A Lesson on Proof of Programs with Frama-C. Invited Tutorial Paper

Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles

CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
`firstname.lastname@cea.fr`

Abstract. To help formal verification tools to make their way into industry, they ought to be more widely used in software engineering classes. This tutorial paper serves this purpose and provides a lesson on formal specification and proof of programs with FRAMA-C, an open-source platform dedicated to analysis of C programs, and ACSL, a specification language for C.

Keywords: deductive verification, Frama-C, ACSL, program specification, teaching.

1 Introduction

Recent advances on proof of programs based on deductive methods allow verification tools to be successfully integrated into industrial verification processes [1, 2]. However, their usage remains mostly confined to the verification of the most critical software. One of the obstacles to their deeper penetration into industry is the lack of engineers properly trained in formal methods. A wider use of formal verification methods and tools in industrial verification requires their wider teaching and practical training for software engineering students as well as professionals.

This tutorial paper presents a lesson on proof of programs in the form of several exercises followed by their solutions. It is based on our experience in teaching at several French universities over the last four years. This experience shows that, for the majority of students, theoretical courses (like lectures on Hoare logic [3] and weakest precondition calculus [4]) are insufficient to learn proof of programs. We discuss the difficulties of the lesson for a student, necessary background, most frequent mistakes, and emphasize some points that often remain misunderstood. This lesson assumes that students have learned the basics of formal specification such as precondition, postcondition, invariant, variant, assertion.

In our lesson, we use FRAMA-C [5, 6], an open-source platform dedicated to the analysis of C programs and developed at CEA LIST. Being open-source is an important advantage for teaching: FRAMA-C is available on all major Linux distributions, and can be easily installed by a local network administrator at any university. FRAMA-C gathers several static analysis techniques in a single collaborative framework. In particular, two different plug-ins are dedicated to proof of programs: JESSIE [7, 8] and WP [9]. The latter is newer and aims to be better integrated into the rest of the platform. Up to now, we have used JESSIE in our lessons because it is more stable and its level of integration into the platform is not an issue for teaching.

All static analyzers of FRAMA-C, including JESSIE, share a common specification language, called ACSL [10]. This language allows FRAMA-C analyzers to collaborate in an effective way [11]. Before proving programs, the students must learn to formally specify them, so we include ACSL in our lesson which thereby mixes program specification and program verification. ACSL syntax was designed to stay close to C, and students do not have any problem to learn it on-the-fly with a language manual at hand. Thus we do not include a detailed presentation of ACSL in this paper.

The paper is organized as follows. The lesson is presented in three parts: discovery of the JESSIE tool (Section 2), specification of C programs in ACSL (Section 3) and their verification with JESSIE (Section 4). Section 5 presents our teaching experience feedback. Section 6 provides some related work and concludes.

2 Introductory exercices

2.1 Safety checks for arithmetic overflows

Question 1. Run JESSIE to prove the following program:

```
/*@ ensures x >= 0 && \result == x || x < 0 && \result == -x;
    assigns \nothing; */
int abs(int x) { return ( x >= 0 ) ? x : -x; }
```

Can you explain the unproved safety property? Write a precondition restricting the values of `x`, for example, to the interval $-1000 \ldots 1000$ and re-run the proof.

Answer. The postcondition for the given program is proved, but JESSIE reports an unproved safety check for an arithmetic overflow risk in `-x`. Since `INT_MIN = - INT_MAX - 1`, the expression `-x` provokes an overflow for `x = INT_MIN`. Restricting the values of `x` to $-1000 \ldots 1000$ in the precondition avoids this risk, and the proof succeeds for the complete specified program:

```
/*@ requires -1000 <= x <= 1000;
    ensures x >= 0 && \result == x || x < 0 && \result == -x;
    assigns \nothing; */
int abs(int x) { return ( x >= 0 ) ? x : -x; }
```

Discussion. Arithmetic overflows are responsible for well-known critical software crashes, but most students ignore these issues. JESSIE helps them to understand this point. Notice that the weakest possible precondition avoiding overflows in `abs` would be `x > INT_MIN`. Finally, the contract has an **`assigns`** clause that specifies that `abs` is not supposed to modify the global state of the program. We will go back on this clause in the next section.

2.2 Safety checks for pointer validity

Question 2. Consider the following function swapping the values referred by its inputs:

```
int swap(int *p1, int *p2) { int tmp = *p1; *p1 = *p2; *p2 = tmp; }
```

a) Specify the postcondition and run JESSIE to prove the program. Explain the results you observe. Add a precondition and re-run the proof.

b) Explain the role of the **`assigns`** clause you put in the postcondition. (Did you?) Give an example of a wrong implementation that would be proved by JESSIE without it.

Answer. **a)** We first add the following postcondition:

```
/*@ ensures \old(*p1) == *p2 && \old(*p2) == *p1;
    assigns *p1, *p2; */
```

Here `\old(*p1)` refers to the value of `*p1` before the call. JESSIE proves the postcondition, but indicates a safety alarm at each dereference of the pointers `p1` and `p2`. Indeed, the validity of these pointers is supposed to be guaranteed by the caller, so it should be explicitly specified by the precondition as follows:

```
/*@ requires \valid(p1) && \valid(p2);
    ensures \old(*p1) == *p2 && \old(*p2) == *p1;
    assigns *p1, *p2; */
int swap(int *p1, int *p2) { int tmp = *p1; *p1 = *p2; *p2 = tmp; }
```

After running JESSIE again, we see that the safety properties are now proved as well.

b) The **`assigns`** clause specifies here which variables can be modified by the function. Without this clause, the following erroneous implementation can be proved:

```
int shared; // a global variable that should not be modified in swap
int swap(int *p1, int *p2) { shared = *p1; *p1 = *p2; *p2 = shared; }
```

Discussion. The students often forget to specify validity of memory accesses. This exercise insists on this point and shows safety checks for pointer validity in JESSIE. Another common error is a missing or incorrect **`assigns`** clause. Considering counter-examples is an excellent way to make students aware of the problem.

3 Lesson on program specification

The goal of this lesson is to formally specify a well-known but non-trivial function that searches an element in a sorted array. This can be split in several steps of increasing difficulty to introduce the most useful ACSL constructs.

3.1 Function contracts

Question 3. Write the ACSL formal specification corresponding to the following informal specification of function `find_array`. Explain its clauses.

```
/* [find_array(arr, len, query)] returns any index [idx] of the sorted array
   [arr] of length [len] such that arr[idx] == query. If such an index does not
   exist, it returns -1. */
int find_array(int* arr, int len, int query);
```

Answer. Here is a correct answer which provides **`requires`**, **`ensures`** and **`assigns`** clauses.

```
/*@ requires len >= 0;
    requires \valid(arr+(0..(len-1)));
    requires \forall integer i, j; 0 <= i <= j < len ==> arr[i] <= arr[j];
    ensures (\exists integer i; 0 <= i < len && arr[i] == query) ==>
             0 <= \result < len && arr[\result] == query;
    ensures (\forall integer i; 0 <= i < len ==> arr[i] != query) ==>
             \result == -1;
    assigns \nothing; */
int find_array(int* arr, int len, int query);
```

The three preconditions respectively say that:

– the given array length must be positive or zero,
– the array must contain at least `len` valid memory cells which can be safely read,
– the array must be sorted.

The two **`ensures`** clauses respectively state that:

– if the array contains `query` at some index between `0` and `len-1`, then the value **`\result`** returned by the function is between these bounds, and `arr[\result] == query`,
– if all the elements of the array are different of `query`, the returned value is `-1`.

The **`assigns`** clause specifies here that no memory location may be modified by the function since it must have no observable effect on the memory from the outside.

Discussion. Maybe surprisingly, the main mistake is not related to big clauses like complex quantifications and implications. Students actually tend to forget implicit specifications on the length of the array and the validity of its cells (usually they do not forget the **`assigns`** clause after previous exercises). At this point, it is important to emphasize this specific weakness of informal specifications which often contain implicit, unwritten parts. Additionally, we can show that writing several **`requires`** and **`ensures`** clauses may be clearer than writing a single clause of each type with a big conjunction.

3.2 Behaviors

Question 4. Modify the previous ACSL specification in order to use two distinct behaviors corresponding to whether the element `query` is found or not. Explain your changes.

Answer. Here is a correct answer which defines two behaviors `exists` and `not_exists`.

```
/*@ requires \forall integer i, j; 0 <= i <= j < len ==> arr[i] <= arr[j];
    requires len >= 0;
    requires \valid(arr+(0..(len-1)));

    assigns \nothing;

    behavior exists:
      assumes \exists integer i; 0 <= i < len && arr[i] == query;
      ensures 0 <= \result < len;
      ensures arr[\result] == query;

    behavior not_exists:
      assumes \forall integer i; 0 <= i < len ==> arr[i] != query;
      ensures \result == -1; */
int find_array(int* arr, int len, int query);
```

The **`assumes`** clauses are the activation conditions of the behaviors: if in some behavior this clause is valid, the **`ensures`** clause of this behavior must be satisfied.

Discussion. The usual students' question is the difference between an **`assumes`** clause (which is an assumption) and a **`requires`** clause (which is also allowed in a behavior and contains a requirement which must be satisfied by the caller, so it needs a proof). It is also important to explain that behaviors correspond to specifying by cases and to show that this new specification is much clearer than the previous one which is equivalent and uses implications instead.

3.3 Logical predicates

Question 5. Modify the previous specification to define and use two logical predicates:

- `sorted` which states that a given array is sorted,
- `mem` which states that an element belongs to a given array.

Answer. Here is a correct answer.

```
/*@ predicate sorted(int* arr, integer length) =
      \forall integer i, j; 0 <= i <= j < length ==> arr[i] <= arr[j];

    predicate mem(int elt, int* arr, integer length) =
      \exists integer i; 0 <= i < length && arr[i] == elt; */

/*@ requires sorted(arr,len);
    requires len >= 0;
    requires \valid(arr+(0..(len-1)));

    assigns \nothing;

    behavior exists:
      assumes mem(query, arr, len);
      ensures 0 <= \result < len;
      ensures arr[\result] == query;

    behavior not_exists:
      assumes ! mem(query, arr, len);
      ensures \result == -1; */
int find_array(int* arr, int len, int query);
```

The first **`requires`** clause of the function now uses the predicate `sorted` while the **`assumes`** clauses of both behaviors use predicate `mem`.

Discussion. Some students have difficulties with this question since they do not remark that the **`assumes`** clauses of both behaviors are the exact opposite to each other: a minimal logical background is actually required here. After that, we can explain the difference between a predicate (a parameterized logical proposition), a logic function (which *defines* a logical term depending on its parameters), and a programming function (which *computes* a value depending on its parameters). The three notions allow the user to write cleaner code/specification without redundancy.

3.4 Testing the specification

Students often feel more comfortable when they can interact with the computer to gain confidence in their answer. Unlike in the verification phase of Sec. 4, usual specification process does not allow such interactions, except for type-checking the ACSL contract. It is possible to overcome this issue by providing a test function that calls the specified function on sample cases: the specification written by the students must then imply the assertions of the test function.

Question 6. Check with JESSIE that your specification allows to prove the assertions of the following function (note that the pre-condition of the last call should not be satisfied, as we deliberately give an unsorted array to `find_array`).

```
void main () {
  int array[] = { 0, 4, 5, 5, 7, 9 };
  int idx = find_array(array,6,7);
  /*@ assert idx == 4; */
  idx = find_array(array,6,5);
  /*@ assert idx == 2 || idx == 3; */
  idx = find_array(array,5,9);
  /*@ assert idx == -1; */
  array[0] = 6;
  // pre-condition should be broken
  idx = find_array(array,4,6);
}
```

Answer. The answers to Questions 3, 4 and 5 pass this test.

Discussion. This question allows students to catch some specification errors by themselves. In particular, the fact that a successful call must return a value between `0` and `len-1` is often overlooked. When missing, the first two assertions of `main` are not provable: nothing in ACSL prevents some memory cell `arr[i]` with an index `i` outside of `0..5` from containing `query` as well. Although some guidance may be required to go from noticing that an assertion is not proved up to writing a correct specification of `find_array`, this approach is very helpful for testing the specification and explaining specification problems.

3.5 Modular verification and function calls

Question 7. In the following program, the functions `abs` and `max` are declared but not defined.

```
// returns absolute value of given integer x>INT_MIN
int abs ( int x );

// returns maximum of x and y
int max ( int x, int y );

// returns maximum of absolute values of given integers x>INT_MIN and y>INT_MIN
int max_abs( int x, int y ) {
  x=abs(x);
  y=abs(y);
  return max(x,y);
}
```

a) Specify the three functions and prove the function `max_abs`.

b) Remove the precondition of the function `max_abs` and re-run the proof. Observe and explain the proof failure.

c) Remove the postcondition of the function `max` and re-run the proof. Observe and explain the proof failure.

Answer. **a)** Here is a specified program that is proved by JESSIE. The specifications are pretty much self-explanatory based on the previous examples.

```
#include<limits.h>
/*@ requires x > INT_MIN;
    ensures x >= 0 && \result == x || x < 0 && \result == -x;
    assigns \nothing; */
int abs ( int x );
```

```
/*@ ensures \result >= x && \result >= y;
    ensures \result == x || \result == y;
    assigns \nothing; */
int max ( int x, int y );

/*@ requires x > INT_MIN && y > INT_MIN;
    ensures \result >= x && \result >= -x &&
      \result >= y && \result >= -y;
    ensures \result == x || \result == -x ||
      \result == y || \result == -y;
    assigns \nothing; */
int max_abs( int x, int y ) {
  x=abs(x);
  y=abs(y);
  return max(x,y);
}
```

b) If the precondition of the function `max_abs` is removed, JESSIE does not manage to prove that the precondition of `abs` is satisfied each time it is called. Indeed, in modular verification the precondition must be ensured by the caller, and it cannot be proved if the inputs of `max_abs` can be equal to `INT_MIN`.

c) If the postconsition of the function `max` is removed, JESSIE cannot prove the post-condition of `max_abs`. Indeed, the proof of the caller relies on the contract of the callee (whose code is not necessarily defined in the same file).

Discussion. This question illustrates specific roles of preconditions and postconditions in modular verification. Note that pre- and postconditions of the caller and of the callee have dual roles in the caller's proof. The precondition of the caller is assumed and the postcondition of the caller must be ensured. On the contrary, the precondition of the callee must be ensured by the caller, while the postcondition of the callee is assumed in the caller's proof.

4 Lesson on program verification

Once a correct specification has been written, the next exercises deal with verifying that a given implementation is conforming to this specification. The main difficulty here consists in finding appropriate *loop invariants* for each loop of the program. An invariant is a property that holds when entering the loop the first time and is preserved by one step of the loop. In other words, it must hold after 0 step, and if we assume that it holds after n steps, it must hold after $n+1$ step. By induction, the invariant thus holds when we exit the loop[1]. Moreover, the invariants are *the only thing* that is known about the state of the program after the loop. They must thus be strong enough to allow us to prove post-conditions, but not too strong, or we won't be able to prove the invariants themselves. Finding the correct balance requires some training as explained below.

4.1 Safety

Question 8. Write loop invariants to prove all safety properties for the following implementation of `find_array`.

[1] Loop termination is handled in the next section.

```
int find_array(int* arr, int length, int query) {
  int low = 0;
  int high = length - 1;
  while (low <= high) {
    int mean = low + (high -low) / 2;
    if (arr[mean] == query) return mean;
    if (arr[mean] < query) low = mean + 1;
    else high = mean - 1;
  }
  return -1;
}
```

Answer. The following invariants show that `low` and `high` (thus `mean`) are within `arr`'s bounds:

```
/*@ loop invariant 0 <= low;
    loop invariant high < length; */
```

Discussion. Students usually don't have issue finding these invariants, as they arise quite naturally from the loop structure itself. An important point however is that `low <= high` is *not* an invariant, as it is not preserved by the last step of an unsuccessful search, where we end up with `low == high + 1`.

4.2 Loop invariants

Question 9. Prove that the invariants written in Question 8 hold (fix them if necessary).

Answer. The invariants above are correct.

Question 10. Write the loop invariants that allow to prove the post-conditions of behaviors `exists` and `not_exists` and prove that they are correct.

Answer. The following invariants are necessary:

```
/*@ loop invariant \forall integer i; 0 <= i < low ==> arr[i] < query;
    loop invariant \forall integer i; high < i < length ==> arr[i] > query; */
```

Discussion. These invariants are much more difficult to write than the ones that ensure safety. Indeed, they do not stem directly from the code itself. Rather, they provide the main correctness argument of the algorithm: because the array is sorted, the array elements to the left of `low` are smaller than `query`, while the array elements to the right of `high` are greater than `query`. Hence, we can restrict the search to the interval `low..high`.

4.3 Loop variant and program termination

Question 11. Provide a **`loop variant`** that ensures that the loop always terminates.

Answer. We need a positive integer expression decreasing at each step, so we take:

```
/*@ loop variant high - low + 1; */
```

Discussion. A helpful hint to find a loop variant without giving a formal definition is to look for an upper bound for the number of remaining loop iterations. At each step we decrease the diameter of the interval `low..high` where the element `query` can still be found, hence `high-low+1` gives such an upper bound. Program termination is usually left at the end of the lesson as it is mainly orthogonal to the other proof obligations.

5 Teaching feedback and discussion

Our experience shows that a deep theoretical course on Floyd-Hoare logic is not mandatory for the practical session on proof of programs. After comparing students having missed the lectures with others who have attended a complete theoretical course, we can say that, for program specification and proof exercises, good programming skills seem even more helpful than good knowledge of the underlying theory. Theoretical courses by themselves are definitely not sufficient to learn proof of programs.

The exercises of this lesson correspond to the difficult points that should be thoroughly exercised in practice. First, it is important to write correct specifications (Section 3). Proving a function `f` with a wrong specification is a very common error. For instance, a too strong precondition that prevents calling `f` in legitimate contexts or a too weak postcondition that forgets to state something about the state after the execution of `f` will not be detected by running JESSIE on `f` alone. When writing a postcondition, most students focus on the returned values and forget to specify that the function under verification does not modify variables when it is not supposed to do so.

Section 3.4 shows how students can "test" their specification before attempting to verify the implementation. Nevertheless, the instructor should check the specification of each student even if everything is proved at the end.

A major difficulty in program verification is to understand the role of an invariant. Many students need some time to understand that a loop invariant is a summary of the effects of the n first steps of the loop and that this is the only thing that is known on the state of the program after these steps. Thus, writing an invariant strong enough to enable proving the annotations below the loop (including post-conditions) and preserved by a loop step is often a delicate task.

Another important difficulty of proof of programs with automatic tools is analysis of proof failures. Basically, a proof failure can be due to an incorrect implementation (a bug), a wrong specification or the incapability of the automatic prover to prove the the required property. In the first case, it is sufficient to fix the bug. In many cases, test generation can help to find a counter-example. In the second case, the unproved property is not necessarily the erroneous one. Attentive analysis of the proof obligation may help to understand its proof failure. The problem can be due to an earlier incorrect or missing clause (such as a precondition, a loop invariant of the current loop, a loop invariant of a previous, or outer, or inner loop, the contract of a previously called function, etc.). Additional statements (assertions, lemmas, stronger loop invariants, etc.) may help the prover in some cases. They can also help the user to understand which part of a complex property is too difficult for the automated prover. Finally, when nothing else works, an interactive proof assistant (such as Isabelle, Coq and PVS) can be used to finish the proof.

6 Related work and conclusion

Usage of tools in formal verification courses is getting some traction [12]. In particular, the KeY tool for Java is used at a couple of universities [13]. Like KeY, FRAMA-C and JESSIE benefit from their open-source nature and the fact that they target existing languages, already known to students. However, they are used for the moment at a very small number of institutions [14], and very limited introductory material with exercices is available [15], [16, Chap. 9]. FRAMA-C is also part of an experiment in online programming training [17].

In this paper, we demonstrated by a small practical session how JESSIE can be used for teaching formal software verification. Our experience shows that JESSIE is perfectly adequate for this purpose since it benefits from an expressive specification language, adequate documentation, ease of use and installation. We hope that this work will be helpful in teaching proof of programs and will contribute to the introduction of formal methods based techniques and tools into industrial software engineering.

References

1. Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., Schoen, D.: Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach. In: the World Congress on Formal Methods in the Development of Computing Systems. (1999) 1798–1815
2. Delmas, D., Duprat, S., Baudin, P., Monate, B.: Proving temporal properties at code level for basic operators of control/command programs. In: 4th European Congress on Embedded Real Time Software. (2008)
3. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1969) 576–580 and 583
4. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8) (1975) 453–457
5. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C User Manual. (October 2011) `http://frama-c.com`.
6. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C, a program analysis perspective. In: the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012). Volume 7504 of LNCS., Springer (2012) 233–247
7. Moy, Y.: Automatic Modular Static Safety Checking for C Programs. PhD thesis, University Paris 11 (January 2009)
8. Moy, Y., Marché, C.: Jessie Plugin Tutorial
9. Correnson, L., Dargaye, Z.: WP Plug-in Manual, version 0.5. (January 2012)
10. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. (February 2011)
11. Correnson, L., Signoles, J.: Combining Analyses for C Program Verification. In: the 17th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2012). Volume 7437 of LNCS., Springer (2012) 108–130
12. Feinerer, I., Salzer, G.: A comparison of tools for teaching formal software verification. Formal Aspects of Computing **21**(3) (2009)
13. KeY Project: Uses of KeY for teaching `http://www.key-project.org/teaching/`.
14. Frama-C: Uses of Frama-C for teaching `http://bts.frama-c.com/dokuwiki/doku.php?id=mantis:frama-c:teaching`.
15. Burghardt, J., Gerlach, J., Hartig, K., Pohl, H., Soto, J.: ACSL by Example. A fairly complete tour of ACSL features through various functions inspired from C++ STL. Version 7.1.0 (for Frama-C Nitrogen).
16. Almeida, J.C.B., Frade, M.J., Pinto, J.S., de Sousa, S.M.: Rigorous Software Development, An Introduction to Program Verification. Undergraduate Topics in Computer Science. Springer (2011)

17. Quan, T., Nguyen, P., Bui, T., Le, T., Nguyen, A., Hoang, D., Nguyen, V., Nguyen, B.: iiOSProTrain: An Interactive Intelligent Online System for Programming Training. Journal of Advances in Information Technology **3**(1) (2012)