E-ACSL, a runtime verification tool for safety and security of C programs (tool paper)

Julien Signoles, Nikolai Kosmatov, Kostyantyn Vorobyov

E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper)

Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov

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

Abstract

This tool paper presents E-ACSL, a runtime verification tool for C programs capable of checking a broad range of safety and security properties expressed using a formal specification language. E-ACSL consumes a C program annotated with formal specifications and generates a new C program that behaves similarly to the original if the formal properties are satisfied, or aborts its execution whenever a property does not hold. This paper presents an overview of E-ACSL and its specification language.

Keywords: Runtime Verification Tool, Runtime Assertion Checking, Memory Debugger, Formal Specification Language

1 Introduction

E-ACSL is a runtime verification tool within Frama-C [14], a framework dedicated to source code analysis of C programs[1]. E-ACSL is capable of checking various safety and security properties at runtime including (but not limited to) complex functional specifications, well ordering of function calls, information flow leakage and a broad range of undefined behaviors, focusing on such issues as division by zero, integer overflows, validity of pointer dereferences and accesses to uninitialized memory. These properties are expressed as source code annotations written using a formal specification language also called E-ACSL.

The E-ACSL tool consumes a C program annotated with specifications written in the E-ACSL specification language and outputs a new C program that embeds an inline monitor generated from the formal E-ACSL specification. At runtime the monitored program behaves similarly to the original if the formal properties are satisfied, or aborts its execution if any property is violated. It is worth noting that since Frama-C provides several plug-ins to generate E-ACSL annotations from high-level or implicit specifications, most usages of E-ACSL do not require annotating programs manually. Verifying properties at runtime with E-ACSL is thus a mostly automatic process.

Even though several papers have already described such E-ACSL components as its specification language [8], memory model [29, 28], static analyses to optimize code generation [12, 13], and possible usages in conjunction with other Frama-C plug-ins [17, 21, 2], this paper aims at presenting a quick overview of the whole tool.

Outline Section 2 briefly describes the E-ACSL specification language. Section 3 gives a sketch of the tool's design. Section 4 points out several possible practical usages of E-ACSL and Section 5 summarizes results of experiments and case studies involving practical runtime verification of C programs with E-ACSL.

[1] Frama-C, including E-ACSL, is freely available from http://frama-c.com

```
 1 /*@ requires 2 <= n <= 10000;
 2     requires \forall integer j; 0 <= j < n ==> \valid(t+j);
 3     ensures \forall integer j; 0 <= j < n ==>
 4       ( t[j] != 0 <==>
 5         (j >= 2 && (\forall integer d; 2 <= d < j ==> j%d != 0) ) );
 6     assigns t[0..n-1]; */
 7 void primes(int *t, int n) {
 8   int i, k;
 9   t[0]=t[1]=0;
10   /*@ loop invariant I1: 2 <= i <= n;
11       loop invariant I2: \forall integer j; 0 <= j < i ==>
12         ( t[j] != 0 <==>
13           (j >= 2 && (\forall integer d; 2 <= d < j ==> j%d != 0) ) );
14       loop assigns i, k, t[0..n-1];
15       loop variant n-i; */
16   for(i=2; i < n; i++) {
17     /*@ loop invariant I3: 2 <= k <= i && t[i] == 1;
18         loop invariant I4: \forall integer j; 0 <= j < i ==>
19           ( t[j] != 0 <==>
20             (j >= 2 && (\forall integer d; 2 <= d < j ==> j%d != 0) ) );
21         loop invariant I5: \forall integer d; 2 <= d < k ==> i % d != 0 ;
22         loop assigns k, t[0..i];
23         loop variant i-k; */
24     for(k=2, t[i]=1; k < i; k++) {
25       if( i % k == 0){
26         t[i]=0;
27         break;
28       }
29     }
30   }
31 }
32
33 int main(){
34   int i, a[100];
35   primes(a, 200); // provokes illegal memory access
36   /*@ assert \base_addr(&i) != \base_addr(a); */
37   /*@ assert \forall integer j; 0 <= j < 100 ==> \initialized(a + j); */
38 }
```

Figure 1: Example of an annotated C program primes computing prime numbers

2 E-ACSL Specification Language

The E-ACSL specification language is formally described in the reference manual [25], and its overview is presented in an earlier article [8]. This section gives a brief summary of its main features.

The E-ACSL specification language [25] is an executable subset of the ACSL specification language [5]. This language is a contract-based behavioural specification language *à la* Eiffel [19] whose design was inspired by JML [18]. ACSL logic is a typed first-order logic whose terms are pure (*i.e.* side-effect free) C expressions extended with keywords and built-ins to handle C specificities.

Fig. 1 shows an example C program with ACSL annotations. Given an array `t` of `n` integers, the function `primes` writes a non-zero value to array element `t[j]` if `j` is a prime number, and 0 otherwise. The function `main` calls `primes` on the integer array `a` of 100 elements, but erroneously specifies its size as 200 instead of 100. The execution of `main` may thus provoke runtime errors.

The contract of the function `primes` is given via annotations at lines 1–6 of Fig. 1. The precondition clauses at lines 1–2 assume that the value of `n` is between 2 and 10000, and the array `t` has valid cells for all indices `j` between 0 and `n-1`. It is expressed as validity of the pointer `t+j` to the corresponding cell `t[j]` using the builtin predicate `\valid`. For a pointer `p`,

\valid(p) states that memory location *p can be safely read and written. The postcondition of the function states that for every index j, t[j] is non-zero if and only if j is prime. The fact that j is prime is expressed as being greater or equal to 2, and having no divisor d such that 2<=d<j. The clause at line 6 states that the elements of array t are the only variables that can have a different value after the function call.

The annotations at lines 10–15 and 17–23 express loop contracts. Each contract contains loop invariants, a loop assigns clause and a loop variant. Loop invariants must be true before the loop and after each loop iteration. The clause loop assigns indicates variables whose values can change during the execution of the loop. A loop variant is an integer expression that must be non-negative whenever a loop iteration starts, and must strictly decrease at each iteration, thus allowing a deductive verification tool to deduce the termination of the loop after a finite number of steps.

Two other builtin predicates of ACSL are illustrated by assertions at lines 36–37. The first one specifies that the memory blocks containing variable i and array a do not have the same base address. In other words, they are two different blocks. The ACSL construct \base_addr(p) denotes the base (i.e. start) address of the memory block pointer p belongs to. The assertion at line 37 specifies that all array elements are initialized. The ACSL predicate \initialized(p) expresses that the memory location *p referred to by pointer p has been initialized.

Using the deductive verification plug-in Frama-C/Wp, one can easily prove that function primes respects its ACSL contract. As expected, the proof that the call to primes at line 35 respects its precondition will fail (for the precondition at line 2) because the validity for all required indices (up to $200 - 1 = 199$) is not ensured here. The assertion at line 36 is proved as well (while that at line 37 is left unknown since Wp does not support the \initialized predicate yet). While detailed function and loop contracts are mandatory to obtain a complete proof and/or meaningful proof failures using a deductive verification tool, runtime assertion checking can be applied without having to manually specify all of them.

Although inspired by JML, ACSL was designed for static analysis and notably program proof by means of Dijkstra's weakest precondition calculus (WP) [9]. In particular, several constructs have no executable meaning, *e.g.* quantification over infinite sets. That is why E-ACSL was designed as a large subset of ACSL such that all its terms and predicates can be evaluated at runtime. For instance, each quantified integer variable in E-ACSL must be syntactically bounded in order to restrict quantification to finite integer intervals. Actually, the most practical ACSL constructs are still part of E-ACSL. For example, all ACSL annotations of the program of Fig. 1 also belong to E-ACSL. E-ACSL includes mathematical integers and built-in memory predicates and functions even if checking such constructs at runtime efficiently is challenging.

The executable nature of E-ACSL implies one fundamental semantic difference with ACSL: *undefinedness*. Indeed, ACSL is based on standard mathematical logic in which any well-typed construct is well-defined. In particular, any predicate is either true or false (even if possibly unprovable). It is not the case in E-ACSL because some constructs, *e.g.* the expression 1/0, cannot be evaluated at runtime without errors. Consequently, E-ACSL is based on 3-valued logic in which these constructs are undefined. For instance, the predicate 1/0 == 1/0 is valid in ACSL (by reflexivity of equality) but is undefined in E-ACSL. Both logic remain nevertheless consistent: 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.

3 Monitor Generation Scheme

The E-ACSL runtime verification tool is implemented as a plug-in of Frama-C [14], a framework dedicated to source code analysis of C programs. Frama-C is in charge of preprocessing, parsing, typing and providing an abstract syntax tree (AST) representing the input C code annotated with ACSL annotations. E-ACSL in turn generates a new AST that represents the output C program that embeds monitors generated from E-ACSL annotations.

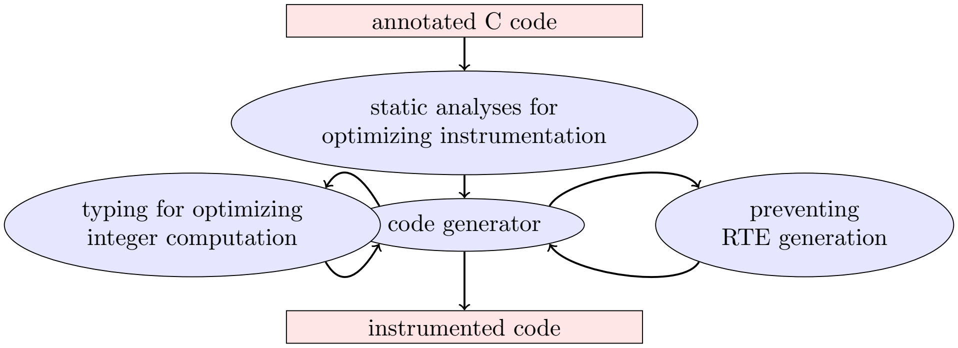
annotated C code
static analyses for optimizing instrumentation
typing for optimizing integer computation
code generator
preventing RTE generation
instrumented code

Figure 2: E-ACSL Translation Scheme.

Fig. 2 shows a translation scheme of the E-ACSL plug-in. Two static analyses are executed prior to code generation. One analysis identifies the goto-like statements which go outside blocks in order to soundly insert necessary code when exiting a block, while the other one finds out an over-approximation of the statements which are necessary to monitor in order to verify memory properties (if any) [13]. For instance, considering a pointer p of type int* and a program containing a single annotation \valid(p), it is only necessary to monitor statements which assign p and its potential aliases.

Two other analyses are performed on the fly during a code generation phase. First, the type system [12] deals with E-ACSL mathematical integers: for any E-ACSL term it computes the smallest C type that can safely encode it. If the term cannot be safely represented by a C type, the generated code uses the Gmp multiple precision arithmetic library[2] to encode it. For instance, assuming a typical 64-bit architecture, if the type of both x and y is int, this type system infers that the result of x+1 fits in a long.

Finally, the RTE plug-in of Frama-C is used to prevent undefined behaviors in the generated code. For instance, for the following E-ACSL annotation /*@ assert x / y == 0; */, RTE automatically generates an additional assertion y != 0 to prevent potential division by zero at runtime.

4 Possible Usages

E-ACSL comes with a companion script called e-acsl-gcc.sh that simplifies the process of invoking Frama-C and E-ACSL to generate an inline monitor m from an input C program p, and further compile and link it against the E-ACSL runtime library using a C compiler (*gcc* by default).

[2] http://gmplib.org

From a user's viewpoint, the main issue is thus writing formal annotations. This process, however, can often be automated. The most important usages of E-ACSL are the following. Notable only the fifth one requires manual annotations.

1. Checking the absence of a wide class of undefined behaviors;
2. Checking high-level properties;
3. Improving detection capabilities of testing;
4. Complement static analyzers;
5. Debugging specifications.

Here is a few details about these activities.

First, checking the absence of a broad class of undefined behaviors is possible due to the Frama-C plug-in RTE [11] that automatically generates E-ACSL assertions to prevent such undefined behaviors as arithmetic overflows, undefined downcasts, array out-of-bound accesses and invalid pointer dereferences. Once generated by RTE, these annotations are translated into runtime checks by E-ACSL. It is worth noting that, when focusing on detection of invalid memory accesses E-ACSL operates similar to a memory debugger such as Address Sanitizer [23] or Valgrind [20]'s MemCheck [24].

Consider an example program in Fig. 1. Even if no E-ACSL annotations are provided by the user, RTE can be used to add assertions verifying the validity of memory accesses. Thanks to these checks out-of-bounds memory accesses in function `primes` at line 35 can be detected by E-ACSL without manual annotation effort.

Second, in checking high-level properties E-ACSL also relies on other Frama-C plug-ins to generate annotations. For instance, the Aoraï [27] plug-in can generate annotations from a Büchi automaton (or, equivalently, an LTL formula) that models function call ordering, *e.g.* "function f must always be called twice before function g". This has been proven useful in practice when modeling function relationship of library APIs. Another example is the SecureFlow plug-in. SecureFlow is a source-to-source transformation that encodes information flows of an input program into itself [1, 2]. It is then possible to verify with E-ACSL that the generated code does not contain information flow leakage, *e.g.* that no confidential data leaks onto a public channel.

Third, all the properties that E-ACSL validates at runtime may theoretically be verified by means of static techniques including abstract interpretation, model checking or WP. However, practical tool limitations may prevent some verifications. In any case, it is often costly to verify properties statically: verifying a subset of properties statically and delegating verification of the remaining properties to E-ACSL may be a good compromise between exhaustiveness of static verification and the cost of the verification process.

Consider an example program in Fig. 1. Suppose line 35 is replaced by `primes(a, 100)`. In this case, even if the loop contracts are not provided by the user, E-ACSL can check that the call to `primes` verifies its precondition before the call, and its postcondition after the call. In contrast, the deductive verification plug-in Wp is not able to check the postcondition without the loop contract. For the original version presented in Fig. 1, E-ACSL detects that the call `primes(a, 200)` on line 35 does not respect its precondition before the call. E-ACSL is also capable of checking the assertions on lines 36 and 37 (even if no other annotations are provided).

Fourth, improving detection capabilities of testing mainly rely on the first two items. For instance, many undefined behaviors remain undetected by testing because they do not lead to program crash or incorrect results on the tested inputs and user system. Coupling E-ACSL to a test case generation tool like PathCrawler [6] or a fuzzing tool such as American Fuzzy Lop[3] may significantly improve their detection rate of errors.

Fifth, before proving complex functional specifications by means of WP, it may be reasonable to test them. Indeed, in case of proof failure, a user must investigate the cause of the bug which requires time and expertise [22]. It is particularly difficult for non-experts [16]. Among others, one possible cause is that the code does not match the specification because one of them is wrong: in that case, testing their compliance in a preliminary step (by means of the fourth E-ACSL usage) improves confidence that the proof failure comes from another cause.

5 Practical Runtime Verification with E-ACSL

One of the strengths of E-ACSL is its practical application to runtime verification of C code. Over the past several years E-ACSL has been experimentally evaluated and used in a number of case studies involving runtime verification of benchmarked and industrial-strength code [15, 13, 4, 2, 21, 29, 28]. This section now briefly summarizes the results of these experiments.

5.1 Competitions on Runtime Verification

One key evaluation of E-ACSL has been performed by an independent committee during Competitions on Runtime Verification (CRV) organized in 2014 [3] and in 2015 [10] as part of the International Conference on Runtime Verification. E-ACSL participated in the C program monitoring track both in 2014 and in 2015. The 2016 edition of the C track did not take place due to a lack of participants.

Each competition was split into three phases. During the first phase the participants were invited to submit their benchmarks to a public repository. In the second, monitor collection phase, the participants were invited to submit their monitors generated from the shared code. During the final evaluation phase the qualified tools were evaluated by running the generated monitors and ranked using different criteria such as memory or CPU utilization overhead and correctness of the runtime analysis.

In the 2014 installment of CRV four runtime verification tools (E-ACSL, RiTHM-1, RTC and ARTiMon) had registered for competing in the track on online monitoring of C programs but only E-ACSL, RiTHM-1 and RTC submitted their solutions for evaluation. In 2014 E-ACSL had won the second place, falling behind the RTC tool[4]. In 2015, where overall six tools registered for the track (E-ACSL, MarQ, RiTHM-v2.0, RTC, RV-monitor and TimeSquareTrace) and only two tools (E-ACSL and RV-monitor) submitted their solutions, E-ACSL arrived at the first place[5].

5.2 Runtime Detection of Memory Errors

One of the benefits of E-ACSL is its ability to detect memory errors such as buffer overflows, illegal dereferences, double free and similar vulnerabilities.

At a specification level, memory error detection via E-ACSL uses built-in annotations including such key predicates as `\valid(p)`, `\initialized(p)`, `\base_addr(p)`, `\offset(p)` and `\block_length(p)`. For a given pointer p, these predicates respectively denote the validity of p

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

[4] https://www.rv-competition.org/2014-2

[5] https://www.rv-competition.org/2015-2

(i.e., whether p is safe to dereference), if the value pointed to by p has been initialized by previously writing to it, the base (start) address of the memory block p belongs to, the byte-length of that block and the byte offset of p relative to the base address.

Memory-related annotations of E-ACSL allow to reason about memory allocated by the program at a block-level, resulting in a much more powerful analysis when compared to state-of-the-art memory debuggers [29]. For instance, for a dereference `*(p+i)` the E-ACSL expression `\base_addr(p) == \base_addr(p+i)` specifies that `p+i` belongs to the same memory block that `p` points to. This detects an out-of-bounds access if both `p` and `p+i` point to two different allocated areas. The E-ACSL specification language can express a large variety of contract properties [8] that go far beyond out-of-bounds checks, where block-level properties can be also very useful.

To support E-ACSL memory predicates, code generated by E-ACSL relies on a C memory monitoring library. This is such that memory modifications made by the program at runtime are stored in the library by the monitor and checking annotations corresponds to querying it. Before program instrumentation, static analysis is performed to safely remove unnecessary memory instrumentation to improve efficiency of the monitor [13]. Memory monitoring in E-ACSL has initially been implemented using a compact prefix tree called patricia trie [15]. The trie stores per-block metadata in its leaves and uses base addresses of memory blocks as keys associated with trie nodes. Routing from the root to a leaf is ensured by internal nodes of the trie.

Initial experiments that involved checking memory-related annotations in a number of small programs (see [15]) indicated that results produced by E-ACSL are precise. Yet, due to complexity of trie lookups, even in small code samples such an approach is prone to producing high runtime overheads of over 100 times compared to performance of unmonitored programs [13]. Further experiments with computationally intensive benchmarks from SPEC CPU datasets [26] revealed that overheads of memory analysis using a patricia trie can exceed 600 times when compared to unobserved executions [29].

To aid high overheads incurred by the monitoring library built upon a patricia trie, it has been replaced by a novel shadow memory monitoring scheme [29] allowing to capture bounds of memory blocks and benefit from constant-time lookups.

Empirical evaluation of the E-ACSL implementation using shadow memory has been conducted for a problem of checking validity of pointer and array accesses over SPEC CPU programs (see [29]). This evaluation compared performance overheads of E-ACSL to the overheads incurred by state-of-the-art memory debuggers and an implementation of E-ACSL using a patricia trie. This experimentation has shown that for the purpose of verifying validity of memory accesses runtime overheads of E-ACSL (avg. ×19) are significantly better than the overheads of a patricia trie approach (avg. ×77.5) and comparable to the runtime overheads of such state-of-the-art memory debuggers as MemCheck (avg. ×15.5) or Dr. Memory (avg. ×18). Furthermore, memory overheads of E-ACSL shadow memory monitoring (avg. ×2.9) were on average the lowest of all tools including E-ACSL patricia trie implementation. It is worth noting that E-ACSL as it stands now has undergone several optimization passes that reduced the average runtime overhead over SPEC CPU programs to only 13 times when compared to runs of original programs.

5.3 Case Studies Using E-ACSL

CURSOR

E-ACSL has also been used to analyze industrial-strength code using third-party tools. For instance, a technique called CURSOR [21] uses the Frama-C Value plugin to generate assertions for dynamically verifying a number of Common Weakness Enumeration (CWE) [7] vulnerabilities and further dynamically verify them using E-ACSL.

To support their approach, Pariente and Signoles applied CURSOR to several Apache libraries (of approximately 8,000 LOC) containing file management and sanitizing functions relevant to security analysis. As a result of its static analysis, CURSOR generated 340 assertions that were further converted to executable code using E-ACSL, leading to a 22% increase of the source code base and an average runtime overhead of less than 3 times.

Secure Flow

In another instance E-ACSL has been used alongside a hybrid flow-sensitive information flow analysis of C programs called Secure Flow [2] to dynamically enforce an information flow policy protecting against timing attacks on a cryptographic library. E-ACSL has been applied to protect against memory errors and ensure correctness of the information flow analysis via Secure Flow.

During the experimentation (reported in [2]) Secure Flow and E-ACSL analyzed 14 cryptographic algorithms belonging to LibTomCrypt library[6]. In this study E-ACSL has shown slowdowns of approximately 15 times compared to unobserved executions. This result is consistent with experimentation using SPEC CPU benchmarks [29].

6 Conclusion

In this tool paper we have given a broad overview of E-ACSL– an online verification tool for C programs. E-ACSL is built atop Frama-C, an industrial-strength framework for source code analysis of C programs. At its core E-ACSL transforms an input program P annotated with formal specifications written in the E-ACSL specification language into a program instrumented with a monitor automatically generated from the specification. A run of the monitored program is aborted is any of the properties given by the specification is violated, otherwise, the behaviour of the monitored program is functionally equivalent to the input program.

The E-ACSL specification language allows for specification of a wide range of properties including such issues as well-ordering of function calls, information flow leakage, integer overflows and memory errors. Most of these properties can be generated automatically using Frama-C, for instance via the RTE plug-in capable of generating annotations verifying the absence of undefined behaviors at runtime.

E-ACSL is a runtime verification tool that has a variety of practical applications. Over the past several years E-ACSL has been applied to a number of problems in runtime analysis of benchmarked and industrial strength code including detection of spatial and temporal memory errors, enforcing information flow security properties and CWE vulnerabilities.

Future Work

At the present stage of its implementation E-ACSL supports generating C monitors from most common constructs provided by its specification language. However, more work is required to

[6] http://www.libtom.net/LibTomCrypt/

support the E-ACSL specification language in full. The second direction of future work relates to application of static analyses during the monitor generation phase. Even though E-ACSL already uses sound static techniques to reduce the amount of instrumentations (and consequently reduce runtime overheads), better static analyses are likely to improve the overheads results. Also these analyses and the E-ACSL code generator are quite complex pieces of code: future work includes their formalization and proof of correctness. Finally, E-ACSL lacks support for detecting concurrency issues such as deadlocks or data races. We are looking to address such concurrency problems in the future.

Acknowledgment

The authors would like to the anonymous reviewers from their helpful feedback. This work has received funding for the S3P project from French DGE and BPIFrance.

References

[1] Assaf, M., Signoles, J., Totel, E., Tronel, F.: Program transformation for non-interference verification on programs with pointers. In: International Information Security and Privacy Conference (SEC'13) (Jul 2013)

[2] Barany, G., Signoles, J.: Hybrid Information Flow Analysis for Real-World C Code. In: Tests and Proofs (TAP'17) (Jul 2017), to appear

[3] Bartocci, E., Bonakdarpour, B., Falcone, Y.: First international competition on software for runtime verification. In: Proceedings of the International Conference on Runtime Verification. LNCS, vol. 8734, pp. 1–9. Springer (2014)

[4] Bartocci, E., Falcone, Y., Bonakdarpour, B., 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. International Journal on Software Tools for Technology Transfer pp. 1–40 (Apr 2017)

[5] Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html

[6] 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'09). pp. 70–78 (May 2009)

[7] Common Weakness Enumeration: A community developed dictionary of software weakness types, http://cwe.mitre.org

[8] Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Symposium on Applied Computing (SAC'13) (Mar 2013)

[9] Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)

[10] Falcone, Y., Nickovic, D., Reger, G., Thoma, D.: Second international competition on runtime verification CRV 2015. In: Proceedings of the International Conference on Runtime Verification. LNCS, vol. 9333, pp. 405–422. Springer (2015)

[11] Herrmann, P., Signoles, J.: Annotation generation: Frama-C's RTE plug-in, http://frama-c.com/download/frama-c-rte-manual.pdf

[12] Jakobsson, A., Kosmatov, N., Signoles, J.: Rester statique pour devenir plus rapide, plus précis et plus mince. In: Journes Francophones des Langages Applicatifs (JFLA'15) (Jan 2015), in French

[13] Jakobsson, A., Kosmatov, N., Signoles, J.: Fast as a Shadow, Expressive as a Tree: Optimized Memory Monitoring for C. Science of Computer Programming pp. 226–246 (Oct 2016)

[14] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing 27(3), 573–609 (2015)

[15] Kosmatov, N., Petiot, G., Signoles, J.: An optimized memory monitoring for runtime assertion checking of C programs. In: Proceedings of the International Conference on Runtime Verification. pp. 167–182 (September 2013)

[16] Kosmatov, N., Prevosto, V., Signoles, J.: A lesson on proof of programs with Frama-C. In: International Conference on Tests and Proofs (TAP'13) (Jun 2013), invited Tutorial Paper

[17] Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: International Conference on Runtime Verification (RV'13) (Sep 2013)

[18] Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design, chap. 12, pp. 175–188. Springer (Oct 1999)

[19] Meyer, B.: Applying "Design by Contract". Computer 25(10) (Oct 1992)

[20] Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Notices 42(6), 89–100 (June 2007)

[21] Pariente, D., Signoles, J.: Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures. In: Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'17) (Jun 2017)

[22] 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). pp. 130–150 (Jul 2016)

[23] Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: A fast address sanity checker. In: Proceedings of the USENIX Annual Technical Conference. pp. 309–319. USENIX Association (June 2012)

[24] Seward, J., Nethercote, N.: Using Valgrind to detect undefined value errors with bit-precision. In: Proceedings of the USENIX Annual Technical Conference. pp. 17–30. USENIX (2005)

[25] Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language, http://frama-c.com/download/e-acsl/e-acsl.pdf

[26] Standard Performance Evaluation Corporation: SPEC CPU (2006), http://www.spec.org/benchmarks.html

[27] Stouls, N., Prevosto, V.: Aoraï plug-in tutorial, http://frama-c.com/download/frama-c-aorai-manual.pdf

[28] Vorobyov, K., Kosmatov, N., Signoles, J., Jakobsson, A.: Runtime detection of temporal memory errors. In: To appear in the Proceedings of the International Conference on Runtime Verification. Springer (September 2017)

[29] Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow state encoding for efficient monitoring of block-level properties. In: Proceedings of the International Symposium on Memory Management. pp. 47–58. ACM (June 2017)