

Runtime Assertion Checking with Frama-C

Nikolay Kosmatov and *Julien Signoles*



list

Runtime Verification 2013 Tutorial

September 24th, 2013

(long no
[for it < C
C); if (0
tmp2 =
st of the

tmp2[j] = (t <= (Nb1 - 1)) ? tmp1[j] : (t <= (Nb1 - 1)) ? tmp2[j] : (t <= (Nb1 - 1)) ? tmp1[j] : tmp2[j]; /* Then the second pass looks like the first one: */
tmp1[0] = 0; k = 0; k++ tmp1[j] = mc2[0][k] * tmp2[k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
i = 1; tmp1[0] >= 1; /* Final rounding: tmp2[0] is now represented on 9 bits. */ if (tmp1[0] < -256) tmp2[0] = -256; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] = tmp1[0];



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
- ...

► ...



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 **customize** the runtime verification
- ▶ how to **combine** runtime verification with other analyses

(long no
for i <= 4
C1) if (0)
tmp2 =
st of the

tmp2[0] = (t <= (Nb1 - 1)) else if (tmp1[0]) >= (t <= (Nb1 - 1)) tmp2[0] = (t <= (Nb1 - 1)) - 1; else tmp2[0] = tmp1[0]; /* Then the second part looks like the first one: "Nb1 - 1" is replaced by "Nb2 - 1" and "Nb1" by "Nb2".
tmp1[0] = 0; k = 0; k++ tmp1[0] = mc2[0][k] * tmp2[k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*(MC1 - 1) + MC2*M1
i = 1; tmp1[0] >= 1; /* Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] < -256) tmp2[0] = -256; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] = tmp1[0];



Presentation of Frama-C

Context

Frama-C Overview

ACSL and E-ACSL

First Steps

Runtime Verification

Checking Assertions

Function Contract

Integers

Errors in Annotations

Memory-Related Annotations

Customization

Runtime Monitor Behavior

Incomplete Program

Combinations with Other Analyzers

Generating Annotations Automatically

Mixing Static Verification and Runtime Assertion Checking



Presentation of Frama-C

Context

Frama-C Overview

ACSL and E-ACSL

First Steps

Runtime Verification

Checking Assertions

Function Contract

Integers

Errors in Annotations

Memory-Related Annotations

Customization

Runtime Monitor Behavior

Incomplete Program

Combinations with Other Analyzers

Generating Annotations Automatically

Mixing Static Verification and Runtime Assertion Checking



- ▶ 90's: CAVEAT, an Hoare logic-based tool for C programs at CEA
- ▶ 2000's: CAVEAT used by Airbus during certification process of the A380 (DO-178 level A qualification)
- ▶ 2002: Why and its C front-end Caduceus at INRIA
- ▶ 2006: Joint project to write a successor to CAVEAT and Caduceus
- ▶ 2008: First public release of Frama-C (Hydrogen)
- ▶ today:
 - ▶ Frama-C Fluorine (v9)
 - ▶ Multiple projects around the platform
 - ▶ A growing community of users
 - ▶ and of plug-ins developers
 - ▶ Trust-In-Soft, startup based on Frama-C technologies



- ▶ A framework for modular analysis of C code.
- ▶ <http://frama-c.com>
- ▶ Developed at CEA LIST (Software Safety labs) and INRIA Saclay (Toccata team).
- ▶ Released under LGPL license (Fluorine v3 in June 2013)
- ▶ Kernel based on CIL (Necula et al. at Berkeley).
- ▶ ACSL annotation language.
- ▶ Extensible platform
 - ▶ Collaboration of analyses over same code
 - ▶ Inter plug-in communication through ACSL formulae.
 - ▶ Adding specialized plug-ins is easy





- ▶ Dassault's internal plug-ins [Pariante & Ledinot, FoVeOOs 2010]
- ▶ **Taster**: coding rules (Atos/Airbus) [Delmas & *al.*, ERTSS 2010]
- ▶ **Fan-C**: flow dependencies (Atos/Airbus) [Duprat & *al.*, ERTSS 2012]
- ▶ simple **Concurrency** plug-in (Adelard) [first release in 2013]
- ▶ various academic experiments (mostly security and/or concurrency related)
- ▶ others close private plug-ins (CEA, others companies in France, US, ...)



ACSL: ANSI/ISO C Specification Language

Presentation

- ▶ like JML or Spec# for C programs
- ▶ based on Eiffel-like contracts
- ▶ allows the users to specify functional properties of their programs
- ▶ designed for static analyzers
- ▶ already used in large-scale industrial projects
- ▶ allows communication between various plugins
- ▶ independent from a particular analysis
- ▶ ACSL manual at <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, a specification language

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

Differences with ACSL:

- ▶ few restrictions
- ▶ compatible semantics changes
- ▶ manual at

frama-c.com/download/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**

(long no
[for 0 <=
C1]; if (0
tmp2 =
st of the

tmp2[0] = (t <= (N-1) ? t : else if (tmp1[0] >= (t <= (N-1) ? t : else tmp2[0] = tmp1[0]; /* Then the second pass looks like the first one: "for (k = 0; k < 8; k++) tmp1[0][k] = mc2[0][k] * tmp2[k][0] / 7" The [i,j] coefficient of the matrix product MC2*TMP2, that is, *MC2*[i](TMP1) = MC2*[i](MC1*M1) = MC2*[i]M1 * MC1*[i] = 1 * tmp1[0][i] >= 1. /* Final rounding: tmp2[0][0] is now represented on 9 bits. */ if (tmp1[0][0] < -256) m2[0][0] = -256; else if (tmp1[0][0] > 255) m2[0][0] = 255; else m2[0][0] = tmp1[0][0];



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 behavior if no annotation is violated

(long no
[for it <= 0
C1); if (0)
tmp2 =
st of the

tmp2[0] = (t <= 0 ? (n1 - t) : else if (tmp1[0] >= (t <= 0 ? (n1 - t) : 0; else tmp2[0] = tmp1[0]; /* Then the second pass looks like the first one: "MC2" *
tmp1[0][k] = 0; k = 0; k++ tmp1[0][k] = mc2[0][k] * tmp2[k][0]; /* The [k] coefficient of the matrix product MC2*TMP2, that is: *MC2*[TMP1] = MC2*(MC1*M1) = MC2*M1 * MC1 *
l = 1; tmp1[0][l] >= 1; /* Final rounding: tmp2[0][0] is now represented on 9 bits. *if (tmp1[0][0] < -256) m2[0][0] = -256; else if (tmp1[0][0] > 255) m2[0][0] = 255; else m2[0][0] = tm



Linux:

- ▶ packages on Debian, Ubuntu, Fedora, ...
- ▶ compile from sources using OCaml package managers:
 - ▶ Godi: <http://godi.camlcity.org/godi/index.html>
 - ▶ Opam: <http://opam.ocamlpro.com/>

Windows:

- ▶ Godi
- ▶ Wodi: <http://wodi.forge.ocamlcore.org/>

Mac OS X:

- ▶ Binary package available
- ▶ Source compilation through homebrew



- ▶ this tutorial is mainly based on E-ACSL
- ▶ E-ACSL is a CEA's external Frama-C plug-in
- ▶ compile sources once Frama-C is installed
- ▶ <http://frama-c.com/download/e-acsl/e-acsl-0.3.tar.gz>

in this tutorial:

Frama-C Fluorine-20130601 + E-ACSL 0.3



Executables

- ▶ `frama-c`: console-based interface
- ▶ `frama-c-gui`: Graphical User Interface

Testing the installation

- ▶ `frama-c -help`: list of available plug-ins
- ▶ `frama-c -kernel-help`: options provided by the kernel
- ▶ `frama-c -e-acsl-help`: E-ACSL specific options



Manuals

- ▶ <http://frama-c.com/support.html>
- ▶ E-ACSL webpage: <http://frama-c.com/eacsl.html>
- ▶ ``frama-c -print-share-path`/manuals`
- ▶ `man frama-c`
- ▶ inline help
 - ▶ `frama-c -kernel-help`
 - ▶ `frama-c -plugin-help`

Support

- ▶ frama-c-discuss@gforge.inria.fr
- ▶ tag frama-c on <http://stackoverflow.com>



Presentation of Frama-C

Context

Frama-C Overview

ACSL and E-ACSL

First Steps

Runtime Verification

Checking Assertions

Function Contract

Integers

Errors in Annotations

Memory-Related Annotations

Customization

Runtime Monitor Behavior

Incomplete Program

Combinations with Other Analyzers

Generating Annotations Automatically

Mixing Static Verification and Runtime Assertion Checking



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



goal: check each value of `m` in function `main`.

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

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



Example 1: max

Solution 1: use C assertions

```
#include <assert.h>

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

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



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

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

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

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



- ▶ **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

```
(long n)
{ for (i = 0; i < n; i++)
  tmp2 = ...
  // ...
}
```

```
tmp2[i] = (i < (N-1) ? tmp1[i] : 0) else if (tmp1[i] >= 0) tmp2[i] = (i < (N-1) ? 0 : tmp1[i]); /* Then the second pass looks like the first one:
tmp1[0] = 0; k = 0; k++ tmp1[k] = mc2[k] * tmp2[k] / ^ The [k] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1 *
i = 1; tmp1[0] >= 1; */ Final rounding: tmp2[0] is now represented on 9 bits: *if (tmp1[0] < -256) tmp2[0] = -256; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] = tmp1[0];
```



goal:

specify function `absval` which computes the absolute value of its argument.

```
int absval(int x) { return x>0 ? x : -x; }
```

(long no
for it is
C1); if (0
tmp2 =
st of the

tmp2[0] = (t <= (Nb1 - 1)) ? tmp1[0] : (t <= (Nb1 - 1)) ? tmp2[0] : (t <= (Nb1 - 1)) ? tmp1[0] : tmp2[0]; /* Then the second pass looks like the first one: "MC1" */
tmp1[0] = 0; k = 0; k++ tmp1[0] = mc2[0][k] * tmp2[k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is: *MC2*[TMP1] = MC2*(MC1*M1) = MC2*M1*(MC1*M1) */
i = 1; tmp1[0] >= 1; /* Final rounding: tmp2[0] is now represented on 9 bits. */ if (tmp1[0] < -256) tmp2[0] = -256; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] = tmp1[0];



Example 2: absval

Solution

```
/*@ 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.



Example 2: absval

Solution, fixed

```
#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`



- ▶ 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
- ▶ **disjoint behaviors** states that given behaviors do not overlap



```
#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**




```
/*@ ensures \result > 0; */
unsigned long long my_pow
(unsigned int x, unsigned int n)
{
    unsigned long long res = 1;
    while (n) {
        if (n & 1) res *= x;
        n >>= 1;
        x *= x;
    }
    return res;
}
```

- ▶ the generated program **does not** require GMP



```
/*@ ensures \result > 0;
   @ behavior two:
   @   assumes n == 2;
   @   ensures \result % n == 0;
   @   ensures (\result + 1) % n == 1; */
unsigned long long my_pow
(unsigned int x, unsigned int n);
```

- ▶ the generated program **requires GMP**



- ▶ 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



goal: specify the following function

```
int is_dividable(int x, int y) {
    return x % y == 0;
}
```



Example 3: dividable

Solution

```
/*@ behavior yes:
    @   assumes x % y == 0;
    @   ensures \result == 1;
    @ behavior no:
    @   assumes x % y != 0;
    @   ensures \result == 0; */
int is_dividable(int x, int y) {
    return x % y == 0;
}
```

- ▶ $x \% y$ may be undefined
- ▶ if undefined, E-ACSL prevents its execution by **reporting an error**



Example 3: dividable

Solution, improved

```
/*@ requires y != 0;
   @ behavior yes:
   @   assumes x % y == 0;
   @   ensures \result == 1;
   @ behavior no:
   @   assumes x % y != 0;
   @   ensures \result == 0; */
int is_dividable(int x, int y) {
    return x % y == 0;
}
```

- ▶ adding an **extra annotation** is usually better
- ▶ make the requirement explicit



- ▶ 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)`



- ▶ specification may require **values at different program points**
- ▶ $\backslash\text{at}(e, L)$ refers to the value of expression e at label L
- ▶ some predefined labels:
 - ▶ $\backslash\text{at}(e, \text{Here})$ refers to the current state
 - ▶ $\backslash\text{at}(e, \text{Old})$ refers to the pre-state
 - ▶ $\backslash\text{at}(e, \text{Post})$ refers to the post-state
- ▶ $\backslash\text{old}(e)$ is equivalent to $\backslash\text{at}(e, \text{Old})$

(long no
for it
C1); if (0
tmp2 =
st of the

tmp2[0] = (t <= (Nb1 - 1)) ? else if (tmp1[0]) >= (t <= (Nb1 - 1)) ? else tmp2[0] = tmp1[0]; /* Then the second part looks like the first one: "Nb1
tmp1[0][k] = 0; k = 0; k++ tmp1[0][k] = mc2[0][k] * tmp2[k][0]; /* The [k] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*(MC1
l = 1; tmp1[0][l] >= 1; /* Final rounding: tmp2[0][0] is now represented on 9 bits. */ if (tmp1[0][0] < -256) m2[0][0] = -256; else if (tmp1[0][0] > 255) m2[0][0] = 255; else m2[0][0] = tmp1[0][0];



goal: specify the following function which swaps its arguments

```
void swap(int *p, int *q) {
    int tmp = *q;
    *q = *p;
    *p = tmp;
}
```



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

- ▶ 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)



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

```
\forall x_1, \dots, x_n;
  a_1 <= x_1 <= b_1 && ... && a_n <= x_n <= b_n
==> p
```

```
\exists x_1, \dots, x_n;
  a_1 <= x_1 <= b_1 && ... && a_n <= x_n <= b_n
&& p
```



Example 5: sum of matrices

A more advanced example about pointers and quantification

goal: specify the following function which sums two square matrices

```
typedef int* matrix;
matrix sum(matrix a, matrix b, int size);
```



```
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);
```



Which kind of error 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
- ▶ **memory leak:** use more memory at the end of the execution than at the beginning.
 - ▶ use the special variable `__memory_size`

(long n;
for (i = 0;
i < n; i++)
tmp2 =
sum of the

tmp2[i][j] = (i < n) ? (tmp2[i][j] >= 0 ? (i < n) ? tmp2[i][j] : 0 : else tmp2[i][j] = tmp1[i][j] ; /* Then the second part looks like the first one. */
tmp1[i][j] = 0; k = 0; k++ tmp1[i][j] = mc2[i][k] * tmp2[k][j] ; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*(MC1
i = 1; tmp1[i][j] >= 1; */ Final rounding: tmp2[i][j] is now represented on 9 bits. *if (tmp1[i][j] < -256) tmp2[i][j] = -256; else if (tmp1[i][j] > 255) tmp2[i][j] = 255; else tmp2[i][j] = tmp1[i][j];



- ▶ clause `loop invariant` before a loop body
- ▶ indicates **invariant properties** in a loop
- ▶ a loop invariant is **valid** if and only if:
 - ▶ it holds before entering the loop
 - ▶ it holds at the end of the loop body, after each iteration
- ▶ necessary annotations for **proof of programs** with loops

```
(long n)
{ for (i = 0; i < n; i++)
  C1; if (i < n)
    tmp2 = ...
  // at the
```

```
tmp2[i] = (i <= n-1) ? tmp[i] : tmp[i] >= 0 ? (i <= n-1) ? tmp[i] : (i <= n-1) ? 0 : else tmp2[i] = tmp1[i]; /* Then the second pass looks like the first one:
tmp1[0] = 0; k = 0; k++ tmp1[k] = mc2[0][k] * tmp2[k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
i = 1; tmp1[0] >= 1; /* Final rounding: tmp2[0] is now represented on 9 bits: *if (tmp1[0] < -256) tmp2[0] = -256; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] = tmp1[0];
```



goals:

- specify the following function which searches an element `elt` in a sorted global array `A`
- provide loop invariants

```
int A[10];

int search(int elt) {
    int k;
    for(k = 0; k < 10; k++)
        if(A[k] == elt) return 1;
        else if(A[k] > elt) return 0;
    return 0;
}
```




```
int A[10];

/*@ requires \forall integer i;
    @      0 <= i < 9 ==> A[i] <= A[i+1];
    @      behavior exists:
    @          assumes \exists integer j;
    @              0 <= j < 10 && A[j] == elt;
    @              ensures \result == 1;
    @      behavior not_exists:
    @          assumes \forall integer j;
    @              0 <= j < 10 ==> A[j] != elt;
    @              ensures \result == 0; */
int search(int elt);
```



```
#include "linear_search_spec.c"

int search(int elt) {
    int k;
    /*@ loop invariant 0 <= k <= 10;
       @ loop invariant \forall integer i;
       @ 0 <= i < k ==> A[i] < elt; */
    for(k = 0; k < 10; k++)
        if(A[k] == elt) return 1;
        else if(A[k] > elt) return 0;
    return 0;
}
```



Presentation of Frama-C

Context

Frama-C Overview

ACSL and E-ACSL

First Steps

Runtime Verification

Checking Assertions

Function Contract

Integers

Errors in Annotations

Memory-Related Annotations

Customization

Runtime Monitor Behavior

Incomplete Program

Combinations with Other Analyzers

Generating Annotations Automatically

Mixing Static Verification and Runtime Assertion Checking



- ▶ E-ACSL calls the function `e_acsl_assert` for each annotation
- ▶ by default, this function **fails iff the input boolean expression corresponding to the annotation is false** (i.e. 0)
- ▶ a failure generates an error message and aborts the execution
- ▶ possible to **customize the behavior** of the generated code by providing its **own definition** of `e_acsl_assert`



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

void e_acsl_assert
  (int predicate, char *kind, char *fct,
   char *pred_txt, int line)
{
  if (! predicate) {
    printf(
      "%s failed at line %d in function %s.\n\n"
      "The failing predicate is: \n%s.\n",
      kind, line, fct, pred_txt);
    exit(1);
  }
}
```



```
#include <stdio.h>

void e_acsl_assert
    (int predicate, char *kind, char *fct,
     char *pred_txt, int line)
{
    if (! predicate) {
        FILE *f = fopen("logfile.log", "a");
        fprintf(
            f,
            "%s failed at line %d in function %s.\n\
The failing predicate is:\n%s.\n",
            kind, line, fct, pred_txt);
        fclose(f);
    }
}
```



- ▶ possible to run E-ACSL on **code without a main** (e.g. library) or containing **undefined functions**
- ▶ correct only if there is no memory-related annotations
- ▶ **BE CAREFUL** with memory-related annotations:
 - ▶ may need the option `-e-acsl-full-mmodel` for correctness
 - ▶ less efficient generated code
 - ▶ if no main is provided, may need to call `__e_acsl_memory_init` (resp. `__e_acsl_memory_clean`) at the beginning (resp. at the end) of the main before linking



- ▶ 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**
- ▶ option **-e-acsl-prepare** must be used in case of running an analysis before E-ACSL

(long no
for it <= 0
C1); if (0)
tmp2 =
se of the

tmp2[0] = (t <= 0 ? (n1 - t) : else if (tmp1[0] >= 0) (t <= (n1 - t) ? (t <= (n1 - t) ? 0 : else tmp2[0] = tmp1[0]; /* Then the second part looks like the first one: MC2 *
tmp1[0] = 0; k = 0; k++) tmp1[0] = mc2[0][k] * tmp2[k]; /* The [i,j] coefficient of the matrix product MC2 * TMP2, that is, * MC2 * (TMP1) = MC2 * (MC1 * M1) = MC2 * M1 * MC1 *
i = 1; tmp1[0] >= 1; /* Final rounding: tmp2[0] is now represented on 9 bits: if (tmp1[0] < -256) tmp2[0] = -256; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] = tmp1[0];



again the example of **sum of matrices**:

no need of writing assertions since RTE generates them

```
int main(void) {
    int a[] = { 1, 1, 1, 1 };
    int b[] = { 2, 2, 2, 2 };
    matrix c = sum(a, b, 2);
    free(c);
    // /*@ assert \valid(&c[0]); */
    // /*@ assert \valid(&c[2]); */
    int trace = c[0] + c[2];
    return 0;
}
```



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
 - ▶ 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 precondition 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)



- ▶ **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)
- ▶ must use `-e-acsl-prepare` when running Wp



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



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

Time for the final demo!

```
(long no
{ for (i = 0; i < C; i++)
  tmp2 = ...
  // ...
}
```

```
tmp2[i] = (i < (Nb1 - 1)) ? tmp1[i] : tmp2[i]; /* Then the second pass looks like the first one:
tmp1[i] = 0; k = 0; k++ tmp1[i][k] = mc2[i][k] * tmp2[k]; /* The [i][k] coefficient of the matrix product MC2*TMP2, that is: *MC2*[i](TMP1) = MC2*[i](MC1*TM1) = MC2*TM1[i] * MC1[i][k]
i = 1; tmp1[i][0] >= 1; /* Final rounding: tmp2[i][0] is now represented on 9 bits: *if (tmp1[i][0] < -256) tmp2[i][0] = -256; else if (tmp1[i][0] > 255) tmp2[i][0] = 255; else tmp2[i][0] = tmp1[i][0];
```



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 **customize** E-ACSL
- ▶ how to **combine** E-ACSL with other analyses
 - ▶ RTE
 - ▶ WP
 - ▶ Value
 - ▶ PathCrawler



- ▶ **user manuals:** <http://frama-c.com/download.html>
- ▶ **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).

