Teaching formal methods to future engineers

Catherine Dubois, Virgile Prévosto, Guillaume Burel

Teaching Formal Methods to Future Engineers

Catherine Dubois[1], Virgile Prevosto[2], and Guillaume Burel[1]

[1] ENSIIE, Samovar, CNRS, Évry, France
{catherine.dubois, guillaume.burel}@ensiie.fr
[2] Institut LIST, CEA, Université Paris-Saclay, Palaiseau, France
virgile.prevosto@cea.fr

Abstract. Formal methods provide systematic and rigorous techniques for software development. We are convinced that they must be taught in Software Engineering curricula. In this paper, we present a set of formal methods courses included in a Software Engineering & Security track of ENSIIE, École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise, a French engineering school delivering the « Ingénieur de l'ENSIIE » degree (master level). These techniques have been taught over the last fifteen years in our education programs in different formats. One of the difficulty we encounter is that students consider these kinds of techniques difficult and requiring much work and thus are inclined to choose other courses when they can. Furthermore, students are strongly focused on the direct applicability of the knowledge they are taught, and they are not all going to pursue a professional career in the development of critical systems. Our experience shows that students can gain confidence in formal methods when they understand that, through a rigorous mathematical approach to system specification, they acquire knowledge, skills and abilities that will be useful in their professional future as Computer Scientists/Engineers.

1 Introduction

Formal methods provide systematic and rigorous techniques for reliable software development. Many industries developing critical systems have already adopted formal methods with significant successes (see e.g. [14] for railway experience). Knowing these techniques and methods helps enhancing the quality of software, even in contexts were full-fledged formal verification is not employed. We are thus convinced that formal methods must to be taught in Software Engineering curricula whatever the professional orientation of the future engineers. In this paper, we present a set of formal methods courses included in a Software Engineering & Security track at ENSIIE, École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise, a French engineering school delivering the « Ingénieur de l'ENSIIE » degree (master level). These techniques have been taught over the last fifteen years in our education programs in different formats, especially regarding hourly volumes and elective/compulsory nature. These formal methods courses also reflect a long tradition of research in formal methods at ENSIIE.

The paper is organised as follows. Section 2 presents ENSIIE, its curriculum and specialised tracks. Section 3 quickly introduces the Software Engineering & Security track, emphasizing the courses where formal methods play an important role. Each of these courses is then detailed in a dedicated section (Sections 4, 5 and 6). We then conclude in Section 7.

2 ENSIIE

École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (ENSIIE, `https://www.ensiie.fr`) is one of the oldest French institutions offering a degree of Engineer (master level) in computer science. Since its creation in 1968, almost 3,000 engineers have graduated from this institution.

Like for the majority of engineering schools in France, most students are admitted at ENSIIE through a selective entrance examination that requires at least two years of preparation with an intensive program in Mathematics and Physics (*Classes Préparatoires aux Grandes Écoles* in French, a very selective curriculum for the first two years in college). ENSIIE hosts about 500 students (around 150 new students per year for a 3-years curriculum).

Students follow a threefold curriculum[3]:

- Information Technology (40%): software engineering, systems and networks, security, artificial intelligence, virtual reality, games and video gaming, robotics, high performance computation;
- Applied Mathematics (30%): operational research, optimisation, data science, machine learning, financial mathematics;
- Business organisation (30%): economy, finance, management, business organisation, entrepreneurship.

A considerable amount of time (11 months during the whole studies) is spent working in companies or research laboratories, corresponding to 3 internships distributed during the study period.

ENSIIE curriculum is organised in 6 semesters or 3 years. Semesters 1 and 2 (first year) form the common core of training with courses in the three main areas previously cited - computer science and engineering, applied mathematics and management - and humanities. This first year has a bachelor level or L3 level according to French educational system. Semesters 3, 4 and 5 mainly correspond to elective technical courses (they are completed by management and humanities). Students can freely choose their courses but specialised tracks are proposed. Because of quotas imposed in some courses, the choices of a student are accepted according to their academic results and personal professional motivations. Semester 6 is dedicated to a 6 months internship leading to a dissertation and a defence evaluated by a jury. These four last semesters end up with a master level. During ENSIIE third year (Semesters 5 and 6), students

[3] Course catalogue can be found at `https://www.ensiie.fr/wp-content/uploads/2018/05/ensiie_course_catalogue.pdf`

can be enrolled in a research oriented Master (2nd year) in applied mathematics or computer science by attending selected courses from the engineer and master programs. In that case they may have a double degree.

Four specialised tracks are offered: Applied Mathematics (financial analysis, statistics, data science, operational research), Software Engineering & Security (SE & S) (software architecture, systems, formal methods, security), Numerical Interactions (virtual & augmented reality, artificial intelligence), High Performance Computing/Big Data (HPC architecture and operating system, clusters, compilation, numeric simulation). In each track, there are also compulsory and elective courses. Besides, there is also a *free* track in which students are allowed to choose courses from the four previous tracks, composing a menu *à la carte*.

In the rest of the paper, we focus on the SE & S track. Most of the acronyms for courses titles used in this paper stand for names in French. We decided to keep them for a better matching with the official course catalogue.

3 Software Engineering & Security Track

Common core contains some courses related to basics in computer science[4]: imperative programming (C), database design, operating systems, functional programming (OCaml), logic, Web programming (PHP, Javascript) and networks, object oriented programming (Java, C++). Programming projects developed by a team of several students accompany the previous courses.

At this stage, a first formal highlight is given with logic and functional programming. The former course forms the basis for teaching formal methods while the latter introduces students to types, induction, termination and correctness.

Let us focus now on Semesters 3 and 4 (Master 1 level) in SE & S track. S3 courses are mainly compulsory: Agile Project Management, Advanced Functional Programming (IPF), Formal languages, Software Validation and Verification (VVL), Assembly Language and Compilation, Software Engineering (IGL). Students can choose between Operational Research and a course about Security and Protocols. The course entitled IGL introduces students to the principles of Software Engineering and trains them in modelling with UML. It also provides some knowledge about model-driven engineering and quality collaborative project management. Semester S4 is more flexible in the sense that students have some choice, e.g. they can choose between a course about formal methods (MFDLS) and a course about semi numerical algorithms. Then, they have the choice between a course about models of computations (CAL) and a course about the design of privacy-by-design applications. Until 2017, they could also take a class about concurrency and verification by model checking (PVC). However, for structural reasons, this course has not been taught during the last years, but it will be proposed again in 2019-2020, in Semester 4, with a similar content.

Semester 5 (Master 2 level) proposes a large choice of courses. We focus on PROG1 and PROG2 that, among others, belong to the SE & S track. The

[4] In parenthesis appear the languages used to illustrate the different concepts.

former focuses on formal proof and formal semantics and the latter on abstract interpretation and deductive verification.

We consider the following set of courses, VVL, MFDLS, CAL, PVC, PROG1, PROG2 as *the formal methods track* or, shorter, *the formal track*. All of them are 42 hours long (including lectures, tutorials, lab sessions and exams), except VVL which is only 21 hours long. As the number of students is quite low, lectures and tutorials are usually mixed.
VVL introduces students to testing (both black- and white-box testing), and proof of programs (Hoare Logic). Besides lectures and tutorial classes, lab sessions are organized where students use Junit [5], PathCrawler [13] and the Frama-C [12] platform (in particular its deductive verification WP plugin).
CAL, as its name suggests, focuses on calculability and presents several equivalent philosophies and models for computation: Turing machines, partial recursive functions, lambda-calculi. At this point, notions of complexity can be introduced. Eventually, Gödel's first incompleteness theorem is discussed. In order to make these notions more concrete, lab sessions are organized, for example to implement Turing machines.
PVC is concerned with basic concepts of concurrent programming and verification. With these lectures, students acquire in particular the main techniques to verify dynamic properties of concurrent programs (deadlock freeness and more advanced properties) using a model-checker, here SPIN.
We focus below on the three remaining courses, MFDLS, PROG1 and PROG2[5].

In Table 1, we can find the numbers of students that registered in the different courses we focus on in this paper. As mentioned above, the acronyms are related to the French titles. Thus VVL stands for *Software Validation and Verification*, MFDLS for *Formal Methods for Reliable Systems*, CAL for *Models of Computation*, PVC for *concurrency and model-checking*, PROG1 for *Formal Proof and Semantics* and PROG2 for *Static Analysis*. Finally, IGL stands for *Software Engineering*, and IPF for *Advanced Functional Programming*. We can see that these numbers are quite stable over years.

Semester	*Course Title*	$2018-2019$	$2017-2018$	$2016-2017$
$S3$	IGL	57	57	63
$S3$	IPF	64	69	67
$S3$	**VVL**	59	59	67
$S4$	**MFDLS**	25	29	32
$S4$	**CAL**	19	34	28
$S4$	**PCV**			29
$S5$	**PROG1**	17	20	15
$S5$	**PROG2**	19	18	15

Table 1. Numbers of students

[5] The authors of this paper are teaching these courses.

As said before, IGL, IPF and VVL are compulsory courses for the SE & S track. With a very few exceptions (because of the free track), students registered in MFDLS and CAL have been enrolled in VVL in the previous semester. A large number of students take both MFDLS and CAL (50% in 2018-2019). In Semester 5, most of students taking PROG1 have taken MFDLS or at least VVL. PROG1 and PROG2 are taught to the same students, except a very small number of students taking only PROG1.

4 Formal Methods for Reliable Systems (MFDLS)

The course about software validation and verification (VVL) introduces students to formal proofs of programs when programs are annotated with assertions (pre and post-conditions, loop invariants and variants). It is their first encounter with formal specifications. MFDLS makes them go further on that direction with state-based formal methods like B [1] and Event-B [2] and the correct-by-construction development process. The B method was used until spring 2019 when we decided on switch to Event-B. Focus is put on modelling and refinement. We also introduce some security notions, more precisely the main control access policies and show that security issues may also be formalised and integrated to a functional model.

Why moving from B to Event-B while the B version was a well-oiled machine? A first answer would be that Event-B being the *recent* variant of the B method, it should be preferred for teaching newer generations of engineers. However, it is not an easy task. Indeed, B is devoted to developing software with a very long life cycle and it has demonstrated its capacities on large industrial projects (e.g. Paris Meteor line 14)), while Event-B is rather a language for modelling systems [14]. However, they share the same foundations: set theory, predicate language, state-based method and refinement. We believe that Event-B refinement is easier and more natural for beginners in formal methods than B refinement. They can understand quite easily the so-called parachute paradigm [2] even if they have difficulties when it comes to implementing it on examples. Finding good gluing invariants remains a hard task, both in B and Event-B. Furthermore, Event-B, with its rather weak language of actions (no if/while substitutions) helps sending the message that modelling and programming are two very distinct activities.

The 2019 schedule is as follows (just replacing Event-B with B will give the previous schedules). Usually, 2 sequences of 3h30 each are scheduled per week on a period of 7 weeks. The first sequence contains an introduction to formal methods and Event-B as well as a presentation of set theory (sets, relations, functions). The second sequence is devoted to - pen and pencil - exercises from simple models requiring only sets to models with sets and relations as variables. For example, one exercice concerns a small system with users that register, log in and log out, revisited with passwords and then with black-listed users. Then, students have a hands-on sequence with Rodin (`http://wiki.event-b.org`) and ProB (`https://www3.hhu.de/stups/prob/`) where they play with or implement some of the models written previously. Faults may have been introduced

by the instructor. Sequence 4 is devoted to a formal approach of the semantics of actions and proof obligations. It is also the occasion to review some concepts from logic such as term, formula, free/bound occurrence of a variable and proof rule. Sequence 5 is a lab session, where students learn how to do simple interactive proofs with Rodin. At that time, around the middle of the course, students are evaluated on their ability to manipulate set theory and write some models. This pen-and-pencil evaluation takes place in Sequence 6 and is an hour long. Refinement is then taught and practised during 2,5 sequences with again some practice with Rodin and ProB. A peer-correction of the previous evaluation (described in more details below) takes place meanwhile. For the rest of the course, focus is put on security and control access policies (DAC, MAC and RBAC) with lectures and tutorials. In particular, we study the RBAC encoding (invariants mainly) done within B by Huynh et al. [10] and the combination of a functional model and a security policy. In one of the last sequences, an industrial partner visits us and gives a talk illustrating some real case studies (usually about transportation systems), that motivate students a lot. This talk often opens not only summer internships, but also (and more often) long internships in Semester 6. During the last sequence (Seq. 12), students have to defend their project whose subject has been given in the middle of the course.

The course is illustrated with many examples, from simple to more complex (e.g. Bridge example is studied with the help of Abrial's slides and some youtube videos) giving them good *patterns* to reuse. We encourage both proof and animation though Rodin and ProB but we insist a lot on the differences with respect to verification and validation.

The project has to be realized by 2-persons teams and usually a list of 3 subjects is proposed to the classroom. Most of the projects have security aspects: secure management of medical records, voting system, DAC, ... They are usually case studies inspired by research papers, e.g. in 2019 a reporting management system inspired from [19] that integrates a control access policy close to RBAC but with state-dependent access rights, or a simplified control air traffic control system inspired from [11]. With the description of the system, a refinement plan is proposed. The project is part of the evaluation for 50% of the final mark. Students pass this course with very few exceptions.

Let us come back to the peer correction of the first evaluation that we have been doing for 2 years now. For the moment we do not use any tool for that purpose, so some manual manipulation of assignment papers are required to ensure anonymity of both the corrector and the author. The main benefit for the students is to understand that there are different acceptable solutions. For the teacher it is more work because a solution sheet must be carefully prepared and a double check is necessary. Furthermore, as we do the peer evaluation during a class, the teacher is very much solicited and has to individually help some correctors.

A recurrent difficulty for some students, both in B and in Event-B, is the real nature of invariants and the link with proofs. They do understand proof obligations corresponding to preservation of the invariants by the events. As

we noticed while reviewing projects, an informal requirement for a bike sharing system like « A damaged bicycle can not be borrowed by any user » is usually reflected in the pre-condition of an operation modeling the action of borrowing a bike but it is more rarely part of the invariant. As said before in Section 1, most of our students have a good background in mathematics. However we can notice that we spend more time to practice set theory and more precisely relational operators because students have less knowledge about that field for some years. We plan to use a set interpreter and an intensive individual training to make it through. For this course, we do not see too much disparity in students' mathematics background. The difference lies in their ability to abstraction.

5 Mechanized Formal Proof and Semantics (PROG1)

This course is equally divided into 2 modules, Mechanized Formal Proof (MFP) and Semantics of Programming Languages (SPL) running in parallel, with one sequence (3h30) for each one in a week. The 2 modules are independent, however the common mathematical tool is the notion of inference rule for proving but also for specifying semantics. Students pass this course with very few exceptions.

MFP is devoted to interactive proving and also automatic proving at an introductory level. Thus, in this module we first step into the Coq interactive theorem prover (`https://coq.inria.fr/`), used here as an environment to write functional programs, specifications and proofs. We benefit from the fact that our students have studied functional programming and practised OCaml (at least in their common core for most of them), they are used to functions, recursive functions, inductive data types, pattern matching, types and functions as first class values. Hence, they can move from OCaml to Coq quite easily regarding writing code. The first two sequences are hands on, students are introduced to inductively defined predicates and proofs using tactics, up to proofs by induction. At the end of these two sequences, a Coq project is assigned to the students: usually functions on lists (from simple to more elaborate ones, e.g. a simplified version of count-down, sorting function, queue implementation, set as interval list). Projects are done by pairs and must be submitted at the end of the course with a small report using coqdoc.

Then we come back to logic with a reminder of natural deduction for first order logic and a highlight on intuition/classical settings. A quick presentation (which is just a reminder for most students) of pure lambda-calculus and simply typed lambda-calculus (STLC) is done. We then link both worlds by presenting the Curry-Howard (CH) isomorphism. This isomorphism is illustrated on STLC and minimal natural deduction. A blackboard proof is done, describing a process/algorithm to go from a natural deduction proof to a STLC term and back. It is checked on simple examples inside Coq. Then extensions are studied (pairs/conjunction and sum types/disjunction). We do not go further in the Barendregt cube [3], but we insist on the idea that when logical features are added, the language is extended too. Presenting all this lasts 3 sequences with lectures and tutorials. The part about CH isomorphism is considered as diffi-

cult by students. To make it more concrete, we plan to make them implement the production of the lambda term by enriching the tactical prover provided in Chapter 16 of [8].

In the last sequence, students are introduced to automated theorem provers (that they have already encountered in the proof part of VVL when they used the WP Frama-C plugin). We quickly have a look at the DPLL algorithm and implement, during a short lab session, an SMT solver by combining glucose[6] as a SAT solver and glpsol (a tool from the library GLPK [7]) as a solver for linear arithmetic[8].

An exam is organized at the end of the module and the project is part of the evaluation for 50% of the final mark.

The previous module is complemented by a module (SPL) about semantics of programming languages. Students are taught dynamic operational semantics with small step and big step format. Different programming paradigms are revisited (because they have all been practised in other courses in previous semesters). Sequence 1 starts with a language of arithmetic expressions with variables, illustrating the notions of evaluation and environment. Then, we build on this language to formalize the semantics of a small imperative language leading to the notion of execution. Besides tutorials, practical sessions allow students to implement interpreters for the previous languages in OCaml. Then, we move on to a small functional language (Mini-ML) allowing for the introduction of lambda abstractions, closures, and call-by-name vs. call-by-value. Here again, a lab session is organized to develop an OCaml interpreter for Mini-ML, and we also investigate the notion of higher order abstract syntax. A tutorial is usually organized to study other features such as inheritance (using Featherweight Java following the presentation in [17]) or blocks (where locations are introduced). The module ends with a presentation of the K system (`http://k-framework.org/`) [18], which is an environment for specifying and animating formal semantics, followed by a practical session about this system, going back to the previous simple imperative language.

An exam is organized at the end of the module. Students have to submit the results of some practical sessions, which will account for 30% of the final mark. The main difficulty that the students encounter is the handling of inductive rules that describe the semantics. Although inductive systems are taught already since the logic course of the common core, the students struggle in linking their intuition of the behaviour of programming languages with the design of inductive rules.

---

[6] `https://www.labri.fr/perso/lsimon/glucose/`

[7] `https://www.gnu.org/software/glpk/`

[8] The lab session text is at the following url `http://web4.ensiie.fr/~guillaume.burel/download/PR_TP.pdf`

6 Static analysis and Deductive Verification (PROG2)

The course contains 6 sequences (3h30 each), giving a brief overview of static analysis and abstract interpretation. It uses a fairly classical minimal imperative language (assignment, test and while loop) as illustration. In parallel students have to work on a project detailed below. Students pass this course with very few exceptions.

The first sequence recalls notions about operational semantics (which in theory have been seen by the students in their previous courses) and introduces the notion of control-flow graphs, concrete execution traces and collecting semantics. The second sequence defines the main grounding blocks of static analysis: lattices and fixpoints, with examples of forward and backward analyses as well as over- and under- approximations. We then move on to Galois connections and insertions and define abstract execution over the sign domain. Widening is seen in the fourth sequence (together with narrowing) and illustrated over intervals. Finally, we present reduced product by showing how the combination of sign and parity information can give more precise results than each piece seen in isolation. The last sequence is dedicated to the presentation of the Eva plugin [7] of Frama-C [12] and a lab session were students use Eva to prove the absence of runtime errors in small C functions, usually extracted from open-source libraries (see for instance `https://gitlab.com/vprevosto/stan/wikis/2018-2019/tp` for the exercises given this year).

The most important message we try to convey is that it is possible to obtain correct, mathematically backed results about programs, including ones written in real-world languages (hence the last course). As an aside, we also put forward the importance of having precise definitions of the semantics of the various programming languages elements one is working with.

Generally speaking, students do not have a very strong background in logic, which is particularly seen during the lecture on Galois connections and insertions that is usually felt as particularly difficult to grasp.

The main frame of the course is quite stable since the last few years. A small change in the lectures organization has been made possible by the relatively small numbers of students taking the course. While each sequence is formally divided into a lecture followed by a tutorial session, in practice, giving an exercise as soon as the corresponding notion has been introduced proved very beneficial. A more radical change would be to move from pen and paper exercises to lab sessions where they would have to implement these notions, e.g. in OCaml, Why3 or Coq. Such a change would however imply a huge preparation beforehand, and even if students tend to prefer programming rather than doing more theoretical exercises it is not completely clear whether these activities will help them understanding better the theoretical notions that are presented. Indeed a two hours session is very short for proposing something in Coq, or even Why3. On the other hand, an exercise in OCaml would make them focus on an implementation, leaving out the proofs that it is correct. Furthermore, such exercises might interfere with the projects that are described in the next paragraph.

In parallel to the main course, students are asked to work in pairs on a project, consisting in first reading a research article and summarizing it during a short presentation to the whole class, and second doing some software development related to the article. There are usually two categories of subjects for the projects. For each of them, one or two individual subjects are selected, depending on the number of students enrolled in the course. All in all, at most 2 or 3 groups are working on the same subject. The first category is based on an article about static analysis or abstract interpretation and the associated assignment typically consists in implementing the algorithm described in the paper. After many years, where we asked the implementation to take the form of a Frama-C plug-in (or in one occasion of a new domain for Eva), we chose this year to restrict the task to a simple academic language similar to the one presented in the lectures (https://gitlab.com/vprevosto/stan). While letting the students interact with a real framework can be more formative, the complexity of Frama-C's API was a big hurdle to pass before being confronted to the static analysis itself. The two articles this year were Antoine Miné's *A New Numerical Abstract Domain Based on Difference-Bound Matrices* [15] and David Monniaux' and Laure Gonnord's *Cell Morphing: from Array Programs to Array-free Horn Clauses* [16], the latter being probably a bit too ambitious.

The second category is dedicated to deductive verification, with an article on program proofs and a subject consisting in implementing, specifying and proving a small algorithm. Again, this year we shifted from imposing the use of the WP plug-in of Frama-C (and thus a C implementation) to propose Why3 [9], so that students do not have to fight C's idiosyncrasies in addition to think about the best way to write their function contracts and loop invariants. The two articles were *Ghost for Lists: A Critical Module of Contiki Verified in Frama-C* by Allan Blanchard, Nikolai Kosmatov and Frédéric Loulergue [6], and *Secure Information Flow by Self-Composition* by Gilles Barthe, Pedro D'Argenio and Tamara Resk [4]. For the former, the associated subject was the basic operations of the skip list data structure, while for the latter it consisted in the Kruskal algorithm for computing maximal spanning trees over graphs.

7 Conclusion

We presented in this paper a *formal* track offered to students engaged in a Software Engineering & Security curriculum in an engineering school. This has been happening for more than 15 years with variants. Some of our students who have followed this set of courses have a job where they use formal methods every day but a lot of them do not. We interviewed a few of the latter about benefits they got from this formal track in their professional life while they do not apply formal methods directly[9]. To quote one of them, « I think that all the notions we learn about analysis of a program, its source code, and its behaviour, allow us to better understand what we are developing, to better understand what is

[9] Answers can be found at `http://web4.ensiie.fr/~dubois/interviews_FMTEA19.pdf`

happening when we write this or that instruction in our code. ». And to quote another one « Formal methods gave me rigor in software design ». We believe that this formal track gives a solid basis to students who want to continue down the *formal* direction (Phd or job relying on formal methods) because it covers a large panel of techniques for specification and verification. For those who go to more traditional development, this formal track gives them rigor, rigor and rigor. This also gives them, when the time comes, the memory that formal tools exist and can help them in a more reliable development.

We would like to thank all the colleagues who participated or participate to that set of formal courses. We cite some of them (in any order): S. Blazy, R. Laleau, J. Signoles, X. Urbain, P. Courtieu, F. Gervais, G. Berthelot, A. Mammar, T. Le Gall, R. Rioboo, C. Mouilleron, D. Watel, J. Falampin, C. Métayer, N. Kushik, A. Djoudi. Finally, we mention and thank late P. Facon who introduced a course at ENSIIE about formal specification with VDM in the late 90s and thus opened a specific route.

References

1. J. Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.
2. J. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
3. H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.
4. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.
5. S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp, and C. Stein. *JUnit 5 User Guide*. `https://junit.org/junit5/docs/current/user-guide/`.
6. A. Blanchard, N. Kosmatov, and F. Loulergue. Ghosts for lists: A critical module of contiki verified in frama-c. In *NFM*, volume 10811 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2018.
7. S. Blazy, D. Bühler, and B. Yakobowski. Structuring abstract interpreters through state and value abstractions. In A. Bouajjani and D. Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 112–130. Springer, 2017.
8. C. Dubois and V. Ménissier-Morain. *Apprentissage de la programmation avec OCaml*. Hermès Sciences, 2004.
9. J. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.
10. N. Huynh, M. Frappier, A. Mammar, R. Laleau, and J. Desharnais. A formal validation of the RBAC ANSI 2012 standard using B. *Sci. Comput. Program.*, 131:76–93, 2016.

11. A. Jarrar and Y. Balouki. Formal modeling of a complex adaptive air traffic control system. *CASM*, 6:6, 2018.
12. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.
13. N. Kosmatov, N. Williams, B. Botella, and M. Roger. Structural unit testing as a service with pathcrawler-online.com. In *SOSE*, pages 435–440. IEEE Computer Society, 2013.
14. T. Lecomte, D. Déharbe, É. Prun, and E. Mottin. Applying a formal method in industry: A 25-year trajectory. In S. A. da Costa Cavalheiro and J. L. Fiadeiro, editors, *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*, volume 10623 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2017.
15. A. Miné. A new numerical abstract domain based on difference-bound matrices. *CoRR*, abs/cs/0703073, 2007.
16. D. Monniaux and L. Gonnord. Cell morphing: From array programs to array-free horn clauses. In *SAS*, volume 9837 of *Lecture Notes in Computer Science*, pages 361–382. Springer, 2016.
17. B. C. Pierce. *Types and programming languages*. MIT Press, 2002.
18. G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
19. I. Vistbakka and E. Troubitsyna. Towards Integrated Modelling of Dynamic Access Control with UML and Event-B. *arXiv e-prints*, May 2018.