Runtime Assertion Checking and its Combinations with Static and Dynamic Analyses

Nikolay Kosmatov and *Julien Signoles*

Tests & Proofs 2014 Tutorial

July 25th, 2014

Motivation

Runtime verification of
rigorous, mathematical **semantic properties of a C program**

- safety properties:
  - no division by zero
  - no arithmetic overflow
  - validity of memory accesses
  - ...
- functional properties:
  - function preconditions must be satisfied by the caller
  - function postconditions must be satisfied by the callee
  - ...
- ...

Our goal

In this tutorial, we will see:

- how to specify a C program with the E-ACSL specification language
- how to detect errors at runtime with the E-ACSL plug-in of Frama-C
- how to combine runtime verification with other analyses

Presentation of Frama-C

- Frama-C Overview
- E-ACSL

Runtime Verification

- Assertions
- Function Contracts
- Integers
- Memory-Related Annotations

Combinations with Other Analyzers

- Runtime Errors
- Tests Generation and RAC
- Deductive Method and RAC
- Abstract Interpretation and RAC

- Presentation of Frama-C
  - Frama-C Overview
  - E-ACSL
- Runtime Verification
  - Assertions
  - Function Contracts
  - Integers
  - Memory-Related Annotations
- Combinations with Other Analyzers
  - Runtime Errors
  - Tests Generation and RAC
  - Deductive Method and RAC
  - Abstract Interpretation and RAC

Frama-C at a glance

- Framework of Analyses of ISO C 99 Code
- Developed at CEA LIST (Software Security labs) and INRIA Saclay (Toccata team).
- Released under LGPL license (Neon, March 2014)
- ACSL annotation language.
- Plug-in based extensible platform
  - Collaboration of analyses over same code
  - Inter plug-ins communication through ACSL formulae.
  - Adding new (open/close-source) plug-ins is easy
- Used in several industrial contexts

http://frama-c.com

ACSL: ANSI/ISO C Specification Language

- like JML or Spec# for C programs
- based on Eiffel-like contracts
- allows the users to specify behavioral functional properties of their programs
- designed for static analyzers
- independent from a particular analysis/tool
- lingua franca of Frama-C analyzers

http://frama-c.com/acsl

- first-order logic
- pure C expressions (side-effect-free expressions)
- C types + $\mathbb{Z}$ (integer) and $\mathbb{R}$ (real)
- built-ins predicates and logic functions, particularly over pointers:
  - `\valid(p)`
  - `\valid(p+0..2)`,
  - `\separated(p+0..2,q+0..5)`,
  - `\block_length(p)`
  - ...
- ...

E-ACSL: Executable-ACSL

E-ACSL, a specification language

- (large) executable subset of ACSL
- annotations may be evaluated at runtime

Main differences with ACSL:

- remove unexecutable ACSL constructs (e.g. axiomatics)
- compatible semantics changes

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

Benefits:

- being executable allows to be understandable by dynamic tools (testing tools, monitors)
- being based on ACSL allows to be supported by existing Frama-C analyzers
- being translatable into C allows to be supported by other analysis tools for C

E-ACSL, a Frama-C plug-in

- converts an annotated C program p into another one p'
- p' fails at runtime whenever an annotation is violated
- p' and p have the same functional behavior if no annotation is violated

What and why?

- ensure properties at some program points
- defensive programming

How?

- C macro `assert` provided by `assert.h`
  - takes a C expression of type `int` as argument
- E-ACSL clause `assert`
  - takes an E-ACSL predicate as argument
  - much more expressive than C "boolean" expressions

Example 1: max

```c
int max(int x, int y) {  return x<y ? x : y;  }

int main(void) {
  int m = max(0, 0);
  /*@ assert m == 0; */ // assert(m == 0);
  m = max(-4, 3);
  /*@ assert m == 3; */ // assert(m == 3);
  return 0;
}
```

- generate the C code in file `a.c` with:

```
frama-c -e-acsl max.c \
        -then-on e-acsl -print -ocode a.c
```

Function Contract

Principle

- goal: specification of imperative functions
- approach: give assertions (i.e. properties) about the functions
  - precondition is supposed to be true on entry (ensured by callers of the function)
  - postcondition must be true on exit (ensured by the function if it terminates)
- nothing is guaranteed when the precondition is not satisfied
- termination may or may not be guaranteed (total or partial correctness)

- the precondition is verified when entering the function
- the postcondition is verified when exiting the function
- the contract is thus verified for each function call

`absval` computes the absolute value of its argument.

```
/*@ ensures (x >= 0 ==> \result == x)
  @       && (x <  0 ==> \result == -x); */
int absval(int x) { return x>0 ? x : -x; }
```

- that is actually wrong when the argument is `INT_MIN`.

frama C Software Analyzers

Example 2: absval

`absval` computes the absolute value of its argument.

```
/*@ ensures (x >= 0 ==> \result == x)
  @       && (x <  0 ==> \result == -x); */
int absval(int x) { return x>0 ? x : -x; }
```

- that is actually wrong when the argument is `INT_MIN`.

```
#include <limits.h>

/*@ requires x > INT_MIN;
  @ ensures (x >= 0 ==> \result == x)
  @      && (x <  0 ==> \result == -x); */
int absval(int x) { return x>0 ? x : -x; }
```

- preprocessing annotations requires to use the option `-pp-annot`

Behaviors

- Global precondition (**requires**) and postcondition (**ensures**) apply to all cases
- Behaviors refine global contract in particular cases
- For each **behavior** (case):
  - the subdomain is defined by **assumes** clause
  - additional constraints are given with local **requires** clauses
  - the behavior's postcondition is defined by **ensures** clauses, ensured whenever **assumes** condition is true
- **complete behaviors** states that given behaviors cover all cases (not supported by the E-ACSL plug-in yet)
- **disjoint behaviors** states that given behaviors do not overlap (not supported by the E-ACSL plug-in yet)

Example 2: absval

Solution, improved

```c
#include <limits.h>

/*@ requires x > INT_MIN;
  @ behavior pos:
  @    assumes x >= 0;
  @    ensures \result == x;
  @ behavior neg:
  @    assumes x < 0;
  @    ensures \result == -x;
  @ complete behaviors;
  @ disjoint behaviors; */
int absval(int x) { return x>0 ? x : -x; }
```

- ACSL and E-ACSL use mathematical integers
- many advantages compared to bounded integers
  - automatic theorem provers work much better with such integers than with bounded integers arithmetics
  - specify without implementation details in mind
  - still possible to use bounded integers when required
  - much easier to specify overflows
- yet runtime computations may be more difficult

- E-ACSL uses GMP to represent mathematical integers
- try to avoid them as much as possible (interval-based type system)
- no GMP in the previous examples
- indeed few GMP's in practice
- only used when the annotations talk about (potentially) very big integers
- in such a case, the generated code must be linked against GMP

Example 3: pow

partial specification of pow

```
/*@ ensures \result > 0;
  @ behavior pos:
  @   assumes x > 0;
  @   ensures \result % x == 0;
  @   ensures (\result + 1) % x == 1; */
unsigned long long my_pow
  (unsigned int x, unsigned int n)
```

- the generated program requires GMP

- E-ACSL provides several built-in predicates to talk about pointers
- `\valid(p)`: is p valid?
- `\initialized(p)`: is $*p$ initialized?
- `\base_addr(p)`: base address of the block containing p
- `\block_length(p)`: length of the block containing p
- `\offset(p)`: offset of p from `base_addr(p)`
- also provides assigns clause to talk about memory locations which may change (not supported by the E-ACSL plug-in yet).

Refering to another state

- specifications may require values at different program points
- `\at(e,L)` refers to the value of expression `e` at label `L`
- some predefined labels:
  - `\at(e,Here)` refers to the current state
  - `\at(e,Old)` refers to the pre-state
  - `\at(e,Post)` refers to the post-state
- `\old(e)` is equivalent to `\at(e,Old)`

Example 4: swap

```
/*@ requires \valid(p);
  @ requires \valid(q);
  @ ensures *p == \old(*q);
  @ ensures *q == \old(*p);
  @ assigns *p \from \old(*q);
  @ assigns *q \from \old(*p); */
void swap(int *p, int *q)
/* { ... } */
```

- the generated code is machine-dependent: add `-machdep x86_64` on an x86-64 architecture
- the generated program must be linked against the E-ACSL memory library
- E-ACSL tries to minimize the instrumentation (dataflow analysis)

Quantification

- E-ACSL is based on a first order logic
- it provides finite existential and universal quantifications over terms
- quantifications must be guarded

$$\texttt{\textbackslash forall}\ \tau\ x_1, \ldots, x_n;\quad a_1 \mathrel{<=} x_1 \mathrel{<=} b_1\ \&\&\ \ldots\ \&\&\ a_n \mathrel{<=} x_n \mathrel{<=} b_n \ \texttt{==>}\ \texttt{p}$$

$$\texttt{\textbackslash exists}\ \tau\ x_1, \ldots, x_n;\quad a_1 \mathrel{<=} x_1 \mathrel{<=} b_1\ \&\&\ \ldots\ \&\&\ a_n \mathrel{<=} x_n \mathrel{<=} b_n \ \&\&\ \texttt{p}$$

Example 5: sum of matrices

A more advanced example about pointers and quantification

```
typedef int* matrix;

/*@ requires size >= 1;
  @ requires \forall integer i, j;
  @    0 <= i < size && 0 <= j < size ==>
  @    \valid(a+i*size+j) && \valid(b+i*size+j);
  @ ensures \forall integer i, j;
  @    0 <= i < size && 0 <= j < size ==>
  @    \valid(\result+i*size+j) &&
  @    \result[i*size+j] ==
  @      a[i*size+j]+b[i*size+j];
  @ */
matrix sum(matrix a, matrix b, int size);
```

Example 5: sum of matrices

Error detection

Which memory errors are we able to detect here?

- spatial error: invalid memory access due to out-of-bounds offset or array index
- temporal error: invalid memory access to a deallocated memory object (use after free)
- memory leak: use more memory at the end of the execution than at the beginning.
  - use the special variable `__memory_size`

Generating Annotations Automatically

C code may have runtime errors

- Frama-C plug-ins may generate annotations
- the RTE plug-in generates an annotation for each potential runtime error
- possible to run RTE, then to run E-ACSL
- automatic detection of each runtime error

Errors in annotations?

- ACSL logic is total and 1/0 is logically significant
  - help the user to write simple specification like u/v == 2
  - 1/0 is defined but not executable
- E-ACSL logic is 3-valued
  - the semantics of 1/0 is "undefined"
  - lazy operators `&&`, `||`, `_?_:_`, `==>`
  - correspond to Chalin's Runtime Assertion Checking semantics
  - consistent with ACSL: valid (resp. invalid) E-ACSL predicates remain valid (resp. invalid) in ACSL
  - Evaluating an undefined term must not crash

E-ACSL & RTE in a whole

Initial user-annotated C Code

RTE plug-in

Initial C Code with RTE annotations

E-ACSL plug-in

Instrumented C Code

RTE plug-in on generated C expression

Intrusmented C Code with RTE annotations

E-ACSL plug-in on each new annotation

Instrumented C Code preventing RTEs

E-ACSL & RTE in a whole

Initial user-annotated C Code

RTE plug-in

Initial C Code with RTE annotations

E-ACSL plug-in

Instrumented C Code

RTE plug-in on generated C expression

Intrusmented C Code with RTE annotations

E-ACSL plug-in on each new annotation

Instrumented C Code preventing RTEs

E-ACSL & RTE in a whole

Initial user-annotated C Code

↓ RTE plug-in

Initial C Code with RTE annotations

→ E-ACSL plug-in

Instrumented C Code

↓ RTE plug-in on generated C expression

Intrusmented C Code with RTE annotations

← E-ACSL plug-in on each new annotation

Instrumented C Code preventing RTEs

E-ACSL & RTE in a whole

Initial user-annotated C Code

↓ RTE plug-in

Initial C Code with RTE annotations

→ E-ACSL plug-in

Instrumented C Code

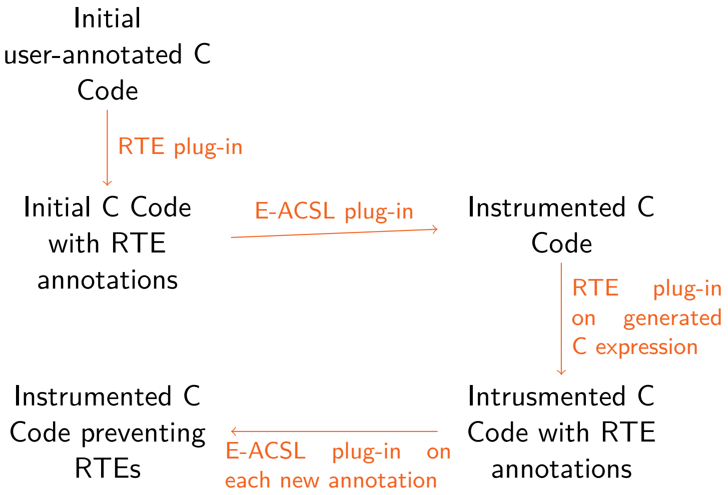
↓ RTE plug-in on generated C expression

Intrusmented C Code with RTE annotations

← E-ACSL plug-in on each new annotation

Instrumented C Code preventing RTEs

E-ACSL & RTE in a whole



Example 6: is_dividable

dividability of elements of arrays

no need of writing assertions since RTE generates them

```
/*@ requires \forall integer i; 0 <= i < len ==>
        \valid(num+i) && \valid(denum+i)
        && \valid(result+i);
  @ ensures  \forall integer i; 0 <= i < len ==>
        result[i] == (num[i] % denum[i] == 0 ? 1 : 0);
  @*/
void is_dividable
  (int *num, int *denum, int *result, int len) {
  for(int i = 0; i < len; i++)
    if (num[i] % denum[i] == 0) result[i] = 1;
    else result[i] = 0;
}
```

Mixing E-ACSL and PathCrawler

Runtime assertion checking e.g. E-ACSL

+ provides a powerful tool to detect various kinds of errors
+ supports expressive specifications and provides an unambiguous verdict
- requires (representative) test inputs to run the code with

Structural test generation e.g. PathCrawler

- may have restricted support of errors and specification features (due to symbolic execution, its memory model, . . .)
- cannot always provide a verdict automatically
+ can generate a test suite with a rigorous coverage

Combine E-ACSL and PathCrawler to check the specification at runtime on a test suite with a rigorous coverage

Verifying Annotations Statically

- Frama-C comes with various static analyzers
- some aim at statically verifying a program
  - may guarantee the absence of runtime error
  - may ensure that a program satisfies its ACSL specification
- usually require extra work by the user
  - adding extra annotations (assertions, loop invariants, etc)
  - parameterizing the tool
  - writing stubs
- what to do when all the code is not statically verified?

may also use E-ACSL on such cases

Plug-in Wp

- based on Dijkstra's weakest preconditon calculus
- generates theorems (proof obligations) to ensure that a code satisfies its ACSL specification
- uses automatic/interactive theorem provers to verify these theorems
- is able to verify complex specifications
- requires to manually add extra annotations (e.g. loop invariants)

Mixing E-ACSL and Wp

Main ideas

- idea 1: dynamically check with E-ACSL the properties which are not statically proved with Wp.
- idea 2: use E-ACSL to test your specification before trying to prove it with Wp
  - use pre-existing test suites
  - write test cases manually
  - generate test cases with an automatic test generation tool like the PathCrawler plug-in of Frama-C
- the annotations proved by Wp are not converted by E-ACSL and so not checked at runtime (except if the option `-e-acsl-valid` is set)

goal: formally specify `binary_search` according to its informal specification

```
/* Takes as input a sorted array, its length,
   and an int to search for.
   Returns the index of a cell which contains
   the searched value.
   Returns -1 if the key is not present in the
   array. */
int binary_search(int *a, int length, int key);
```

Plugin Value

- based on Cousot's abstract interpretation
- computes over-approximations of possible values of variables at each program point
- evaluates simple E-ACSL annotations
- is able to statically ensure the absence of RTE
- generates extra E-ACSL annotations when it cannot guarantee the absence of RTE

Mixing altogether

- possible to combine Value, WP + E-ACSL
- even possible to send E-ACSL results back into Frama-C

Time for the final demo!

Conclusion

We have seen:

- how to specify a C program with the E-ACSL specification language
- how to detect errors at runtime with the E-ACSL plug-in of Frama-C
- how to combine E-ACSL with other analyses
  - RTE
  - WP
  - Value
  - PathCrawler

Bibliography

- M. Delahaye, N. Kosmatov, and J. Signoles.
  Common specification language for static and dynamic analysis of C programs.
  Symposium on Applied Computing 2013 (SAC'13).
- N. Kosmatov, G. Petiot, and J. Signoles.
  An optimized memory monitoring for runtime assertion checking of C programs.
  Runtime Verification 2013 (RV'13).
- P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski.
  Frama-c: a Software Analysis Perspective.
  Software Engineering and Formal Methods 2012 (SEFM'12).
- L. Correnson and J. Signoles.
  Combining Analyses for C Program Verification.
  Formal Methods for Industrial Case Studies (FMICS'12).