Online Teaching of Verification of C Programs in Applied Computer Science

Matthias Güdemann(✉)

University of Applied Sciences (UAS) Munich, Munich, Germany
matthias.guedemann@hm.edu

Abstract. This is a report on teaching formal methods in the form of program verification for Master students in an applied computer science setting. The course was taught fully online, using recorded videos, synchronous sessions, the learning management system Moodle (https://moodle.org/), a distributed version control system and mostly biweekly graded practical assignments.

The first objective was to use the C language. It is a very relevant language in the sectors where verification is used in industry. The students already know the language, it also has interesting properties which can make verification challenging and shows the importance of edge cases in verification. The second objective was to teach the use of mature, industrial-strength tools in order to make the skills transferable to the later work situation of the students. This required tools that are actually used in industry to analyze C programs. The third objective was to introduce different verification approaches and to show the strengths and potential limitations of each. The selected approaches were deductive verification, abstract interpretation and model checking.

To achieve these goals, Frama-C with its WP and EVA plugin, the model checker CBMC and the Z3 SMT solver were selected. Because of the applied setting it was desired to use examples which did not require the use of interactive theorem proving for deductive verification.

1 Introduction

Teaching formal methods and program verification is an important part of computer science education. Just writing specifications of programs is often hard for the students and having to do so is a very valuable experience in its own. Being able to prove properties of programs gets more and more widespread in many domains, in particular as program security gets ever more important.

At the same time, formal methods are often considered to be a theoretic or academic topic without clear application in practice. In particular in an applied computer science setting where the focus is less on research and theoretical foundations and more on applicable topics. Often it is also functional programming languages and dependent types which are used in program verification. This has the advantage of having the Curry-Howard isomorphism as a clear correspondence between programs and proofs. The downside is that while functional programming aspects are used more and more in modern programming languages,

© Springer Nature Switzerland AG 2021
J. F. Ferreira et al. (Eds.): FMTea 2021, LNCS 13122, pp. 18–34, 2021.
https://doi.org/10.1007/978-3-030-91550-6_2

programming purely in functional languages is still limited to very few niches. Therefore, it is unlikely that many of the students will do this later in their jobs.

Choosing C as a language for program verification has some extra challenges but also benefits. C has a lot of rather special and difficult aspects. At the same time it is widely used in domains where program verification is mandatory. Also, there exist different industrial strength verification tools for C which allows for hands on experiments in verification on real programs.

The rest of the paper is structured as follows: Sect. 2 gives some background on the university, C verification and the lecturer. Section 3 introduces the verification tools that were selected for the course. Section 4 gives some detail on the online teaching setting and Sect. 5 gives an overview of the exercise assignments the students had to complete. Section 6 reports on the challenges the students faced and the evaluation of the course by the students. Section 7 concludes the paper with some outlook on the changes for the next iteration of the course.

2 Background

2.1 University of Applied Sciences

The German University of Applied Science (UAS) is traditionally a type of university with a principal focus on teaching applied topics. For the professors at UAS it is a requirement to have worked at least 3 years outside academia and most have at least 5 years of industrial experience. Teaching is generally focused on current topics and with applicability in industry in mind.

In recent years the image of UAS is changing and the focus changes in the direction of research, in particular applied research, i.e., topics which show promise of commercial use in a short time-period. This shift to research has also changed the topics that are taught in computer science curricula. In Munich this has led to the introduction of formal methods teaching in the form of program verification and model-checking. Such a focus is still rather uncommon at UAS but the experience shows that the students do see the merits of formal methods if taught in a way that shows real applicability.

2.2 C Program Verification

Choosing C as the target language for program verification was a compromise between the complexity of the properties to show and the applicability of the topic in a later industrial setting. C is still a widely used language in embedded systems and safety critical domains which is an important domain for program verification.

The complexity of the properties is limited due to C not being designed with verification in mind. On the contrary, C has aspects like undefined or implementation defined behavior which makes verification tricky. At the same time this offers interesting topics for discussion in the classroom when implicit assumptions the students have turn out to be false.

Using C for verification can be applicable in an industrial context because C is still widely used. This is because compilers exist for almost every architecture and also because it allows very fine-grained control which can be essential in embedded systems or low level development like operating systems. This important position of C also means that there exist industrial grade verification tools. Learning these tools is also a potential source of readily applicable knowledge for the students.

3 Verification Approaches and Tools

The choice of verification approaches and of tools is closely connected. It was important to select modern tools that are capable of reading real C programs and not just subsets of C or abstracted languages which lack some of the really challenging aspects. We chose verification approaches, deductive verification, abstract interpretation and software bounded model-checking. This was done to show their respective strengths and weaknesses of these approaches.

Because of the curriculum of the UAS there was no real background in mathematical logic, therefore it was important have more or less full automation support even for deductive verification. Introducing interactive theorem proving would have taken too much time for the course.

The Frama-C framework [3,10] offers support for expressing properties in the Ansi C specification language (ACSL) [2]. This provides a good integration into C programs as annotations of functions or definitions of logic functions. At the time of the course the current version of Frama-C was 21.0. C programs with ACSL annotations can be compiled and executed just as normal C programs without these annotations. All ACSL annotations are expressed in specially formatted comments.

3.1 Deductive Verification

For deductive verification the Frama-C/WP plugin was chosen. It uses ACSL specifications of functions and uses preconditions and loop annotations to create proof obligations to prove the properties. The proof obligations can be discharged by different external tools. Frama-C uses the Why3 platform [9] which supports different SMT solvers, first order logic theorem provers or external interactive theorem provers. There is an extensive work on formalization and verification of standard algorithms in C available [6].

The following code illustrates an ACSL annotation in the form of a two part loop invariant and a loop variant.

```
/*@
  loop invariant 0 <= i <= n;
  loop assigns i;
  loop variant n - i;
 */
for(int i = 0; i < n; i++) { ... }
```

The following code illustrates an ACSL annotation of a function contract. It specifies that the postcondition of the function is that b points to the original pointer of a and that a points to the original pointer of b, i.e. the values at the pointers have been exchanged when the function returns. The original pointer is marked as *old*, one can specify own labels in addition to standard labels that are available.

```
/*@
  ensures *b == \old(*a);
  ensures *a == \old(*b);
 */
void swap (int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

3.2 Abstract Interpretation

For abstract interpretation the Frama-C extended value analysis (EVA) [4] plugin was chosen. It allows for fully automatic verification of ACSL specifications using abstract interpretation and related techniques. It uses internal abstract domains and different options that control the analysis, e.g., the number of internal states to analyze.

The advantage of this kind of analysis is that if no error is reported, then no runtime error is possible in the program. If an error is reported, this means that in the abstraction a runtime error can occur, so then there *may* be a problem in the concrete program.

```
int f(int a) {
  int x, y, sum, result;
  if(a == 0) {
    x = 0; y = 5;
  } else {
    x = 5; y = 0;
  }
  sum = x + y;
  result = 10 / sum;
  return result;
}
```

Given the above code the plugin manages to prove the absence of a runtime error in the form of division by zero. This is shown in the below output of Frama-C. While in the resulting value sets the variables x and y can have either value 0 or 5, the value of $sum = x + y$ cannot have the value 0 because x and y cannot have value 0 at the same time.

```
[eva] done for function f
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function f:
  x in {0; 5}
  y in {0; 5}
  sum in {5}
  result in {2}
[eva:summary] ====== ANALYSIS SUMMARY ======
  ----------------------------------------------------------------
  1 function analyzed (out of 1): 100% coverage.
  In this function, 8 statements reached (out of 8): 100% coverage.
  ----------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ----------------------------------------------------------------
  0 alarms generated by the analysis.
  ----------------------------------------------------------------
  No logical properties have been reached by the analysis.
  ----------------------------------------------------------------
```

3.3 Software Bounded Model-Checking

For software bounded model-checking the CBMC [11] tool was chosen. It allows for specification of assertions directly in C code and uses bit-precise model-checking using SAT and SMT solving to verify or disprove the properties. From a teaching point of view, the possibility to get counterexamples for false properties is a very interesting feature of model-checking.

In addition to CBMC, pure SMTLIB2 SMT solving in the form of Z3 [8] and CVC4 [1] was used to illustrate the formalization of lemmas in the form of satisfiability of constraint problems. Frama-C/WP also uses SMT solvers for deductive verification, but the proof obligations and encoding of these problems were not detailed in the lecture.

CBMC allows for textual generation of verification conditions in different formats. This is a useful feature to illustrate how the C programs are transformed into single static assignment (SSA) and then translated into constraint problems for SMT solving.

In the following code the assertion specifies that the sum of parameter x and y cannot be zero.

```c
  int f(int n, int x, int y) {
    int divisor = 0x12345678 - x + (y << 1);
    assert (divisor != 0);
    return n / divisor;
}
```

CBMC translates this into the following constraint problem. This is direct output of CBMC, slightly shortened to reduce it to the essential part.

```
{-22} f::n!0@1#1 = nondet_symbol
{-23} f::x!0@1#1 = nondet_symbol
{-24} f::y!0@1#1 = nondet_symbol
{-25} f::1::divisor!0@1#2
       = 305419896 + shl (f::y!0@1#1, 1) + -f::x!0@1#1
|---------------------
{1} ¬(f::1::divisor!0@1#2 = 0)
```

The first 3 lines of the constraint problem state that the three parameters of the function f are equivalent to a nondeterministic value, i.e., the constraints 22, 23 and 24 are always fulfilled. The constraint 25 then defines that the local variable *divisor* of the function f is equal to the right side which corresponds to $305419896 + (y \ll 1) - x$. These constraints describe the program in single static assignment (SSA) in the form of equivalences that relate the variables and parameters.

From these constraints CBMC then tries to deduce the property in the form of the assertion $\neg(divisor = 0)$. This is done by negating the property, i.e., adding the additional constraint $divisor = 0$. An SMT solver then checks satisfiability of the conjunction of the constraints and the negated property. If this is satisfiable then there exist parameter n, x and y such that $divisor = 0$. A satisfying assignment comprises a counterexample to the property.

For this program and property it is of course possible to choose the function parameters in such a way that a division by zero is possible. The following shows the counterexample as generated by CBMC. This means that with x equal to -1841559528 and y equal to 1073993936 the calculated value of *divisor* is 0, the value of n is not important. The possibility to get counterexamples is very useful: “It is impossible to overestimate the importance of the counterexample feature” [7].

```
State 33 file div.c function __CPROVER__start line 5 thread 0
----------------------------------------------------
  n=0 (00000000 00000000 00000000 00000000)

State 34 file div.c function __CPROVER__start line 5 thread 0
----------------------------------------------------
  x=-1841559528 (10010010 00111100 00001000 00011000)

State 35 file div.c function __CPROVER__start line 5 thread 0
----------------------------------------------------
  y=1073993936 (01000000 00000011 11011000 11010000)

State 36 file div.c function f line 6 thread 0
----------------------------------------------------
  divisor=0 (00000000 00000000 00000000 00000000)
```

```
Violated property:
  file div.c function f line 7 thread 0
  assertion divisor != 0
  divisor != 0
```

4 Online Teaching

Due to the restrictions because of the COVID-19 pandemic, the course was taught fully online. In addition, because of the restrictions concerning in-person exams, grading was done on practical exercises. Each exercise was for around 2 weeks and could be completed in teams of two students or alone.

Online teaching worked quite well. The exercises were organized via github classroom[1] which allows for easy creation of repositories from templates for the students.

The course was held in the following way: each week there was an asynchronous part where new material was distributed as recorded videos and slides. At the normal lecture date there was a synchronous session where the material was presented in more detail. In the synchronous part the students were also asked to complete several small multiple-choice quizzes per session. Most sessions also included live-demos of the relevant aspects of the currently used tools.

We also employed Rocket.Chat[2] which proved to be very helpful to exchange code snippets to discuss problems or questions for the practical exercises. It also integrates Jitsi[3] to support video calls and live screen sharing.

To prevent most kinds of compatibility problems we decided to use a standardized virtual machine as the software platform. The VM was based on a standard Ubuntu Linux with the different tools preinstalled. Frama-C can easily be installed via the *opam* package manager for OCaml. CBMC and Z3 are directly available as packages in Ubuntu.

5 Exercise Selection

Due to the fact that a basic course in mathematical logic was not compulsory in the students' curriculum it was necessary to start with basics of specification using first order logic. From these foundations we then switched to formal specification and Hoare logic.

For each exercise we give a short paragraph on the preceding preparation lectures, the task itself and the goal of the exercise. Tools like Frama-C, SMT solvers and CBMC were presented in live-demo sessions in the synchronous sessions.

[1] https://classroom.github.com.
[2] https://rocket.chat/.
[3] https://jitsi.org/.

5.1 Exercise 1—Informal Specification

graded no/**time** 1 week

Preparation. In the lecture before this exercise the students got an introduction to propositional logic.

Task. The first exercise was to clone a repository which contained a single C file and to analyze informally what the function in the C file would do. The students were asked to compile the file, execute it and to validate their guess what the function *f* computes.

```c
int f(int n) {
  int s = 0;
  int i = 1;
  while (i <= n) {
      s = s + i;
      i++;
  }
  return s;
}
```

The next step was to write down a specification of what the function computes. This specification was intended to be informal and it also was the first time the students had to write a specification on their own.

Finally, the students were asked to think about edge cases for which the function might not fulfill the specification.

Goal. The intention of this exercise was to familiarize the students with C programs, to get an idea about the difficulties to express precisely what a function is intended to do and also with the fact that machine integers do not always behave like unbounded integers.

5.2 Exercise 2—First Order Logic

graded no/**time** 1 week

Preparation. In the lecture before this exercise the students got an introduction to first order logic with many different examples of formalized properties.

Task. The next exercise was to express the specification of exercise 1 as a first order logic formula. Still, in free-form, not yet in a standardized way like ACSL.

Goal. The intent here was to familiarize the students with the challenge to use logic to correctly specify a property.

5.3 Exercise 3—Hoare Logic

graded yes/**time** 1 week

Preparation. In the lecture before this exercise the students got an introduction to different approaches to program testing and coverage criteria, proof trees and Hoare logic. For Hoare logic reasoning a simple imperative language was introduced to explain the separate rules for the different language constructs.

Task. This exercise was the first graded exercise in the course. It included simple properties which had to be proven using Hoare logic and manually writing down the proof tree of the Hoare rule applications.

This included the calculation of the weakest precondition of the following programs.

```
  // which precondition is required for postcondition y > 1?
  y := x + 1;

  // which precondition is required for postcondition z > 0?
  y := x + 1;
  x := y + 1;

  // which precondition is required for postcondition z > 0?
  z := x * y;
```

The next part was to specify a loop invariant such that with the precondition $n \geq 0$ the postcondition $acc = 2 * n$ holds. It was also asked to give the loop variant which guarantees termination.

```
acc := 0;
i := 0;
while (i < n)
  acc := acc + 2;
  i := i + 1;
```

The last part then asked to generalize the *while* rule to a rule for *for* loops, i.e. to specify how a Hoare-style rule for *for* loops would have to look like in order to prove correctness of the Hoare triple.

Goal. The intent of this exercise was to familiarize the students with Hoare logic reasoning which is at the base of deductive verification. Loop invariants (and variants) generally have to be specified manually. Understanding how the Hoare logic rules for loops work is a very important concept in verification of imperative programs.

5.4 Exercise 4—Deductive Verification Using Frama-C

graded yes/**time** 1 week

Preparation. In the lecture before this exercise the students got an introduction to the ACSL specification language for C and to Frama-C. This consisted mainly of the ACSL operators for first order logic and the specific keywords to express function preconditions, properties, assertions and loop invariants.

For Frama-C this included running the command line version of the tool on an annotated C file and the interpretation of the resulting output messages.

Task. The next exercise was based directly on exercise 3. For this exercise, the programs were given as C programs, the pre- and postconditions and the loop invariants had to be expressed as ACSL annotation before and after each statement, corresponding to the Hoare triple. The goal was to prove the postconditions from the specifications and loop invariants using Frama-C/WP in a fully automatic way. The exercise also included the formalization of a lemma for multiplication and to check which of the SMT solvers was capable to verify the lemma automatically.

Goal. The intent of this exercise was to familiarize the students with ACSL specifications and with using Frama-C. We limited the use to the command line interface which is more than adequate for tasks like these. The exercise did not yet use fixed-width machine integers, any runtime warnings were to be ignored.

5.5 Exercise 5—Arrays

graded yes/**time** 2 weeks

Preparation. In the lectures before the students got a reminder on peculiarities of the C language, in particular pointers, as well as an introduction on control flow graphs (CFG) as program representations.

Task. The next exercise dealt with more complex specifications. There were three parts. The first part was to specify the return value of a function that computes the minimum of two integers.

```c
int min(int x, int y) {
  int z = x < y ? x : y;
  return z;
}
```

The second part was the first task to include arrays. For a given array of integers and its length, the index of the minimal element was to be returned. If no such element exists, then a special value had to be returned.

```
int min_array(int* arr, int len) {
  if (len == 0)
    return -1;

  int min = 0;

  int i;
  for (i = 0; i < len; i++) {
      if(arr[i] < arr[min])
        min = i;
    }
  return min;
}
```

The third part consisted of finding the smallest non-negative value in a sorted array. The specification here included specifying that the values in the array are sorted in a non-decreasing order.

```
int min_pos_array(int* arr, int len) {
  if (len == 0)
    return -1;

  for (int i = 0; i < len; i ++) {
      if (arr[i] >= 0)
        return i;
    }
  return -1;
}
```

Goal. The intent of this exercise was to familiarize the students with function contracts in ACSL in addition to the statement annotations. Already the specification of minimum is non-trivial, several solutions only specified that the return value should be less than or equal to both input parameters.

For the sorted array, several of the students specified a pairwise predicate, comparing only direct successor elements. This led to problems with the automatic provers. The SMT solvers required a global specification of a sorted array in order to provide fully automated proofs. This illustrated the difference between a specification which is good for verification and a specification which would be easy to translate into an efficient implementation.

5.6 Exercise 6—Runtime Errors

graded yes/**time** 2 weeks

Preparation. In the lecture before this exercise the students got an introduction into the possible runtime errors of C programs. This also included the different warnings that Frama-C/WP produces to prevent runtime errors from appearing. This includes mainly integer overflow/underflow, pointer validity and aliasing.

Task. The next exercise was split into two parts. The first part was to add preconditions to most of the former exercises in such a way that any runtime errors were excluded. Frama-C/WP provides an option to generate proof-obligations for showing the absence of runtime errors.

The second part was the implementation and specification of a variant of the famous *fizz-buzz* program. In this form it incorporated 3 arrays of same length. At each index divisible by 3 and 5 the first of the arrays should have a value 1 and the two others a value 0. At each index divisible by 3 the second should have value 1 and at each index divisible by 5 only the third array should hold the value 1.

Goal. The intent of this exercise was to familiarize the students with all different kinds of possible runtime errors in a language like C. This does not only include potential integer overflow or illegal memory access, but also aliasing in form of overlapping arrays.

A fully complete and correct specification of the fizz-buzz function proved to be tricky. The main implementation variant was first to zero all arrays and then fill the arrays with values 1 where appropriate. Unfortunately this solution requires a more complex loop invariant than using a single loop and filling each array at each index with the correct value 0 or 1. It also illustrated well that specifying *what* a program does exactly can be more difficult than implementing this functionality.

5.7 Exercise 7—Abstract Interpretation

graded yes/**time** 1 week

Preparation. In the lectures before the students got an introduction to abstract interpretation. This includes a simple sign domain as example and an overview of different properties that can be verified by abstract interpretation.

Task. The next exercise dealt with abstract interpretation (AI). Frama-C provides the EVA plugin which does a form of AI. Unfortunately, from a didactic perspective, this plugin is quite advanced and it is not possible to reduce the domains to simple ones like the sign domain only. While it is possible to deactivate the normal C-domain, this is discouraged by the authors of Frama-C because it is unlikely to work as expected. There seems to be a gap in the set of analysis tools for C which are based on abstract interpretation which are well adapted for teaching.

Therefore, the exercise itself was divided in a theoretical and a practical part. In the theoretical part, the students were asked to define an abstract domain using first unbounded integer intervals and to define abstract addition and multiplication for this domain. Then the domain changed to fixed bit-width integer intervals with the same task and to note the differences.

The practical part consisted of working through the Frama-C/EVA tutorial [5]. This provides some insight into how such a tool can be used to analyze C code and how to understand the functioning of an unknown program, but it allows for less learning how AI works for real programs.

5.8 Exercise 8—Bounded Model Checking

graded yes/**time** 2 weeks

Preparation. In the lectures before this exercise the students got an introduction to SMT solving and bounded model checking. Specifically the SMTLIB2 format was presented as a standard interchange format for modern SMT solvers.

For bounded model checking, the single static assignment (SSA) form was introduced and it was shown how a program in this form can be expressed in SMTLIB2 using different underlying logics.

Finally, the CBMC tool was presented with the required options like loop unwinding. It was also shown how assumptions can be used to formalize specific properties and how standard coverage properties can be generated based on CFG representations.

Task. The last exercise dealt with bit-precise model-checking of C programs in the form of CBMC and formalizing a problem directly as SMT constraint problem. The first part of the exercise was to analyze the famous *fast inverse square root* program used in a popular 3D-shooter game in 1999[4]. The students were asked to formalize the property that the relative error of this routine was below a threshold for an interval of possible input parameters. The following code[5] is available under a GPL license, the comments have been removed.

```
float Q_rsqrt( float number )
{
        long i;
        float x2, y;
        const float threehalfs = 1.5F;
        x2 = number * 0.5F;
        y  = number;
        i  = * ( long * ) &y;
        i  = 0x5f3759df - ( i >> 1 );
        y  = * ( float * ) &i;
        y  = y * ( threehalfs - ( x2 * y * y ) );
        return y;
}
```

[4] https://en.wikipedia.org/wiki/Fast_inverse_square_root.

[5] https://github.com/id-Software/Quake-III-Arena/blob/master/code/game/q_math.c.

In the second part of the exercise the students were asked to specify an invariant and to verify using CBMC that the following C program computes the absolute value of a given input `float`.

```
float myabs(float v) {
  if(v < 0)
    return -v;
  else
    return v;
}
```

The last part of the exercise was to formalize and prove the following lemma as an SMT problem in `QF_FP` logic, where *float* is the set of 32-bit IEEE 754 floating-point values.

$$\forall x, y \in float : x \times y < 0 \rightarrow (x < 0 \lor y < 0)$$

Goal. The intent of this exercise was to familiarize the students with the way how invariants (in the form of assertions) can be formalized to prove non-trivial properties of C programs. The formalization of a lemma in the form of a satisfiability problem was intended to familiarize the students with the approach to prove a property by showing that the negation is unsatisfiable.

6 Evaluation

Overall the course worked quite well. There were almost no technical problems, mainly due to the fact that a pre-installed virtual machine was provided. The performance of the VM was more than enough, in particular with hardware-accelerated virtualization.

6.1 Challenges for Students

A big challenge for the students was to understand the reason why deductive proofs did not work. There are mainly two reasons for this: i) the property might not be fulfilled or ii) the pre-conditions or loop invariants are not strong enough to prove the post-condition.

Frama-C lists the proof-obligations which cannot be discharged. The challenge is that these are reported in the form of first order logic which is non-trivial to map back to the original C source code. The main options to alleviate that problem is to use named annotations which allows for more fine-grained reporting. It is also possible to use the Frama-C GUI which shows the annotations at the source code. Still, in both options the proof obligation is provided encoded in first order logic which is non-trivial to understand.

Overall it is clear that deductive verification is very powerful and at that same time also quite challenging to do. In particular for loop invariants the automated tool support is limited. In the end it is necessary to fully understand the program and also to understand the peculiarities of the C language. Therefore, we consider this more a feature than a problem, verification requires understanding of both the problem and the programming language in order for someone to be able to formalize and solve a problem and then to prove the correctness thereof.

6.2 Results

The overall results of the course were quite good. All students that participated in doing the exercises passed the course. The grades start at 1.0 (best) and go down to 4.0 (worst), 5.0 represents a failure to pass the course. The average grade was 1.52 with the worst grade being 2.3.

The traditional format is to have a written or oral test at the end of the semester. Due to the pandemic this was changed to grade the exercises directly and do a short interview to check whether the students did the work themselves. In these interviews it often became obvious for the students why certain properties would not be proven or what was lacking to have a fully correct specification.

Runtime errors due to overflow seem to be a common knowledge, understanding those did not pose any difficulty to the students. The main challenges were correctly specifying properties and understanding aliasing in C.

For correctness of specifications it might make sense to stress more to check that wrong results are actually not validated. An example would be to show that a specification does not validate an incorrect input to fizz-buzz where the respective entries hold a value of 1 but the others are not necessarily 0 (a rather common error in the specifications).

6.3 Student Evaluation of the Course

At the UAS Munich, every course is evaluated by the participating students. The evaluation is done close to the end of the semester, but before the final test and therefore before the grades are known. Overall 11 of the 17 students in total in the course did respond to the survey. The full results are available in German[6], a summary is shown in Table 1 and Table 2, numbers represent the percentage of the students.

Table 1. Summary of student responses for the course

	Too small	Small	Good	Much	Too much
The amount of learning matter is	0	9.1	81.8	9.1	0
The pace of the course is	0	0	100	0	0
For me the requirements are	0	9.1	81.8	9.1	0
The share of self-learning is	0	0	90.9	9.1	0

[6] https://guedemann.org/downloads/Evaluierung_Programmverifikation.pdf.

Table 2. Summary of student responses for their experience

	Very negative	Negative	Neutral	Positive	Very positive
I find the topic is more interesting than I did before	0	0	0	45.5	54.5
I learned a lot in the course	0	0	0	36.4	63.8
I enjoy participating in the course	0	0	9.1	36.4	54.5
I would recommend the course	0	0	9.1	0	90.9
Rating I would give to the course	0	0	0	20	80

Having the standardized virtual machine and software installation allowed for live demos and parallel participation of the students. Exchanging code via Rocket.Chat proved to be very efficient in comparison with screen sharing. Most code used in program verification is rather short, so exchange via text is feasible. Working mostly with command line tools (Frama-C has a GUI but use was mostly text oriented) proved well suited to this mode of online teaching.

This view was shared by the students, one of the free text evaluations said:

> "The communication via Rocket.Chat works very well, better than expected. I like the polls during the synchronous lecture as this invites active participation."

7 Conclusion and Outlook

Overall the course and its content was well-received by the students. Using C as language for verification made the course interesting because of the applicability to real world programs. Choosing Frama-C and CBMC as the main tools for the analysis and verification proved to be beneficial as these are mature, industrial strength tools. Using a virtual machine greatly reduced the technical problems with installation and compatibility.

The choice of exercises was for the most part rather conventional and probably more on the easy side. A next iteration should probably include one or two more challenging tasks than this first time. Unfortunately it is not easy to choose good exercise problems for deductive verification, in particular if fully automatic verification is desired. Since the last iteration there is a new version of Frama-C which might have more options for interactive proofs than just resorting to the Coq interactive theorem prover. We feel that interactive theorem proving is a separate topic which would require using a different language than C for verification. This would reduce the applicability of the learning matter for our use-case.

Another challenge is finding a tool for abstract interpretation of C programs which works well with very simple domains. It would be possible to add such features to CBMC. There is an implementation of interval domains based on CBMC called `intervalAI`[7]. Unfortunately it is limited to the interval domain only, and also does not compile on current versions of `gcc` as is based on an older version of CBMC.

For a next iteration of this course this will likely change the sequence of topics and also shift the focus of the practical exercises from abstract interpretation to model-checking. Model-checking has the feature that a property that cannot be proven results in a counterexample which can be analyzed. This provided excellent direct feedback which allows for teaching about special and edge cases of fixed width vector based arithmetic of integers and IEEE 754 floating-point. Abstract interpretation will likely become a more theoretical topic, to be used on a whiteboard. It is a powerful technique in practice but without proper tool support it is difficult to teach in an applied setting.

References

1. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
2. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI C specification language. CEA-LIST, Saclay, France, Technical report v1 2 (2008)
3. Blanchard, A.: Introduction to C program proof with Frama-C and its WP plug-in. https://allan-blanchard.fr/frama-c-wp-tutorial.html
4. Bühler, D.: EVA, an evolved value analysis for Frama-C: structuring an abstract interpreter through value and state abstractions. Ph.D. thesis, Rennes 1 (2017)
5. Bühler, D., et al.: Eva-the evolved value analysis plug-in. https://frama-c.com/download/frama-c-eva-manual.pdf
6. Burghardt, J., Gerlach, J., Hartig, K., Pohl, H., Soto, J.: ACSL by example. DEVICE-SOFT project publication. Fraunhofer FIRST Institute (2010)
7. Clarke, E.M.: The birth of model checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 1–26. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0_1
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Filliâtre, J.C., Paskevich, A.: Why3 – where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems, pp. 125–128. Springer, Heidelberg (2013)
10. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015)
11. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26

[7] https://github.com/sukrutrao/IntervalAI.