appropriate tools are employed. Having an environment tailored to the style and content of a particular lecture is especially beneficial. We argue that current state of related technology is such that it is possible to implement a tool that can be adapted to a particular style of classes with manageable effort. We illustrate our point by showing a tool called HAHA (Hoare Advanced Homework Assistant) and presenting its effectiveness in teaching Hoare logic.

1 Introduction

It is important to shape developers' culture so that more refined methods of software cultivation that involve reasoning on programs are well understood and perceived as practical. One important way to achieve this is to present these methods to future programmers during their curriculum in an attractive, modern way. The experience of our faculty, most probably shared by other places, shows that typical student sees the subject of Hoare logic as tedious, boring, obscure and, above all, utterly impractical. Its principles are, in general, poorly understood, and the whole concept is believed by many to be no more than a weird part of computer science history even though it plays a fundamental role in the dynamically growing field of program verification.

It is doubtless that this sorry state of affairs has multiple reasons. However there is one issue, which contributes greatly to this problem and which we hope to at least partially resolve. It is the issue of tools that are used to teach Hoare logic. In many cases, said tools consist of a pen and few sheets of paper. These respectable instruments have proven to be an indispensable educational aid over centuries. Still, it is a tedious task to verify a while program on a sheet of paper. It is much harder than convincing oneself about correctness of the very same program in a less formal way. What is worse, the chances of making a mistake can be argued to be similar for both approaches. The fact that the whole logical inference is performed on paper, without any aid from the computer (save,

⋆ This work was partly supported by Polish government grant N N206 493138.

perhaps, in the matter of typesetting) further enforces the view that Hoare logic is impractical. Tediousness of the verification process is also an issue for lecturers, as it turns authoring and grading assignments into truly formidable tasks.

The most direct way of tackling the troubles we just described is to use automated formal verification software to facilitate checking the correctness of Hoare programs. Systems such as ESC/Java [7,24], Frama-C [5,8], Verifast [18], Microsoft VCC [9], KeY [3], to mention only few, have not yet made their way into the everyday toolbox of an average developer [19], but have proven to be usable in the verification process of software projects.

The design of these powerful systems is geared towards applications in large scale software development, rather than education, and these two goals are, at least in some aspects, conflicting. One of the reasons is that rich feature set of languages such as C or Java (e.g. the complexity of heap handling in Verifast [18] is an additional burden in getting the basic ideas through), as well as the need to minimise the load that the verification system puts on a programmer during creation and evolution of real code, invariably result in complexity of the underlying formal logic, which limits its use in an introductory course.

There are tools such as KeY-Hoare [6] or Why3 [13], which are designed with the goal in mind to serve as educational aids, but were started from general purpose verification tools. Unfortunately, variants of Hoare logic which are implemented by them are seldom compatible with these from existing teaching materials of courses in particular faculties. This means pedagogical experience of tutors should be at least to some descent lost when such a tool is adopted for instruction. It should also be noted that high automation they offer can distract students and allow them to solve exercises by trial and error with little regard to the design of the logic. This calls for a tool in which only basic techniques are implemented, but which can easily be adapted to different courses.

One more crucial point here is that the usefulness of the tools for teaching is usually presented through qualitative descriptions. Hardly anyone tries to use statistical methods of quantitative psychology [27] to estimate the impact that introduction of a tool has on the educational process. We address the discussed shortcomings of previous approaches and propose a Hoare logic teaching assistant, HAHA, designed specifically to teach the logic. Moreover, we present a quantitative study of the impact adoption of such tool has on teaching results.

The paper is structured as follows. Section 2 gives an overview of the requirements for the tool and its current features. The experiment concerning the results of teaching students with our tool is presented in Section 3. This is followed by a presentation of related work in Section 4. We conclude in Section 5

2 Presentation of HAHA

Requirements for the tool The basis of our study are the tutoring activities in our faculty. We gathered requirements on a tool to support teaching in the following two scenarios. The first of them involves giving instruction on Hoare logic in a standard undergraduate course on semantics and verification of programs. The second one involves teaching advanced topics in software verification.

As a secondary non-functional requirement on the development we took the constraint that the internal design of the tool must be such that the tool is relatively easy adaptable to other teaching environments.

Teaching Hoare logics We would like the teaching process to be similar to the one that was run before. This means we require the syntax of the programs to be close to original Hoare logic with possible extensions, but in the manner easily digestible by students who are acquainted with programming languages such as Pascal or Java. In addition, we would like the process of verification to give the impression that it is done as part of program development, in particular we would not like to change the environment to the one of interactive prover to assist in discharging verification conditions. Instead we would like the students to give assertions between instructions that are subsequently verified by an automatic theorem prover. In case the prover cannot discharge the supplied conditions we would like to make it possible to give it additional help rather by means of additional axioms than through interactive proof. The latter solution would only be possible if significantly more time was available to instruct students how to do interactive proofs. It is an unavoidable feature of this part of software verification that it requires deep technical fluency in the proving technology [25,15].

Traditionally assignments in our courses suggest students to fill in all assertions in program code. Contrary to the common trend to make only loop invariants obligatory, we decided that we want to force students to fill in all intermediate assertions. This might seem surprising, since the need to write many formulae increases the amount of work necessary to create a verified program. However, we believe that, in the case of a teaching aid, our approach is more beneficial. First, this suggests students to match the assertions with relevant Hoare logic rules and in this way it reinforces the process of teaching the logic. Second, it gives the students a tangible experience of how much information must be maintained at each step of the program to make it execute correctly — the process of making the verification work is also a tangible experience of how it is easy to overlook transformation of some tiny detail in this information packet.

Teaching advanced verification methods Second scenario is to teach students advanced software verification methods. Our faculty has a course on verification methods during which students learn how to use various verification tools. A further step is to teach them how the methods work so that they are able to develop new such methods in future. In this case, only implementation of an existing verification method can give the students the real working understanding of similar methods and can give the feeling on how much effort an implementation of such a procedure requires. For this kind of activity one needs a platform to do experiments with implementations of various verification methods.

Alternatives to development Before we started development of our own tool we reviewed the literature on existing tools (see Section 4). We decided to have a closer look at Why3 [4,13] and KeY-Hoare [6]. We verified small programs with both tools to have an impression of how they match our requirements.

Why3 is a very mature tool. It gives the possibility to work with C or Java code and brings Emacs interface with syntax colouring for the native Why language, WhyML. Still, the tool does not have the possibility to enforce filling in assertions between all instructions. One more obstacle, which makes it less suitable for our purposes is the syntax of WhyML. It is not similar to the languages the students are used to work with so it would present some kind of difficulty for them. We could start instruction with Java or C code, but in the end we would have to expose students to the native syntax. Therefore, we gave this option up.

Key-Hoare is another mature tool that is an adaptation of the very attractive KeY platform to the needs of a Hoare logic course. Still, the logic behind the tool is not pure Hoare logic, but Hoare logic with updates. Therefore, the first obstacle in adoption of the tool was the need to extend the lecture material about Hoare logic to include the updates. Another difficulty with Key-Hoare is that it exposes a student to the KeY prover where a considerable number of logic rules can be applied at each step of verification. A student must be instructed which of them should be used and which of them avoided in typical cases. We believed that we did not have enough time during classes to explain it. At last, the prover environment gave a good impression on which logic rules are available at each point, but it also did not give the impression of a typical software development environment and this was one of our requirements.

After taking these observations, we decided to develop our own, based upon Eclipse, environment called Hoare Advanced Homework Assistant, or HAHA[1]. We describe it in more detail below.

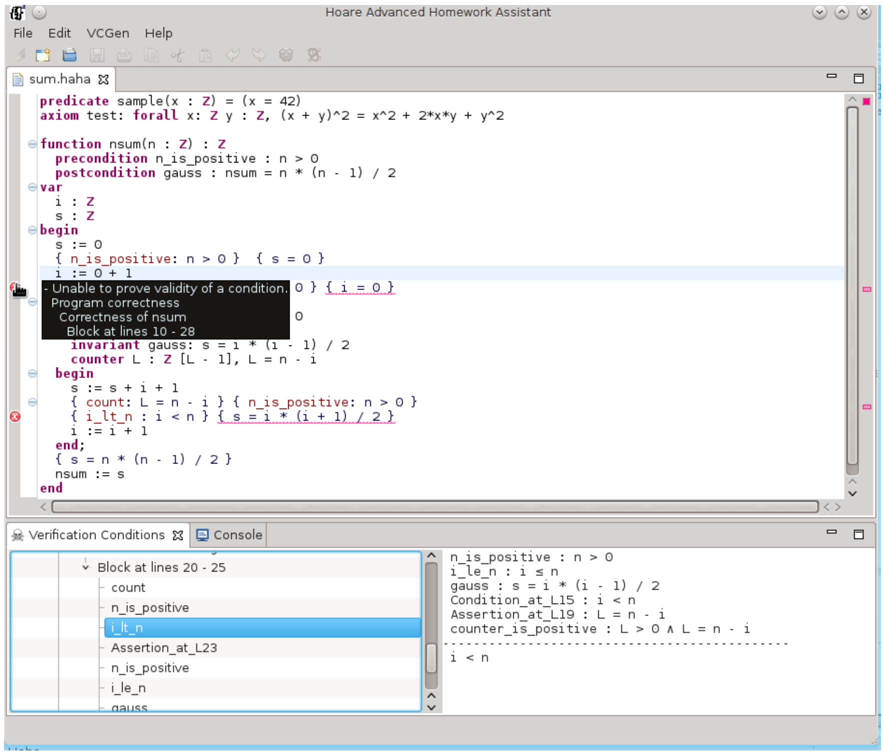
2.1 Overview of HAHA

The user interface of HAHA is presented in Fig. 1. The main pane of the window is filled with the source code of the program one works with. It features an editor for simple while programs and has all features that are expected of a modern IDE, such as syntax highlighting, automated completion proposals and error markers. Once a program is entered, it is processed by a verification conditions generator, which implements the rules of Hoare logic. The resulting formulae are then passed to an automated prover. If the solver is unable to ascertain the correctness of the program, error markers are generated to point the user to the assertions which could not be proven. A very useful feature is the ability to find counterexamples for incorrect assertions. These are included in error descriptions displayed by the editor.

The input language of HAHA is that of while programs over integers and arrays. We designed it so that its mechanisms and data types match those supported by state of the art satisfiability solvers, e.g. Z3 [10] or CVC4 [2]. As the language is fairly standard, we do not give its grammar. Instead, we discuss pivotal features of the HAHA syntax through an example presented in Fig. 2.

The program consists of a single function, which calculates the sum of natural numbers from 1 to n, for a given value of n. The specification simply states that the result (which, just like in Pascal, is represented by a special variable) matches the well known Gauss' formula, as long as the argument is not negative.

[1] The tool is available from `http://haha.mimuw.edu.pl`



```
predicate sample(x : Z) = (x = 42)
axiom test: forall x: Z y : Z, (x + y)^2 = x^2 + 2*x*y + y^2

function nsum(n : Z) : Z
  precondition n_is_positive : n > 0
  postcondition gauss : nsum = n * (n - 1) / 2
var
  i : Z
  s : Z
begin
  s := 0
  { n_is_positive: n > 0 }  { s = 0 }
  i := 0 + 1
- Unable to prove validity of a condition. 0 } { i = 0 }
  Program correctness
   Correctness of nsum                    0
    Block at lines 10 - 28
    invariant gauss: s = i * (i - 1) / 2
    counter L : Z [L - 1], L = n - i
  begin
    s := s + i + 1
    { count: L = n - i } { n_is_positive: n > 0 }
    { i_lt_n : i < n } { s = i * (i + 1) / 2 }
    i := i + 1
  end;
  { s = n * (n - 1) / 2 }
  nsum := s
end
```

```
Block at lines 20 - 25
  count
  n_is_positive
  i_lt_n
  Assertion_at_L23
  n_is_positive
  i_le_n
  gauss
```

```
n_is_positive : n > 0
i_le_n : i ≤ n
gauss : s = i * (i - 1) / 2
Condition_at_L15 : i < n
Assertion_at_L19 : L = n - i
counter_is_positive : L > 0 ∧ L = n - i
------------------------------------------
i < n
```

Fig. 1. The user interface of HAHA.

Here it must be noted that the type `Z`, used in the example program, represents unbounded (that is, arbitrarily large) integers. This is one example of a design choice that would not necessarily be valid for a verifier meant to be used in actual software development. The reason is that errors related to arithmetic overflows are quite common, and are often exploited for malicious purposes. It seems reasonable to require a static analyser to be able to ensure that no such mistakes are present in the checked code. In our setting, these considerations play a less important role, so we were able to choose a simpler model of arithmetic. On the other hand, it might be actually desirable to be able to illustrate difficulties associated with the necessity of avoiding or handling overflows. For this reason we have created a modification of our tool to allow the use of `Int` variables, modelled as 32-bit vectors. The fact that we were able to add such feature with relative ease seems to reinforce our claim about adaptability of HAHA.

Structure of the language appears to be fairly self explanatory. Let us note that, following the example of Eiffel, loop invariants can be named. This is also extended to other types of assertions. The names are useful for documentation purposes, and make error messages and solver logs much easier to read. It is possible to give multiple invariants for a single loop. Similarly, a sequence of

```
predicate sample(x : Z) = (x = 42)
axiom test: forall x: Z y : Z, (x + y)^2 = x^2 + 2*x*y + y^2

function nsum(n : Z) : Z
  precondition n_is_positive : n > 0
  postcondition gauss : nsum = n * (n - 1) / 2
var
  i : Z
  s : Z
begin
  s := 0
  { n_is_positive: n > 0 }  { s = 0 }
  i := 0
  { n_is_positive: n > 0 } { s = 0 } { i = 0 }
  while i < n do
    invariant n_is_positive: n > 0
    invariant i_le_n : i <= n
    invariant gauss: s = i * (i - 1) / 2
    counter L : Z [L - 1], L = n - i
  begin
    s := s + i
    { count: L = n - i } { n_is_positive: n > 0 }
    { i_lt_n : i < n } { s = i * (i + 1) / 2 }
    i := i + 1
  end;
  { s = n * (n - 1) / 2 }
  nsum := s
end
```

Fig. 2. Example program. The function computes the sum of numbers from 1 to $n - 1$ using a loop. The postcondition states that the final result is equal to the result of the Gauss' formula.

assertions is interpreted as a conjunction. This is notably different from the approach taken in some textbooks and course material, in which an implication is supposed to hold between two consecutive assertions not separated by any statement. We believe the former interpretation to be clearer, but, to illustrate the ease with which HAHA can be modified to fit the requirements of a particular course, we have implemented the latter in a variant of our tool. Finally, let us remark that an explicit application of the weakening rule can be unambiguously represented with the help of the `skip` statement.

In the example program, each pair of consecutive statements is separated by assertions. For each instruction the correctness of the two surrounding assertions is checked by the application of the Hoare logic rule for the instruction combined with implicit application of the weakening rule. As long as loop invariants are provided, these midconditions can be inferred automatically through Dijkstra's weakest precondition calculation [11]. However, no such inference is performed in HAHA as a result of the requirement stated in Section 2 above.

HAHA supports proofs of both partial and total correctness. We follow here the traditional pattern used in our classes. To facilitate the latter, we allow a specifically designated invariant to be parametrised by an auxiliary integer variable. The conditions generated for such an invariant, which we call *a counter*, ensure that the loop condition holds precisely when the invariant formula is true for an argument of 0. It must also be proved that, if the invariant holds before the loop for an argument of $L \in \mathbb{N}$, it will hold for a number $L' \in \mathbb{N}$ strictly smaller than L after the loop body is executed. Concrete syntax used by HAHA for the purpose of termination proving can be seen in the example program, although proving termination of this particular loop is not a very challenging task. The expression in square brackets represents the aforementioned value of L'.

This solution is different than frequently used construct of *loop variant* as in Why3 or *decreases* statement as in JML. However, these solutions focus, as mentioned before, on ease of intent specification while we focus on giving the students the impression of which is information a programmer should control to write correct programs. Our solution forces students not only to provide the formula that decreases with each loop iteration, but also to explicate the way *how* the formula is decreased. Students can provide such formula only when they have good command of all the control flow paths of the program. Other courses may opt for the mentioned above design with loop variants. A smooth introduction of such a construct requires binary Hoare calculus, which is a desirable extension of the basic formalism. To extend the portfolio of existing options, we are working on adding support for the binary assertions.

Before we conclude this brief examination of the capabilities of HAHA, let us focus on the very first lines of the example program. They contain definitions of an axiom and a predicate. They are only an illustration, and are not actually used in the remaining code. When creating specifications for more complex procedures, however, it is often desirable to use predicates to simplify notation. Our experience shows that this is especially useful in code operating on arrays, as it tends to employ rather complex assertions. Axioms, in turn, are helpful when the automated solver cannot prove a true formula.

3 Description of the experiment

Experiment environment The base environment of the experiment was the course of *Semantics and program verification* (*Semantyka i weryfikacja programów* in Polish).[2] The students are traditionally instructed on two kinds of classes, namely, on lectures and on blackboard exercises. The lectures give the theoretical background for the material while during blackboard exercises students are exposed to problems such as defining formal semantics for a given intuitive description of a language or verification of a small program using Hoare

[2] A description of the course curriculum can be found in page `http://informatorects.uw.edu.pl/en/courses/view?prz_kod=1000-215bSWP`

logic. The problems are solved in cooperation between students and tutors. The lectures are given to all students who enrolled on the course while the blackboard exercises are given to smaller exercise groups of 15–20 students each.

The course is traditionally divided into three approximately equal in weight topics: operational semantics, denotational semantics, and program verification. Each of them ends with a homework graded by tutors. The basis for the ultimate score of a student is the final exam. The students are given there three problems concerning the three topics of the course. The course has been run several times and is instructed with an established routine. The lectures are given with help of slides that undergo only minor changes each year. The blackboard exercises are given using standard sets of problems to be dealt with.

Experiment setup HAHA tool was introduced to the standard setup of the course. It was used in the course only for the third part of the instruction, where Hoare logic was presented. The whole body of students was divided in two populations: G_0, where the instruction was done traditionally, and G_1, where the instruction was done with help of HAHA. The control population G_0 consisted of 66 students while the population G_1 consisted of 38 ones[3].

All students were warned at the first lecture that some of them will be trained with help of a new tool to avoid shock on their side. We also believe that the Eclipse environment is perceived neither as a hot novelty nor as an antique tool, which is confirmed by the Evans Data Corp. survey [12].

The tool was presented only during the blackboard exercises. It was not mentioned during the lecture except from the initial short message. The group G_0 consisted of three classes groups instructed by two teachers T_1, T_2. The group G_1 consisted of two classes groups instructed by two teachers T_2 and T_3. One of the teachers T_2 instructed both a classes group in G_0 and a classes group in G_1. The teacher T_3 is a co-author of the tool and the current paper. Incidentally, the teacher T_3 was ill during one of the critical classes. He was replaced by the co-author of the paper for this single slot.

The instruction consisted in exposing the students to the same set of problems as in the previous years. However, the examples were shown in the HAHA editor displayed through a projector on a screen visible for the whole group. Students chosen by a tutor had an opportunity to determine the Hoare formulae written in HAHA. After the series of instructional classes finished a homework was given as in the previous years. The students instructed with help of HAHA were encouraged to return their homeworks done in HAHA. The homework assignment was prepared by the teacher T_1 who did it in the previous years, but to ensure it can be prepared smoothly with HAHA it was modified slightly by T_3. In the end 12 students returned their homeworks in HAHA format. All of the homeworks were graded with the maximal number of points.

The results of instructions were checked during the final exam. The students both in G_0 and G_1 were given the same assignments one of which consisted in filling in missing assertions of Hoare logic in a given program, which was typical

[3] The supplementary material with raw data and R source code is available from `http://haha.mimuw.edu.pl/experiment2012/data.zip`

kind of assignment on exams in this subject for several years. Both groups had to do it on a supplied piece of paper that contained the program and free space to fill in with formulae.[4] The assignment was again prepared by T_1. This time there was no influence from the authors of the study to meet the constraints imposed by HAHA. Instead T_1 was explicitly instructed to disregard any experience associated with this tool and to prepare an assignment exactly according to the schemes of the preceding years. After the exam the assignments were collected and graded in points ranging from 0 to 10. The grading was done by teacher T_1, the one who prepared the assignment and who did not do instruction with HAHA. Out of 66 people in G_0 a group G'_0 of 62 students attended the exam and out of 38 in G_1 a group G'_1 of 33 students did[5].

In addition to the assignments, students were given a sheet of paper with an anonymous survey concerning the use of HAHA. The survey had 3 questions:

1. Have you heard about HAHA tool before this exam?
2. Did you use HAHA tool?
3. Did you try to do your homework with HAHA?

In addition students were asked in the questionnaire to share comments about HAHA. The students were given short additional time to fill in the questionnaire.

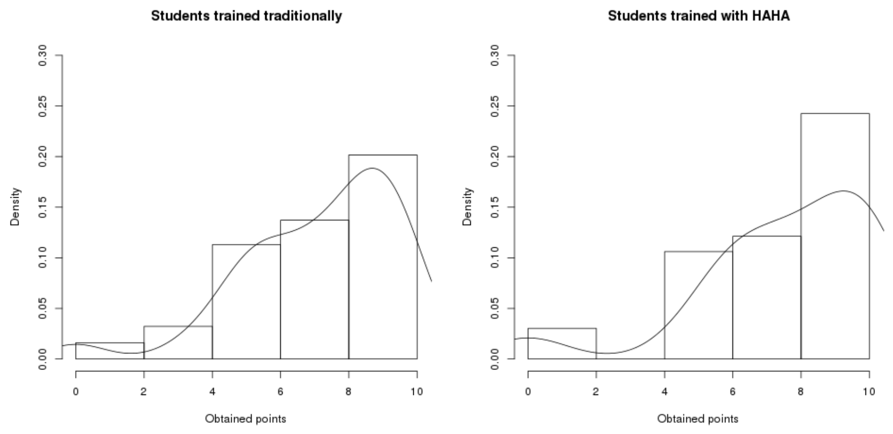


Fig. 3. Scores of students trained traditionally and with HAHA.

Experiment results and discussion The main result of the experiment is the comparison between the performance of students in G'_0 and in G'_1. The histograms of the scores obtained by the groups are presented in Fig. 3. (In both

[4] The texts of the assignments are included in the supplementary documentation of the experiment.

[5] Each of the students had two potential attempts, all the figures are results combined from both of them.

pictures, an idealised distribution of the scores is drawn, as given by R tool.) We assume that students who did not show up on any attempt effectively did not take the training so they cannot be counted. We can immediately see that the distributions are not normal (they are rather bimodal in both cases). This can be confirmed by Shapiro-Wilk normality test, which gives $W = 0.9039$ with the p-value of 0.00014 for the group G'_0 and $W = 0.8225$ with the p-value of 0.000089 for the group G'_1. Therefore, the null hypothesis that the distributions are normal can be rejected with high confidence.

The mean score of G'_0 is $E = 7.181$ with the standard deviation $\sigma = 2.257$ while the mean of G'_1 is $E = 7.603$ with $\sigma = 2.594$. Analogously, the median for G'_0 is $M = 7.75$ while the median for G'_1 is $M = 8$. This suggests that population G'_0 has lower scores in this experiment than the treatment one G'_1.

We would like to strengthen this argument by showing that these differences are not pure coincidence. We give here a substantial evidence that the scores in G'_1 are greater than scores in G'_0, which means that the students taught with help of HAHA perform not worse than students taught in the traditional way. To ascertain this we use the non-parametric Mann-Whitney test, as the use of standard parametric tests assumes the distributions are normal. We can indeed apply the test since its prerequisites are met:

Hypotheses The null hypothesis H_0 for the current study is that the median of scores in G'_0 is greater or equal to the median of scores in G'_1. The alternative hypothesis H_A is that the median in G'_0 are less than the one in G'_1.

Uniformity of populations We also can assume that in case the distribution of results from G'_0 and G'_1 is the same, the probability of an observation from G'_0 exceeding one from G'_1 is equal to the probability of an observation from G'_1 exceeding one from G'_0. We discuss this assumption later, but in summary the background of both groups is the same and their members were chosen randomly from the point of view of the experiment.

Independence We can assume that the observations in G'_0 and G'_1 are independent since they concern answers to the same problem given in disjoint groups in the conditions of an exam during which communication between students was prohibited.

Comparable responses The scores are numeric so they can be easily compared one with another so a single ranking of the subjects can be formed.

The z-score for the test[6] is $Z = 1.358$ with the confidence level $p = 0.087$, which means that we are very close to reject H_0 with statistical confidence that students without HAHA training obtain better results than students taught with HAHA. (If we restrict our study to first attempts of the exam only then $Z = 1.698$, $p = 0.045$ so we could statistically reject H_0. This might be seen as a more appropriate setup as all students are judged based upon the same assignment.)

There are a few issues that should be discussed here.

[6] The results were obtained with help of the R package **coin** [16,17] with its heuristics to handle ties in input data.

Bias of more talented students One possible bias that could affect these results is that the students in G'_1 could have been more talented than average.

First thing to note here is that the team of teachers has no direct impact on the composition of the exercises groups in our faculty. The assignment of students to the groups is done automatically by USOS studies management system [26] based upon plan preferences expressed by the students. Subsequently students can change their assignment by asking teachers of permission to move from one group to another. Therefore, the driving force for the assignment of students to groups is on the side of students. As a result, we can exclude that the groups were intentionally chosen so that more talented students ended up in G'_1.

This can be confirmed by an additional check. The assignments 1 and 2, which consist in defining of an operational and denotational semantics for toy languages, have similar mathematical nature as the one with Hoare logic. In case G'_1 consisted of more talented students, also the scores for the other assignments during the exam should be higher for them. The histograms of the scores obtained for the assignment 1 and 2 are presented in Fig. 4. The mean of the scores in G'_0 for the assignment 1 was $E = 8.08$ while the median was $M = 9$. The mean in G'_1 for the assignment 1 was $E = 7.91$ while the median was $M = 8$. Analogous figures for assignment 2 were $E = 4.25$ and $M = 3.75$ in G'_0 with $E = 4.92$ and $M = 5.2$ in G'_1. Summing up, the control group G'_0 had higher results for assignment 1 and lower for assignment 2. In both cases the differences were not statistically significant ($Z = -1.04$, $p = 0.15$ for assignment 1 and $Z = 0.69$, $p = 0.25$ for assignment 2). As a result, we do not see any significant bias here that could testify that students in G'_1 were more talented.

Problem of instruction by T_3 For the validity of the study it is important to judge more thoroughly the impact of the instruction of the authors of the tool, referred here as T_3. The mean of the exercise group taught by T_3 was $E = 6.46$ and median $M = 7.5$. These results are below the average results of the control group G'_0. Therefore, the results make the mean score of G'_1 only lower and contribute exclusively to strengthening the general conclusion.

Discussion of instruction by T_2 Since one of the tutors instructed both with help of HAHA and without HAHA, it is interesting to compare the results obtained by the corresponding exercise groups $G^2_0 \subseteq G'_0$ and $G^2_1 \subseteq G'_1$. The sizes of the exercise groups were 21 and 20, respectively. The average and median were $E = 7.67$ and $M = 9$ for G^2_0 and $E = 8.35$ and $M = 8.5$ for G^2_1. The Mann-Whitney statistics this time results in $Z = 1.04$ and $p = 0.15$. These results are not statistically significant, but give clear indication in favour of our claim that teaching with HAHA helps students in getting better scores.

Independence and uniformity of populations Important assumption in the experiment is that the results of G'_0 and G'_1 are independent one from the other. Since the experiment lasted several weeks it was in principle possible that students from the tested group would interfere significantly with those in the control group. We have three reasons to think that this was only marginal. First, students from other groups neither volunteered nor demanded to return their homework written in HAHA. In case, the tool was very popular there should

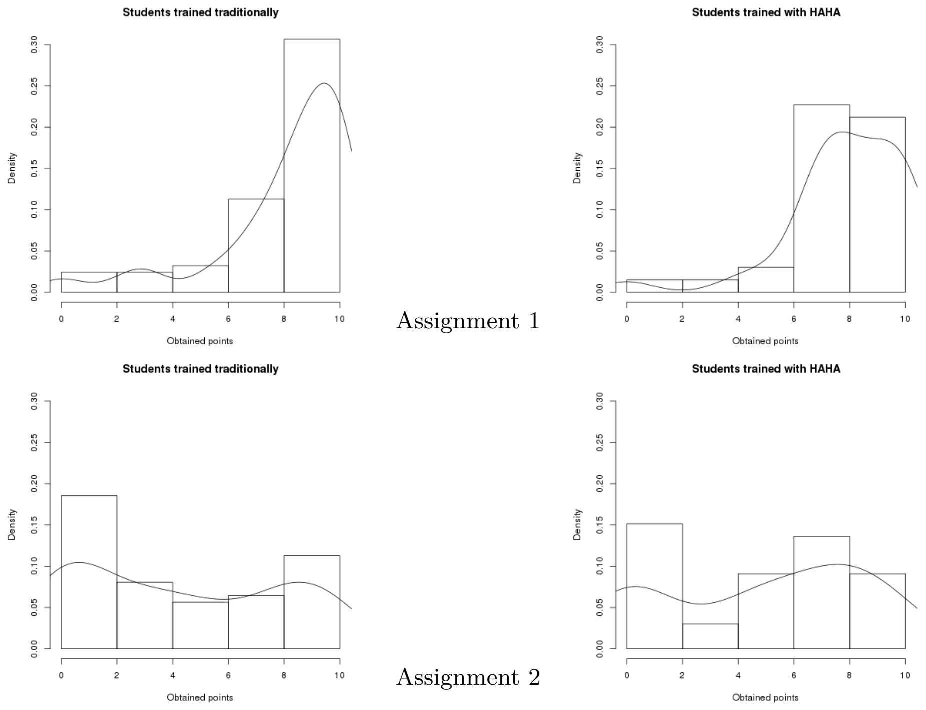


Fig. 4. Scores obtained of students for assignments 1 and 2.

be attempts in G'_0 to return solutions in HAHA (this was not forbidden). Second, the solutions to assignments during the exam were individual, i.e. all kinds of cooperation were explicitly forbidden under standard sanctions. Third, the anonymous survey during the exam in all cases except 17 indicated no familiarity with HAHA (answer ‘no’ to the first question). Therefore, the number of people who admitted familiarity was strictly less than the number of people both in G_1 and G'_1. Moreover, the number of people who answered positively to the question number 2 was exactly equal to the number of people who returned homeworks.

4 Related work

The limited space does not make it possible to give an overview of all program verification tools. We focus here on those with reported applications in teaching.

KeY-Hoare [6] is a tool that serves purposes very similar to HAHA. It uses a variant of Hoare logic with explicit state updates which allows one to reason about correctness of a program by means of symbolic forward execution. In contrast, the assignment rule in more traditional Hoare logics requires backwards reasoning, which can be argued to be less natural and harder to learn. Implementation of the system is based on a modification of the KeY [3] tool.

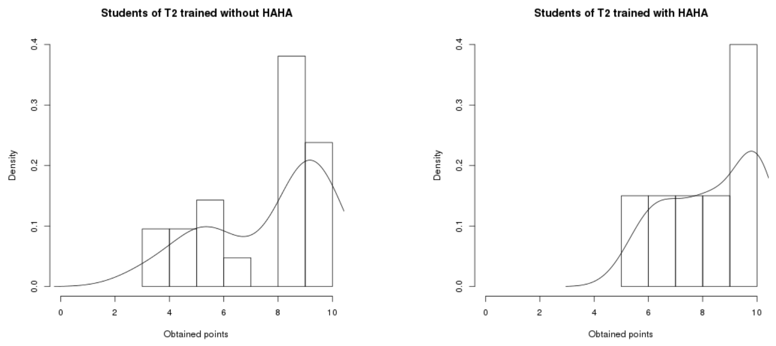


Fig. 5. Scores of students trained by T_2. Note the change in the density scale.

Why3 [4,13] is a platform for deductive program verification based on the WhyML language. It allows computed verification conditions to be processed using a variety of provers, including SMT solvers and Coq. WhyML serves as an intermediate language in verifiers for C [5,8], Ada [21] and Java. It has also been used in a few courses on formal verification and certified program construction.

Another tool used in education that must be mentioned here is Dafny [23]. It can be used to verify functional correctness and termination of sequential, imperative programs with some advanced constructs, such as classes and frame conditions. Input programs are translated to language of the Boogie verifier, which uses Z3 to automatically discharge proof obligations.

Some courses on formal semantics and verification use the Coq proof assistant as a teaching aid [25,15]. Reported results of this approach are quite promising, but the inherent complexity of a general purpose proof assistant appears to be a major obstacle [25]. One method that has been proposed to alleviate this issue is to use Coq as a basis of multiple lectures on subjects ranging from basic propositional logic to Hoare logic [15]. In this way the overhead necessary to learn to effectively use Coq or a similar tool becomes less prominent.

The complexity of general purpose provers is often a troublesome issue in education. One can attempt to resolve this problem by creating tools tailored to specific applications, which sacrifice generality for ease of use. One example of such a system is SASyLF [1], which is a proof assistant used for teaching language theory. Another program worth mentioning here is CalcCheck [20]. It is used to check validity of calculational proofs in the style of a popular textbook [14]. This approach is very similar to what we advocate for teaching Haore logic, as it employs a tool created to fit the style of existing educational material.

5 Conclusions and further work

We have implemented a tool to support instruction of Hoare logic to students. This tool resides in a mainstream software development environment Eclipse, which can give the impression to students that the technique matches contemporary trends in software development tools. It turns out that students instructed with this tool improved they scores almost in statistically significant way (and in statistically significant way in one of considered scenarios).

Since the design of HAHA assumes it can be adapted to other curricula, we invite everybody to consider its adaptation to their own teaching needs. The tool is released with a flexible open-source licence that makes these activities easy.

There are still many ways this endeavour could be extended. The presentation of counterexamples can be enhanced in various ways. Addition of different verification condition generation procedures could open the tool for different settings, for instance as an aid in instruction of first-year students where forcing them to write machine checkable loop invariants can improve their command of programming. It would also be convenient to enable execution of edited programs and their debugging as in GNATprove [22]. Another interesting improvement would be to add the option to automatically generate unit tests based upon counterexamples encountered during program development.

Acknowledgement We would like to thank professor Andrzej Tarlecki for encouragement in development of HAHA and for letting us make our study in the course of his lecture. This study would be impossible without tutoring of Bartek Klin who kindly agreed to instruct students with HAHA. We also thank Ewa Madalińska-Bugaj for preparation of assignment problems and grading them.

References

1. J. Aldrich, R. J. Simmons, and K. Shin. SASyLF: an educational proof assistant for language theory. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*, pages 31–40. ACM, 2008.
2. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. of CAV 2011*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
4. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.
5. S. Boldo and C. Marché. Formal verification of numerical programs: From C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5(4):377–393, Dec. 2011.
6. R. Bubel and R. Hähnle. A Hoare-style calculus with explicit state updates. In Z. Instenes, editor, *Proc. of Formal Methods in Computer Science Education (FORMED)*, ENTCS, pages 49–60. Elsevier, 2008.
7. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.
8. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A software analysis perspective. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Proc. of SEFM'12*, volume 7504 of *LNCS*. Springer, 2012.

9. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion*, pages 429–430. IEEE, 2009.
10. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
11. E. W. Dijkstra. *A Discipline of Programming.* Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1976.
12. Evans Data Corp. *Users' Choice: 2011 Software Development Platforms.* Evans Data Corp, 2011.
13. J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proc. of ESOP'13*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
14. D. Gries and F. B. Schneider. *A logical approach to discrete math.* Springer, New York, NY, USA, 1993.
15. M. Henz and A. Hobor. Teaching experience: Logic and formal methods with Coq. In J.-P. Jouannaud and Z. Shao, editors, *Proc. of CPP'11*, volume 7086 of *LNCS*, pages 199–215. Springer, 2011.
16. T. Hothorn, K. Hornik, M. A. van de Wiel, and A. Zeileis. Implementing a class of permutation tests: The coin package. *Journal of Statistical Software*, 28(8), 2008.
17. T. Hothorn, K. Hornik, M. A. van de Wiel, and A. Zeileis. *Package 'coin': Conditional Inference Procedures in a Permutation Test Framework.* CRAN, 2013.
18. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.
19. B. Johnson, S. Yoonki, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proc. of ICSE'13*, pages 672–681. IEEE, 2013.
20. W. Kahl. The teaching tool CalcCheck a proof-checker for Gries and Schneider's "logical approach to discrete math". In J.-P. Jouannaud and Z. Shao, editors, *Proc. of CPP'11*, volume 7086 of *LNCS*, pages 216–230. Springer, 2011.
21. J. Kanig. Leading-edge ada verification technologies: combining testing and verification with GNATTest and GNATProve – the Hi-Lite project. In *Proceedings of the 2012 ACM Conference on High Integrity Language Technology*, pages 5–6. ACM, 2012.
22. J. Kanig, E. Schonberg, and C. Dross. Hi-lite: the convergence of compiler technology and program verification. In *Proceedings of the 2012 ACM Conference on High Integrity Language Technology*, pages 27–34, New York, NY, USA, 2012. ACM.
23. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Proc. of LPAR'10*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
24. K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical note, Compaq Systems Research Center, Oct. 2000.
25. B. C. Pierce. Lambda, the ultimate TA: using a proof assistant to teach programming language foundations. In *Proc. of ICFP*, pages 121–122, 2009.
26. A. Stępień, K. Kośla, M. Krysiak-Klejnberg, and A. Kudelska. Usosweb enrolling direct enrollment in groups. Technical report, University of Warsaw, 2008.
27. S. J. Trierweiler and G. Stricker. *The Scientific Practice of Professional Psychology.* Springer, 1998.