Frama-C, a Collaborative Framework for C Code Verification. Tutorial Synopsis*

Nikolai Kosmatov and Julien Signoles

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

Abstract. Frama-C is a source code analysis platform that aims at conducting verification of industrial-size C programs. It provides its users with a collection of plug-ins that perform static and dynamic analysis for safety- and security-critical software. Collaborative verification across cooperating plug-ins is enabled by their integration on top of a shared kernel, and their compliance to a common specification language, ACSL.
This paper presents a three-hour tutorial on Frama-C in which we provide a comprehensive overview of its most important plug-ins: the abstract-interpretation based plug-in Value, the deductive verification tool WP, the runtime verification tool E-ACSL and the test generation tool PathCrawler. We also emphasize different possible collaborations between these plug-ins and a few others. The presentation is illustrated on concrete examples of C programs.

Keywords: Frama-C, ACSL, abstract interpretation, deductive verification, runtime verification, test generation, combinations of analyses.

1 Introduction

The last few decades have seen much of the groundwork of formal software analysis being laid. Several angles and theoretical avenues have been explored, from deductive verification to abstract interpretation to program transformation to monitoring to concolic testing. While much remains to be done from an academic standpoint, these techniques have become mature enough to have been successfully implemented and used in industrial settings [1].

However, although verification of C programs is of paramount importance because the C programming language is still the language of choice for developing safety-critical systems and is also routinely used for security-based applications, verifying large C programs remains a time-consuming and challenging task. One of the reasons is related to the C programming language itself since it combines high level features like arrays and low level features like user-controlled memory allocations, bitfields and unions. Another reason comes from weaknesses of each verification technique: dynamic techniques are not bothered by C code complexity but are not exhaustive, abstract interpretation is exhaustive and almost automatic but may be imprecise and cannot verify

* This work has received funding for the S3P project from French DGE and BPIFrance.

complex functional properties, while deductive methods may tackle a broad varieties of properties but require formal specifications and may be less efficient in presence of low level code. One effective way to circumvent this problem is to combine several analyses in order to reduce weaknesses of each one thanks to the others. For instance, abstract interpretation can ensure the absence of most runtime errors, deductive verification can prove most functional properties, while monitoring can check at runtime the remaining properties. Such analysis combinations is the *raison d'être* of FRAMA-C.

The FRAMA-C software analysis platform [2] provides a collection of scalable, interoperable, and sound software analyses for the industrial analysis of ISO C99 source code. The platform is based on a kernel which hosts analyzers as collaborating plug-ins and uses the ACSL formal specification language [3] as a lingua franca. FRAMA-C includes plug-ins based on abstract interpretation, deductive verification, monitoring and test case generation, as well as a series of derived plug-ins which build elaborate analyses upon the basic ones. This large variety of analysis techniques and its unique collaboration capabilities make FRAMA-C most suitable for developing new code analyzers and applying code analysis techniques in many academic and industrial projects.

This article is a companion paper of a 3-hour tutorial which brings participants to a journey into the FRAMA-C world along its main plug-ins. It aims at providing the essence of each technique and tool along with a few illustrating examples. While several tutorials about some parts of FRAMA-C have already been presented in previous conferences [4–8], none of them have already presented all these techniques altogether. Here, after a general overview of FRAMA-C (Section 2), we present deductive verification tool WP [2, 9] (Section 3), abstract interpretation based plug-in VALUE [2, 10] and its recent redesign EVA (Section 4), the runtime verification tool E-ACSL [11, 12] (Section 5) and the test generation tool PathCrawler [13, 14] (Section 6). A last section is dedicated to some of their possible collaborations (Section 7).

2 Overview of FRAMA-C

FRAMA-C is a platform which aims at analyzing source code written in ISO C99. This code may be annotated with formal specifications written in the ACSL formal specification language [3] (presented in Section 3). Recently FRAMA-CLANG has been released as a prototype FRAMA-C extension to handle C++ code. The platform is written in OCAML [15] and based on a plug-in architecture [16]: each analyzer is a plug-in which is linked against the FRAMA-C kernel.

The kernel provides a normalized representation of C programs and ACSL specifications. In addition, the kernel provides several general services for supporting plug-in development and providing convenient features to FRAMA-C's end-users. For instance, messages, source code and annotations are uniformly displayed, whereas parameters and command line options are homogeneously handled. The kernel also allows plug-ins to collaborate with each other either sequentially or in parallel. Sequential collaboration consists in a chain of analyses that perform operations one after another, while parallel collaboration combines partial analysis results from several analyzers to complete a full program verification. Examples of collaborations will be provided in Section 7. In particular, the kernel consolidates analysis results to provide the users with a synthesis

of what is proven and ensure logical consistency when verifying dependent properties: the proof of a property P by analyzer A may depend on the validity of another property Q whose proof is done by analyzer B [17].

The FRAMA-C kernel is open source (under LGPL v2.1), as well as many of its plug-ins. Several plug-ins are presented in Figure 1. Many important plug-ins are ded-

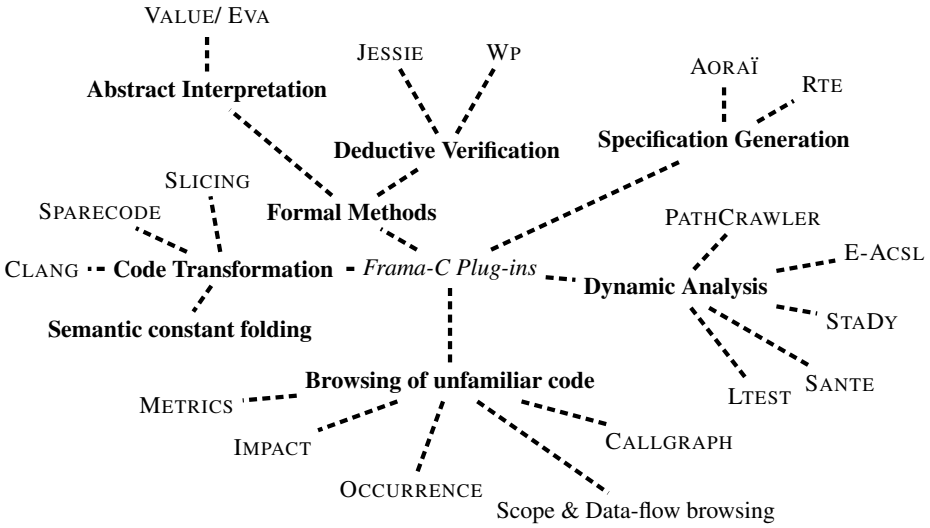

Fig. 1: FRAMA-C plug-in gallery

icated to program verification. First, FRAMA-C comes with a powerful abstract interpretation framework based on VALUE, which aims at computing over-approximations of possible values of program variables at each program point. VALUE is presented in Section 4. Next, FRAMA-C provides two alternative plug-ins for deductive verification: JESSIE (which is now deprecated) and WP. These plug-ins aim at verifying that a given C code satisfies its specification expressed as ACSL annotations. ACSL language and WP plug-in are presented in Section 3. Finally, FRAMA-C provides dynamic verification through the E-ACSL plug-in which aims at verifying annotations at runtime. This plug-in is presented in Section 5. Another dynamic tool, PATHCRAWLER[1] is dedicated to test case generation and is presented in Section 6. Plug-ins SANTE, STADY and LTEST implement different collaborations between static and dynamic analyses. They are introduced in Section 7.

Other plug-ins are not directly program verifiers. Some of them aim at helping the users to better understand a source code they are not familiar with: plug-in METRICS computes some useful metrics about the code, plug-in OCCURRENCES displays in the

[1] unlike many other FRAMA-C analyzers, PATHCRAWLER is currently not open source but is available through the online test generation service http://pathcrawler-online.com.

FRAMA-C GUI all occurrences of a particular left-value (taking into account aliasing), while a few other plug-ins compute scope and dataflow dependency information. Plug-in CALLGRAPH computes the callgraph, taking into consideration function pointers as soon as VALUE has been executed.

Some plug-ins perform program transformations. Plug-in SEMANTIC CONSTANT FOLDING replaces constant variables by their numerical values and propagates them along the dataflow by taking into account aliasing. Plug-in SLICING simplifies the code by removing code fragments that are irrelevant with respect to a given program property (e.g. preserve the effects of a particular statement). Plug-in SPARECODE can be seen as a particular case of SLICING which removes dead code. Plug-in IMPACT computes the values and statements impacted (directly or transitively) by the side effects of a given statement. It is the forward counterpart of the usual (backward) slicing, but it does not necessarily generate a new program: by default it just highlights the impacted statement.

FRAMA-C also allows analyzers to generate new ACSL annotations which encode specific properties. Plug-in AORAÏ takes as input a Büchi automaton or an LTL formula and generates ACSL annotations which encode the corresponding temporal property that can be verified by other means. In the same spirit, plug-in RTE generates an ACSL annotation for every possible undefined behavior of the source code. For instance, it generates a guard `y` $\neq$ `0` before a division by `y` in the source code.

3 Specification and Deductive Verification with FRAMA-C / WP

3.1 Specification of C Programs with ACSL

ACSL (ANSI/ISO C Specification Language) [3] is a formal behavioral specification language offered by FRAMA-C and shared by different FRAMA-C analyzers. It allows its users to specify functional properties of C programs similarly to Eiffel [18] and JML [19]. It is based on the notion of function contract. The *contract* of a function f specifies the preconditions that are supposed to be true before a call of f (i.e. ensured by the caller), and the postconditions that should be satisfied after the call of f (and should be thus established during the verification of f). The preconditions are specified in **`requires`** clauses, while the postconditions are stated in **`ensures`** clauses. An additional type of postconditions, specified in an **`assigns`** clause in ACSL and used for the so-called *frame rule*, states a list of locations of the global memory state that may have a different value before and after the call. When the contract of f contains such a clause, all locations that are not mentioned in it must have the same value before and after the call of f. Function contracts can be also represented in the form of different behaviors.

Predicates used in annotations are written in typed first-order logic. Variables have either a C type or a logical type (e.g. `integer` or `real` for mathematical integer or real numbers). The user can define custom functions and predicates and use them in annotations together with ACSL built-ins. Indeed, ACSL features its own functions and predicates to describe memory states. In particular, regarding memory-related properties, **`\valid`**`(p)` expresses validity of a pointer `p` (i.e. being a non-null pointer which can be safely accessed by the program); **`\base_addr`**`(p)`, **`\block_length`**`(p)`,

```
1 /*@ requires n ≥ 0 && \valid(t+(0..n-1));
2     assigns \nothing;
3     ensures \result ≠ 0 ⟺
4       (\forall integer j; 0 ≤ j < n ⟹ t[j] == 0);
5 */
6 int all_zeros(int *t, int n) {
7   int k=0;
8   /*@ loop invariant 0 ≤ k ≤ n;
9       loop invariant \forall integer j; 0 ≤ j < k ⟹ t[j] == 0;
10      loop assigns k;
11      loop variant n-k;
12  */
13  while(k < n){
14    if (t[k] ≠ 0)
15      return 0;
16    k++;
17  }
18  return 1;
19 }
```

Fig. 2: Function `all_zeros` specified in ACSL (file `all_zeros.c`).

and **`\offset`**`(p)` express respectively the base address, the size of the memory block containing `p` and the offset of `p` inside it (in bytes), while **`\initialized`**`(p)` is true whenever the pointed location `*p` has been initialized. We refer the reader to [3] for detailed documentation of all ACSL features.

Example of Specifications Figure 2 illustrates a C function `all_zeros` specified in ACSL. This function receives as arguments an array `t` and its size `n` and checks whether all elements of the array are zeros. If yes, it returns a nonzero value, and 0 otherwise. The function contract contains a precondition (line 1) and postconditions (lines 2–4). The precondition states that the input array contains `n` valid memory locations at indices `0..(n-1)` that can be safely read or written, and that the size `n` is non negative. This property must be ensured by the caller and should be thus specified in the precondition. The **`assigns`** clause at line 2 states that the function is not allowed to modify any non-local variable. Without this clause, an erroneous implementation writing zeros in all elements of the array and returning 1 would be considered correct with respect to the contract. Finally, the clause at lines 3–4 states that the result is nonzero if and only if all elements of the array are equal to zero. The loop contract at lines 8–12 will be discussed in the next section.

Figure 3 provides another example of a specified function. This function is only declared and takes as arguments an array `t` of size `n` and some element `elt`. It must return an index `i` such than `t[i] = elt`, or -1 if there is no such index. The precondition (line 1) and the **`assigns`** clause (line 2) are similar to the ones of the function `all_zeros`. The postcondition is expressed through two named behaviors. They correspond to the two different cases of the contract. First, the behavior `present` states

```
1 /*@ requires len ≥ 0 && \valid(t+(0..len-1));
2     assigns \nothing;
3     behavior present:
4       assumes \exists integer i; 0 ≤ i < len && t[i] == elt;
5       ensures 0 ≤ \result < len && t[\result] == elt;
6     behavior absent:
7       assumes \forall integer i; 0 ≤ i < len ⟹ t[i] ≠ elt;
8       ensures \result == -1;
9     disjoint behaviors;
10    complete behaviors;
11  */
12 extern int find_value(int *t, int len, int elt);
```

Fig. 3: Function `find_value` specified in ACSL.

that, if the searched element `elt` is present in the array (line 4), the function's result is an index with the expected property (line 5). The behavior `absent` corresponds to the opposite case (line 7). In that case, the function returns -1 (line 8). Additionally the **`disjoint behaviors`** clause states that these behaviors are mutually exclusive (line 9), while the **`complete behaviors`** clause indicates that their cover all the possible cases of the function (line 10). In other words, being both disjoint and complete guarantees that one and only one behavior applies at each function call.

3.2 Deductive Verification with FRAMA-C / WP

Among other formal software verification techniques, deductive program verification consists in establishing a rigorous mathematical proof that a given program respects its specification. When no confusion is possible, one also says for short that deductive verification consists in "proving a program". The weakest precondition calculus proposed by Dijkstra [20] reduces any deductive verification problem to establishing the validity of first-order formulas called *verification conditions*. The WP plug-in [2, 9] of FRAMA-C performs weakest precondition calculus for deductive verification of C programs. Various automatic SMT solvers, such as Alt-Ergo, CVC4 and Z3, can be used to prove the verification conditions generated by WP.

Example of Proof Let us illustrate deductive verification with WP on the example of Figure 2. The command `frama-c-gui -wp all_zeros.c` runs the proof with WP on this example and shows the results in the FRAMA-C GUI. Suppose first that the user has specified the contract at lines 1–4 without writing the loop contract at lines 8–12. In this case, the proof of the postcondition will not be successful. Indeed, in presence of loops, since the number of loop iterations is unknown, the deductive verification tool requires a *loop invariant*, i.e. an additional property on the program state that is true before the loop and after each complete loop iteration. It can be specified in a loop contract using **`loop invariant`** and **`loop assigns`** clauses. The clause at line 8 specifies the interval of values of the loop variable `k`. The clause at line 9 specifies

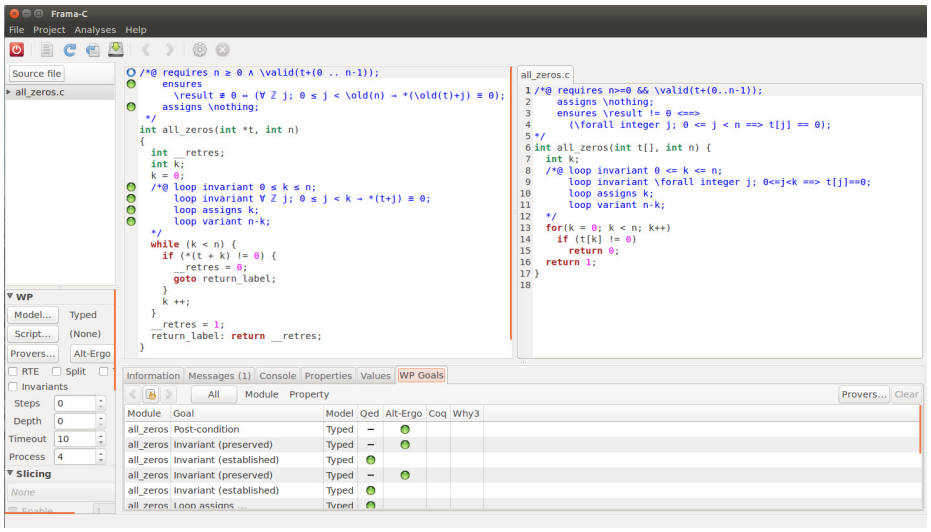

Fig. 4: Successful proof for the program of Figure 2 with FRAMA-C/WP.

that all elements at indices `0..(k-1)` are equal to 0 (that is indeed true after any complete loop iteration otherwise the loop execution was interrupted at line 15). Similarly to **`assigns`**, the **`loop assigns`** clause specifies the variables (but both global and local ones in this case) that may change their value during the loop. The loop contract can also contain a **`loop variant`**, which defines a decreasing natural measure corresponding to an upper bound of the number of remaining loop iterations and is used to prove that the loop terminates. In this example, `n-k` provides such a bound (cf. line 11).

On the complete program of Figure 2 with the loop contract, WP successfully proves that this function respects its specification. In addition, it is possible to make WP check the absence of runtime errors using the option `-wp-rte`. In this case, thanks to the array validity assumed at line 1 and the interval of values specified at line 8, WP successfully proves that array access at line 14 is valid and the arithmetic operation at line 16 does not overflow.

WP*'s models* Deductive methods rely on *models*. For C programs, arithmetic models provide abstract representations of machine integers and/or floats, while memory models are abstractions of the program memory. These models are a trade-off between simplicity (making proof more automatic) and expressivity (being able to deal with more properties or programs at the price of making proof harder). WP's internal engine is generic, tries to simplify verification conditions and does not depend on a particular model [21]. WP actually comes with several different arithmetic and memory models. For arithmetics, the users can choose between mathematical integers or machine integers and between real numbers or floats. Machine integers make proofs easier, but the user must ensure the absence of integer overflows by other means (usually by using the

`-wp-rte` option). Using reals converts float operations to real ones without rounding (that is unsafe with respect to norms, but tractable), while the float model introduces correct rounding but the proofs are rarely automatic and often require the use of a proof assistant (like Coq or PVS).

The WP's *Hoare* memory model, directly inspired by the historic definition of weakest precondition calculus, is very simple. However, it assumes a program with no pointer to be sound. A common programming C pattern is nevertheless to use pointers for function arguments passed by reference. In such cases, their adresses are never taken and so they are not aliased if they were not aliased when calling the function. Thus, it remains safe to use the a *Hoare*-like model: that is the purpose of the *Reference Parameters* model (shortly *Ref*). The last provided model is the *Typed* model which allows powerful reasoning on heap data. The special mode *Typed+Ref* uses the *Typed* model for expressiveness but is automatically able to detect when using the simpler *Ref* model is safe (making the proof more automatic).

For additional detail on specification with ACSL and deductive verification with WP, the reader may refer to articles [2, 21], dedicated tutorials [6, 22] and the WP manual [9].

4 Value Analysis with FRAMA-C / VALUE and EVA

The Value Analysis plug-in of FRAMA-C (VALUE for short) [10] automatically computes sets of possibles values for the variables of an analyzed program at each program point, by means of abstract interpretation [23]. It also warns about potential runtime errors. These objectives and means are shared with commercial tools like Polyspace [24] or Astrée [25]. However, VALUE has also distinct goals. First, it is *not* application directed: it aims at being directly usable on any C code in any applicative domain, from low level system libraries to safety-critical applications.[2] One consequence is that VALUE relies on an efficient generic domain which can nevertheless be less precise than specific domains designed for specific code like digital filters [26]. This drawback is circumvented by the FRAMA-C ecosystem: what cannot be proven by VALUE may still be proven by another plug-in, possibly a dedicated one.

Another originality of VALUE comes from its presence in the FRAMA-C ecosystem: one would like to reuse what it has computed in other plug-ins. Consequently, VALUE keeps the computed possible values of each variable of the program at each program point. This information is available in the FRAMA-C GUI and helps the user to better understand VALUE's results. An illustrative example for PolarSSL 1.1.7[3] is presented in Figure 5. It also allows derived analyses like slicing to be sound. In particular it helps them to safely interpret function pointers and to find out potential aliasing. In this way, several plug-ins have been developed by academic and industrial users for specific goals in a safe way without spending too much time with pointer intricacies [27–30].

[2] This goal is not yet reached but progress is regularly made in that direction and it is still an objective (which is not shared by other widely used tools, as far as we know).

[3] See https://tls.mbed.org/.

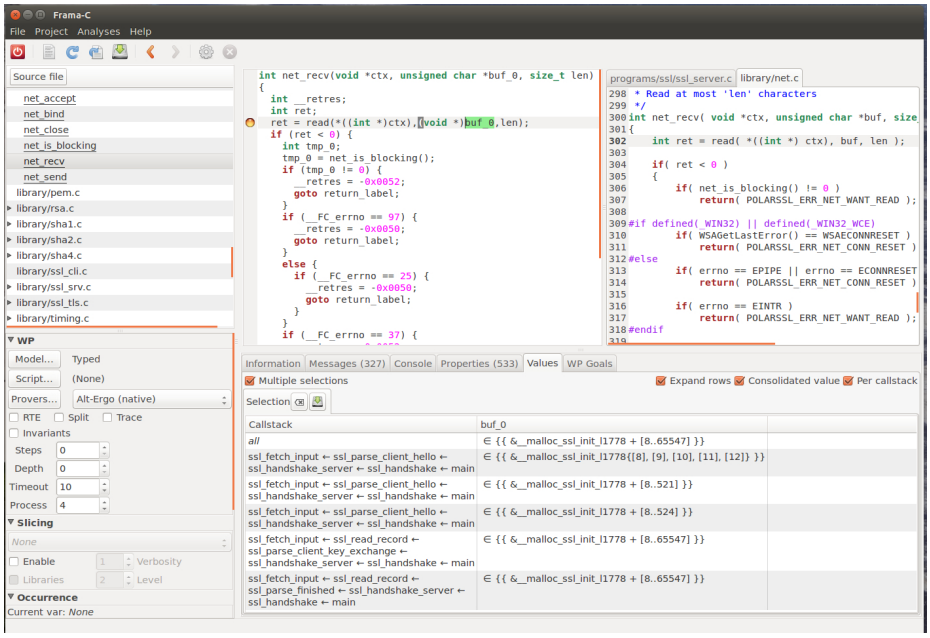

Fig. 5: FRAMA-C GUI with VALUE's results on PolarSSL's function `net_recv`. It displays the possible values of the function's parameter `buf` *per* callstack.

Abstract domains VALUE has hard-coded domains which can not be changed easily. They have been chosen for their good compromise between precision and efficiency and rely on heavily optimized datastructures and algorithms [31].

Integers are represented either by an exact set of possible values (when such a set remains small), or by intervals with congruence information (when the set of possible values becomes large). For example, an **`int`** variable could have values in the domain `[1..41], 1%2`, which means any positive odd integer smaller or equal to 41. Congruence is of particular interest to express offset properties like "the pointer `p` is a 32-bit aligned offset from `&t[0]`". Note that this domain is *not* relational: it does not keep any relation between program variables. For instance, if $x = y$, VALUE only knows that both variables have the same possible set of values (say, the interval $[a; b]$) . It does not know that x and y have the same value in $[a; b]$.

Floating points are represented by IEEE 754 double-precision finite intervals. Rounding is performed when necessary (e.g. from simple-precision floats to double-precision). Infinities and NaN are considered as undesirable errors and reported as such.

Pointers are seen as a pair of a base address and an offset. This way, it is possible to verify the absence of buffer overflow by checking that the offset is positive and smaller than the size of the base. Consequently, VALUE's abstract representation of a pointer is a set of possible base addresses associated with possible offsets (in bytes). A base address can be either the address of a local or global variable, the address of a function formal parameter, the address of a literal string constant, or the special `NULL` base which is used to encode absolute addresses (denoted by their offsets). For instance, let `P` be a global pointer and `t` be a local array of 16-bit integers. Then a pointer `Q` could have values in the set `{ NULL; &P[0..24],0%8; &t[4..10],0%2 }`. It means that pointer `Q` is either null, or equal to pointer `P` with an offset (in bytes) divisible by 8 between 0 and 24, or refers to one of the cells `t[2],...,t[5]`.

In addition to one of the above-mentioned abstract values, VALUE associates to each memory location a flag which indicates if its contents may contain an indeterminate value like uninitialized local variables (ISO C99 standard [32], Section 6.2.4). Having a memory location containing such a value is not an error *per se* but accessing it is. The abstract memory representation maps each base address to a size and a chunk of memory cells. Each chunk itself maps a consecutive range of bits to abstract values. This representation is untyped and so can precisely interpret unions, bitfields and heterogeneous pointer conversions.

Parameterization Abstract interpreters are automatic tools. However they rarely give useful results when running from scratch on a new large C program, because the analysis quickly diverges due to approximations. These tools always need parameterization to get more precise tractable results while consuming a decent amount of resources (computation time and computer memory). VALUE has a large variety of parameters [10], two of which are presented below.

The most important one is named `slevel`. It is an instance of trace partitioning [33] and, in particular, allows the user to unroll loops up to a certain limit. It may be set per loop, per function or for the whole program. Instead of having a single analysis state that approximates all the values of all possible executions, it allows the analysis to keep up to n separated states in parallel, which improves precision.

Another important way of parameterization is case splitting through ACSL trivial disjunctions (and `slevel`). Consider the following simple example.

```
1 /*@ ensures \result ≥ 0; */
2 int f(char x) {
3   if (x == 0) return 0;
4   else return x * x;
5 }
```

Without case splitting, VALUE is not able to prove the function's postcondition. Indeed, since VALUE is not relational and x may take any value from -128 to 127, it can only conclude that the returned value at line 4 belongs to the interval $[-16256..16384]$. However, the user may introduce the trivial assertion `/*@ assert x <0 || x ==0 || x >0; */` before line 3 and set `slevel` to 3. This way, VALUE will split the 3 cases of the disjunctive predicate and keep them separated. In each case, it is able to verify the postcondition[4]. Of course, it is also able to prove the newly introduced trivial assertion. Interestingly, case splitting and aggressive trace partitioning often compensate for the absence of relational domains. It is true in this simple example, and it remains true in much larger applications.

Eva: Evolved Value Analysis Since the very last open source release of FRAMA-C (namely, FRAMA-C Aluminium), an Evolved version of VALUE, named EVA, is available. It aims at reconciling the variety of target programs of VALUE with application-specific abstract domains, which allows to improve precision and/or efficiency in particular cases. Consequently, EVA transforms VALUE from a monolithic analyzer with hard-coded domains to a generic extendable analysis parameterized by cooperating abstract domains. In FRAMA-C Aluminium, the focus was made on supporting the very same domains as in VALUE for compatibility: case studies demonstrate that EVA gets comparable analysis time for better results. New domains will be introduced in the next releases of FRAMA-C, like support for Apron's domains [34] and Venet's Gauge [35].

5 Runtime Verification with FRAMA-C / E-ACSL

FRAMA-C was initially designed as a static analysis platform. Later on, it was extended to provide dynamic analysis tools as well. First, PATHCRAWLER, a preexisting test case generation tool (presented in Section 6), has been partially rewritten to become a FRAMA-C plug-in. Second, a runtime verification tool, namely E-ACSL, has been implemented as a FRAMA-C plug-in. The E-ACSL tool is the purpose of this section.

Since FRAMA-C was originally oriented towards static verification, ACSL has the same bias. In particular, it is based on mathematical logic that cannot be dynamically verified in its entirety. Consequently, an "executable" subset of this specification language has been designed, in which each annotation has an executable meaning. This specification language is also called E-ACSL ("E" stands for "executable"). Given a C program p annotated in E-ACSL, the FRAMA-C plug-in E-ACSL generates another C

[4] Interval arithmetic guarantees that the product of two numbers of the same sign is positive here.

program which observationally behaves like p if each annotation is satisfied, or reports the first failing annotation and exits otherwise. Section 5.1 introduces the annotation language, while Section 5.2 presents an overview of the tool.

5.1 E-ACSL Specification Language

The E-ACSL specification language [11, 36] is a large strict subset of ACSL. It excludes ACSL constructs which have no significance at runtime. For instance, it includes neither mathematical lemmas nor axiomatics. There is no termination property as well, for example, to specify that a function does not terminate: it could not be verified in finite time at runtime. However, loop variants and **`decreases`** clauses —which are respectively used to prove termination of loops and recursive functions by specifying a measure which strictly decreases at each iteration/invocation— are still present because their verification only depends on (at most) two previous loop/function body runs.

Quantifications The most important restriction of E-ACSL is certainly that every quantified variable must be syntactically bounded to a finite interval (whose bounds are not necessarily constant). For instance, if `arr` is an array of `len` cells, the predicate

$$\texttt{\textbackslash forall integer i; 0}\leq\texttt{i<len} \Longrightarrow \texttt{arr[i]>0} \qquad (1)$$

means that every cell of `arr` is positive. However, because of an unbounded quantification over `x`, the ACSL predicate `\forall integer x, (2*x)%2 ==0` (stating that every even integer is dividable by 2) does not belong to the E-ACSL language. This restriction is not a strong limitation in practice because quantifications in program properties are usually constrained by the program context.

Integers Example (1) of the previous paragraph illustrates that E-ACSL also supports mathematical integers in the same way as ACSL: E-ACSL remains compatible with tools supporting ACSL (in particular, other FRAMA-C plug-ins). It is still possible to use modular arithmetic in specifications through casts. For instance, the term `(`**`int`**`)(INT_MAX+1)` is interpreted as `INT_MIN`.[5] Although mathematical integers make the runtime verification harder, they can be safely implemented by using machine integers in almost all practical cases (see Section 5.2).

Undefinedness The most important change with respect to ACSL is the introduction of undefined terms and predicates *à la* Chalin [37] through tri-valued logic. Indeed, undefined terms like `1/0` would lead to an undefined C behavior if executed, while they introduce no issue in static tools: these tools just cannot prove any non-trivial property containing such terms except tautologies like `1/0==1/0` (by commutativity of equality). The E-ACSL semantics of such terms and predicates is undefined in order to overcome this issue.

Another important source of undefinedness is memory accesses like `*p` and `t[i]`. Tools supporting the E-ACSL language must ensure that undefined terms and predicates are never evaluated. Section 5.2 explains how our FRAMA-C plug-in handles

[5] Unlike the ISO C99 standard, ACSL and E-ACSL explicitly specify the semantics of cast overflows through modular interpretations (see ACSL reference manual [3, Section 2.2.4]).

them. In order to limit the impact of undefinedness, logical operators like `&&`, `||` and $\Longrightarrow$ are lazy in E-ACSL. For instance, the interpretation of `n≠0 && 10/n==m` is always well-defined. This semantics change remains nevertheless consistent with the original ACSL semantics: for any E-ACSL predicate p, if p is valid (resp. invalid) in ACSL then p is either valid (resp. invalid) or undefined in E-ACSL. Conversely, if p is valid (resp. invalid) in E-ACSL then p is also valid (resp. invalid) in ACSL. This fundamental property ensures tool compatibility between ACSL and E-ACSL.

5.2 E-ACSL Inline Monitoring Tool

The FRAMA-C plug-in E-ACSL is a program transformation tool: it takes as input a C program p annotated with E-ACSL specifications and generates another C program which observationally behaves like p if each annotation is satisfied, or stops on the first failing annotation otherwise. In other words, E-ACSL generates an *online* (more precisely, *inline*) monitor [38] for a C program based on its formal specification. This inline monitor is heavily optimized: E-ACSL got the second place of the first Competition of Runtime Verification tool (CRV) in 2014 [39], then won the second competition in 2015 (in the category of online monitoring of C programs in both cases).

Figure 6 shows how simple the E-ACSL transformation looks like in simple cases[6]: it mainly converts an ACSL assertion into an executable assertion through the use of a dedicated C function `e_acsl_assert`[7] which behaves by default in the same way as the standard C macro **`assert`** and can be customized by the end-user. However, a closer look at this simple example illustrates that the transformation is not as easy as it may sound. Indeed, E-ACSL generates **`long long`** integers `1LL` and `0LL` in order to perform the computation in this (bigger) type and ensure the absence of **`int`** overflows in `y-1`.[8] This section proposes a short overview of the transformation scheme which allows E-ACSL to generate efficient-but-sound code.

```
int div(int x, int y) {
  /*@ assert y-1 ≠ 0; */
  return x/(y-1);
}
```

```
int div(int x, int y) {
  /*@ assert y-1 ≠ 0; */
  e_acsl_assert(y-1LL ≠ 0LL);
  return x/(y-1);
}
```

Fig. 6: Naive E-ACSL translation. Original code (left) *vs.* translated code (right).

[6] The generated code shown in this paper is compliant with a 64-bit x86 architecture.

[7] It actually takes additional arguments in order to provide informative user feedback when a property is violated. They are omitted for clarity.

[8] The C99 semantics of subtraction ensures that, in the generated code, `y` is converted to **`long long`** through the usual arithmetic conversion before computing the subtraction (see ISO C99 standard [32, Sections 6.3.1.8 and 6.5.6]).

Implementing Mathematical Integers E-ACSL uses the GMP library[9] in order to implement mathematical integers. For instance, Figure 7 presents the generated code for the previous example of function `div` when forcing E-ACSL to use GMP for integer operations. GMP integers are actually pointers that must be allocated and deallocated. In the

```
1  int div(int x, int y) {
2    /*@ assert y-1 ≠ 0; */
3    mpz_t e_acsl_y, e_acsl_1, e_acsl_sub, e_acsl__2;
4    int e_acsl_ne;
5    mpz_init_set_si(e_acsl_y,(long)y);          // e_acsl_y = y
6    mpz_init_set_si(e_acsl_1,1L);               // e_acsl_1 = 1
7    mpz_init(e_acsl_sub);
8    mpz_sub(e_acsl_sub, e_acsl_y, e_acsl_1); // e_acsl_sub = y-1
9    mpz_init_set_si(e_acsl__2,0L);              // e_acsl_2 = 0
10   e_acsl_ne = mpz_cmp(e_acsl_sub, e_acsl__2); // y-1 == 0
11   e_acsl_assert(e_acsl_ne ≠ 0);                // runtime check
12   mpz_clear(e_acsl_y); mpz_clear(e_acsl_1);   // deallocations
13   mpz_clear(e_acsl_sub); mpz_clear(e_acsl__2);
14   return x/(y-1);
15 }
```

Fig. 7: Translation of function `div` by using GMP.

example, lines 5–7 and 9 allocate (and initialize at the same time) four GMP integers, while lines 12–13 free them. Integer operations are performed through function calls. In our example, the subtraction is computed at line 8 and the comparison is done at line 10. The runtime check at line 11 consists in checking the result of this comparison.

Although safe, this translation scheme through GMP is quite heavy and inefficient: compare it with the direct translation scheme presented in Figure 6 to see how more complex it is. Doing this GMP translation for every integer operation is not practical, but it allows us to translate any mathematical operations in a safe way. Consequently, E-ACSL implements a (sub-)type system based on interval inference which infers, for every integer term, the smallest C type that may contain all its possible values [11, 40]. It is either a C integral type or a GMP. In our `div` example, it allows E-ACSL to safely use the type **`long long`** to perform the subtraction without overflow. Our experiments have demonstrated that almost no GMP code is generated by E-ACSL, except if the initial code does contain (signed or unsigned) long long integers. It is worth noting that AdaCore has adapted this solution to SPARK 2014 in order to allow its users to specify mathematical properties without worrying about overflows while preserving efficiency at runtime.

Preventing Undefined Behaviors In Section 5.1, we have said that every tool which aims at supporting the E-ACSL language must ensure that undefined terms and predi-

[9] See http://gmplib.org/.

cates are never executed. To reach this goal, the E-ACSL plug-in relies on the FRAMA-C plug-in RTE. As explained in Section 2, this plug-in generates an ACSL annotation with a guard to prevent every possible undefined behavior of the source code. All the annotations generated by RTE are actually E-ACSL-compliant and the RTE's API allows a developer to generate such annotations for a particular code fragment (for example, a C expression).

Consequently, when generating some code fragment C, E-ACSL asks RTE to generate annotations to prevent undefined behavior in C. Then it converts them into additional code fragment C' thanks to its own translator. No recursion is required because RTE's generated annotations never contain undefined terms or predicates: C' is always free of undefined behaviors. Figure 8 illustrates this translation scheme on a simple example: when translating the predicate `u/v==2`, E-ACSL generates an annotation `v` $\neq$ `0` thanks to the RTE plug-in. This extra annotation is then turned into C code by E-ACSL itself.

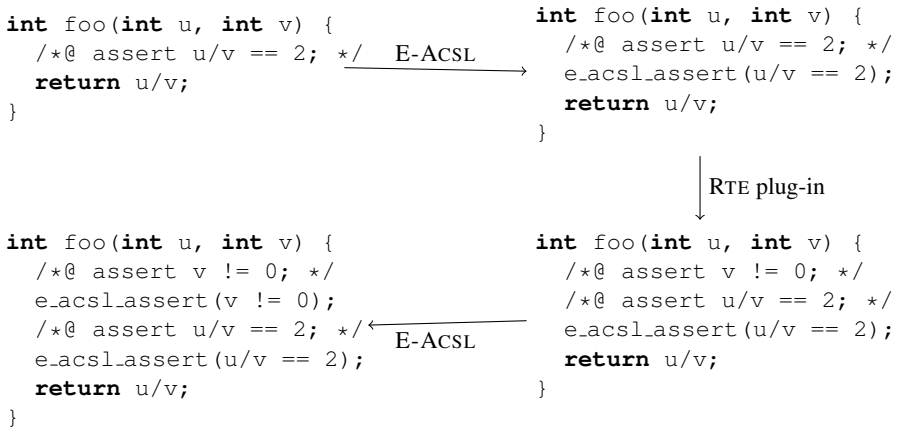

Fig. 8: Preventing runtime errors in the code generated from specifications.

Supporting Memory-Related Constructs An important feature of E-ACSL is memory-related constructs (introduced in Section 3.1), which allow the users to express complex properties about program memory. In particular, the RTE plug-in may use them to generate annotations preventing memory-related errors like dereferencing an invalid pointer: if the RTE plug-in has been executed on the original code in order to generate annotations for possible undefined behaviors, E-ACSL may be used to detect them at runtime.

In the general case, handling such constructs requires to query the program memory at runtime, for instance, to check whether some data has been fully initialized, to get the length of a memory block, or to get the offset of a pointer from its base address. For

this purpose, E-ACSL comes with its own memory runtime library (mRTL) to be linked against the generated code [41]. This code records program memory modifications in a dedicated mRTL datastore, which can then be queried to evaluate memory-related E-ACSL constructs. Figure 9 shows such an instrumentation: memory allocations, deallocations and initializations are stored in the mRTL store, and checking an assertion requires to query the store.

However, this instrumentation is expensive: it is desirable to avoid it whenever possible. In our example, every line marked as *useless* is indeed not necessary since we are only interested in checking the validity of `p` at line 16 (that is, checking whether `p` is an initialized pointer that refers to a memory location which can be safely accessed by the program). It is worth noting that line 7 which stores the allocation of the local variable `x` must be kept because of the alias between `p` and `&x` is created at line 9: `p` is indeed valid because it is the address of this local variable.

In order to limit this instrumentation, E-ACSL implements a backward dataflow analysis that soundly over-approximates the memory locations to be monitored [40, 42]: all other locations (all the lines marked as *useless* in our example) can safely be untracked by the monitor.

6 Test Case Generation with PATHCRAWLER

For structural unit testing of C code, FRAMA-C offers a test case generation tool, called PATHCRAWLER [13, 14]. Given a C source code with a function under test f, it tries to generate test cases that *cover* (i.e. activate) all feasible execution paths in f, that is, to achieve the *all-paths* test coverage criterion. Its method combines symbolic execution, concrete execution and constraint solving similarly to Dynamic Symbolic Execution tools like DART/CUTE, PEX, SAGE, KLEE, etc. [43].

The main steps of the method are presented in Figure 10. First, a chosen (partial) program path π is symbolically executed in order to construct its *path predicate* φ_π, that is, the constraints over the values of input variables that ensure the execution of π. Next, a constraint solver is used to solve the set of constraints φ_π. PATHCRAWLER relies on the COLIBRI constraint solver also developed at CEA List. If it succeeds, the resulting solution provides a test datum that covers the target path π. This test datum is then executed concretely on an instrumented version of the function in order to record the complete path and program outputs, and to double-check that it covers the target path π. If φ_π has no solution, path π is infeasible (i.e. impossible to activate). Finally, the next path to be covered is chosen. The tool continues similarly for all program paths that are explored in a depth-first search. When the number of paths is too large for an exhaustive path coverage, the user can limit their exploration to paths with at most k consecutive iterations of loops (k-*paths* criterion).

PATHCRAWLER is sound, meaning that each test case activates the test objective for which it was generated. This is verified by concrete execution. PATHCRAWLER is also complete in the following sense: if the tool manages to explore all feasible paths of the program, then the absence of a test for some path means that this path is infeasible, since the tool does not approximate path constraints [14, Section 3.1].

```
1  int main(void) {
2    int x, y, z, *p;
3     // local variable allocations
4      __store_block(&p, 4U);
5      __store_block(&z, 4U);                    // useless
6      __store_block(&y, 4U);                    // useless
7      __store_block(&x, 4U);
8      __full_init(&p);  // initialization of p
9    p = &x;
10     __full_init(&x);  // initialization of x  // useless
11   x = 0;
12     __full_init(&y);  // initialization of y  // useless
13   y = 1;
14     __full_init(&z);  // initialization de z  // useless
15   z = 2;
16   /*@ assert \valid(p); */
17     // runtime check
18     {
19       int __e_acsl_initialized;
20       int __e_acsl_and;
21       __e_acsl_initialized = __initialized((void *)(&
             p),sizeof(int *));
22       if (__e_acsl_initialized) {
23         int __e_acsl_valid;
24         __e_acsl_valid = __valid((void *)p, sizeof(int));
25         __e_acsl_and = __e_acsl_valid;
26       }
27       else __e_acsl_and = 0;
28       e_acsl_assert(__e_acsl_and);
29     }
30   *p = 3;
31     // memory deallocation
32     __delete_block(&p);
33     __delete_block(&z);                       // useless
34     __delete_block(&y);                       // useless
35     __delete_block(&x);
36   return 0;
37 }
```

Fig. 9: E-ACSL memory instrumentation.

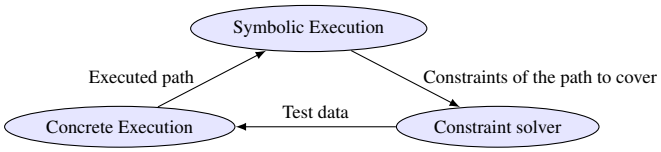

Fig. 10: Main steps of the PATHCRAWLER method.

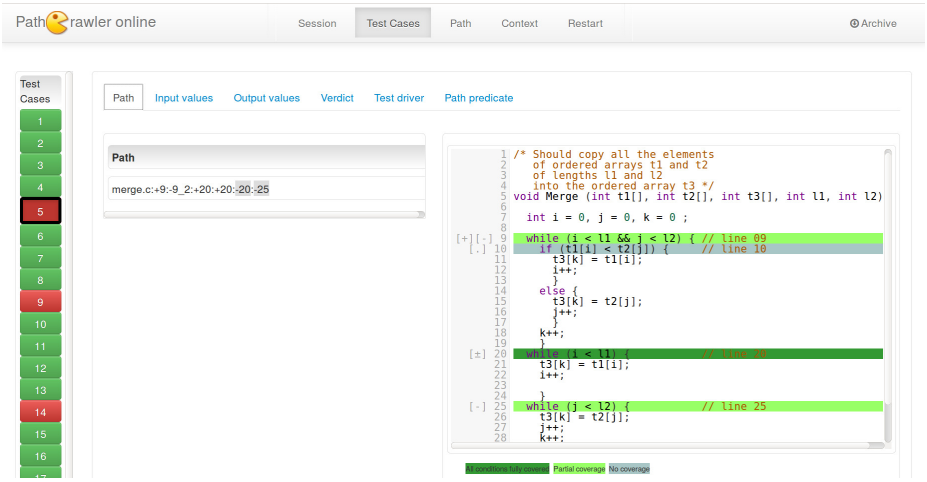

Fig. 11: Example of test case generation results on pathcrawler-online.com where the user can find the test data, executed paths and branches, path predicates, concrete and symbolic outputs, pass or fail verdicts, etc.

PATHCRAWLER can accept user-provided test parameters that indicate the chosen strategy (*all-paths* or *k-paths*) and a precondition specifying the desired value intervals and relationships between input variables. They should be carefully specified in order to avoid generation of inadmissible test data. PATHCRAWLER can be used through the online test generation service http://pathcrawler-online.com/. Figure 11 illustrates the results of a test generation session with this service. The reader can find more information on the tool and its usage in [4, 5, 13, 14].

Recently, a new efficient variant of dynamic symbolic execution has been proposed for a rich set of test coverage criteria [44]. In this approach, test generation is highly optimized in order to avoid both unnecessary redundant attempts to cover a test objective and an exponential blow-up of the search space (in particular, by removing the constraints of a test objective from the constraint store while trying to cover other objectives). This technique has been implemented in the LTEST toolset [45] on top of PATHCRAWLER.

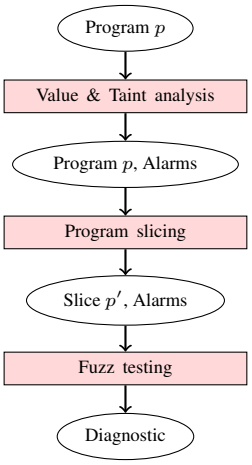

Fig. 12: Methodology of the Flinder-SCA tool.

7 Combinations of Analyses

Various combinations of analyses have been designed and implemented within FRAMA-C. In this section, we present a few of them where different static and dynamic analyzers are advantageously combined together in FRAMA-C.

The SANTE method [46, 47] aims at detecting runtime errors and combines three FRAMA-C analyzers. First it runs value analysis to detect potential errors, or *alarms*. Next, it runs slicing in order to simplify the program with respect to these alarms by preserving possible erroneous behavior. Finally, test generation with PATHCRAWLER tries to cover these alarms and trigger potential erroneous situations. PATHCRAWLER can confirm an alarm as a real bug, or sometimes, when it manages to cover all paths without triggerring the alarm, establish that it is safe (i.e. a false alarm). In this combination, the analyzers are complementary: error detection with abstract interpretation based value analysis is complete but imprecise, while testing is precise but incomplete since it is in general not exhaustive. Slicing removes irrelevant code, simplifies the search space and thus makes testing more efficient.

The SANTE method was recently extended to security flaw detection and successfully applied to the Heartbleed vulnerability in OpenSSL library [48]. Its methodology is shown in Figure 12. In addition to value analysis that detects runtime errors, taint analysis is used to identify alarms that can be impacted by potentially malicious input values and are likely to be exploitable. After a program simplification step with slicing, a dynamic analysis step (with the fuzzing tool Flinder) is used to try to trigger the alarms. This work also demonstrates the possibilities of collaboration of FRAMA-C analyzers with external tools: indeed, taint analysis and fuzz testing tools used in this project were implemented by two industrial partners.

Another interesting collaboration where dynamic analysis also improves a static verification technique is realized by the STADY tool [49]. During deductive verification,

when some proof fails, STADY runs test generation to help the validation engineer to understand the reason of each proof failure and illustrate it by a counterexample.

Inversely, static analysis can be beneficial for dynamic analysis. In the context of the LTEST testing toolset [45], a combination of VALUE and WP is efficiently used to detect infeasible test objectives and therefore to avoid the waste of time of covering them during test generation [50]. Another combination, where static analysis helps to optimize runtime verification by removing irrelevant monitoring code, has been mentioned in Section 5.

8 Conclusion

Modern software has nowadays become increasingly critical and widely expanded in various domains of our life. Bugs and security flaws may have very expensive costs and sometimes lead to dramatic consequences. In this context, practical and efficient tools for software analysis and verification are necessary to ensure a high level of safety and security of software.

In this paper we have presented a synopsis of a tutorial on FRAMA-C, a rich and extensible platform for analysis of C code. FRAMA-C has been successfully applied in several industrial [28, 51–54] and academic projects [27, 29, 55–59], and has become a reference for teaching software verification in several universities and engineering schools all around the world (including France, Germany, United Kingdom, Portugal, Russia, Brazil, China, United States). We have described its main analyzers based on abstract interpretation, deductive verification, runtime assertion checking and test case generation. These analyzers are publicly available in open-source or online versions. We have also emphasized a few combinations of analyses that appear to be practical and complementary to each other. FRAMA-C provides a convenient and powerful platform for combining different analyzers and development of new ones.

References

1. Boulanger, J.L., ed.: Industrial Use of Formal Methods: Formal Verification. Wiley-ISTE (2012)
2. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing **27**(3) (2015) 573–609
3. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. http://frama-c.com/acsl.html.
4. Kosmatov, N., Williams, N., Botella, B., Roger, M., Chebaro, O.: A lesson on structural testing with PathCrawler-online.com. In: International Conference on Tests and Proofs (TAP 2012). Volume 7305 of LNCS., Springer (2012) 169–175
5. Williams, N., Kosmatov, N.: Structural testing with PathCrawler: Tutorial synopsis. In: International Conference on Quality Software (QSIC 2012), IEEE (2012) 289–292
6. Kosmatov, N., Prevosto, V., Signoles, J.: A lesson on proof of programs with Frama-C. In: International Conference on Tests and Proofs (TAP 2013). Volume 7942 of LNCS., Springer (2013)
7. Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: International Conference on Runtime Verification (RV 2013). Volume 8174 of LNCS., Springer (2013) 386–399

8. Kosmatov, N., Signoles, J.: Runtime assertion checking and its combinations with static and dynamic analyses - tutorial synopsis. In: International Conference on Tests and Proofs (TAP 2014). Volume 8570., Springer (2014) 165–168
9. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z.: WP Plug-in Manual. http://frama-c.com/wp.html.
10. Cuoq, P., Yakobowski, B., Prevosto, V.: Frama-C's value analysis plug-in. http://frama-c.com/download/value-analysis.pdf.
11. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: the 28th Annual ACM Symposium on Applied Computing (SAC 2013), ACM (2013) 1230–1235
12. Signoles, J.: E-ACSL User Manual. http://frama-c.com/download/e-acsl/e-acsl-manual.pdf.
13. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: European Dependable Computing Conference (EDCC 2005). Volume 3463 of LNCS., Springer (2005) 281–292
14. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: International Workshop on Automation of Software Test (AST 2009), IEEE (2009) 70–78
15. Cuoq, P., Signoles, J.: Experience report: Ocaml for an industrial-strength static analysis framework. In: International Confererence on Functional Programming (ICFP 2009). (2009) 281–286
16. Signoles, J.: Software Architecture of Code Analysis Frameworks Matters: The Frama-C Example. In: Workshop on Formal Integrated Development Environment (F-IDE 2015). (2015) 86–96
17. Correnson, L., Signoles, J.: Combining Analyses for C Program Verification. In: International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2012). Volume 7437 of LNCS., Springer (2012) 108–130
18. Meyer, B.: Object-oriented Software Construction, Second Edition. Object-oriented Series, Prentice Hall, New York (1997)
19. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accomodates both runtime assertion checking and formal verification. In: International Symposium on Formal Methods for Components and Objects (FMCO 2002). Volume 2852 of LNCS., Springer (2002) 262–284
20. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8) (1975) 453–457
21. Correnson, L.: Qed. computing what remains to be proved. In: NASA International Symposium on Formal Methods (NFM 2014). Volume 8430 of LNCS., Springer (2014) 215–229
22. Burghardt, J., Gerlach, J., Lapawczyk, T.: ACSL by Example. (2016) https://gitlab.fokus.fraunhofer.de/verification/open-acslbyexample/blob/master/ACSL-by-Example.pdf.
23. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages (POPL 1977), ACM Press (1977) 238–252
24. Deutsch, A.: Static verification of dynamic properties (2003) PolySpace White Paper.
25. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: European Symposium on Programming (ESOP 2005). Volume 3444 of LNCS., Springer (2005) 21–30
26. Feret, J.: Static Analysis of Digital Filters. In: European Symposium on Programming (ESOP 2004). Volume 2986 of LNCS., Springer (2004) 33–48
27. Berthomé, P., Heydemann, K., Kauffmann-Tourkestansky, X., Lalande, J.F.: Attack model for verification of interval security properties for smart card C codes. In: Programming Languages and Analysis for Security (PLAS 2010), ACM (2010) 1–12

28. Cuoq, P., Delmas, D., Duprat, S., Moya Lamiel, V.: Fan-C, a Frama-C plug-in for data flow verification. In: Embedded Real-Time Software and Systems Congress (ERTS^2 2012). (2012)
29. Demay, J.C., Totel, E., Tronel, F.: SIDAN: a tool dedicated to software instrumentation for detecting attacks on non-control-data. In: International Conference on Risks and Security of Internet and Systems (CRiSIS 2009), IEEE (2009) 51–58
30. TrustInSoft: tis-ct blog post http://trust-in-soft.com/tis-ct/.
31. Bonichon, R., Cuoq, P.: A mergeable interval map. Studia Informatica Universalis **9**(1) (2011) 5–37
32. ISO/IEC 9899:1999: Programming languages – C
33. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: European Symposium on Programming (ESOP 2005). Volume 3444 of LNCS., Springer (2005) 5–20
34. Jeannet, B., Miné, A.: Apron: A Library of Numerical Abstract Domains for Static Analysis. In: Computer Aided Verification (CAV 2009). (2009) 661–667
35. Venet, A.: The Gauge Domain: Scalable Analysis of Linear Inequality Invariants. In: Computer Aided Verification (CAV 2012). (2012) 139–154
36. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language. (May 2015) http://frama-c.com/download/e-acsl/e-acsl.pdf.
37. Chalin, P.: Engineering a sound assertion semantics for the verifying compiler. IEEE Transactions on Software Engineering **36** (2010) 275–287
38. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Engineering Dependable Software Systems. Volume 34 of NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press (2013) 141–175
39. Bartocci, E., Bonakdarpour, B., Falcone, Y., Colombo, C., Decker, N., Klaedtke, F., Havelund, K., Joshi, Y., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang., Y.: First International Competition on Runtime Verification. Rules, Benchmarks, Tools and Final Results of CRV 2014. Submitted.
40. Jakobsson, A., Kosmatov, N., Signoles, J.: Rester statique pour devenir plus rapide, plus précis et plus mince. In: Journées Francophones des Langages Applicatifs (JFLA 2015). (2015) In French.
41. Kosmatov, N., Petiot, G., Signoles, J.: An Optimized Memory Monitoring for Runtime Assertion Checking of C Programs. In: the 4th International Conference on Runtime Verification (RV 2013). Volume 8174 of LNCS., Springer (2013) 167–182
42. Jakobsson, A., Kosmatov, N., Signoles, J.: Expressive as a tree: Optimized memory monitoring for C. Submitted.
43. Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: International Conference on Software Engineering (ICSE 2011), ACM (2011) 1066–1071
44. Bardin, S., Kosmatov, N., Cheynier, F.: Efficient leveraging of symbolic execution to advanced coverage criteria. In: International Conference on Software Testing, Verification and Validation (ICST 2014), IEEE (2014) 173–182
45. Bardin, S., Chebaro, O., Delahaye, M., Kosmatov, N.: An all-in-one toolkit for automated white-box testing. In: International Conference on Tests and Proofs (TAP 2014). Volume 8570 of LNCS., Springer (2014) 53–60
46. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: the ACM Symposium on Applied Computing (SAC 2012), ACM (2012) 1284–1291
47. Chebaro, O., Cuoq, P., Kosmatov, N., Marre, B., Pacalet, A., Williams, N., Yakobowski, B.: Behind the scenes in SANTE: a combination of static and dynamic analyses. Autom. Softw. Eng. **21**(1) (2014) 107–143

48. Kiss, B., Kosmatov, N., Pariente, D., Puccetti, A.: Combining static and dynamic analyses for vulnerability detection: Illustration on heartbleed. In: International Haifa Verification Conference (HVC 2015). Volume 9434 of LNCS., Springer (2015) 39–50
49. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: Your proof fails? testing helps to find the reason. In: International Conference on Tests and Proofs (TAP 2016). Volume 9762 of LNCS., Springer (2016) 130–150
50. Bardin, S., Delahaye, M., David, R., Kosmatov, N., Papadakis, M., Traon, Y.L., Marion, J.: Sound and quasi-complete detection of infeasible test requirements. In: International Conference on Software Testing, Verification and Validation (ICST 2015), IEEE (2015) 1–10
51. Bishop, P.G., Bloomfield, R.E., Cyra, L.: Combining testing and proof to gain high assurance in software: A case study. In: International Symposium on Software Reliability Engineering (ISSRE 2013), IEEE (2013) 248–257
52. Cuoq, P., Hilsenkopf, P., Kirchner, F., Labbé, S., Thuy, N., Yakobowski, B.: Formal verification of software important to safety using the Frama-C tool suite. In: International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies (NPIC & HMIT). (2012)
53. Delmas, D., Duprat, S., Moya-Lamiel, V., Signoles, J.: Taster, a Frama-C plug-in to enforce Coding Standards. In: Embedded Real-Time Software and Systems Congress (ERTS^2 2010)
54. Pariente, D., Ledinot, E.: Formal verification of industrial C code using Frama-C: a case study. In: International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010). (2010)
55. Ceara, D., Mounier, L., Potet, M.L.: Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences. In: the 3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2010). (2010) 371–380
56. Ayache, N., Amadio, R., Régis-Gianas, Y.: Certifying and reasoning on cost annotations in C programs. In: Formal Methods for Industrial Critical Systems (FMICS 2012). (2012)
57. Carvalho, N., da Silva Sousa, C., Pinto, J.S., Tomb, A.: Formal Verification of kLIBC with the WP Frama-C Plug-in. In: NASA International Symposium on Formal Methods (NFM 2014). Volume 8430 of LNCS., Springer (2014) 343–358
58. Gavran, I., Niksic, F., Kanade, A., Majumdar, R., Vafeiadis, V.: Rely/Guarantee Reasoning for Asynchronous Programs. In: International Conference on Concurrency Theory (CONCUR 2015). (2015) 483–496
59. Nguena-Timo, O., Langelier, G.: Test Data Generation for Cyclic Executives with CBMC and Frama-C: A Case Study. Electronic Notes in Theoretical Computer Science **320** (2016) 35–51