# The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform

Patrick Baudin[1], François Bobot[1], David Bühler[1], Loïc Correnson[1], Florent Kirchner[1], Nikolai Kosmatov[2], André Maroneze[1], Valentin Perrelle[1], Virgile Prevosto[1], Julien Signoles[1], Nicky Williams[1]

[1]Université Paris-Saclay, CEA, List, Software Safety and Security Lab, Palaiseau, France
[2]Thales Research and Technology, Palaiseau, France

## ABSTRACT

The C programming language remains popular for system-level programming and embedded code in many critical domains, where the consequences of errors can be extremely costly or even dramatic. Verification and validation of such programs is crucial to make the software-dependent services reliable and secure. This paper presents a panorama of Frama-C, a popular platform for C program analysis and verification. It relies on a careful architectural design, in which different analyzers rely on a common kernel and share a common specification language. The key success factors of the platform are the soundness of its tools, a wide range of available analyzers and a rich ecosystem. This overview presents the main design choices of the platform, its basic analyzers, and their advanced uses for a large set of software verification tasks.

**ACM Reference Format:**
Patrick Baudin[1], François Bobot[1], David Bühler[1], Loïc Correnson[1], Florent Kirchner[1], Nikolai Kosmatov[2], André Maroneze[1], Valentin Perrelle[1], Virgile Prevosto[1], Julien Signoles[1], Nicky Williams[1]. 2019. The Dogged Pursuit of Bug-Free C Programs:, The Frama-C Software Analysis Platform. In *Proceedings of Commun. ACM (CACM).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

The C programming language is a cornerstone of computer science. Designed by Ritchie and Thompson at Bell Labs as a key element of Unix engineering, it was rapidly adopted by system-level programmers for its portability, efficiency, and relative ease of use compared to assembly languages. It remains today, nearly 50 years after its creation, still widely used in software engineering.

But C is a hard language to wield. Its native design choices giving a large freedom to the developer—the same that gave its popularity—clash with the requirements of modern development practices such as strong typing, encapsulation, or genericity. Given its ubiquity in software engineering, this has had noticeable safety and, more recently, cybersecurity impacts. The use of verification techniques, and in the case of systems with high-confidence requirements, formal methods, can address these shortcomings.

Indeed, *formal methods* is a set of techniques based on logic, mathematics, and theoretical computer science which are used for specifying, developing and verifying software and hardware systems. By relying on solid theoretical foundations, formal methods is able to provide strong guarantees about those systems. In particular, program analysis techniques focus on the program code *after* it has been written or even compiled. Such techniques are called *sound* if their results are correct with respect to the behavior of the program under analysis.

Unfortunately, implementing such techniques for C programs is hard. Indeed, the same issues that make C programming very error-prone also tend to complicate the task for formal-methods based analyzers. It is very easy to write an illegal program whose behavior is *undefined* by the C standard: it can for instance provoke a crash, or sometimes silently corrupt memory and lead to arbitrary results. Examples of such behaviors include division by 0, illegal memory access and reading uninitialized variables. In particular, the fact that C allows a direct access, through casts and pointer arithmetic, to the sequence of bytes that contain the concrete representation of an object in memory is a major impediment to any attempt to reason on these objects at a more abstract level. Yet, many functions from the C standard library, starting with `memcpy` for copying an object to another location in memory, will trigger such low-level accesses, not to mention their presence in many parts of user-defined code.

Frama-C [26] is a C code analysis platform that attempts to tackle this complex issue. It is developed at CEA List with a few key ideas at its core. First and foremost, it acknowledges the fact that there is no silver bullet in software verification, in the sense that no single technique will ever be able to succeed in assessing all properties a user can be interested in. Thus, the platform should foster collaborations between various techniques, by letting individual analyzers exchange information about the properties they can handle as well as the hypotheses they make along the analysis (in the hope that another analyzer may be able to validate them). In a similar manner, the platform was meant to be easily extensible, in particular, by third-party developers. This is also reflected by the choice of the LGPL license for open-source releases of the tool, which allows the development of proprietary plug-ins as long as any change made to the core platform is contributed back. Finally, Frama-C is meant to be usable by software engineers that are not necessarily experts in formal methods. This implies providing as much automation as possible, as well as assessing the performances of the platform on real-world case studies. The purpose of this paper is to provide a panorama of the platform, its key design choices and its uses.

Since its first public release in 2008, Frama-C has been continuously evolving. An active R&D is conducted to bring well-established program analysis techniques (such as abstract interpretation or weakest precondition calculus) to the level of industrial-strength tools. In parallel, novel techniques are developed for specific analysis tasks, for example, for specification and verification of specific kinds of properties coming into focus with the increasing

*CACM, 2019, City*
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

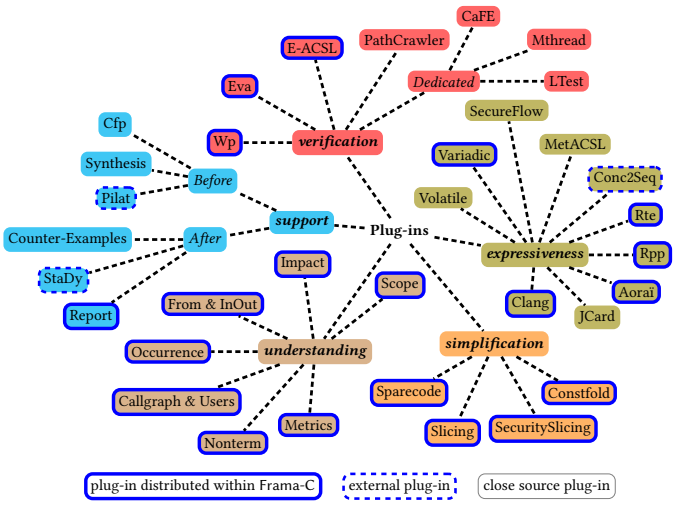


**Figure 1: Frama-C plug-in gallery.**

complexity of modern software, or enhancing existing techniques with new approaches. This paper illustrates efforts of both kinds.

## 2 OVERVIEW OF THE PLATFORM

Frama-C allows users to analyze a given C program, better understand (or even simplify) it and assess properties about it. The user can for instance explore the program structure and compute some metrics on it. Program properties can be explicitly expressed as annotations written in the formal specification language ACSL (described below). They can be validated by partial, dynamic verification or formally verified by rigorous, static verification.

Frama-C is not a single tool, but a framework that groups together several tools. Each tool is provided as a plug-in. The latest open-source release, Frama-C 21-Scandium contains 27 plug-ins. Frama-C offers an extensible and collaborative setting: anyone can develop and provide new plug-ins, and the plug-ins can collaborate with each other in different ways.

### 2.1 Different Plug-ins for Different Analyses

Figure 1 shows a selection of Frama-C plug-ins. **Verification plug-ins** are the most important ones. The value analysis plug-in Eva focuses on detecting *undefined behaviors* (often called *runtime errors*) and tries to prove their absence. For example, for the code `if(*p<0) *p = -(*p);` where `p` is of type `int*`, it has to check that: (i) reading and writing `*p` is safe, and (ii) `-(*p)` does not overflow, that is, $*p \neq -2^{31}$, because the type `int` (over 32 bits) can only express values $-2^{31}...(2^{31}-1)$. For a division, it has to check that the denominator is not 0. Eva does not require additional annotations: potential runtime errors can be deduced from the code.

On the contrary, proving program-specific, *functional* properties requires first to specify them as ACSL annotations. For the previous example, such annotations can state that it computes the absolute value of `*p`. Then such properties can be proved using the deductive verification plug-in Wp. It can also require additional proof-guiding annotations or even a user-guided, *interactive* proof.

Sometimes, when such properties are not (yet) proved, the user can automatically verify them at runtime for a given execution using E-ACSL. The user can also automatically generate test inputs and check for these inputs that the program behaves as expected using PathCrawler. A few other plug-ins are specialized, such as CaFE for temporal properties, MThread for concurrency properties and LTest for test automation.

Several plug-ins are aimed at **supporting the verification process**, either before or after the run of verification plug-ins. Cfp [1] prepares the analysis with Eva for a given library function specified with an ACSL contract by inferring a suitable analysis context. Synthesis automatically generates a function body implementing a given function contract. Pilat [16] infers necessary proof-guiding annotations for loops (as polynomial loop invariants) for a proof with Wp. In case of a proof failure, StaDy. and Counter-Examples aim at generating a counter-example. Report summarizes what has been (or not yet) verified.

Other plug-ins help verification engineers to **better understand the analyzed code**: From, InOut, Impact, Scope, and Occurrence detail dependency and scope information related to memory locations. Callgraph and Users provide information about function calls, while Nonterm warns about non-terminating code. Metrics provides some code metrics.

A few plug-ins are program transformers that **simplify the analyzed code**. Constfold performs constant propagation, while Slicing removes pieces of code that are irrelevant with respect to a specific criterion. Sparecode and SecuritySlicing [32] perform specialized simplifications, removing non-executable, *dead* code or code irrelevant for confidentiality/integrity properties.

Last but not least, several plug-ins **extend the expressiveness of other analyzers**. Frama-Clang and JCard target C++ and JavaCard code, Volatile and Variadic specifically deal with volatile memory locations and variadic functions, while RTE, Aoraï, RPP, MetACSL, Conc2Seq, and SecureFlow automatically generate ACSL properties from higher-level or implicit specifications.

### 2.2 Plug-in Collaboration

No program analysis technique is perfect by nature: many program analysis problems are *undecidable*. In other words, it is impossible to create a tool capable to solve them for all programs. However, some approaches and tools are more efficient for particular kinds of properties or programs than others. To take benefit from the strengths of different tools, Frama-C promotes analyzer collaboration. It can be used to decompose verification work and comes in two different flavors: *sequential* and *parallel*.

**Sequential collaboration** consists in using the result of an analyzer as input to another one. It can also consist in generating annotated C code that encodes a verification problem in such a way that it can be understood by another analyzer. Such collaborations are allowed by the plug-ins in *support* and *expressiveness* categories of Fig. 1. Several examples are provided below.

**Parallel collaboration** consists in using several analyzers to verify program properties, each analyzer verifying a subset of properties. For instance, absence of undefined behaviors can be verified by Eva, while functional properties can be proved by Wp. Eventually, the few remaining properties may be checked at runtime by E-ACSL. Frama-C ensures the consistency of partial results emitted

by the analyzers, and summarizes what has been verified and what remains to be [14].

### 2.3 Platform Architecture

Frama-C plug-ins are based on a *kernel* that provides key services to both end-users and plug-in developers. The kernel contains three main components: (1) basic services (such as program parsing) that build a normalized representation (called Abstract Syntax Tree, or AST) of the analyzed program, (2) specialized services (e.g. exploring and manipulating the program AST including ACSL annotations) for code analyses and (3) general-purpose libraries. Altogether, they provide a large API, providing useful services to analyzers and facilitating plug-in development. This makes it possible to develop, within a few days, a brand-new prototype analyzer supporting most C constructs.

## 3 ACSL SPECIFICATION LANGUAGE

For specifying C code, Frama-C offers ACSL, the ANSI/ISO C Specification Language[1]. ACSL clauses (*annotations*) are written in special comments `//@...` or `/*@... */`. While ACSL is a fairly rich language, we give in this paper only a very brief description. Interested readers can refer to existing tutorials[2] for an in-depth presentation.

As we mentioned above, Frama-C can be used to check that no input leads to a runtime error (RTE) in a given program. Such checks can be generated by the RTE plug-in as ACSL assertions, for verification by other plug-ins (Eva, Wp, or E-ACSL). An ACSL assertion (**assert** clause) can be put anywhere in the code to indicate that a property must hold at this particular point. For the code example `if(*p<0) *p = -(*p);` we considered above, RTE generates the following (simplified) assertions:

```
//@ assert \valid(p);
if( *p < 0 ){
  //@ assert *p>INT_MIN;
  *p = - (*p);
}
```

These assertions indicate precisely the required properties: (i) pointer `p` is *valid*, that is, `*p` can be safely read/written, and (ii) `*p` should be greater than the minimal value of type **int**. Lines 13–14 of Fig. 4 show an assertion to prevent a division by 0.

Obviously, such properties only ensure the absence of undefined behaviors. They do not mean that the program behaves as intended. In order to verify its *functional* properties—the intended behavior—it is necessary to have a precise, formal description of what this intended behavior is. Such a description can also be expressed in ACSL.

A key ingredient of ACSL is the notion of *function contract*, which can be traced back to Eiffel and Meyer's *Design by Contract* [31]. Basically, a (function) contract defines some constraints on the state in which the function might be called (the *precondition*), and in exchange provides some guarantees about the state in which it returns control to its caller (the *postcondition*). It is also important to define which parts of the state (i.e. which variables or memory locations) can be modified during the execution of the function (the *frame rule*). Thanks to it, the caller knows that everything that is not in the frame is left untouched.

[1] https://github.com/acsl-language/acsl/releases/tag/1.14
[2] https://github.com/fraunhoferfokus/acsl-by-example/raw/master/ACSL-by-Example.pdf, https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf

```
1  /*@ requires \valid(p);
2      requires *p > INT_MIN;
3      assigns *p;
4      ensures ( \old(*p) ≥ 0 ⇒ *p == \old(*p) ) ∧
5        ( \old(*p) < 0 ⇒ *p == -\old(*p) );
6  */
7  void pabs(int *p){
8    if (*p < 0)
9      *p = -(*p);
10 }
```

**Figure 2: Example of ACSL function contract.**

```
1 int i = 0, j = 10, k = 12;
2 /*@ loop invariant 0 ≤ i ≤ 10;
3     loop invariant i+j == 10;
4     loop assigns i,j;
5 */
6 while (i < 10) { i++; j--; }
7 //@ assert j == 0;
8 //@ assert k == 12;
```

**Figure 3: Example of ACSL loop contract.**

Figure 2 shows a possible contract for a simple function with the considered conditional statement. The **requires** clauses express the precondition (lines 1–2), denoted $\text{Pre}_{\text{pabs}}$. It states that function `pabs` expects to be called with an argument `p` that is a valid pointer, and the pointed value is greater than the minimal value of type **int**. This precondition guarantees the absence of runtime errors in the function. The **ensures** clause (lines 4–5) expresses the postcondition $\text{Post}_{\text{pabs}}$, which states that the resulting value of `*p` is the absolute value of its initial (old) value. Furthermore, `pabs` is supposed to modify only `*p`, as indicated by the **assigns** clause on line 3.

Another important ingredient of ACSL is a *loop contract*. Placed in front of a loop, it contains clauses providing additional information to reason about the loop behavior. It includes a *loop invariant* stating properties that hold when entering the loop for the first time and are preserved after each loop step. Hence, by induction, they also hold at the end of the loop, regardless of the number of steps. As for functions, loops also have frame rules, introduced by **loop assigns**. Figure 3 illustrates these annotations (see lines 2–4) on a very simple loop manipulating `i` and `j` together in order to keep their sum constant, while leaving variable `k` untouched. Lines 7–8 contain two assertions that hold after the loop. We will illustrate below how loop contracts help to reason for programs with loops.

ACSL uses first-order logic formulas, with integer and real arithmetic. Unlike the bounded C types, ACSL's integer and real types are unbounded. In particular, this makes it easier to write annotations stating the absence of arithmetic overflow. For instance, assuming `x` and `y` are C variables of type **int** (hence also their sum), the following assertion will guarantee that their sum can be safely computed in C, without triggering an overflow:

```
1 /*@ assert INT_MIN ≤ x+y ≤ INT_MAX; */
```

Finally, ACSL features several built-in predicates for stating properties over the pointers manipulated by the program.

## 4 CORE ANALYSES OF THE PLATFORM

This section introduces the four core Frama-C plug-ins, namely Eva that focuses on detecting undefined behaviors, Wp that aims

```
1  int x, y, sum, res;
2  // D²x = D²y = D²sum = D²res = {Uninit}
3  if(getchar()=='*'){
4    x = 0; y = 0;
5  // D⁵x = {0}, D⁵y = {0}
6  }else{
7    x = 1; y = 1;
8  // D⁸x = {1}, D⁸y = {1}
9  }
10 // D¹⁰x = {0, 1}, D¹⁰y = {0, 1}
11 sum = x + y;
12 // D¹²sum = {0, 1, 2}
13 //@ assert sum ≠ 0;
14 res = 10/sum;
```

(a)

```
1  int x, y, sum, res;
2  // D²x = D²y = D²sum = D²res = {Uninit}
3  if(getchar()=='*'){
4    x = 0; y = 1;
5  // D⁵x = {0}, D⁵y = {1}
6  }else{
7    x = 1; y = 0;
8  // D⁸x = {1}, D⁸y = {0}
9  }
10 // D¹⁰x = {0, 1}, D¹⁰y = {0, 1}
11 sum = x + y;
12 // D¹²sum = {0, 1, 2}
13 //@ assert sum ≠ 0;
14 res = 10/sum;
```

(b)

**Figure 4: Eva illustrated on two toy examples, (a) and (b).**

at proving functional properties, E-ACSL that checks properties at runtime, and PathCrawler that generates test cases.

## 4.1 Chasing Undefined Behaviors with Eva

The Eva plug-in provides a configurable and automatic analysis of the whole program, intended to prove the absence of undefined behaviors. This term refers to instructions for which the C standard imposes no requirements, leading to crashes and more generally unpredictable execution flow. They can in particular result in security vulnerabilities and attackers frequently exploit such illegal instructions in order to steal data or execute malware. Eva detects most undefined behaviors, such as invalid memory accesses, uninitialized memory reads, divisions by zero, signed integer overflows, undefined bit shifts and invalid pointer comparisons. It can also treat as erroneous some behaviors that are allowed by the standard but often unwanted by developers, such as unsigned integer overflows or exceptional floating-point values (e.g. infinities).

Eva is based on a technique called *abstract interpretation*. The goal of the analysis is to compute a set of possible values for each variable at each program point. Since computing these sets precisely is undecidable (as we explained in the Plug-In Collaboration section), Eva uses *abstractions* to over-approximate them. For instance, if the set of values $D_v^l$ of a variable $v$ at program point $l$ is $\{1, 3, 5, \ldots, 97, 99\}$, it can be approximated as the integer interval $[1, 99]$. If $v$ takes value $v_0$ at point $l$ for some execution, $v_0$ will necessary belong to the approximated set $D_v^l$. The contrary is not true: the set $D_v^l$ can contain values that $v$ never takes at point $l$ in practice. Thus, the computed abstractions build a sound over-approximation of all possible behaviors. As a consequence, Eva is sound: its analysis is *exhaustive* and reports *all* undefined behaviors that could happen in an execution of a program.

Let us illustrate how Eva analyzes the toy example of Fig. 4a, that gives the body of function `main`. It expects the user to type a character (cf. line 3). In the majority of cases, the else branch is activated and the program executes without errors. But if the user types '`*`', the program executes the then branch and tries to divide by 0 on line 14. We use line numbers to refer to program points $l$. At line 2, the sets of values computed by Eva (shown in comments on line 2) for the four variables contain only the special "Uninitialized" value. After the assignements of line 4 (resp., 7), the new domains of `x` and `y` are shown on line 5 (resp., 8), the others being unchanged. On line 10, the domains coming from both branches are merged. Therefore, the computed over-approximated set of values for `sum` on line 12 is $D_{\text{sum}}^{12} = \{0, 1, 2\}$, even if the value 1 is not possible in practice. Since $0 \in D_{\text{sum}}^{12}$, the assertion on line 13 cannot be proved, and Eva reports a potential division by 0. This is a *true alarm*: the division by 0 can happen, and Eva detects it. Other runtime errors can be detected similarly. For instance, if the assignment of `y` is removed on line 4, Eva computes $D_y^5 = \{\text{Uninit}\}$ and $D_y^{10} = \{\text{Uninit}, 1\}$, and reports an alarm for reading an unitialized variable on line 11.

Approximations often lead to *false alarms*: correct code can also be flagged as a potential error. It can be seen on the example of Fig. 4b, where the computed over-approximated set of values for `sum` on line 12 is again $D_{\text{sum}}^{12} = \{0, 1, 2\}$, while only value 1 occurs in practice. Based on this over-approximated set, Eva cannot prove the assertion on line 13 and reports a potential division by 0 while it can never happen: this is a false alarm. To avoid it, the user can use *trace partitioning*, i.e. make the analysis consider both paths separately to make the analysis more precise. Eva will continue the analysis of both branches without merging their values on line 10: in both cases (with $D_x^{10} = \{0\}, D_y^{10} = \{1\}$ and $D_x^{10} = \{1\}, D_y^{10} = \{0\}$) Eva will compute $D_{\text{sum}}^{12} = \{1\}$ and prove the absence of the error.

To limit the burden of false alarms while maintaining a reasonable analysis time, a balance must be reached between precision and efficiency. Typically, Eva is used in an iterative process where the analyst uses the result of one analysis run to finely tweak the next one, by configuring the abstractions and partitioning. To make complex settings easily accessible for non-expert users, Eva provides a meta-option `-eva-precision N`, with `N` between 0 and 11, which conveniently adjusts a dozen of underlying options (including trace partitioning [30]). Any $\texttt{N} \geq 1$ avoids the false alarm for Fig. 4b.

Eva provides various means of expressing abstractions (called *abstract domains*) that can be enabled and tuned on a case-by-case basis. The default abstract domain represents integer values as small discrete sets or intervals with a linear congruence information, floating-point values as intervals following the IEEE 754 standard, and pointers as possible offsets for each potential base address. It accurately represents arrays, structures and unions. Various additional domains (such as *gauges* [39], *numerors* [25], numerical domains provided by Apron[3]) bring more expressiveness but slow down the analysis.

*Studying the results.* The main output of Eva is an exhaustive list of potential undefined behaviors, or *alarms*, expressed as ACSL assertions. Each alarm should be reviewed to determine if it reveals a real bug or is a false alarm caused by the analysis approximations. False alarms might be disproved by other Frama-C plug-ins. It is possible to inspect (cf. Fig. 5), at each program point and for each call stack, the values computed for each variable and expression.

Eva is tightly integrated with other tools of the platform providing to them detailed information about its results. These results are used by many other plug-ins. Notably, Studia highlights all statements reading or writing a given memory location, allowing the user to jump between the sink of a bug (where it can be observed) and its source (the actual culprit). InOut computes the memory zones read and written by a function, summarizing its

[3] http://apron.cri.ensmp.fr/library/

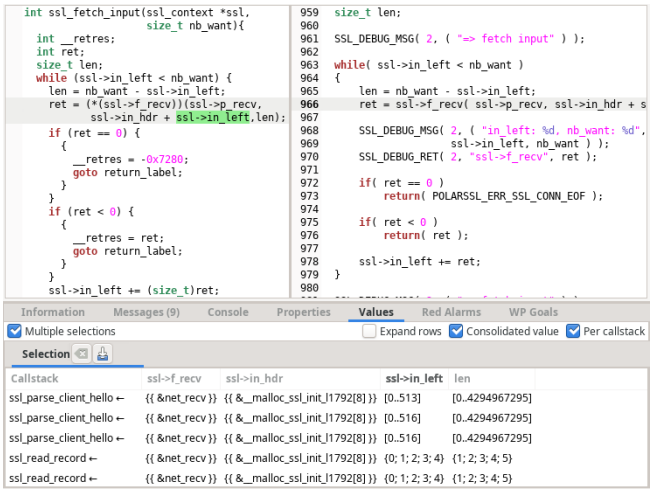


**Figure 5: Frama-C graphical interface allows any C expression to be inspected for its possible runtime values, as computed by Eva. Pointers, structured and scalar values are expressed in a concise but precise notation. Each callstack is separated, with filtering and grouping capabilities.**

```
1 int x=-42, y=36;
2 int *q=&x;
3 // Pre_pabs holds
4 pabs(q);
5 //@ assert x==42;
6 //@ assert y==36;
```

```
1 int x=INT_MIN;
2 int *q=&x;
3 // Value of *q
4 // is INT_MIN.
5 // Pre_pabs fails
6 pabs(q);
```

```
1 int a[2]={-42,0};
2 int *q=&a[0];
3 // Pointer q+2
4 // is invalid.
5 // Pre_pabs fails
6 pabs(q+2);
```

**Figure 6: Wp illustrated on toy examples, (a), (b) and (c).**

dependencies. Finally, Metrics estimates the analysis code coverage and reports the statements proven unreachable by Eva.

*Usage.* Eva handles the subset of C99 commonly used in embedded software. Dynamic allocation is supported, but often leads to imprecise results. The analysis is fully context-sensitive: function calls are inlined, and recursive functions are not supported. Eva has been highly optimized for years to achieve scalability on large programs, and has already been successfully applied to verify safety-critical codes, especially in the nuclear industry [33].

## 4.2 Proving Functional Properties with Wp

*Deductive verification* aims at proving that functional properties of a program hold in all cases. It is usually performed in a *modular* way, function by function, where the caller's proof can rely on the callee's contract, proved separately. The Wp plug-in is a modern and effective implementation of this approach for C and ACSL.

Let us illustrate this approach on the code of Fig. 6a (say, giving the body of function `main`) that calls the function of Fig. 2. After line 2 of Fig. 6a, pointer `q` refers to `x`, thus `x` and `*q` are aliases. On this code, Wp deduces the first assertion from the value −42 of `*q` before the call and the postcondition of `pabs` (cf. lines 4–5 in Fig. 2). Since other variables cannot be modified by the call of `pabs` (cf. line 3 in Fig. 2), Wp also proves the second assertion of Fig. 6a.

However, a call to a function guarantees to ensure its postcondition after the call only if its precondition is respected before the call. Thus, Wp must check that the precondition of `pabs` (cf. lines 1–2 in Fig. 2) is respected before the call: here, indeed, pointer `q` is valid and the pointed value is not `INT_MIN`. The precondition cannot be proved for the code of Fig. 6b, where the pointed value is `INT_MIN`. Its proof also fails for the code of Fig. 6c, where `q` refers to the first cell of an array of two integers, hence `q+2` is invalid: dereferencing it would be an out-of-bound access (i.e. an undefined behavior).

In the modular approach, the function contract of the callee must be proved separately. For the code of Fig. 2, Wp successfully proves that the implementation of `pabs` respects its contract.

Deductive verification for programs with loops usually relies on loop contracts that must be specified by the user. To illustrate it on a toy example, consider the code of Fig. 3. For the loop contract, Wp must verify that the loop invariant is indeed true before the loop and is preserved by each new loop iteration, and that the loop frame rule is indeed true. Thanks to the loop invariant, at line 7 Wp knows that $0 \leq i \leq 10$, $i + j = 10$, and since the execution exited the loop, $i \geq 10$. From these conditions, it deduces that $i = 10$ and therefore $j = 0$, that proves the assertion on line 7. The assertion on line 8 is deduced from the frame rule (line 4) since the value of $k$ cannot be changed by the loop.

To perform deductive verification, Wp relies on *Hoare Logic* and *Weakest Precondition Calculus*. At a high level, Wp compiles C code and ACSL contracts into mathematical theorems (called *verification conditions* and expressed as first-order logic formulas) that provide *sufficient* conditions to entail the validity of the expected functional properties. These theorems use various mathematical theories (including integer and real arithmetics, anonymous functions, arrays and records). They are then sent to automated theorem provers (or SMT solvers, like Alt-Ergo, Z3 or CVC4) to be checked for validity. Alternatively, one can also use a proof assistant like Coq.

Naive implementations of *weakest precondition calculus* are known to have exponential cost and cannot be used on complex programs. Moreover, modelling the semantics of C memory access with aliasing and low-level encoding of data is known to be a challenge for automated reasoning. Wp has been developed since 2008 with an industrial target in mind, and benefits from well-known modern techniques to make it efficient.

Wp implements a generic backward calculus engine to produce verification conditions by weakest precondition calculus [28]. It is parameterized by a *memory model*, defining a specific representation of memory locations in the resulting verification conditions. Wp features various memory models which combine known techniques [22] to propose different balancing between efficiency and expressiveness, and some heuristics to select which model(s) to apply on a given program.

Wp offers several backends for discharging the generated verification conditions with automated SMT solvers and proof assistants, either natively or via the Why3 [21] platform. The complexity of the generated verification conditions is dramatically reduced by Qed [13], a generic and extensible simplification engine of Wp, helping to discharge some corner cases of theories that are still issues for mainstream SMT solvers. Finally, Wp features an extensible proof tactic engine to *interactively* split complex proofs into smaller ones, possibly executing custom decision procedures.

Altogether, these features make Wp an efficient implementation of deductive verification to prove functional properties of C/ACSL programs. A recent industrial use-case in avionics [9] reports that

98.5% of the 3315 C functions were proved by Wp, where only 2.3% functions required the interactive termination of some proofs.

## 4.3 Checking Properties at Runtime with E-ACSL

Runtime assertion checking is the process of verifying specifications (historically, assertions) at runtime, i.e. when the program is being executed. It was popularized by the programming language Eiffel in the late 1980s to support defensive programming. At the turn of the millennium, this approach was adopted by dedicated formal specification languages for mainstream programming languages like JML for Java or Spec# for C#.

In the context of C and Frama-C, ACSL would be the language of choice for runtime assertion checking. However, being primarily designed for deductive verification, it needed adjustments for runtime checking. In addition, verifying expressive properties at runtime for a language like C in a sound and efficient way is challenging and requires original solutions.

*4.3.1 Specification Language Adjustments.* As explained above, ACSL is based on mathematical logic. In particular, it contains several constructs that have no computational meaning, such as lemmas and axioms, or unbounded quantifications. Therefore, they were removed from the *executable* subset of ACSL dedicated to runtime assertion checking: the E-ACSL specification language[4].

Another important issue of ACSL with respect to runtime checking is its logic-based semantics that assigns a (possibly unspecified) value to each construct. For instance, the predicate $0/0 \equiv 0/0$ is necessarily true in ACSL by reflexivity of equality. This semantics helps formal reasoning made by the Wp plug-in and associated provers. However, it is problematic at runtime since terms like $0/0$ cannot be safely executed. Consequently, E-ACSL considers that the semantics of such terms is actually undefined (relying on Chalin's strong validity principle [11] and three-valued logic). Undefined terms and predicates must never be executed.

*4.3.2 Compiling Formal Properties into Executable Code.* Compiling E-ACSL annotations into C code is the purpose of the E-ACSL plug-in [38] of Frama-C. The instrumented code it produces checks the annotations at runtime and reports failures. For instance, using the E-ACSL plug-in to check at runtime the code of Fig. 6a confirms that the annotations (including the assertions, the pre- and postcondition of `pabs`) are verified, while for Fig. 6b,c the failing preconditions are detected and reported to the user.

At a first glance, the compilation process may look quite easy. For instance, the E-ACSL assertion `/*@ assert z ≠ 0;*/` is compiled to the C assertion `assert(z ≠ 0);`. However, in general it is not always so simple to generate both sound and efficient code, as shown below on two illustrative cases.

*Arithmetic.* For the E-ACSL assertion `/*@ assert x+1 ≤ INT_MAX;*/`, it would be unsound to generate the C code `assert(x+1 ≤ INT_MAX);` since at runtime `x+1` might overflow, while in the ACSL specification, as we explained above, it is computed over (unbounded) mathematical integers. Consequently, E-ACSL generates specific code[5]

[4] http://frama-c.com/download/e-acsl/e-acsl.pdf
[5] based on GNU multiple precision library, https://gmplib.org/

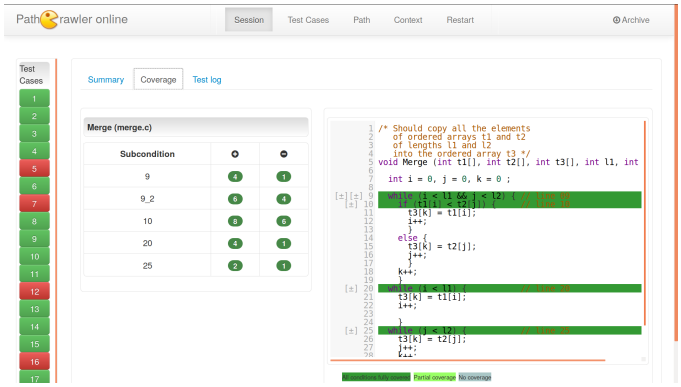


**Figure 7: Results of a test generation session with PathCrawler-online illustrating condition coverage of the generated test cases. Failed test cases are shown in red. For each test case, its inputs, outputs and the activated path can be inspected. Any gaps in the coverage of the function are also explained.**

to perform the computations precisely and to remain sound. To remain efficient, it still generates (more efficient) machine arithmetic based code when it is sound to do so. For instance, assuming that the type of `x` is `int` on a standard 64-bit architecture, the previous assertion is compiled to `assert((long)x+1L ≤ (long)INT_MAX);` to execute the addition and comparison without overflow over the larger C type `long`.

*Memory Properties.* An important feature of the specification language are memory properties such as `\valid(p)`. In order to soundly and efficiently evaluate such properties, memory-related operations (allocations, deallocations and assignments) in the original code that are relevant for the memory properties of interest are recorded in a dedicated datastructure.

## 4.4 Generating Test Cases with PathCrawler

Another dynamic analysis plug-in of Frama-C is PathCrawler [40]. Given a C program and a specific function in it, PathCrawler generates unit test cases for this function. Basically, it explores (a subset) of program paths and tries to generate test inputs for each of them. PathCrawler follows the so-called *concolic* test generation technique (also called *Dynamic Symbolic Execution*) since it combines *symbolic execution* of the program with a usual (*concrete*, i.e. non symbolic) execution of the compiled code. Symbolic execution represents the execution of a program path symbolically, with undetermined values of program inputs. It relies on the path predicate defining the values of the input variables that activate the chosen path. To find a set of concrete values satisfying the path predicate, i.e. test inputs for the path, PathCrawler relies on the Colibri constraint solver, also developed at CEA List. A concrete execution of the generated test on an instrumented version of the program is used to confirm the executed path and to optimize the test generation process.

To ensure that the test inputs are realistic, and avoid detecting bugs which would never arise in legitimate function calls, the user can provide a precondition limiting the admissible input values. If the user also provides an *oracle* function, that compares the outputs

produced by the test with the expected behavior, then PathCrawler will automatically report a *pass* or *fail* verdict for each test.

Since 2009, PathCrawler has an online version[6] (see Fig. 7) that allows the user to provide a C file (or choose one of the available examples), generate test cases and explore the results.

# 5 TELL FRAMA-C WHAT YOU WANT TO VERIFY

Together with the expressive power of ACSL specifications, the core analyzers presented in the previous section allow the verification of a very large class of properties about C programs. However, the bare ACSL language makes it sometimes difficult to express other kinds of properties. In that case, there exist various specialized plug-ins dedicated to ease the task of writing the formal specification of a property of interest.

In many cases, such a plug-in offers a dedicated Domain-Specific Language (DSL) for writing the property and operates by instrumenting the code under analysis with additional ACSL annotations and/or C instructions so that the verification of standard ACSL annotations on the instrumented code with the core analyzers implies that the original DSL formulas hold on the original code.

## 5.1 Verify Sequences of Events: Aoraï and CaFE

It is often necessary to verify that a set of events during a program execution follow a particular order, for example:

> a call to function `send_private_data()` must always be preceded by a call to function `authenticate()` returning `0`, without a call to function `logout()` in-between.

Such properties (often expressed in *temporal logic* [12]) can be verified for any given execution using an *automaton*. In our example, it consists of three states encoding the current status of the execution: user non authenticated (initial state), user authenticated (and not yet logged out), or error (i.e. private data sent without being authenticated). The transitions between states naturally follow the observed function calls, except that the error state cannot be left. The first two states are accepting (i.e., the property is respected as long as the execution ends in one of them), while the last one means the property fails for the given execution.

Two Frama-C plug-ins are dedicated to such properties. Aoraï [26], simply adds C variables representing the states, together with the appropriate transition functions and ACSL annotations ensuring we end up in an accepting state. Checking the validity of these annotations is then left to one of the main analysis plug-ins described in the previous section. A more recent plug-in, CaFE, can handle additional properties including nested function calls. CaFE is based on a refined version of classical temporal logic, CaRet [2], and relies on model-checking techniques [12].

## 5.2 Verify Relational Properties: RPP

On the contrary to an ACSL contract, that specifies what is supposed to happen during a single call to the corresponding function, *relational properties* examine the relations that may exist between several executions of either the same or different functions. An interesting example of this class of properties is *non-interference*:

```
1 int sec, pub;
2 void noleak(){
3  pub=pub+10;
4  sec=sec+pub; }
5 void leak(){
6  pub=pub+sec; }
```

```
1 int sec1, pub1, sec2, pub2;
2 /*@ requires pub1==pub2;
3     ensures  pub1==pub2; */
4 void wrapper_leak(){
5  pub1=pub1+sec1; /* 1st call */
6  pub2=pub2+sec2; /* 2nd call */ }
```

**Figure 8: (a) A C code, and (b) RPP transformation for `leak`.**

given a partition of the variables into public and private ones, one wants to ensure that any two executions starting in states where public variables have the same values always end up in states where public variables have the same values. In other words, the public result should not depend upon the values of private variables. Figure 8a illustrates a function `noleak` that respects this property: the public variable `pub` does not depend on the secret variable `sec`. This property is not true for function `leak`, where `pub` depends on `sec`.

RPP [8] is a Frama-C plug-in that offers an extension of ACSL to formally specify relational properties (involving any number of executions of any number of functions). RPP then uses a form of self-composition [6] to generate a wrapper function (composing the executions of the functions involved in the relational property) with an ACSL contract such that its proof implies the relational property for the original code. For instance, the wrapper of Fig. 8b simulates two executions of `leak` with equal public values (line 2), but this equality after these executions (line 3)—the non-interference—cannot be proved: the public result depends on a secret variable.

An important benefit of RPP's transformation is that it also allows the use of a proven relational property as a hypothesis in subsequent proofs, following the modularity of the standard deductive verification approach.

## 5.3 Enforce Global Properties: MetACSL

It is often the case that one wants to enforce a given property across the whole program. For instance, we may associate a confidentiality level with each memory location and check that a read access is never done from a location with a higher level than that of the current user, and dually, a write is never performed into a lower-level location. While these kinds of properties could in theory be expressed with standard ACSL annotations, they would spread everywhere in the program. In practice, it can be difficult to write them all by hand without making a mistake and to convince ourselves that the set of annotations is indeed complete.

The recently started MetACSL plug-in [37] seeks to alleviate this issue by automatically generating these ACSL annotations from a single higher-level property expressed in a small DSL extending ACSL to indicate the contexts in which the property must hold. It has been tested over various examples to establish security properties (confidentiality and integrity) and is currently being assessed over more realistic case studies.

## 5.4 Prove Concurrent Programs: Conc2Seq

While most of its plug-ins focus on sequential program analysis, Frama-C also offers an experimental plug-in, called Conc2Seq, for deductive verification of concurrent programs [7]. Similarly to the CSec approach[7], it performs a dedicated code transformation of a given concurrent program into a sequential one, which simulates

[6] http://pathcrawler-online.com/

[7] http://www.southampton.ac.uk/~gp1y10/cseq/cseq.html

concurrent executions of the code in several threads by interleaving the executions of indivisible (*atomic*) blocks in various ways, defined non-deterministically. Conc2Seq also automatically transforms specifications of the initial program into specifications for the resulting program. The variables of various threads are represented by arrays in the simulating program, so that the user can add guiding annotations relating these variables between them to help the proof. Thanks to this transformation, the Wp plug-in can be used to verify the resulting sequential program. If the proof of the annotations for it is successful, the initial concurrent program respects its specification.

### 5.5 Specify Test Objectives: LAnnotate

Test objectives give another example of specific annotations that can be added for analysis using Frama-C core plug-ins. Various structural test coverage criteria (e.g. functions, statements, decisions or branches, conditions, conditions-decisions) can be treated in a unified way provided that the corresponding test objectives are expressed in the code in the generic form of elementary coverage targets. Such a coverage target, also called a *label* [4], is basically a predicate inserted in a particular location. A label is covered by a test when the execution of the test reaches this location and satisfies the predicate. For a given test coverage criterion, the LAnnotate plug-in [4] inserts the corresponding labels, and other plug-ins can be used to reason about them. In particular, PathCrawler supports the label coverage criterion (and therefore, all coverage criteria that can be expressed using labels) and offers an efficient test generation for labels. Other usages of labels will be discussed in the next section.

## 6 GO BEYOND RAW ANALYSES RESULTS

The previous section presented several analyses that apply core plug-ins after a relatively lightweight adaptation (often, via instrumentation) of properties of interest into properties they can directly handle. For more complex analysis problems, this is not sufficient: the target properties can require an advanced code or specification transformation or even a dedicated reasonning. Their analysis can still rely on some of the core analyzers, but has to extend or adapt them in a more significant way. We present here a few examples.

### 6.1 Counter-Examples for Unproven Annotations: StaDy

Manual analysis of proof failures during deductive program verification can be a very complex and time-consuming task. Such a failure can be due to an error in the code or in the specification itself, a missing or too weak specification for a called function or a loop, or lack of time or simply incapacity of the prover to finish a particular proof. Using a combination of deductive verification (with Wp) and test generation (with PathCrawler), the StaDy plug-in [35] helps to classify proof failures into several categories and provides a counter-example illustrating the issue. The translation of ACSL annotations (preconditions, postconditions, etc.) into their counterparts supported by test generation is not straigtforward. For example, to support unbounded integers in ACSL annotations during both concrete and symbolic execution, operations with unbounded integers are translated in two different ways: directly into unbounded integers supported by the constraint solver for symbolic execution, and using a dedicated library for execution of unbounded integers for a concrete execution.

While StaDy was mainly designed for using with Wp, it can also be applied to alarms reported by Eva. Such alarms being reported as unproven assertions, StaDy can be applied to generate counter-examples for some of them (thus showing that they are not false alarms) and facilitate the analysis of alarms by the verification engineer. This is another illustration of the benefits of sharing the same specification language between different analyzers.

### 6.2 Infeasible Test Objectives: LUncov

Section 5.5 illustrated how generic test objectives—or labels—allow PathCrawler to support test-case generation for various test coverage criteria. An important issue in testing is related to *infeasible* (i.e. uncoverable) test objectives that cannot be covered by any test case. Infeasible test objectives lead to an imprecise computation of coverage for a given test suite, and a waste of resources for trying to cover them. Detection of infeasible test objectives—which is in general undecidable—is thus an important task in testing.

An efficient approach to identify infeasible test objectives is to use static analysis. This is the purpose of the LUncov plug-in [4] of Frama-C. It translates a label with predicate $p$ into an assertion with the negated predicate $\neg p$ at the same location. The label is uncoverable if and only if the resulting assertion is always true. LUncov implements various analysis techniques relying on value analysis using Eva and weakest precondition calculus using Wp. In particular, it provides an advanced combination of both tools, where Eva is used to compute the domains of program variables and then shares this information with Wp, to make it more precise.

### 6.3 Program Simplification: Slicing

*Slicing* is a program transformation technique that takes as input a program and a so-called *slicing criterion* (e.g. to preserve the value of a given variable at a given program point) and outputs a simplified C program that preserves the property defined by the slicing criterion. The pieces of code necessary to ensure the preservation property (or for a correct compilation) of the resulting program are kept, while all other, irrelevant instructions are removed. Slicing helps the end-user to focus on a particular point of interest. It also facilitates other analyses by reducing the size of the code they must deal with. The Frama-C slicing tool proposes numerous slicing criteria including preserving read and written memory locations at particular program points, function calls, return values, ACSL annotations or statements. It soundly relies on Eva to compute aliasing and dependency information. Therefore, it may over-approximate its results by keeping pieces of code that are actually not relevant for the selected criterion. However, it never removes anything relevant.

### 6.4 Information Flow: SecureFlow

Information flow properties denote properties of the dependencies between the outputs and the inputs of the program. The most common example is non-interference, presented above. It expresses the absence of information leak. The SecureFlow plug-in [3] lets the user annotate each declaration with a `public` or `private` attribute and uses a dataflow analysis to verify the absence of information

leak. It also relies on Eva's results for determining which locations (hence, with which confidentiality level) pointers might refer to.

# 7 MORE THAN A TOOLSET: AN ECOSYSTEM

## 7.1 Frama-C Community

Since its initial release in May 2008, Frama-C managed to attract an active community of users and plug-in developers. Its open-source license (LGPL 2.1) played of course an important role in this development, facilitating its integration into many Linux distributions (the oldest package, from Debian, dating back to 2009 and Frama-C 3.0 Lithium), and into the main repository of the opam package manager that handles software written in OCaml as is the case of Frama-C. As of July 2020, opam reports around 200 monthly downloads (via opam) of the latest release, Frama-C 21.1 Scandium, that appeared in June 2020.

Naturally, these public releases are accompanied with various communication channels[8], including a mailing list, a bug tracker, and a dedicated StackOverflow tag. Frama-C's blog[9] is also a good way to inform users about what is going on in the platform.

An important part of Frama-C development is funded through collaborative projects, mainly at French and European level. Apart from CEA itself, these projects usually gather a mix of academic and industrial partners in order to explore new research directions while keeping sure that they are relevant to real-world problems. Among these projects, we can in particular mention the French RNTL project CAT and its successor U3CAT[10], funded by ANR, which were fundamental for building the grounding blocks of the platform. Later on, European projects Stance and Vessedia help broadening Frama-C's target properties to cybersecurity.

## 7.2 Teaching with Frama-C

Frama-C is intensively used for teaching. It became difficult to keep track of all universities where the toolset is used in various program analysis or verification courses. In France, where the platform was born and is developed, there are dozens of departments relying on Frama-C for teaching every year. To give just a few examples, Ecole Polytechnique, CentraleSupélec, École Normale Supérieure, ENSIIE, almost all universities in and around Paris, but also in Besançon, Bordeaux, Bourges, Grenoble, Lille, Lyon, Orléans, Rennes, Toulouse, and many others. Frama-C is also increasingly used in other countries. They include, for instance, Austria, Brazil, China, Germany, Portugal, Russia, UK, USA. Among the analyzers of the open-source distribution, most popular for teaching are probably Eva, Wp and E-ACSL. PathCrawler is also actively used for teaching thanks to its online version, PathCrawler-online, allowing the user to explore advanced test generation results. Finally, Frama-C has often been used for trainings in industrial companies and for tutorials on program verification at premier international conferences including ASE, FM, iFM, ISSRE, POPL, SAC, TAP, QSIC, etc.

## 7.3 Collaborations and Industrial Applications

Long-time partnerships started around Frama-C's precursor Caveat, which was developed in the 1990s in close collaboration with the teams at Airbus, and leveraging automated reasoning capabilities from Inria's Alt-Ergo solver.

As Caveat went into industrial production, the development around Frama-C continued these collaborations and engaged them in assisting design decisions. Notably, this took the form of a domain specific language for low-level specifications that compiles into ACSL for deductive verification with Wp, or into a system similar to E-ACSL for runtime verification. This system, called NWOW [9], has been deployed at large scale for the development of onboard critical software, and will be extended to other applications.

Other partnerships started, mid-2000s, with Électricité de France (EDF) and Areva for energy production systems. In particular, EDF reported [33] that Frama-C's Value analysis plug-in (predecessor of Eva) improved the analysis of a 39 kLoC nuclear power plant shutdown system, allowing the demonstration of the absence of intrinsic run-time errors. After some experimentation with different tools, Frama-C was chosen to analyze the code. Today, an ongoing collaboration with EDF focuses on the analysis of larger code bases. R&D efforts between EDF, Framatome and CEA study further usage of Frama-C for other safety-critical software.

Frama-C has also been used for verifying software in other industrial domains, notably by Fraunhofer FOKUS [36] and Mitsubishi for rail and Brazil's TIA for space applications [18]. The 2010s saw a broadening of this base, and an extension from safety-critical software into cybersecurity. The capabilities of Frama-C were put to use by NASA in air traffic management [24], SRI International in gamified cybersecurity [20], Bureau Veritas in marine and offshore [27], and Thales and ANSSI in communication systems [19].

Test generation with PathCrawler was recently evaluated by MERCE (Mitsubishi Electric R&D Centre Europe). After developing additional tooling around PathCrawler, MERCE evaluated automatic test generation over industrial code of about 80,000 lines. In this experiment, 86% of functions were successfully covered in 8 hours. MERCE estimated that automatic test generation with PathCrawler could bring an effective benefit factor of more that 230 for test input generation in the company. Those very good results are very encouraging for an adoption of the technology in the business units [5].

Beyond applications, the extensibility of the platform also allowed tool developers to abstract from the groundwork of code parsing and data structure design, and to focus on new types of verification. Early on, Inria experimented with deductive verification in the Jessie plug-in, later on extended and adapted by ISPRAS in AstraVer [29]. Adelard investigated lightweight concurrency, while teams at Atos implemented dataflow conformity capabilities [15] and prototyped IDE integrations. The field of cybersecurity also proved fertile in academic developments, giving rise to the Stac plug-in from Verimag [10] or the Celia plug-in from Université Paris Diderot [17]. Finally, the mid-2010s modernization brought about with the Eva plug-in allowed for another level of extensibility, at the level of its abstract domains. This was quickly adopted to interface with developments in this field from Verimag, including the Apron domain library and the VPL verified polyhedron library [23].

[8] see https://frama-c.com/support.html
[9] https://blog.frama-c.com
[10] https://frama-c.com/u3cat.html

Similarly, in the context of European projects Stance[11] (FP7) and Vessedia[12] (H2020), Search Lab developed Frama-C plug-ins dedicated to generate counter-examples in the spirit of StaDy (Section 6.1) but based on external test case generators, namely Search Lab's own tool Flinder[13] and later the AFL fuzzer[14]. In these projects, Dassault Aviation also designed a methodology based on Eva, E-ACSL and two home-made plug-ins in order to detect security vulnerabilites and deploy runtime counter-measures when necessary [34]. It has been experimented on a few modules of Apache.

## 8 CONCLUSION

Since its first public release more than 10 years ago, the Frama-C framework has demonstrated its adequacy to successfully address very diverse verification tasks. One of the main factors of this success is undoubtedly the key design idea of a modular analysis platform, where developing a specialized plug-in and having it communicate with others should be as easy as possible. Another important aspect is the fact that the development of Frama-C has been fueled by collaborative projects, that strive to maintain a balance between exploring new research directions and targeting existing industrial code. This is still true to this day, with lines of research towards new programming languages (C++, Rust), cybersecurity and privacy properties, verifying AI-based applications or using AI in verification among others. We hope the readers will try out Frama-C[15] and will find it useful for their verification activities.

## REFERENCES


[1] Michele Alberti and Julien Signoles. 2017. Context Generation from Formal Specifications for C Analysis Tools. In *Logic-based Program Synthesis and Transformation (LOPSTR'17)*.

[2] Rajeev Alur, Kousha Etessami, and P Madhusudan. 2004. A temporal logic of nested calls and returns. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*.

[3] Gergö Barany and Julien Signoles. 2017. Hybrid Information Flow Analysis for Real-World C Code. In *Tests and Proofs (TAP'17)*.

[4] Sébastien Bardin, Omar Chebaro, Mickaël Delahaye, and Nikolai Kosmatov. 2014. An All-in-One Toolkit for Automated White-Box Testing. In *Tests and Proofs (TAP'14)*.

[5] Sébastien Bardin, Nikolai Kosmatov, Bruno Marre, David Mentré, and Nicky Williams. 2018. Test Case Generation with PathCrawler/LTest: How to Automate an Industrial Testing Process. In *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'18)*.

[6] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 6 (2011).

[7] Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue. 2016. Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs. In *Source Code Analysis and Manipulation (SCAM'16)*.

[8] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, and Virgile Prevosto. 2017. RPP: Automatic Proof of Relational Properties by Self-composition. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*.

[9] Abderrahmane Brahmi, Marie-Jo Carolus, David Delmas, Mohamed Habib Essoussi, Pascal Lacabanne, Victoria Moya Lamiel, Famantanantsoa Randimbivololona, and Jean Souyris. 2020. Industrial use of a safe and efficient formal method based software engineering process in avionics. In *Embedded Real Time Software and Systems (ERTS'20)*.

[10] Dumitru Ceara, Laurent Mounier, and Marie-Laure Potet. 2010. Taint Dependency Sequences: A Characterization of Insecure Execution Paths Based on Input-Sensitive Cause Sequences. In *Int. Conf. on Software Testing, Verification and Validation (ICST'10)*.

[11] Patrice Chalin. 2007. A Sound Assertion Semantics for the Dependable Systems Evolution Verifying Compiler. In *Int. Conf. on Software Engineering (ICSE'07)*.

[12] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. 1986. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *Transactions on Programming Languages and Systems* (1986).

[13] Loïc Correnson. 2014. Qed. Computing What Remains to Be Proved. In *NASA Formal Methods (NFM'14)*.

[14] Loïc Correnson and Julien Signoles. 2012. Combining Analyses for C Program Verification. In *Int. Conf. on Formal Methods for Industrial Case Studies (FMICS'12)*.

[15] Pascal Cuoq, David Delmas, Stéphane Duprat, and Virginia Moya Lamiel. 2012. Fan-C, a Frama-C plug-in for data flow verification. In *Embedded Real Time Software and Systems (ERTS'12)*.

[16] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. 2016. Polynomial invariants by linear algebra. In *Automated Technology for Verification and Analysis (ATVA'16)*.

[17] Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. 2013. Local Shape Analysis for Overlaid Data Structures. In *Int. Symp. on Static Analysis (SAS'13)*.

[18] Rovedy Aparecida Busquim e Silva, Nanci Naomi Arai, Luciana Akemi Burgareli, Jose Maria Parente de Oliveira, and Jorge Sousa Pinto. 2016. Formal Verification With Frama-C: A Case Study in the Space Software Domain. *Transactions on Reliability* (2016).

[19] Arnaud Ebalard, Patricia Mouy, and Ryad Benadjila. 2019. Journey to a RTE-free X.509 parser. In *Symp. sur la sécurité des technologies de l'information et des communications (SSTIC'19)*.

[20] Daniel Fava, Julien Signoles, Matthieu Lemerre, Martin Schäf, and Ashish Tiwari. 2015. Gamifying Program Analysis. In *Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'15)*.

[21] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 – Where Programs Meet Provers. In *European Symp. on Programming (ESOP'13)*.

[22] Jean-Christophe Filliâtre and Claude Marché. 2004. Multi-prover Verification of C Programs. In *Int. Conf. on Formal Methods and Software Engineering (ICFEM'04)*.

[23] Alexis Fouilhé, David Monniaux, and Michaël Périn. 2013. Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra. In *Int. Symp. on Static Analysis (SAS'13)*.

[24] Alwyn Goodloe, César A. Muñoz, Florent Kirchner, and Loïc Correnson. 2013. Verification of Numerical Programs: From Real Numbers to Floating Point Numbers. In *NASA Formal Methods (NFM'13)*.

[25] Maxime Jacquemin, Sylvie Putot, and Franck Védrine. 2018. A Reduced Product of Absolute and Relative Error Bounds for Floating-Point Analysis. In *Int. Symp. on Static Analysis (SAS'18)*.

[26] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Asp. Comput.* (2015).

[27] Florent Kirchner, Franck Sadmi, Sébastien Flanc, Lucas Duboc, Hélène Marteau, Virgile Prevosto, and Franck Vedrine. 2016. Safer Marine and Offshore Software with Formal-Verification-Based Guidelines. In *Embedded Real Time Software and Systems (ERTS'16)*.

[28] K. Rustan M. Leino. 2005. Efficient weakest preconditions. *Information Processing Letters* (2005).

[29] Mikhail U. Mandrykin and Alexey V. Khoroshilov. 2015. High-level memory model with low-level pointer cast support for Jessie intermediate language. *Programming and Computer Software* (2015).

[30] Laurent Mauborgne and Xavier Rival. 2005. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *European Symp. on Programming (ESOP'05)*.

[31] Bertrand Meyer. 1991. *Design by Contract.* Prentice Hall.

[32] Benjamin Monate and Julien Signoles. 2008. Slicing for Security of Code. In *Trusted Computing and Trust in Information Technologies (TRUST'08)*.

[33] Alain Ourghanlian. 2015. Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nucl. Eng. Technol.* (2015).

[34] Dillon Pariente and Julien Signoles. 2017. Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures. In *Symp. sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'17)*.

[35] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. 2018. How Testing Helps to Diagnose Proof Failures. *Formal Asp. Comput.* (2018).

[36] Virgile Prevosto, Jochen Burghardt, Jens Gerlach, Kerstin Hartig, Hans Werner Pohl, and Kim Völlinger. 2013. Formal specification and automated verification of railway software with Frama-C. In *Int. Conf. on Industrial Informatics (INDIN'13)*.

[37] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. 2019. MetAcsl: Specification and Verification of High-Level Properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19)*.

[38] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. 2017. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper. In *Int. Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES'17)*.

[39] Arnaud Venet. 2012. The Gauge Domain: Scalable Analysis of Linear Inequality Invariants. In *Computer Aided Verification (CAV'12)*.

[40] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. 2005. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In *European Dependable Computing Conf. (EDCC'05)*.


[11] https://cordis.europa.eu/project/rcn/105816/brief/en

[12] https://www.vessedia.eu/

[13] https://www.flinder.hu

[14] http://lcamtuf.coredump.cx/afl/

[15] see http://www.frama-c.com/download.html