Experience Report: Teaching Code Analysis and Verification Using Frama-C

Salwa Souaf
CentraleSupélec
Rennes, France
salwa.souaf@centralesupelec.fr

Frédéric Loulergue
Univ Orléans, INSA CVL, LIFO EA 4022
Orléans, France
frederic.loulergue@univ-orleans.fr

Formal methods provide systematic and rigorous techniques for software development. We strongly believe that they must be taught in computer science curricula. In this paper we present the pedagogic rationale and the concrete implementation of two courses on the use of formal methods, sharing some material. These courses promote the usage of formal verification to ensure safety and security of software, exemplified in the domain of the Internet of Things.

1 Introduction

To make formal methods more widely applied, one way is to make them easier to use, in particular by making them more automatic and seamlessly integrated in software engineering tools and processes. However, full automation for deductive verification is not possible and we think it is desirable that more professionals know it is possible to formally specify and verify real-world programs, and have a hands-on experience to do it. Teaching program verification is a way to contribute to this goal.

This paper reports teaching experiences on C programs verification using Frama-C [1, 2], an open-source framework for the analysis and verification of ANSI C99 programs. The approach is to present the concepts mainly through examples and have the students specify and verify real-world code. Sharing some material and a project, we taught two courses in different institutions: Northern Arizona University, USA and INSA Centre Val de Loire, France. Over the years efforts of teaching and integrating formal methods in software engineering courses faced many challenges, some of which are mentioned in [14]. We encountered several of these challenges, but in this paper, we rather describe the more focused and detailed difficulties our students faced.

The paper is organized as follows: Section 2 describes the context including the Frama-C tool and the position of the courses in their respective programs, as well as our audiences. Section 3 is devoted to the approach we took for these courses: the organization, content and assignments, in particular the project. Students' challenges, results and evaluations are given in Section 4. We conclude and mention related work in Section 5.

2 Context and Background

Frama-C is architectured as a set of plugins built around a kernel that provides basic services (such as parsing and access to ASTs of the programs). Plugins use, generate, or verify annotations written in the ACSL (ANSI C Specification Language). Analyses and verification techniques include (but are not limited to) value analysis by abstract interpretation (plugin EVA), deductive verification (plugin WP) and dynamic verification (plugin E-ACSL). The plugin RTE is dedicated to generate ACSL assertions for each expression potentially leading to an undefined behavior. Frama-C is used both in industry

M. Gleirscher, J.v.d. Pol, J. Woodcock (Eds.):
First Workshop on Applicable Formal Methods (AppFM 2021).
EPTCS 349, 2021, pp. 69–75, doi:10.4204/EPTCS.349.5

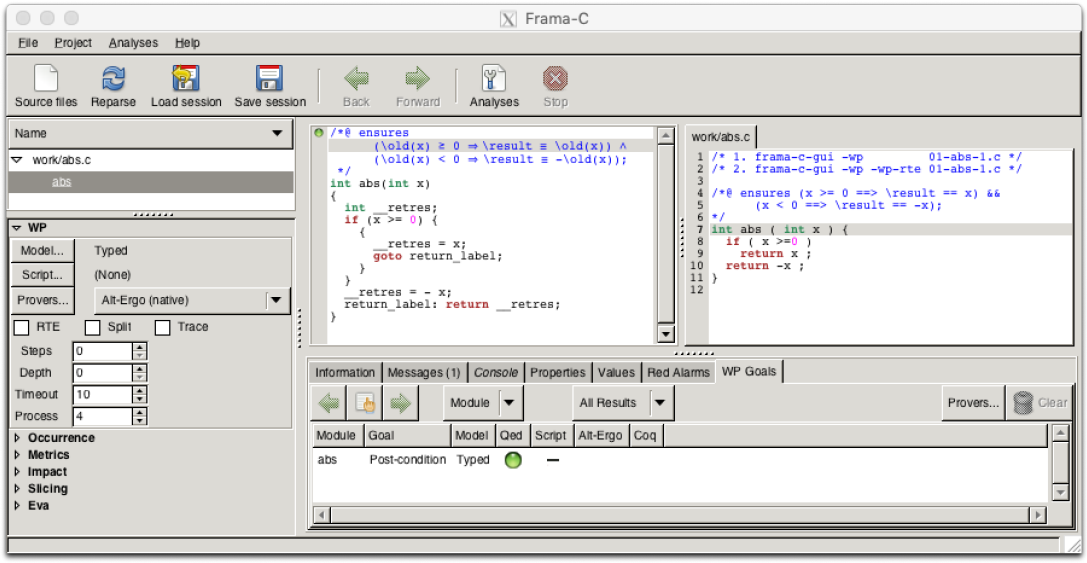

Figure 1: An example of use of Frama-C

and academia. The courses presented in this paper evolved from a series of tutorials given at various international conferences (in particular [5, 6]) and experience from research related to Frama-C (for example [4, 7, 8]). Figure 1 presents a screenshot of the tool while working on one of the first examples of the class with the WP plugin. On the right panel, the original code with an ASCL contract (only a post-condition) on lines 4–5 is displayed. The way the code is considered by the tool as well as a pretty-printed version of the contract is displayed on the left panel. The green bullet indicates that the function is correct with respect to this contract. Indeed the tool is used here without the RTE plugin, therefore there is not any assertion in the code that checks whether there is an overflow when evaluating the expression `-x`. If the RTE plugin is used, such an assertion is generated and the bullet is orange meaning the tool fails to verify that the code respects the contract.

At Northern Arizona University (NAU), the course described in this paper is course CS 451 Mechanized Reasoning about Programs. The only prerequisite were some standard architecture and data structure classes. The focus was on applied formal methods, and we chose to build it around Frama-C. The version of the class described here was taught in Fall 2019 to 37 bachelor students, most of them in their fourth year in the Computer Science program or the Applied Computer Science program. At INSA (Institut National des Sciences Appliquées) Centre Val de Loire, this course was taught to students in their fourth year. INSA is a network of French engineering schools. Each school offers five year programs from which students graduate with a Master's degree. The students who followed this course majored in Security and Computer Technologies. These students have a strong background in programming using different languages but were not exposed to theoretical courses covering formal methods. This course was taught to 23 students during Spring 2020.

3 Teaching Approach

At NAU, CS 451 was 2.5 hours a week, for 15 weeks. In-class exercises were given to students after each concept and related examples were presented, some of which were group exercises. In addition to

8 homework assignements, students had to complete a project in three phases. Finally, a midterm and a final exams were organized. For the homework assignments, in addition to the keys, extensive feedback was presented in class: the most common errors were discussed, as well as the various possible ways to correctly solve the exercise and their respective merit. All the assignments needed to write ACSL annotations and use Frama-C. All the material presented in class was available online, as well as keys for the homework assignments, midterm, and project. A mock midterm exam was organized a week before the exam. No textbook was mandatory for this course. However, the students were referred to Allan Blanchard's tutorial [3] and Jens Gerlach et al. ACSL by Example [10].

For the fourth year students at INSA Centre Val de Loire, the course was divided into 10h40 for the lectures and 10h40 for the in-class work. In order to fully utilize this short time and deliver the course content in a more digestible approach, the decision was made to give the lectures in a tutorial like style to introduce Frama-C and the in-class work as an implementation project with each session devote to a different phases of the project. The students were assessed based on their project work.

Northern Arizona University The course was initially divided into four parts: 6 weeks devoted to the course introduction, basic concepts of ACSL and deductive verification with Frama-C/WP (and RTE), 2 weeks for the foundational aspects: axiomatic semantics (and the reference for this part was [13]), 5 weeks for additional concepts of ACSL and deductive verification with application on case studies and finally 2 weeks for an introduction to static analysis with Frama-C/EVA. The three first parts took longer than planned, and the fourth part was removed. The second and third parts were swapped.

ACSL is basically first-order logic with typed C expressions, typed logical expressions and specific predicates. After presented Frama-C as a whole and basic usage of the tool, the course presented the concept of function contract with pre- and post-conditions. The first contracts contained only non-quantified annotations, and without specific predicates. In ACSL annotations mathematical types are used: a C variable containing an integer is thus considered as a mathematical integer. The plugin RTE adds assertions into the code to verify there is not any runtime error. This includes for example signed integer overflows. The use of RTE was introduced at this point in the course. Then the `\old` predicate, to be used in post-conditions, was introduced. Finally, the `assign` operator, which indicates which parts of the memory are potentially modified was explained. Note that up to this point, pointers were not used in the examples and exercises. How contracts are used in function calls was included in this introductory presentation. This was completed by the presentation of ACSL behaviors which allow to define contracts for sub-parts of the input space, the pointer-specific predicate `\valid`, and for expressing properties on arrays, quantifiers. This introductory part ended with set expressions in ACSL, to express memory validity more concisely. The final example was the specification, using behaviors, of a `find` function on an array of integers, that returns the index of a searched value, or `-1`. Contrary to previous examples and exercises, Frama-C/WP is unable to check the correctness of the implementation with respect to this contract: loop annotations are necessary. The first part of the class ended with all the concepts related to loop annotations: invariants, variants, loop assigns. Loop annotations were related to mathematical induction, and a significant part was devoted to methods to find invariants.

The second part introduced additional ACSL concepts in order to make the project doable: labels, user-defined predicates, the statement of lemmas, logic functions, and finally ghost code.

The goal of the third part was to present axiomatic semantics in a formal way and weakest precondition calculus in an informal way (as a method to prove axiomatic semantics judgments). It however started with the informal presentation of a small imperative language called nano-C (similar to the usual IMP or While languages but with a C syntax) and its structural operational semantics, with a small

modification with respect to IMP/While: sequence is not a command in nano-C, but there are lists of commands (the main program, branchs of conditionals, loop body). This makes the rules of the operational semantics all axioms.

INSA Centre Val de Loire Unlike the version of the course described above, the time dedicated in INSA did not allow an application of a classic detailed presentation of the different ACSL concepts, deductive verification and static analysis. For this specific reason and knowing the background of the students, we took a tutorial-style approach. The idea was to present the different concepts of ACSL and steps of code specification by relying on a concrete small example presented and tested live by the instructor, then a different example for the students to do in class and corrected during the same session. This was a way for the students to practice using the Frama-C tools from the very first sessions, get more familiar with the ACSL specification language as well as keep them invested and attentive throughout the session. We focused on utilizing and presenting, in this order, the RTE, WP and EVA plug-ins as well.

Project: Contiki's memb module As demonstration of the effectiveness and usability of code specification and verification, we proposed a project that aims to specify and verify the `memb` module of the Contiki OS. The subject of this project was inspired from the work presented in [12].

Contiki is an operating system for networked, memory-constrained systems with a focus on low-power wireless Internet of Things devices. Uses for Contiki include systems for street lighting, sound monitoring for smart cities, radiation monitoring, and alarms. It is open-source software. Contiki was developed in 2002, the main focus was on enabling communication in the most constrained devices, with no particular attention given to security. As it matured and as commercial applications arose, communication security was added at different layers, via standard protocols such as IPsec or DTLS. The security of the software itself, however, did not receive much attention.

The `memb` module is Contiki's main memory management module. To avoid fragmentation in long-lasting systems, Contiki does not use dynamic allocation. Memory is pre-allocated in blocks on a per-feature basis, and the `memb` module helps the management of such blocks. It offers a simple API, enabling to initialize a `memb` store, to allocate a block, to free a block, to check if a pointer refers to a block inside the store and to count the number of allocated blocks. The `memb.c` file consists in about 100 lines of C code but is one of the most critical elements of Contiki, as the kernel and many modules rely on it. The Contiki code base involves a total of 56 instances of `memb`. Not all are included in a given Contiki firmware, but a subset is included depending on the application and configuration. `memb` is used for instance for HTTP, CoAP, IPv6 routes, the MAC protocol TSCH, packet queues, etc.

The goal of the project was to formally specify – in ACSL – and verify – using Frama-C/WP – the `memb` module. The API studied in this project is extremely close to the actual Contiki-NG API, but the provided implementation differs for some functions, to make the verification easier for students. Elements to guide them through the different phases were detailed in the document they received at the beginning of each phase. The files that were provided to students were: *logic_defs.h* containing user-defined predicates and logic functions; *lemmas.h* to help the provers to verify the functions with respect to their contracts; *memb.h* headers of the `memb` API; and *memb.c* the full implementation.

The project was divided into three phases in order to better structure the work and have tangible intermediate deliverables. This also eased the amount of work for students and helped them get organized. When faced with a global project description without some guidance, they tend to either be overwhelmed, not knowing where to start, or to push it off to the last minute.

Phase 1 students had to write several user-defined predicates to be able to make function contracts

more concise and readable. These predicates were used to describe properties of `memb` stores. The two functions to specify and verify (including termination) were: `memb_init` and `memb_inmemb`.
Phase 2 focused on the specification and verification (including termination) of two functions of the API: `memb_numfree`, which returns the number of free blocks in the memory store, and `memb_free`, which frees a block, if possible, i.e. if the pointer to free is in the memory store, if it is aligned (c.f. predicate memb aligned), and if it is not already free.
Phase 3 focused on the last function of the API: `memb_alloc`, which returns the address of the first free block if the memory store is not full, and `NULL` otherwise. Unlike the functions of the previous phase, to verify `memb_alloc`, some assertions, in addition to the loop annotations, were needed in the body of the function for the automatic solvers to verify the contract. Some of them were provided, but not all.

4 Students Assessment and Students Evaluation

Students assessment At NAU, at the very beginning of the class, a reminder of propositional calculus truth tables was necessary for some students. On one hand, some students were not acquainted with mathematical notations for logical connectors: ASCL code is written using C syntax for common operations, for example `&&` for conjunction, or specific ACSL syntax, for example `==>` for implication, but by default Frama-C pretty prints these symbols as $\wedge$ and $\Rightarrow$ which was confusing to some students. In the other hand, when presenting the ACSL `\valid` predicate, some students were not comfortable with C-specific features such a pointers, because they essentially programmed using language with automated memory management. In the class CS 396 Principles of Programming Languages, parameter-passing modes are presented. But a significant subset of the students had not taken this class yet, so it was actually necessary also to take some time for C reminders about pointers but also parameter passing.

For an array of size `n`, two usual ACSL patterns are `\forall integer k; 0<= k < n ==> P[k]` and `\exists integer k; 0<= k < n && P[k]` to express that all cells of the array satisfy a property `P` and there exists a cell that satisfies a property `P`. Understanding that using `&&` instead of `==>` in the universal quantification case and vice-versa in the existential case does not work was a challenge for a significant number of students. Most of the students who failed the class were lost at this point.

Loop invariants are of course challenging. We encouraged the students to write loop annotations incrementally. First the loop assigns annotation and simple invariants about the loop counter when there is one. Most students were able to do so. Writing annotations needed to prove post-conditions was of course more difficult. Related to the challenged described in the previous paragraph, in the `find` example (and similar cases) writing a loop invariant that is the negation of an existential in the post-condition was a problem. Making the students explain informally why the loop runs rather than stops helped.

The final grades are given in the following table. 76% of the enrolled students passed the course.

Grade	A	B	C	D	F
Count	13	8	7	2	7

All the passing students were able to read and write a function contract including behaviors, and user-defined predicates. They were also able to write loop annotations, but students with a C grade had difficulties in obtaining a strong enough invariant to allow the proof of post-conditions and some form of logical functions may be challenging. Students with a B grade had problems with writing additional annotations to guide the SMT solvers, when students with a C grade were not able to write such annotations. While all passing students were able to write correct contracts, and were able to check whether pre-conditions contain a logical contradiction, they had sometimes difficulties to write strong enough contracts of called functions to allow the verification of calling functions.

At INSA, the students were assessed based on their final work in the project. The first phase was a test phase to make sure that the majority of the students understood the task at hand and have a good comprehension of the different aspects presented during the class. The second and third phases were the ones taken into consideration in their final grade. The following table summarizes the students grades.

Grade (over 20)	> 16	15–16	13–14	10–12	< 10
Count	4	3	6	4	6

While most of the students were able to write function contacts as well as loop annotations during class for smaller examples, when it came to the project they had difficulties executing. The students with grades less than 12 had difficulties in writing loop annotations specially with in finding an invariant. Students with grades between 13 and 15 had a good understanding of the level of specification needed but failed to find exactly the key annotations to guide the solvers. Finally, some students have shown quite the potential, they were able in a short amount of time to well execute the knowledge acquired in the class.

Students evaluation At NAU, there is a procedure for class evaluation by students: 7 questions with 4 possible responses ranging from strongly disagree (1) to strongly agree (4), and 3 open questions: What suggestions do you have to improve this course? The assignment that most contributed to my learning is. What did you like best about this course? The average score to the 7 questions was 3.5 which denotes a good level of satisfaction. There was not any heavy trend in the answers to open questions, but for the assignment that helped the most: the homework assignments. This is not surprising as these assignments are focused on a few concepts. In-class group exercises were also appreciated.

Suggestions for improvements were often contradictory, for example one student would have liked even more examples while another one would have prefered a focus on concepts. Some comments focused on the last assignment but made it a general comment on the course: in the last phase of the project, the students had to experiment with writing assertions in a function body to guide the SMT solvers. Thwas is not an aspect we exercised a lot in class. For all other concepts and ACSL features, several examples and exercises were presented. One comment was representative of the challenges faces by some students: “have an assignment that explicitly outlines/teaches mathematical logic such as disjunction, conjunction, proof by induction, and the like.”

There were many aspects the students liked best about this course. The only answer with several (similar) occurrences was: “thinking about the code in a different way”.

5 Conclusion and Related Work

In [11], Creuse et al. present a course on formal methods that focuses on deductive verification through Frama-C and SPARK. This course was aimed at students with no strong background in computer science. In their paper, Creuse et al. detail two experiments made at ISAESUPAERO in an engineering program focusing on aerospace industry. Blazy presents an initiation course to formal methods in [9]. She presents her approach and methodology in teaching deductive verification using the Why3 platform. This course was aimed at third-year students at the University of Rennes 1 in France.

Our, rather successful, approach is a hands-on approach using Frama-C, for fourth-year students. In addition to the important concepts of deductive verification, pre- and post-conditions, loop invariants and variants, the students experienced by themselves that it is possible in practice to formally specify and verify real-world C programs using a tool.

References

[1] *Frama-C*. Available at `https://frama-c.com/`.

[2] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles & Nicky Williams (2021): *The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform*. *Commun. ACM* 64(8), p. 56–68, doi:10.1145/3470569.

[3] Allan Blanchard (2019): *Introduction to C Program Proof using Frama-C and its WP plugin*. Available at `https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf`.

[4] Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre & Frédéric Loulergue (2016): *Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs*. In: *Int. Working Conf. on Source Code Analysis and Manipulation (SCAM'16)*, IEEE, pp. 67–72, doi:10.1109/SCAM.2016.18.

[5] Allan Blanchard, Nikolai Kosmatov & Frédéric Loulergue (2018): *A Lesson on Verification of IoT Software with Frama-C*. In: *International Conference on High Performance Computing and Simulation (HPCS)*, IEEE, Orléans, France, pp. 21–30, doi:10.1109/HPCS.2018.00018.

[6] Allan Blanchard, Nikolai Kosmatov & Frédéric Loulergue (2018): *Secure Your Things: Secure Development of IoT Software with Frama-C*. In: *IEEE Cybersecurity Development Conference (SecDev)*, IEEE, pp. 126–127, doi:10.1109/SecDev.2018.00026.

[7] Allan Blanchard, Nikolai Kosmatov & Frédéric Loulergue (2019): *Logic against Ghosts: Comparison of two Proof Approaches for a List Module*. In: *ACM Symposium on Applied Computing (SAC)*, ACM, pp. 2186–2195, doi:10.1145/3297280.3297495. Best Paper Award.

[8] Allan Blanchard, Frédéric Loulergue & Nikolai Kosmatov (2019): *Towards Full Proof Automation in Frama-C using Auto-Active Verification*. In: *Nasa Formal Methods*, LNCS, Springer, pp. 88–105, doi:10.1007/978-3-030-20652-9_6.

[9] Sandrine Blazy (2019): *Teaching Deductive Verification in Why3 to Undergraduate Students*. In Brijesh Dongol, Luigia Petre & Graeme Smith, editors: *Formal Methods Teaching*, Springer International Publishing, Cham, pp. 52–66, doi:10.1007/978-3-030-32441-4_4.

[10] Jochen Burghardt & Jens Gerlach (2019): *ACSL by Example*. Available at `https://github.com/fraunhoferfokus/acsl-by-example`.

[11] Léo Creuse, Claire Dross, Christophe Garion, Jérôme Hugues & Joffrey Huguet (2019): *Teaching Deductive Verification Through Frama-C and SPARK for Non Computer Scientists*. In Brijesh Dongol, Luigia Petre & Graeme Smith, editors: *Formal Methods Teaching*, Springer International Publishing, Cham, pp. 23–36, doi:10.1007/s00165-014-0326-7.

[12] Frédéric Mangano, Simon Duquennoy & Nikolai Kosmatov (2016): *Formal Verification of a Memory Allocation Module of Contiki with Frama-C: A Case Study*. In: *Risks and Security of Internet and Systems (CRiSIS)*, *LNCS* 10158, Springer, pp. 114–120, doi:10.1007/978-3-319-54876-0_9.

[13] Hanne Riis Nielson & Flemming Nielson (1992): *Semantics with applications – a formal introduction*. Wiley.

[14] Maria Spichkova. & Anna Zamansky. (2016): *Teaching of Formal Methods for Software Engineering*. In: *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering - COLAFORM, (ENASE 2016)*, INSTICC, SciTePress, pp. 370–376, doi:10.5220/0005928503700376.