Software Analyzers

Runtime Annotation Checking with Frama-C

The E-ACSL Plug-in

Julien Signoles

June 13, 2024 @ Frama-C Days

CEA-List, Université Paris-Saclay, Software Safety and Security Lab

Runtime Annotation Checking with Frama-C: The E-ACSL Plug-in

Thibaut Benjamin and Julien Signoles

Abstract Runtime Annotation Checking (RAC) is a lightweight formal method consisting in checking code annotations written in the source code during the program execution. While static formal methods aim for guarantees that hold for any execution of the analyzed program, RAC only provides guarantees about the particular execution it monitors. This allows RAC-based tools to be used to check a wide range of properties with minimum intervention from the user. Frama-C can perform RAC on C programs with the plug-in E-ACSL. This chapter presents RAC through practical use with E-ACSL, shows advanced uses of E-ACSL leveraging the collaboration with other plug-ins, and sheds some light on the internals of E-ACSL and the technical difficulties of implementing RAC.

Key words: runtime annotation checking, inline monitoring, dynamic analysis, memory debugging.

Runtime Annotation Checking (RAC) [Huisman & Wijs, 2023]

"The basic idea of runtime annotation checking is that as a program is executed, every precondition and postcondition is checked by simply evaluating the predicate, followed by a test whether the outcome of this evaluation is true."

[1] Customizable behavior

Runtime Annotation Checking (RAC) [Huisman & Wijs, 2023]

"The basic idea of runtime annotation checking is that as a program is executed, every precondition and postcondition is checked by simply evaluating the predicate, followed by a test whether the outcome of this evaluation is true."

E-ACSL [Signoles et al, 2017] [Benjamin & Signoles, 2024]

Frama-C plug-in that takes as input a C program *p* and ACSL annotations and generates a new C code that monitors the annotations. When executed:

- behaves similarly to *p* if every ACSL annotation is valid;
- stops[1] on the first invalid annotation otherwise

[1] Customizable behavior

Outline

Running E-ACSL by Example

```c
#include <stdio.h>
#include <stdlib.h>

int main () {
  int *a, *b;
  a = (int *) malloc (10 * sizeof (int));
  b = (int *) malloc (3 * sizeof (int));
  for(int i = 0; i <= 10; i++) {
    //@ assert (i < 10);
    a[i] = i;
  }
  printf ("Done!\n");
  return 0;
}
```

```
> e-acsl-gcc.sh -c first.c
> ls -1
a.out          // normal binary (as compiled by gcc)
a.out.e-acsl   // monitored binary after E-ACSL instrumentation
a.out.frama-c  // monitored C file generated by E-ACSL
 first.c       // user source file
```

```
> ./a.out.e-acsl
 first.c: In function 'main'
 first.c:9: Error: Assertion failed:
          The failing predicate is:
          i < 10.
          With values at failure point:
          - i: 10
Abandon (core dumped)
```

`e-acsl-gcc-sh`: convenient script that calls Frama-C and the C compiler appropriately

It also works on more complex specifications!

```
1  /*@ requires \valid(a+(0..length-1));
2    @ requires \forall integer i, j;
3    @    0 <= i <= j < length ==> a[i] <= a[j];
4    @ requires length >=0;
5    @ behavior exists:
6    @   assumes \exists integer i; 0<=i<length && a[i] == key;
7    @   ensures 0<=\result<length && a[\result] == key;
8    @ behavior not_exists:
9    @   assumes \forall integer i; 0<=i<length ==> a[i] != key;
10   @   ensures \result == -1;
11   @ complete behaviors;
12   @ disjoint behaviors; */
13 int binary_search(int* a, int length, int key) {
```

```
18   while (low<high) { // instead of low <= high
```

```
27 int main() {
28   int t[5] = { 1, 2, 3, 4, 5 };
29   return binary_search(t, 5, 5);
30 }
```

```
> e-acsl-gcc.sh -c search.c
> ./a.out.eacsl
search.c: In function 'binary_search'
search.c:7: Error: Postcondition failed:
  The failing predicate is:
  exists:
    0 <= \result < \old(length)
    && *(\old(a) + \result) == \old(key).
  With values at failure point:
  - \result: -1
Abandon (core dumped)
```

- checking unproved properties of static analyzers (e.g., Eva, WP)
- extending test suites with monitoring for catching hardly-observable defects
- checking non-ACSL properties, automatically, with the help of dedicated plug-ins
  - absence of undefined behaviors (RTE)
  - ordering of function calls and returns (Aoraï)
  - system level properties (MetACSL)
  - Virgile Prevosto's talk this afternoon!
- checking a few other properties automatically
  - format string in printf- or scanf-like functions
  - calls to critical libc functions, e.g. `memset` or `memcpy`
  - memory consumption

Checking Undefined Behaviors: Example

checking undefined behaviors automatically?

just give `--rte=all` to `e-acsl-gcc.sh`

```c
1 int main ()
2 {
3   int size = 3;
4   int p[size];
5   for (int i = 0; i <= 3; i++)
6     p[i] = 0;
7   return 0;
8 }
```

```
> e-acsl-gcc.sh -c --rte=all search.c
> ./a.out.eacsl
undef.c: In function 'main'
undef.c:6: Error: Assertion failed:
        The failing predicate is:
        rte/mem_access:
                \valid(p + i).
        With values at failure point:
        - rte: mem_access: \valid(p + i): 0
        - sizeof(int): 4
        - i: 3
        - p: 0x7ffcaffdb010
Abandon (core dumped)
```

RAC is a lightweight formal method

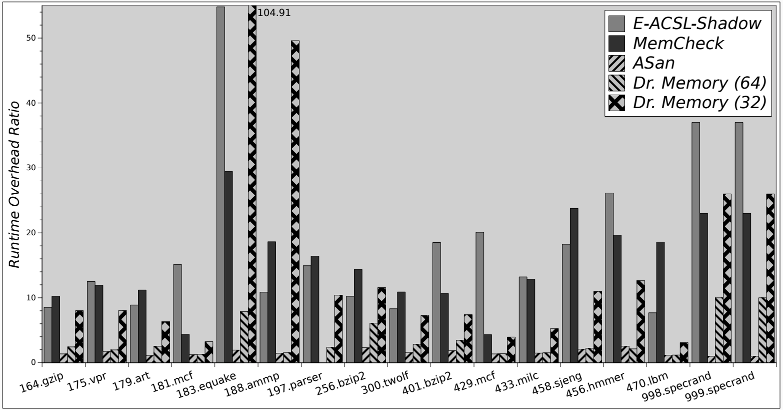
criteria for evaluating runtime (annotation) checkers:

- expressivity: the more formal properties a RAC tool is able to check, the better.
- transparency: the instrumentation should not interfere with the behavior of the original program, beyond interrupting the execution when detecting an invalid property.
- soundness: the instrumented program should check the annotations accurately (always detects the bug)
- correctness = transparency + soundness
- efficiency: to be practical, it is necessary to limit the time and memory overheads induced by the instrumentation.

Expressivity for Undefined Behaviors

Defect Type	E-ACSL	Google's Sanitizers
Dynamic Memory	94% (81/86)	78% (67/86)
Static Memory	✓ (67/67)	96% (64/67)
Pointer-related	56% (47/84)	32% (27/84)
Stack-related	35% (7/20)	70% (14/20)
Resource	99% (95/96)	60% (58/96)
Numeric	93% (100/108)	59% (64/108)
Miscellaneous	94% (33/35)	49% (17/35)
Inappropriate Code	– (0/64)	– (0/64)
Concurrency	– (0/44)	73% (32/44)
Overall	71% (430/604)	57% (343/604)

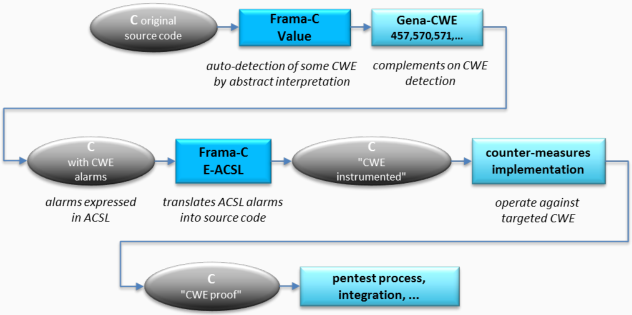
Detection Capabilities over Toyota ITC Benchmark [Vorobyov et al, 2018]



$\times$17 time-overhead; $\times$2.4 memory overhead on SPEC-CPU

speed comparable to Valgrind; slower than AddressSanitizer
less memory-overhead than these tools [Vorobyov et al, 2017]

Use Case: Generating Security Counter-Measures



[Pariente & Signoles, 2017]

First, use automatic static analysis to detect vulnerabilities

Then, switch to fast runtime monitoring

Experimented on modules from Apache/OpenSSL

RAC is a compilation technique

- RAC compiles assertions into executable code
  - input: `/*@ assert x+1 == 0; */`
  - output: `assert (x+1 == 0);`
- may look straightforward
  - "The run-time checker [of Spec#] is straightforward" [Barnet et al., 2011]

RAC is a compilation technique

- RAC compiles assertions into executable code
  - input: `/*@ assert x+1 == 0; */`
  - output: `assert (x+1 == 0);`
- may look straightforward
  - “The run-time checker [of Spec#] is straightforward” [Barnet et al., 2011]
- really straightforward??
  - maybe not: “the run-time overhead [of Spec#] is prohibitive” [Barnet et al., 2011]
  - maybe not: the example above is unsound, in general

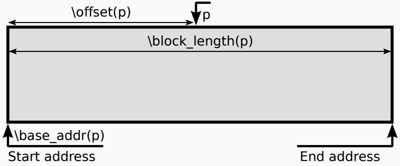
dedicated library (GMP in C) for integers and rationals

```
/*@ assert x + 1 == 0; */
mpz_t e_acsl_1, e_acsl_2, e_acsl_3, e_acsl_4;
int e_acsl_5;
mpz_init_set_si(e_acsl_1, x);                 // e_acsl_1 = x
mpz_init_set_si(e_acsl_2, 1);                 // e_acsl_2 = 1
mpz_init(e_acsl_3);
mpz_add(e_acsl_3, e_acsl_1, e_acsl_2);   // e_acsl_3 = x + 1
mpz_init_set_si(e_acsl_4, 0);                 // e_acsl_4 = 0
e_acsl_5 = mpz_cmp(e_acsl_3, e_acsl_4); // x + 1 == 0
e_acsl_assert(e_acsl_5 == 0);                // runtime check
mpz_clear(e_acsl_1); mpz_clear(e_acsl_2); // deallocate
mpz_clear(e_acsl_3); mpz_clear(e_acsl_4);
```

sound [Benjamin & Signoles, 2023a] but not efficient

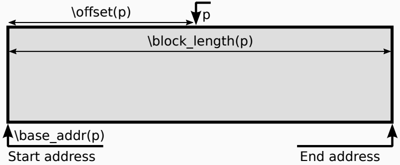
- dedicated type system [Kosmatov et al, 2020], extended to an abstract interpreter [Benjamin & Signoles, 2023b] for being sound and efficient
  - use machine bounded numbers and arithmetic whenever possible
  - use GMP otherwise
- only a few GMPs integers in practice
  - very efficient in practice
- implemented in E-ACSL for integer and rational numbers

- how to compile `\valid(p)` or `\initialize(p)`?
- standard solution: shadow memory
  - implemented in memory debuggers, e.g., Address Sanitizer [Serebryany et al, 2012]
  - cannot evaluate block-level properties



Compiling Memory Properties

- how to compile `\valid(p)` or `\initialize(p)` ?
- standard solution: shadow memory
  - implemented in memory debuggers, e.g., Address Sanitizer [Serebryany et al, 2012]
  - cannot evaluate block-level properties



- E-ACSL's custom shadow memory [Vorobyov et al, 2017]
- issue: heavy instrumentation, so not very efficient
- solution: dedicated dataflow analysis [Ly et al, 2018]
  - monitor only the over-approximated necessary memory locations

- using E-ACSL is quite easy, yet find hard-to-catch bugs
  - can be combined efficiently with plug-ins generating ACSL annotations
- scientific challenge: be expressive, sound and efficient altogether
  - mathematical numbers
    - integers
    - rational numbers
    - what about real numbers?
  - memory properties
    - `assigns` clauses? [Lehner, 2011]
    - what about concurrency?
  - multi-state properties, i.e. `\old` and `\at`
    - partial solutions do exist, the most recent being [Filliâtre & Pascutto, 2022]
    - one is implemented in E-ACSL (unpublished), can be improved
  - inductive and axiomatic definitions?
    - will be in Frama-C 30-Zinc, up to some extend
  - more optimizations

- M. Huisman, A. Wijs *A Concise Guide to Software Verfication. From Model Checking to Annotation Checking.* Springer Nature, 2023
- J. Signoles, N. Kosmatov, K. Vorobyov *E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper.* Int. Work. on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES), 2017
- T. Benjamin, J. Signoles *Runtime Annotation Checking with Frama-C: The E-ACSL Plug-in.* Guide to Software Verification with Frama-C: Core Components, Usages, and Applications. N. Kosmatov, V. Prevosto, J. Signoles, eds, 2024
- K. Vorobyov, N. Kosmatov, and J. Signoles *Detection of Security Vulnerabilities in C Code using Runtime Verification.* Int. Conf. on Tests and Proofs (TAP), 2018
- K. Vorobyov, J. Signoles, and N. Kosmatov *Shadow State Encoding for Efficient Monitoring of Block-level Properties.* Int. Symp. on Memory Management (ISMM), 2017
- D. Pariente and J. Signoles *Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures.* Symp. sur la Sécurité des Technologies de l'Information et des Communications (SSTIC), 2017
- M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, H. Venter *Specification and Verification: The Spec# Experience.* Communications of the ACM, 2011
- T. Benjamin, J. Signoles *Formalizing an Efficient Runtime Assertion Checker for an Arithmetic Language with Functions and Predicates.* Int. Symp. on Applied Computing (SAC), 2023
- N. Kosmatov, F. Maurica, and J. Signoles *Efficient Runtime Assertion Checking for Properties over Mathematical Numbers.* Int. Conf. on Runtime Verification (RV), 2020
- T. Benjamin, J. Signoles *Abstract Interpretation of Recursive Logic Definitions for Efficient Runtime Assertion Checking.* Int. Conf. on Tests and Proofs (TAP), 2023
- K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov *AddressSanitizer: A Fast Address Sanity Checker.* Annual Technical Conf. (ATC), 2012
- D. Ly, N. Kosmatov, F. Loulergue, and J. Signoles *Soundness of a dataflow analysis for memory monitoring.* Work. on Languages and Tools for Ensuring Cyber-Resilience in Critical Software-Intensive Systems (HILT), 2018
- H. Lehner *A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking.* PhD Thesis, ETH Zürich, 2011
- J.-C. Filliâtre, C. Pascutto *Optimizing Prestate Copies in Runtime Verification of Function Postconditions* Int. Conf. on Runtime Verification (RV), 2022