Is Deductive Program Verification Mature Enough to be Taught to Software Engineers?

Marc Schoolderman
Radboud University
Nijmegen, The Netherlands
m.schoolderman@cs.ru.nl

Sjaak Smetsers
Radboud University
Nijmegen, The Netherlands
s.smetsers@cs.ru.nl

Marko van Eekelen[*]
Open University of the Netherlands
Heerlen, The Netherlands
marko.vaneekelen@ou.nl

ABSTRACT

Software engineers working in industry seldom try to apply formal methods to solve problems. There are various reasons for this. Sometimes these reasons are understandable—the cost of using formal methods does not make economic sense in many contexts.

However, formal methods are also often greeted with scepticism. Formal methods are assumed to take too much time, require tools that are too academic, or to be too mathematical to be understood by practice-oriented software engineers.

We tested these assumptions by designing a small course around a framework for program verification, aimed at regular computer science students enrolled in a Master's programme. After four lectures and associated exercises, students were given a small verification task where they had to model and verify a real, non-trivial, C function in Why3.

A significant majority of students managed to prove a non-trivial functional specification of this C function in the time allotted, and many also pointed out inherent flaws of this function discovered during formalization. Participants reported no major difficulties or mental hurdles in learning Why3, and considered its approach to be appropriate for selected components of safety-critical software.

While formal verification tools such as Why3 still have lots of room for improvement, this experience shows that in a short amount of time, software engineers can be taught to use a program verification tool, and obtain usable results without being fully proficient in it. We further recommend that courses on formal methods should also let students explore these as techniques to be applied, instead of only focusing on the theory behind them, as we expect this to gradually lower the barrier to wider acceptance.

CCS CONCEPTS

• **Theory of computation** → Program reasoning; • **Security and privacy** → *Logic and verification*; • **Social and professional topics → Computer science education**.

[*]Also with Radboud University.

This work is part of the research programme 'Sovereign' with project number 14319 which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO). .

KEYWORDS

formal verification, teaching, Why3

ACM Reference Format:
Marc Schoolderman, Sjaak Smetsers, and Marko van Eekelen. 2019. Is Deductive Program Verification Mature Enough to be Taught to Software Engineers?. In *Proceedings CSERC 2019 18-20 November 2019 Computer Science Education Research Conference Larnaca, Cyprus (CSERC '19), November 18–20, 2019, Larnaca, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3375258.3375265

1 INTRODUCTION

Program verification is about as old as the field of computing science itself [17], and the focus of much academic research. Yet, it is seldom applied in industry. In the Sovereign project, we strive to find ways to apply formal methods—and deductive program verification in particular—to practical situations, in cooperation with partners from industry. This has made us acutely aware of the divide between industry on the hand, and academia on the other.

In our discussions, representatives of our industry partners expressed the sentiment that it was all well that *academics* understand the *academic* tools and can use them for selected problems, but that their company employs *software engineers* which, although academically trained, would not be able to learn and understand these techniques in such a way that they can use the effectively—and that formal verification can not hope to get a wider foothold in industry unless this problem was addressed.

This sparked our interest. Clearly, there is no argument that many programs are too large or too complex to be within reach of formal verification with the current tools. However, if the tools themselves are also perceived to be too complex or too theoretical to be understood, that is another problem altogether.

Therefore we were interested in testing the assumption that program verification, and associated computer-aided reasoning tools, are *intrinsically* too much work to get accustomed with for a practice-oriented software engineer.

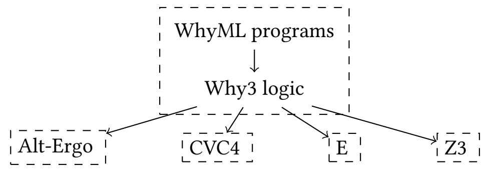
One way to answer such a question is to design a crash course that teaches a formal method to software engineers, and see how they perform at a small verification task. However, since any *academically trained* software engineer would at some point have been a student, we can instead test this assumption by doing the same experiment with university students, which are readily available to us.

Therefore, we selected to teach the Why3 framework for program verification [11] in a short time span, as part of an existing course at Radboud University, and assess the performance of students at a small, but realistic verification task.

In Section 2, we will briefly introduce the Why3 verification framework. Section 3 provides an overview of the choices we made in teaching it to students. Sections 4 and 5 will present and discuss our evaluation of the course. Section 6 provides an overview of related courses. Section 7 contains our conclusions.

2 OVERVIEW OF WHY3

Why3 [11] is a platform for deductive verification of programs. It takes programs written in a dedicated programming language (WhyML), which should be annotated with pre- and postconditions, assertions and loop invariants, and uses a weakest precondition calculus to generate verification conditions in a typed first order logic. These are then subsequently translated to the input formats of various automatic provers, in order to prove them, as visualized in the following diagram:



In Why3, as in many such systems, there is a difference between *programming constructs* and the *logical constructs*. A program is written in WhyML, and annotations are written in a logical language. There is a high degree of similarity between the two: they both share the same expression syntax and allow for the definition of functions. There are also subtle differences.

```
predicate divides (d n: int)
  = exists k: int. n = k * d

predicate prime (p: int)
  = 2 <= p /\ forall i. 1 < i < p -> not divides i p

lemma mod0_implies_divides:
  forall n d. d <> 0 -> mod n d = 0 -> divides d n

let is_prime (n: int): bool
  requires { n > 1 }
  ensures { result <-> prime n }
= if mod n 2 = 0 || mod n 3 = 0 then
    return n <= 3;
  let i = ref 5 in
  while sqr !i <= n do
    invariant { 5 <= !i /\ mod !i 6 = 5 }
    invariant { forall d. 1 < d < !i -> not divides d n }
    if mod n !i = 0 || mod n (!i+2) = 0 then
      return false;
    i := !i + 6
  done;
  assert { forall d. !i <= d /\ divides d n ->
                     divides (div n d) n };
  return true
```

Figure 1: Example of annotated WhyML code

An example of an annotated WhyML program can be seen in Figure 1. The predicate `prime` defines what a prime number is in purely logical terms, whereas the program `is_prime` implements an almost-correct primality test.[1] Since this primality test does not work for $n \leq 1$, this has to be mentioned in its specification. Why3 can quickly prove that this program is correct with respect to this specification by using a combination of Alt-Ergo, CVC4, Z3 and E-Prover.

Furthermore, Why3 supports *ghost code* [10]. This is any code or data that is not necessary to obtain the result of a program, but *can* make its verification easier. The Why3 type system will ensure that code marked as *ghost* can not affect the outcome of a program, allowing the user to use purely logical constructs in computations that only have (*ghost*) side-effects. Furthermore, a user can 'promote' a programming construct to have an effect on the logical level. A function at the programming level can be used to either prove a logical lemma (a so-called *lemma function* or `let lemma`), or provide an axiomatisation of a function in a safe manner (Why3 calls this a `let function`). The similarities between the various levels of Why3 make it fairly easy to get novices started with Why3, at the same time, the subtle differences can also catch new users off-guard.

Most work in Why3 is done inside a graphical user interface where users can transform verification conditions (e.g., splitting a large conjunction into smaller ones), and send them to provers. Most actions are performed with a single mouse click. The user interface also uses colour highlighting to provide users with information about which programming constructs are involved in the current proof state.

3 APPROACH

Our treatment of Why3 was part of an existing course on 'Software Analysis', which is an elective course offered as part of the MSc programme in computing science at Radboud University. The scope of this course was intentionally broad, to treat varying topics in it relating to tools and techniques for analysing software. Formal program verification using Why3 was planned to take up the first half of this course, with the second half veering more towards static analysis. The entry requirement for this course is that students are in possession of a Bachelor's degree in computing science (or equivalent).

By using an existing course that was not advertised primarily as a course on formal methods, theoretical computer science, or theorem proving, we are confident that the students taking this course did not self-select, and form a unbiased representative sample of the computing science student population at Radboud University.

Secondly, to prevent students from deselecting after the start of the course, the guiding principle during teaching was to approach program verification using Why3 as a *tool* that might have a place in a software engineers toolbox, instead of treating formal verification as an intrinsically interesting theoretical topic, where Why3 is only used as a vehicle for demonstrating an application. So, for instance, students were not taught any core principles on which formal verification frameworks are based, such as Hoare logic or

[1]Based on https://en.wikipedia.org/wiki/Primality_test

how to compute weakest precondition calculus.[2] Instead, this was explained only in so far as was necessary to give students an intuition into what Why3 is doing. Also, exercises focused mostly on familiarizing students with features of Why3, similar to a course on programming, and were chosen to enable quick positive feedback.

At the end of this part of the course, students had to write a short report about their work, in which they were also required to include a reflection on Why3. Furthermore, students were asked (but not required) to complete an anonymous survey intended to verify that they were indeed a representative sample of students enrolled in a Master's programme in computing science. In this survey, they are also asked about the time they needed for the course work, and their general disposition towards program verification after taking the course.

3.1 Course structure in detail

Teaching Why3 was split in two parts: in-class teaching with weekly lectures with associated homework exercises, and a small project where students would, in groups of two, tackle a verification challenge in the style of the VerifyThis[3] or SV-Comp[4] competitions. [5]

3.1.1 Teaching Why3 in four weeks. In-class teaching consisted of four weeks of lectures and exercises. The organization per week was as follows:

(1) Outlining a historical background motivating why program verification schemes should employ computer-aided reasoning in order to be feasible, and a first look at Why3 and how to write basic WhyML programs; as well as how to use the logical language of WhyML to write function contracts, invariants, and prove termination.
(2) How the Why3 type system works, and how to reason about mutable data such as arrays. At this point, students were subjected to an interactive demonstration where Kadane's algorithm [3] for finding the maximum subarray was verified, with the intent to set an example to students of how to write more complex logical specifications, discover invariants, and how to interpret the responses of the Why3 IDE.
(3) Techniques in Why3 that can be used in more difficult situations where a proof is not solved automatically—such as cases where a proof by induction is needed. In particular, in this lecture the somewhat difficult concept of `ghost code` and `let lemma`'s [10] was introduced.
(4) Modelling a program in a different, more realistic, programming language in WhyML; for example, how to handle integer overflow, or how to reason about a memory model that supports pointers (adapted from the lectures notes by Filliâtre [12]).

Weekly exercises were designed to fit with the level students were expected to have after each lecture. Students were required to use a Why3 installation on their own system instead of using a web interface for all of these, to have access to the full capabilities and multiple back-end provers. The objective of these exercises were, per week:

(1) Familiarizing with the WhyML syntax and Why3 interface. In particular, students had to finish a partial proof of a Russian peasant multiplication (adapted from an existing course by Bobot [5]). Students were also challenged to rewrite this verified program to perform exponentiation instead, with some hints.
(2) Writing a WhyML program and specification from scratch. Students had to pick a simple sorting algorithm and model it in WhyML, and prove it correct.
(3) Using *let lemma*'s to write inductive proofs, and a slightly harder partial proof (for the factorial function one taken from [17]) that needed to be finished.
(4) Modifying a WhyML program so that it can be used to extract C code.

We estimated that students would need not more than six hours for each exercise. Students were given formative feedback (including fixes to finish their proof efforts, whenever they were very close to a solution).

3.1.2 Verification challenge. After four weeks of in-class teaching, students got their project assignment. Students had to either choose a C functions from a small list of system library routines contained in the CloubLibc [25] library, or a test case that was designed to be similar to a routine for modular addition of 256-bit integers used in the cryptographic library TweetNaCL [4].

Given this C function, the assignment consisted of modelling it reasonably accurately in WhyML, providing a formal specification, and proving that the WhyML model adheres to this specification.

3.1.3 Self-evaluation. Students were given four weeks to complete the verification challenge. Afterwards they had to write a report documenting their formalization, motivating their modelling choices and formal specification; students also had to reflect on what they considered to be the strengths and weaknesses of Why3, and were encouraged to do so from a software engineering viewpoint.

In order to learn more about the learning experience, we also sent a survey asking students about how they self-assessed their programming and mathematical maturity, how much time they spent on the Why3 exercises and the verification challenge, and what their disposition to formal verification was at the end of the course, as an extra check of (our interpretation of) the information contained in their reports.

4 RESULTS

In total 22 students participated seriously in the verification challenge, grouped into 11 teams. One other student made only a rudimentary attempt and would eventually drop out of the course after the part focusing on Why3 was finished.

Out of the 11 teams, nine chose to verify the `strlcat` routine from the CloudLibc library, probably due to familiarity with the operation of string concatenation. Two teams took on the modular addition routine inspired by TweetNaCL, as described in Section 3.1.2.

[2] We do not mean to imply that taking that approach is not a valid didactic approach—however, it would not have been relevant for our research question.
[3] https://www.pm.inf.ethz.ch/verifythis.html
[4] https://sv-comp.sosy-lab.org/
[5] The course material, including exercises and project description is available online at https://cs.ru.nl/~M.Schoolderman/swan2019/

4.1 Verifying strlcat

Out of the nine teams that chose `strlcat`, seven teams produced a WhyML model with a logical specification that was fully verified in Why3. The remaining two teams delivered an incomplete proof. In both cases this seems to be due to teams choosing to build an axiomatisation of the C memory model instead of a simpler approach. We discouraged students from doing this, because it risks introducing logical inconsistencies. In these two cases, however, the axiomatisation was simply not powerful enough to support drawing the necessary conclusions.

The C function `strlcat` is intended to be a safer version of `strcat`, for concatenating null-terminated strings in a manner which is much less likely to cause buffer overruns, but also guaranteeing that the result is a proper null-terminated C string.

There are three major difficulties where the verification effort of this function is not straight-forward:

(1) `strlcat` is not required to (and in fact will not) perform its expected operation in case the two pointers it is passed point to overlapping regions of memory.
(2) In case `strlcat` is called with a `size` argument that is too small, there is a subtle safety mechanism that prevents it from accessing out-of-bounds memory addresses. However, in this case `strlcat` will not concatenate any strings or necessarily produce a result that is null-terminated.
(3) To reason about the length of strings at the specification level, the notion of the "terminating null character" needs to be expressed somehow.

All successful teams used the technique for modelling memory outlined in [12], where memory is modelled as an array of bytes and pointers are indices into this array. To tackle the first problem, students either used different arrays to model separate memory regions, or explicitly added a precondition that the two input strings should not overlap. One team was so precise in this that they proved the code works in some cases of overlap.

The second problem was handled by all teams either by adding the explicit precondition that the `size` argument is proper, or specifying a separate postcondition for the cases where it is improper. Some teams commented (rightly) that this made the formal specification of `strlcat` more intricate than would appear necessary, showing that they were able to draw conclusions about the subtleties of systems-level C code based on their formalization.

The third problem was one which was handled in different ways. Several teams took a hint that existential quantification can often be avoided by using *ghost code* [10], and simply added the location of the null character terminating a string as a *ghost argument* to `strlcat`. One team used direct existential quantification instead to find the null character. This resulted in a much harder, yet still successful proof effort. Several other teams added a logical construct (using a `let function`) which explicitly finds the position of the null character using a loop, but which can still be used in specifications. This essentially involves also proving the correctness of `strlen`, and is an interesting approach that these students discovered themselves.

4.2 Verifying modular addition

Only two teams attempted to verify the modular addition routine; both teams completed verification, where one team performed a thorough analysis, and the the other only proved a simple property.

Even though the modular addition code consisted of the least lines of code of all the options available to students, it was probably the most challenging to verify given that (unlike the CloudLibc routines) it did not have a clear (informal) specification, and has known problems that become apparent during verification.

During this challenge, both teams used Why3's machine integers to prove the absence of signed integer overflow under reasonable input conditions. Both teams were instructed to identify some property that the addition routine preserved. For example, whether if both inputs are already reduced modulo $2^{255} - 19$, it holds that the result will also be in a reduced form.[6] Both teams correctly concluded that such an easy property did not exist. However, one team already reached this conclusion before formalizing the routine. In the end, one team proved that the modular reduction step in the addition routine is never performed if the original inputs were in reduced form. The other team simply proved a bound on the output. Only one team tried to prove that the addition routine actually *adds* numbers; this was achieved with some supervision.

4.3 Self-evaluation

Students were asked to provide an evaluation of Why3 and to assess what role it could have in software engineering. The most often mentioned benefit of Why3 was that it makes performing formal proofs accessible and easier, and that proofs provide a higher degree of confidence in software than simple testing. On the other hand, students reported that the verification task involving only a small piece of code took many hours to complete. Students concluded that they considered Why3 inappropriate for regular software engineering due to this time investment, but well-suited for cases where safety or security of software is more important than economic arguments.

Other interesting general observations raised by students where the following:

- Students were generally positive about the graphical user interface of Why3, which can highlight assumptions and goals in the source code.
- Students criticized the error messages provided in case of a syntax or type error. This is an area where academic tools are often lacking, and this clearly hinders the learning process.
- Multiple students complained about the lack of an online community of Why3 users, on a platform such as StackExchange.

Many students also discussed how they experienced working with Why3 in the process of finding a correct proof; key points raised here were:

- Since Why3 uses automatic provers, proofs can fail for reasons that are not obvious to the users and hard to predict for novices.
- Why3 provides a mechanism to generate counter-examples. While students did report using this, the consensus seemed to

[6] $2^{255} - 19$ is a prime number used in Curve25519 elliptic curve computation.

be that they did not provide meaningful information except in simple cases.

- Similarly, some students were dismayed that an inconsistency in a lemma (or invariant) will allow any subsequent statement to be proven vacuously—giving the user the mistaken impression that the they only have one unproven goal remaining. On the other hand, other teams reported using the *smoke detector* of Why3 to catch these cases with success.
- To explore some program state in depth, several students reported missing the ability to have an interactive debugger in the Why3 IDE.
- Finding loop invariants is clearly the hardest part. Several students found it surprising that Why3 was unable to deduce simple loop invariants, or that a loop invariant also needs to be established if the associated loop is not executed.

Although students reported needing a lot of time to complete the verification challenge, most did not report an obstacle inherent to Why3 while working on the challenge. An obstacle that *was* reported by several students was that to model a C program in WhyML requires a deep understanding of C—a deeper understanding than these students professed to have. These students also pointed out that this translation of C to WhyML should be automated.

4.4 Survey

To learn more about the learning experience, a survey was sent to the 22 students that participated seriously in the verification challenge. One student was at this time no longer at Radboud University and could not be reached. Of the remaining students, 15 responded, for a response ratio of slightly above 70%, which we consider to be acceptably high for a student evaluation.

In general, students reported a much higher confidence in their programming ability than their mathematical ability. All participants reported that they felt at least somewhat skilled in programming (rating themselves at least a 5 on a 7-point Likert scale), whereas two thirds of the students reported their mathematical skill level to be not that great (at most a 3 on a 7-point Likert scale). Only one student reported a higher skill level in mathematics than programming, but this student also reported having taken much more ECTS credit in math and logic courses (120, instead of the average 18).

When asked whether they had any experience in other formal method tools, only a handful students reported having some exposure to systems such as Coq or Uppaal (which are used to support teaching in some bachelor courses at Radboud University). On the other hand, many students reported having used ESC/Java2. This tool is used only briefly at another Master's level course at Radboud University for a simple weekly exercise, and so we do not consider this to have significantly impacted our teaching experiences or jeopardize the representativeness of our student sample. When asked in the survey how they compared learning Why3 to learning a new programming language, 12 students answered that learning Why3 was as hard or slightly harder, with only 3 indicating they found it considerably harder.

The survey results also indicated that the amount of time spent on the weekly exercises was within our estimate and fitting for the number of ECTS points that could be earned for the course. More interestingly, the amount of hours students spent on the verification challenge was reported to be slightly less than 20 hours on average, with the median being 18 hours. In advance, we had budgeted between 24 and 32 hours for the challenge. A histogram of reported hours is shown in Figure 2.

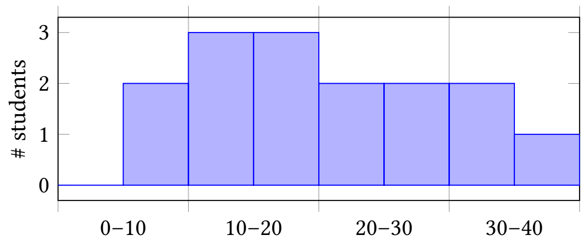


Figure 2: Hours spent on verification challenge by students

5 DISCUSSION

At the start of the verification challenge, we had expected about half of the students to succeed, and another half to deliver only a partial result which either proves only a single property, or where the final result is incomplete due to unproven goals. We also expected students to require around 24 to 32 hours of work. In total, 8 out of the 11 teams managed to complete the challenge with a successful verified result, which we found encouraging. If we assume that the three less successful teams (comprising 6 students) all make up the lower end of the graph of Figure 2, we can still conclude that students completed the challenge in less time than we anticipated.

Students, however, did report needing a lot of time for the verification task. We would like to put this in perspective. The code students were given was representative of systems-level code. Developing such code, including writing a good test suite, takes more time than students probably realize.[7] Compared to that, the amount of time students actually reported is rather low, especially if we take into account that they were all inexperienced users of Why3. To see how much faster they would be if we gave them another similar challenge would be interesting, but we could not justify this being part of the 'Software Analysis' course.

Ultimately, we think that our students were up to the challenge that we put to them, even though they had only received a crash course of four weeks.

5.1 Comparing students to software engineers

Our survey results indicate that our student sample was a typical set of students enrolled in the Master's programme at Radboud University, and did not have a significant prior bias or inclination towards formal methods. For example, in the survey, 60% of the students reported not being familiar with Hoare logic at the start of the course. In the other 40%, only one student reported that he or she used this familiarity while working with Why3. All were familiar with the concept of pre- and postconditions, but had little experience in reasoning about invariants.

[7] In the case of `strlcat`, the function students ended up seeing is the end result of years of intermittent updates and tweaks

Most of these students will, after graduation, apply for software engineering jobs, and so we believe our sample to be representative of a *highly educated* software engineer. Our results, of course, do not apply to all software engineers in possession of a Bachelor's degree in computing science, since our students did choose to enrol in a Master's programme at a university.

5.2 Modelling challenges

As mentioned in Section 4.3, some students reported finding it less satisfying to make a model of C code in WhyML, due to the fact that they felt less confident that they could model C concepts accurately. At another stage in the 'Software Analysis' course, several students also expressed discomfort in reasoning about C programs. This was surprising to us. As one student team put it: "This decreases our confidence that when a WhyML model is proven correct the program in the target language will also be correct." In the anonymous survey, students would also indicate they found modelling the second hardest part in using Why3, after finding loop invariants.

Students identified that a remedy would be to have a tool that either directly verifies C code, or that automatically translates C code into WhyML, but that such a step would inevitably also make verification more difficult. In the survey, 80% of the students indicated that the design of modern programming languages should use formal verification to some degree, which we find consistent with the students' written remarks.

5.3 Using mature tools

In our course, we chose to not use the web interface of Why3[8], but required students to install it on their own systems. The advantage of this was that students could tackle more complicated proofs, since they had access to all the supported powerful automatic provers, and could benefit from all of the features of Why3, such as counterexample generation and the *smoke detector* feature of Why3. Even though there is much available to them in the full Why3 interface which they will not understand (at least at first), we do not find that this impedes the learning process.

The downside here was that installing Why3 is a rather involved process, due to the need to install it (and the *specific* versions of various automated provers that are supported by Why3) from source. To save time, and to ensure most students were on the same platform, we created a binary-only distribution of Why3 and a selection of automated provers for Linux. This worked well, but was a step in distributing Why3 to students that was cumbersome for us and that should not have been necessary.

5.4 Recommendations

Complex programming languages such as C++ are used daily by practitioners that will freely admit to not understanding all the intricate details. The same is also true for formal verification tools. However, in our experience, this is not how they are usually *perceived* in industry. In order to change this, we believe two things are necessary. First, verification tools should not forget to focus on user friendliness; this will lower the barrier to acceptance by engineers that are typically used to work with highly polished tools. Meaningful error messages will also ensure a positive feedback cycle that benefits the learning curve. Second, a university curriculum should not only present computer-aided reasoning tools as part of an (elective) theoretical course on program logics, as this will cement the notion that these tools are mostly an academic pursuit. Students should also be encouraged to explore the application of these tools to small but realistic problems.

[8]http://why3.lri.fr/try/

6 RELATED WORK

Formal verification of software is a research area for which interest is still growing. It is therefore not surprising that attention is paid to it within higher education, also because of the great social importance of secure software.

We note that both fully automated proof tools and semi automated proof assistants are used within education. In a number of cases, this is done as support when teaching students about mathematics topics that are often experienced as difficult. For example, [13] presents the design of the web server ProofWeb for Coq (developed to avoid installation difficulties with different versions of Coq), which is used to teach logic to undergraduate students. Another project [6] uses the Coq proof assistant based on a two-step approach. When teaching students, the authors strictly stick to Coq in the first step. Thereafter, in the second step, they encourage students to gradually convert less formal ordinary textbook proofs into formal Coq proofs. The GeoGebra tool [15] is an automated reasoning tool for discovering theorems on constructed geometric figures, and proving these theorems automatically. The tool is intended to serve as a guiding stick fostering student activities while learning elementary geometry.

In the context of computer science teaching, formal verification is generally introduced at a more advanced level. A theorem prover is not a learning goal in itself but is rather considered as a framework for teaching other subjects. The idea is that using a formal language as a means for introducing new concepts helps student to get a deeper understanding of these. For example, [22] reports positive experiences in teaching students to reason about the correctness of functional programs written in Scheme using the DRACULA programming environment, that uses ACL2 to discharge proof obligations, but where teaching ACL2 itself is not the main objective. As another example, [20] is a textbook on semantics entirely based on the proof assistant Isabelle[9], and the NASA PVS Library[10] contains a full formalization of Nielson and Nielson's textbook [19] on formal semantics. The main advantage of using a proof assistant in the teaching is that it allows students to experiment with their specifications, and to make proofs that are guided by the proof assistant which gives them immediate feedback. RISCAL [26] is a language for modelling algorithms and their properties. This language comes with a tool supporting model development and automatic verification. The tool has been used in two courses at the computer science department of the Johannes Kepler University Linz: (1) The course 'Formal Methods in Software Development' for master students, and (2) the 'Logic' course for undergraduate students. First experiences are promising: a small scale study indicated students seem to perform better if they can use the tool

[9]https://isabelle.in.tum.de/
[10]https://github.com/nasa/pvslib

in its full potential. The PEST framework [7] is similar to RISCAL in the sense that is provides both a specification language and a tool (available as a plug-in for Eclipse) that facilitates automated reasoning. Classroom experiences (the framework was used in two undergraduate courses taught at the computer science department of the University of Buenos Aires) confirm the preliminary results of [26].

In [23] experiences are discussed with teaching formal program specification and verification using the specification language JML and the automated program verification tool ESC/Java2. The authors state that current program verification technology is sufficiently mature for students to use, even as part of courses which are not specifically about formal methods, such as standard programming or software engineering courses. However, the authors also indicate that the use of these tools is better limited to controlled experiments, where the students work with (relatively small) supplied programs, rather than code they develop themselves.

Other experiences are based on the approach in which students are supposed to develop loop invariants before actually writing their code, which is also known as *invariant based programming*. Invariant based programming was already introduced in the 1960s ([18],[14]), and developed further by e.g. [9]. In [1], the results are discussed of using this approach in two different courses. In both courses, the SOCOS [2] environment was used to develop invariant diagrams. The environment computes the verification conditions (VCs) automatically for all transitions in these diagrams, and sends them to either an automatic prover (SIMPLIFY [8]) or to an interactive prover (PVS [21]), similar to the technique that is employed by Why3. One of the courses was an advanced course for graduate students where they were asked to prove program correctness of the generated VCs using PVS. The second course was a beginners course, where the students could discharge the VCs to the SIMPLIFY SMT solver to perform automatic proving, and to PVS for those that could be automatically proved. In both cases, the authors observed that a suitable error reporting mechanism is clearly needed when using these tools in education. In particular, they commented on the difficulties of students when dealing with PVS. They also warn against the pitfall that the tools invite students to use a try-and-debug strategy instead of thinking beforehand about the constraints needed for the invariants.

Another project [24] presents a method to gain insight into the difficulties that students face while developing suitable loop invariants, and assist them in the process. The authors collected data in the background as students attempted to produce verified code with loop invariants. Analysis of this data indicated the kinds of information that can expected, and what kinds of feedback might be useful. In [16] the authors report on an experiment with invariant based programming. They analysed a group of novice students and found that the main difficulty seemed to be lack of skills in formalizing expressions in general, rather than inventing specific invariants. Hence, to successfully use invariant based programming, appropriate training to develop these more general formalization skills is essential.

7 CONCLUSION

As the literature shows, tools for computer-aided reasoning have been used in the classroom successfully for many years, and we encourage this. It is clear that a computer can give students more instant feedback, and is less likely to make a mistake.[11] Also, in logic courses, it can alleviate tedium by offering automation.

The question that we sought to answer is whether computer-aided reasoning tools are also getting mature enough they can not only be used in the classroom for teaching, but that we can also train software engineers in industry to apply them to solve real—although perhaps small—problems in a short amount of time. Our results support the conclusion that this is the case.

Our survey indicates that our student sample is fairly representative for students who have completed a Masters degree in computing science, many of whom will eventually pursue a career as a software engineer. In fact all of our students, being in the possession of a Bachelor's degree, could just as well have been working in this field already.

A purist approach would only allow students to use automation offered by powerful tools only after the student demonstrates proper understanding of the actions that are being automated. When dealing with Why3, this is impossible—due to the fact that it relies on state-of-the-art SMT solvers to prove goals—and, we believe, not necessary. Students (and software engineers) with a sufficient level of higher education will have developed intuitions about reasoning about programs, and have had formal training in programming and logic. They can draw upon these experiences when learning Why3 (or we expect, similar tools) in an applied setting.

As a side effect, we also predict that trying out formal verification tools in a realistic setting on students will provide the developers of these tools with invaluable feedback. Having physical access to a novice user base trying to apply these tools will give insight into what makes them difficult to learn, or where they need to be more powerful. Ultimately, this will result in computer-aided reasoning that will be more usable and powerful for everybody.

ACKNOWLEDGMENTS

The authors thank Léon Gondelman for providing ideas in setting up the course on Why3.

REFERENCES

[1] Ralph-Johan Back. Invariant based programming: basic approach and teaching experiences. *Formal Aspects of Computing*, 21(3):227–244, May 2009.

[2] Ralph-Johan Back and Magnus Myreen. Tool support for invariant based programming. Technical Report 666, TUCS - Turku Centre for Computer Science, Turku, Finland, Feb 2005.

[3] Jon Bentley. Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873, September 1984.

[4] Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. TweetNaCl: A crypto library in 100 tweets. In Diego Aranha and Alfred Menezes, editors, *Progress in Cryptology – LATINCRYPT 2014*, volume 8895 of *Lecture Notes in Computer Science*, pages 64–83. Springer-Verlag Berlin Heidelberg, 2015.

[5] Francois Bobot. MPRI lecture 2-36-1. http://francois.bobot.eu/mpri2018/, 2018.

[6] Sebastian Böhne and Christoph Kreitz. Learning how to prove: From the coq proof assistant to textbook style. In Pedro Quaresma and Walther Neuper, editors, *Proceedings 6th International Workshop on Theorem proving components for Educational software, ThEdu@CADE 2017, Gothenburg, Sweden, 6 Aug 2017.*, volume 267 of *EPTCS*, pages 1–18, 2017.

[11] For instance, we would not have trusted ourselves to correctly check a pen-and-paper proof of the verification challenge described in Section 3.1.2.

[7] Guido de Caso, Diego Garbervetsky, and Daniel Gorín. Integrated program verification tools in education. *Software: Practice and Experience*, 43(4):403–418, 2013.

[8] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.

[9] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, Sep 1968.

[10] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, Jun 2016.

[11] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[12] Jean-Christophe Filliâtre. Deductive program verification with why3: A tutorial. UniGR Summer School on Verification Technology, Systems & Applications 2018, 2018.

[13] Maxim Hendriks, Cezary Kaliszyk, Femke van Raamsdonk, and Freek Wiedijk. Teaching logic using a state-of-the-art proof assistant. *Acta Didactica Napocensia*, 3:35–48, 2010.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[15] Zoltán Kovács, Tomas Recio, and M Vélez. Using automated reasoning tools in geogebra in the teaching and learning of proving in geometry. *International Journal for Technology in Mathematics Education*, 25:33–50, 07 2018.

[16] Linda Mannila. Invariant based programming in education - an analysis of student difficulties. *Informatics in Education*, 9(1):115–132, 2010.

[17] F. L. Morris and C. B. Jones. An early program proof by alan turing. *Annals of the History of Computing*, 6(2):139–143, April 1984.

[18] Peter Naur. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6(4):310–316, Jul 1966.

[19] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[20] Tobias Nipkow and Gerwin Klein. *Concrete Semantics – With Isabelle/HOL*. Springer International Publishing, 2014.

[21] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction—CADE-11*, pages 748–752, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[22] Rex Page, Carl Eastlund, and Matthias Felleisen. Functional programming and theorem proving for undergraduates: A progress report. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*, FDPE '08, pages 21–30, New York, NY, USA, 2008. ACM.

[23] Erik Poll. Teaching program specification and verification using jml and esc/java2. In Jeremy Gibbons and José Nuno Oliveira, editors, *Teaching Formal Methods*, pages 92–104, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[24] C. Priester, Y. Sun, and M. Sitaraman. Tool-assisted loop invariant development and analysis. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, pages 66–70, April 2016.

[25] Ed Schouten. CloudABI, cloud computing meets fine-grained capabilites. https://www.bsdcan.org/2015/schedule/track/Security/524.en.html, 2015.

[26] Wolfgang Schreiner. Theorem and algorithm checking for courses on logic and formal methods. In Pedro Quaresma and Walther Neuper, editors, *Proceedings 7th International Workshop on Theorem proving components for Educational software, THedu@FLoC 2018, Oxford, United Kingdom, 18 july 2018.*, volume 290 of *EPTCS*, pages 56–75, 2018.