DOI:10.1145/3470569

A panoramic view of a popular platform for C program analysis and verification.

BY PATRICK BAUDIN, FRANÇOIS BOBOT, DAVID BÜHLER, LOÏC CORRENSON, FLORENT KIRCHNER, NIKOLAI KOSMATOV, ANDRÉ MARONEZE, VALENTIN PERRELLE, VIRGILE PREVOSTO, JULIEN SIGNOLES, AND NICKY WILLIAMS

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

THE C PROGRAMMING language is a cornerstone of computer science. Designed by Dennis Ritchie and Ken 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. Nearly 50 years after its creation, it is still widely used in software engineering.

But C is a difficult language to wield. Its native design choices give developers much freedom—a reason for its popularity—but that can often 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 are 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 can 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 difficult. 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.

» key insights

- **The C programming language remains popular for system-level programming and embedded code in many critical domains. Verification and validation is crucial to making software-dependent services reliable and secure.**
- **Relying on solid theoretical foundations, formal methods can offer strong guarantees about the software in those systems.**
- **Frama-C, a collaborative, open source platform for code analysis and verification based on formal methods, offers several C code analysis plug-ins.**
- **Frama-C attracted a large community of academic and industrial users, thanks to its open source license and a regular stream of releases since 2008.**

IMAGE BY ANDRIJ BORYS ASSOCIATES

Examples of such behavior include division by 0, illegal memory access, and reading uninitialized variables. In particular, the fact that C allows 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 access, 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: 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 during the analysis (in the hope that another analyzer may be able to validate them).

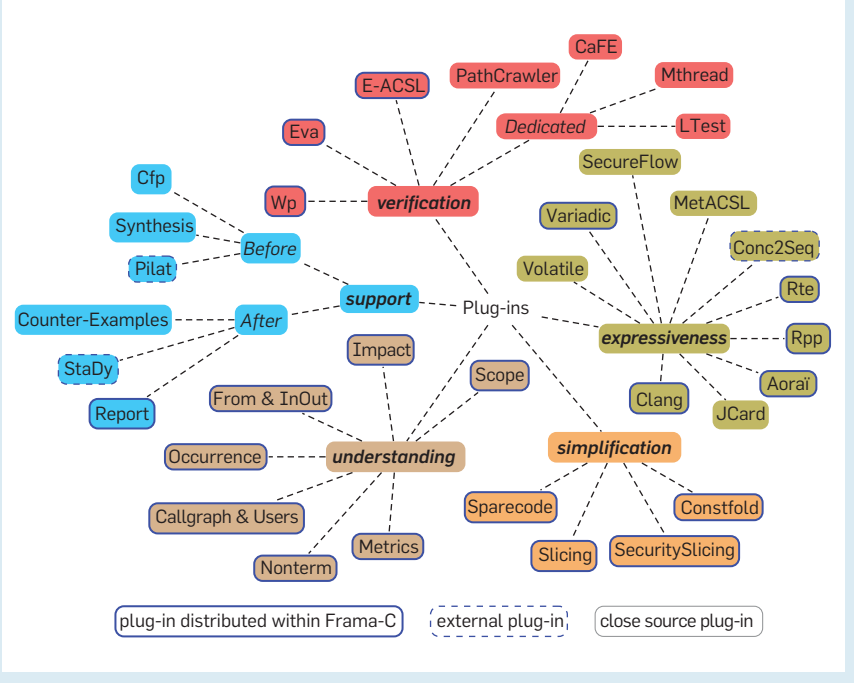
In a similar manner, the platform was meant to be easily extensible, in particular, by third-party developers. This is also reflected by the choice of the LGPL license for open-source releases of the tool, which allows the development of proprietary plug-ins as long as any change made to the core platform is contributed back.

Finally, Frama-C is meant to be usable by software engineers who 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 article 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 complexity of modern software or enhancing existing techniques with new approaches. This article illustrates efforts of both kinds.

Figure 1. Frama-C plug-in gallery.



Platform Overview

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 later). 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 provided as a plug-in. Frama-C 21-Scandium, the latest open-source release (at time of writing), contains 27 plug-ins. Frama-C offers an extensible and collaborative setting: anyone can develop and provide new plug-ins, which can collaborate with each other in different ways.

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<∅) *p = -(*p);` where `p` is of type `int*`, it has to check that reading and writing `*p` is safe, and that `-(*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 (or has not yet) been 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 to 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.

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 of solving them for all programs. However, some approaches and tools are more efficient for particular kinds of properties or programs than others. Frama-C promotes analyzer collaboration to leverage the benefits and strengths of different tools. It can be used to decompose verification work and comes in two different flavors: *sequential* and *parallel*.

Frama-C plug-ins are based on a kernel that provides key services to both end users and plug-in developers.

Sequential collaboration uses the result of one analyzer as the input to another one. It can also generate annotated C code that encodes a verification problem in such a way that another analyzer can understand it. The plug-ins allow such collaborations in the "Support" and "Expressiveness" categories of Figure 1. Several examples are provided below.

Parallel collaboration uses several analyzers to verify program properties, with each analyzer verifying a subset of properties. For instance, Eva can verify the absence of undefined behaviors, while WP can prove functional properties. 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]

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 (for example, 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.

ACSL Specification Language

For specifying C code, Frama-C offers ACSL, the ANSI/ISO C Specification Language.[a] ACSL clauses (*annotations*) are written in special comments //@... or /*@... */. While ACSL is a fairly rich

a https://github.com/acsl-language/acsl/releases/tag/1.14

Figure 2. Example of ACSL function contract.

```
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 3. Example of ACSL loop 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;
```

language, we give only a very brief description in this article. Interested readers can refer to existing tutorials[b] for an in-depth presentation.

As mentioned previously, 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<∅) *p = −(*p); we considered earlier, RTE generates the following (simplified) assertions:

```
//@ assert \valid ( p ) ;
if(*p<∅) {
//@ 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. As we will show later, lines 13–14 of Figure 4 show an assertion to prevent a division by 0.

Obviously, such properties only ensure the absence of undefined behaviors. They do not mean the program behaves as intended. 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 (that is, 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.

Figure 2 shows a possible contract for a simple function with the considered conditional statement. The requires clause expresses the precondition (lines 1–2), denoted Pre_{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 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.

Core Platform Analyses

Frama-C has four core plug-ins: Eva, which focuses on detecting undefined behaviors; WP, which aims to prove functional properties; E-ACSL, which checks properties at runtime; and PathCrawler, which generates test cases.

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 to steal data or execute malware.

Eva detects most undefined behaviors, such as invalid memory accesses,

b https://github.com/fraunhoferfokus/acsl-by-example/raw/master/ACSL-by-Example.pdf, https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf

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 (for example, 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 necessarily 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 Figure 4a, which gives the body of function main. It expects the user to type a character (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 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 assignments 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 $D_{sum}^{12} = \{0, 1, 2\}$, even if the value 1 is not possible in practice. Since $0 \in D_{sum}^{12}$, 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.

Eva can detect other runtime errors 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 uninitialized variable on line 11.

Approximations often lead to false alarms: correct code can also be flagged as a potential error. It can be seen in the example of Figure 4b, where the computed over-approximated set of values for sum on line 12 is again $D_{sum}^{12} = \{0, 1, 2\}$, 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; that is, make the analysis consider both paths separately to glean a more precise analysis. 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 $D_x^{10} = \{1\}, = \{0\}$) Eva will compute $D_{sum}^{12} = \{1\}$ 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 configures the abstractions and partitioning and uses the result of one analysis to finely tweak the next one. 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 underlying options (including trace partitioning[30]). Any $N \geq 1$ avoids the false alarm for Figure 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[c]) bring more expressiveness but slow down the analysis.

Studying the results. Eva's main output 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 (see Figure 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 them with 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 dependen-

c http://apron.cri.ensmp.fr/library/

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

```
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)

cies. 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]

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 Figure 6a (say, giving the body of function main) that calls the function of Figure 2. After line 2 of Figure 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` (lines 4–5 in Figure 2). Since other variables cannot be modified by the call of `pabs` (line 3 in Figure 2), WP also proves the second assertion of Figure 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` (lines 1–2 in Figure 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 Figure 6b, where the pointed value is `INT_MIN`. Its proof also fails for the code of Figure 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-bounds access (that is, an undefined behavior).

In the modular approach, the function contract of the callee must be proved separately. For the code of Figure 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 Figure 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 $0 \le i \le 10$, $i+j=10$, and since the execution exited the loop, $i \ge 10$. 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 arithmetic, anonymous functions, arrays, and records). They are then sent to automated theorem provers (or SMT solvers, such as 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 costs and cannot be used on complex programs. Moreover, modeling 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 precondi-

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.

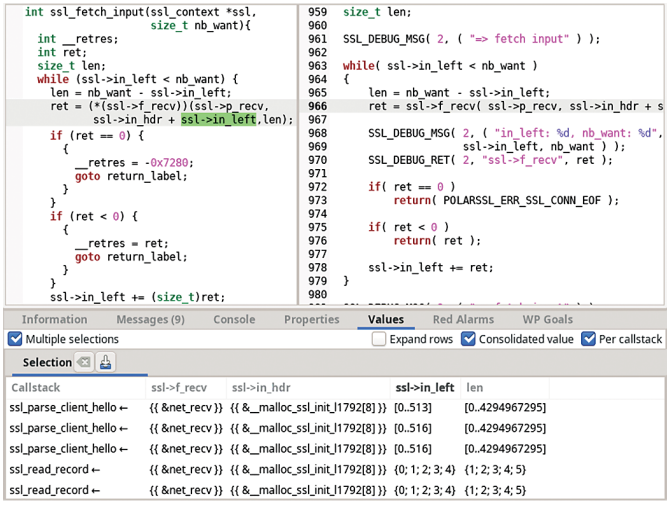


Figure 6. WP illustrated on toy examples (a), (b), and (c).

```
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;
```
(a)

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

```
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);
```
(c)

tion 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[21] 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 generated verification conditions with automated SMT solvers and proof assistants, either natively or via the Why3[22] 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 3,315 C functions were proved by WP, where only 2.3% functions required the interactive termination of some proofs.

Checking properties at runtime with E-ACSL. Runtime assertion checking is the process of verifying specifications (historically, assertions) at runtime, that is, 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, such as 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.

Specification language adjustments. As explained previously, 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.[d]

No single technique will succeed in assessing all properties a user can be interested in. Thus, the platform fosters collaborations between various techniques by any test case.

Another important issue of ACSL, with respect to runtime checking, is its logic-based semantics, which 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 such as 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.

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 the code of Figure 6a at runtime confirms the annotations (including the assertions, the precondition, and postcondition of `pabs`) are verified, while for Figure 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` $z \neq \emptyset$`;*/` is compiled to the C assertion `assert(`$z \neq \emptyset$`);`. However, in general it is not always so simple to generate code that is both sound and efficient, 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

d http://frama-c.com/download/e-acsl/e-acsl.pdf

code[e] to precisely perform the computations 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. Memory properties, such as `\valid(p)`, are an important feature of the specification language. 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 data structure.

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*, that is, 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. PathCrawler relies on the Colibri constraint solver, also developed at CEA List, to find a set of concrete values satisfying the path predicate, that is, test inputs for the path. 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 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[f] (see Figure 7) that allows the user to provide a C file (or choose one of the available examples), generate test cases, and explore the results.

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 sometimes makes it difficult to express other kinds of properties. In that case, various specialized plug-ins exist 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. The plug-in 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 to the original code.

Verify sequences of events: Aoraï and CaFE. It is often necessary to verify that a set of events during a program execution follows a particular order, for example:

A call to function `send_private_data()` must always be preceded by a call to function `authenticate()` returning ∅, 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 (that is, 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 (that is, 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. CaFE, a more recent plug-in, 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]

Verify relational properties: RPP. Contrary to an ACSL contract, which 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: 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 Figure 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 pub-

e Based on GNU Multiple Precision Arithmetic Library: https://gmplib.org/

f http://pathcrawler-online.com

lic 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.

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 performed from a location with a higher level than that of the current user, or that 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.

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

Specify test objectives: LAnnotate. Test objectives offer another example of specific annotations that can be added for analysis using Frama-C core plug-ins. Various structural test coverage criteria (for example, 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 that 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 are examined next.

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, specification transformation, or even a dedicated reasoning. 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 a few examples here.

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.

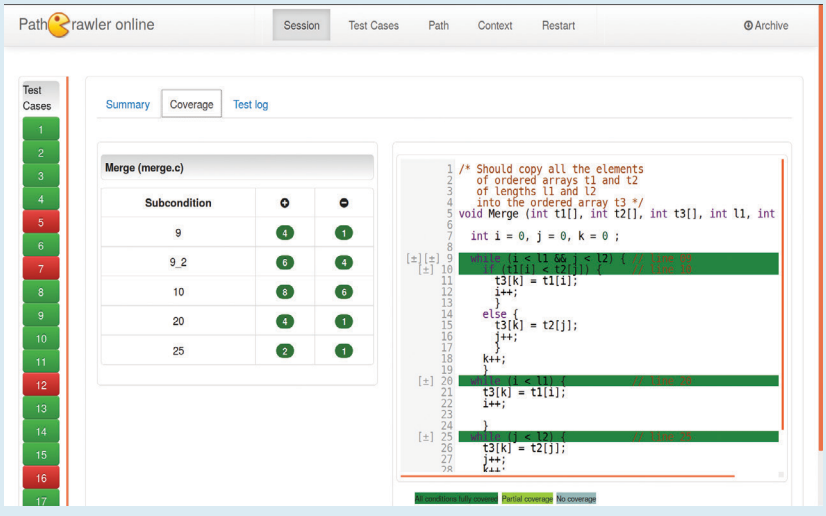


Figure 8. (a) A C code, and (b) RPP transformation for leak.

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

(a)

```
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 */ }
```

(b)

g http://www.southampton.ac.uk/~gp1y10/cseq/cseq.html

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 failures can be due to an error in the code or in the specification itself, a missing or weak specification for a called function or a loop, lack of time, or the 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, and so on) into their counterparts supported by test generation is not straightforward. 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 use 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 counterexamples 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.

Infeasible test objectives: LUncov. Previously, we illustrated how generic test objectives—or labels—allow PathCrawler to support test-case generation for various test coverage criteria. An important issue in testing relates to infeasible (that is, 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]. 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.

Frama-C is intensively used for teaching. Indeed, it became difficult to keep track of all universities where the toolset is used in various program analysis or verification courses.

Program simplification: Slicing. *Slicing* is a program transformation technique that takes as input a program and a so-called *slicing criterion* (for example, 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.

Information flow: SecureFlow. Information flow properties denote properties of the dependencies between the outputs and the inputs of the program. The most common example, presented above, is non-interference. 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.

More Than a Toolset: An Ecosystem

Frama-C community. Since its initial release in May 2008, Frama-C has built an active community of users and plug-in developers. Its open source license (LGPL 2.1) played an important role in this development, facilitating its integration into many Linux distributions (the oldest package, from Debian, dates 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 with Frama-C. As of July 2020, opam reports around 200 monthly downloads (via opam) of the latest release, Frama-C 21.1 Scandium, which was released in June 2020.

Naturally, these public releases are accompanied with various communication channels,[h] including a mailing list, a bug tracker, and a dedicated StackOverflow tag. Frama-C's blog[i] 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, most of which are supported by the French government and the European Union. Apart from CEA itself, these projects usually gather a mix of academic and industrial partners to explore new research directions while keeping sure that they are relevant to real-world problems. Among these projects, we can mention the French RNTL project CAT and its successor U3CAT,[j] funded by ANR, both of which have been fundamental for building the grounding blocks of the platform. Later on, European projects Stance and Vessedia help broaden Frama-C's target properties to cybersecurity.

Teaching with Frama-C. Frama-C is intensively used for teaching. Indeed, 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. Just a few examples include Ecole Polytechnique, CentraleSupélec, École Normale Supérieure, ENSIIE, almost all universities in and around Paris, as well as in Besançon, Bordeaux, Bourges, Grenoble, Lille, Lyon, Orléans, Rennes, Toulouse, and many others. Frama-C is also increasingly used in other countries, including Austria, Brazil, China, Germany, Portugal, Russia, U.K., and the U.S. Among the analyzers of the open source distribution, Eva, WP, and E-ACSL are the most popular plug-ins for teaching. 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 training in industrial companies and for tutorials on program verification at premier international conferences, such as ASE, FM, iFM, ISSRE, POPL, SAC, TAP, and QSIC.

Collaborations and industrial applications. Long-time partnerships began with Frama-C's precursor, Caveat, which was developed in the 1990s in close collaboration with the teams at Airbus and leveraged automated reasoning capabilities from Inria's Alt-Ergo solver. As Caveat went into industrial production, the development around Frama-C continued to nourish these collaborations and engaged them in assisting with 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, 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 in the 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 39kLoC nuclear power plant shutdown system, allowing the demonstration of the absence of intrinsic runtime 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 used 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.[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 eight hours. MERCE estimated that automatic test generation with PathCrawler could bring an effective benefit factor of more than 230 for test input generation in the company. Those very good results are 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]

Similarly, in the context of European projects Stance[k] (FP7) and Vessedia[l]

h https://frama-c.com/support.html
i https://blog.frama-c.com
j https://frama-c.com/u3cat.html
k https://cordis.europa.eu/project/rcn/105816/brief/en
l https://www.vessedia.eu/

(H2020), Search Lab developed Frama-C plug-ins dedicated to generating counter-examples in the spirit of StaDy but based on external test-case generators, namely Search Lab's own tool Flinder[m] and later the AFL fuzzer.[n] In these projects, Dassault Aviation also designed a methodology based on Eva, E-ACSL, and two homemade plug-ins to detect security vulnerabilities and deploy runtime countermeasures when necessary.[34] It has been experimented on a few modules of Apache.

Conclusion

Since its first public release more than 13 years ago, the Frama-C framework has demonstrated its ability 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 holds true to this day, with lines of research toward 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[o] and will find it useful for their verification activities. C

m https://www.flinder.hu

n http://lcamtuf.coredump.cx/afl/

o see http://www.frama-c.com/download.html

References

1. Alberti, M. and Signoles, J. Context generation from formal specifications for C analysis tools. In *Proc. of the 2017 Conf. on Logic-based Program Synthesis and Transformation*.
2. Alur, R., Etessami, K., and Madhusudan, P. A temporal logic of nested calls and returns. In *Proc. of the 2004 Conf. on Tools and Algorithms for the Construction and Analysis of Systems*.
3. Barany, G. and Signoles, J. Hybrid information flow analysis for real-world C code. In *Proc. of the 2017 Conf. on Tests and Proofs*.
4. Bardin, S., Chebaro, O., Delahaye, M., and Kosmatov, N. An All-in-One Toolkit for Automated White-Box Testing. In *Proc. of the 2014 Conf. on Tests and Proofs*.
5. Bardin, S., Kosmatov, N., Marre, B., Mentré, D., and Williams, N. Test case generation with PathCrawler/LTest: How to automate an industrial testing process. In *Proc. of the 2018 Conf. on Leveraging Applications of Formal Methods, Verification and Validation*.
6. Barthe, G., D'Argenio, P., and Rezk, T. Secure information flow by self-composition. *Mathematical Structures in Computer Science 6* (2011).
7. Blanchard, A., Kosmatov, N., Lemerre, M., and Loulergue, F. Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs. In *Proc. of the 2016 Conf. on Source Code Analysis and Manipulation*.
8. Blatter, L., Kosmatov, N., Gall, P., and Prevosto, V. RPP: Automatic proof of relational properties by self-composition. In *Proc. of the 2017 Conf. on Tools and Algorithms for the Construction and Analysis of Systems*.
9. Brahmi, A., Carolus, M., Delmas, D., Essoussi, M., Lacabanne, P., Lamiel, V., Randimbivololona, F., and Souyris, J. Industrial use of a safe and efficient formal method based software engineering process in avionics. In *Proc. of the 2020 Conf. on Embedded Real Time Softw. and Systems*.
10. Ceara, D., Mounier, L., and Potet, M. Taint dependency Ssquences: A characterization of insecure execution paths based on input-sensitive cause sequences. In *Proc. of the 2010 Int. Conf. on Softw. Testing, Verification and Validation*.
11. Chalin, P. A sound assertion semantics for the dependable systems evolution verifying compiler. In *Proc. of the 2007 Int. Conf. on Softw. Engineering*.
12. Clarke, E., Emerson, E., and Sistla, A. Automatic verification of finite-state concurrent systems using temporal logic specifications. *Trans. Programming Languages and Systems* (1986).
13. Correnson, L. Computing what remains to be proved. *In Proc. of the 2014 Conf. on NASA Formal Methods*.
14. Correnson, L. and Signoles, J. Combining analyses for C program verification. *Int. Workshop. on Formal Methods for Industrial Critical Systems (2012)*..
15. Cuoq, P., Delmas, D., Duprat, S., and Lamiel, V. Fan-C, a Frama-C plug-in for data flow verification. In *Proc. of the 2012 Conf. on Embedded Real Time Softw. and Systems*.
16. de Oliveira, S., Bensalem, S., and Prevosto, V. Polynomial invariants by linear algebra. In *Proc. of the 2016 Conf. on Automated Technology for Verification and Analysis*.
17. Dragoi, C., Enea, C., and Sighireanu, M. Local shape analysis for overlaid data structures. In *Proc. of the 2013 Int. Symp. on Static Analysis*.
18. e Silva, R., Arai, N., Burgareli, L., de Oliveira, J., and Pinto, J. Formal verification with Frama-C: A case study in the space software domain. *Trans. Reliability* (2016).
19. Ebalard, A., Mouy, P., and Benadjila, R. Journey to a RTEfree X.509 parser. In *Proc. of the 2019 Symp. sur la Sécurité des Technologies de l'information et des Communications*.
20. Fava, D., Signoles, J., Lemerre, M., Schäf, M., and Tiwari, A. Gamifying program analysis. In *Proc. of the 2015 Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*.
21. Filliâtre, J. and Marché, C. Multi-prover verification of C programs. In *Proc. of the 2004 Int. Conf. on Formal Methods and Softw. Engineering*.
22. Filliâtre, J. and Paskevich, A. Why3—Where programs meet provers. In *Proc. of the 2013 European Symp. on Programming*.
23. Fouilhé, A., Monniaux, D., and Périn, M. Efficient generation of correctness certificates for the abstract domain of polyhedra. In *Proc. of the 2013 Int. Symp. on Static Analysis*.
24. Goodloe, A., Muñoz, C., Kirchner, F., and Correnson, L. Verification of numerical programs: From real numbers to floating point numbers. In *Proc. of the 2013 Conf. on NASA Formal Methods*.
25. Jacquemin, M., Putot, S., and Védrine, F. A reduced product of absolute and relative error bounds for floating-point analysis. In *Proc. of 2018 Int. Symp. on Static Analysis*.
26. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-C: A software analysis perspective. *Formal Asp. Comput.* (2015).
27. Kirchner, F., Sadmi, F., Flanc, S., Duboc, L., Marteau, H., Prevosto, V., and Vedrine, F. Safer marine and offshore software with formal-verification-based guidelines. In *Proc. of the 2016 Conf. on Embedded Real Time Softw. and Systems*.
28. Leino, K. Efficient weakest preconditions. *Information Processing Letters* (2005).
29. Mandrykin, M. and Khoroshilov, A. High-level memory model with low-level pointer cast support for Jessie intermediate language. *Programming and Computer Softw.* (2015).
30. Mauborgne, L. and Rival, X. Trace partitioning in abstract interpretation based static analyzers. In *Proc. of the 2005 European Symp. on Programming*.
31. Meyer, B. *Design by Contract*. Prentice Hall, 1991.
32. Monate, B. and Signoles, J. Slicing for security of code. In *Proc. of the 2008 Conf. on Trusted Computing and Trust in Information Technologies*.
33. Ourghanlian, A. Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nucl. Eng. Technol.* (2015).
34. Pariente, D. and Signoles, J. Static analysis and runtime assertion checking: Contribution to security countermeasures. In *Proc. of the 2017 Symp. sur la Sécurité des Technologies de l'Information et des Communications*.
35. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., and Julliand, J. How testing helps to diagnose proof failures. *Formal Asp. Comput.* (2018).
36. Prevosto, V., Burghardt, J., Gerlach, J., Hartig, K., Pohl, H., and Völlinger, K. Formal specification and automated verification of railway software with Frama-C. In *Proc. of the 2013 Int. Conf. on Industrial Informatics*.
37. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., and Gall, P. MetAcsl: Specification and verification of high-level properties. In *Proc. of the Conf. on Tools and Algorithms for the Construction and Analysis of Systems*.
38. Signoles, J., Kosmatov, N., and Vorobyov, K. E-ACSL, a runtime verification tool for safety and security of C programs. Tool Paper. In *Proc. of the 2017 Int. Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardization for Runtime Verification Tools*.
39. Venet, A. The Gauge domain: Scalable analysis of linear inequality invariants. In *Proc. of the 2012 Conf. on Computer Aided Verification*.
40. Williams, N., Marre, B., Mouy, P., and Roger, M. PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Proc. of the 2005 European Dependable Computing Conf.*

Supplemental material for this article is available online at https://dl.acm.org/doi/10.1145/3470569

Patrick Baudin is a researcher at the Université Paris-Saclay, CEA, List, Palaiseau, France.

François Bobot is a researcher at the Université Paris-Saclay, CEA, List, Palaiseau, France.

David Bühler is a researcher at the Université Paris-Saclay, CEA, List, Palaiseau, France.

Loïc Correnson is a researcher at the Université Paris-Saclay, CEA, List, Palaiseau, France.

Florent Kirchner is head of the department at the Université Paris-Saclay, CEA, List, Palaiseau, France.

Nikolai Kosmatov is a researcher at Thales Research and Technology, Palaiseau, France.

André Maroneze is a researcher at the Université Paris-Saclay, CEA, List, Palaiseau, France.

Valentin Perrelle is a researcher at the Université Paris-Saclay, CEA, List, Palaiseau, France.

Virgile Prevosto is a researcher at the Université Paris-Saclay, CEA, List, Palaiseau, France.

Julien Signoles is a researcher at the Université Paris-Saclay, CEA, List, Palaiseau, France.

Nicky Williams is a researcher at the Université Paris-Saclay, CEA, List, Palaiseau, France.

Watch the authors discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/frama-c