Combined Static and Dynamic Analyses in Frama-C: An Overview

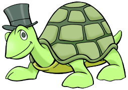
Nikolai Kosmatov

joint work with S.Bardin, B.Botella, O.Chebaro, M.Delahaye, A.Giorgetti, A.Jakobsson, J.Julliand, Y.Le Traon, M.Papadakis, G.Petiot, J.Signoles, K.Vorobyov. . .

2nd COST RV School, Praz-sur-Arly, March 20th, 2018

Static vs. Dynamic analysis techniques

- for a long time, seen as orthogonal and used separately
- more recently, realization of potential synergy and complementarity



Static analysis

Analyzes the source code without executing it

- Instructions reported as safe are safe (complete)
- Detected *potential* errors can be safe (imprecise)

Dynamic analysis

Executes the program on some test data

- Detected errors are really errors (precise)
- Cannot cover all executions (incomplete)

This talk presents some combinations of both approaches in Frama-C

Outline

Frama-C, a platform for analysis of C code

Accelerating runtime assertion checking (RAC) by static analysis (E-ACSL)

Detecting runtime errors by static and dynamic analysis (SANTE)

Deductive verification assisted by test generation and RAC (STADY)

Optimizing testing by value analysis and weakest precondition (LTest)

Conclusion

Outline

Frama-C, a platform for analysis of C code

Accelerating runtime assertion checking (RAC) by static analysis (E-ACSL)

Detecting runtime errors by static and dynamic analysis (SANTE)

Deductive verification assisted by test generation and RAC (STADY)

Optimizing testing by value analysis and weakest precondition (LTest)

Conclusion

A brief history

- 90's: CAVEAT, Hoare logic-based tool for C code 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)
- 2008: First public release of Frama-C (Hydrogen)
- Today: Frama-C v.16 Sulfur
  - Multiple projects around the platform
  - A growing community of users. . .
  - and of developers
- Used by, or in collaboration with, several industrial partners

Frama-C at a glance

Software Analyzers

- A Framework for Modular Analysis of C code
- Developed at CEA LIST
- Released under LGPL license
- Kernel based on CIL [Necula et al. (Berkeley), CC 2002]
- ACSL annotation language
- Extensible plugin oriented platform
  - Collaboration of analyses over same code
  - Inter plugin communication through ACSL formulas
  - Adding specialized plugins is easy
- `http://frama-c.com/` [Kirchner et al. FAC 2015]

ACSL: ANSI/ISO C Specification Language

- Based on the notion of contract, like in Eiffel, JML
- Allows users to specify functional properties of programs
- Allows communication between various plugins
- Independent from a particular analysis
- Manual at `http://frama-c.com/acsl`

Basic Components

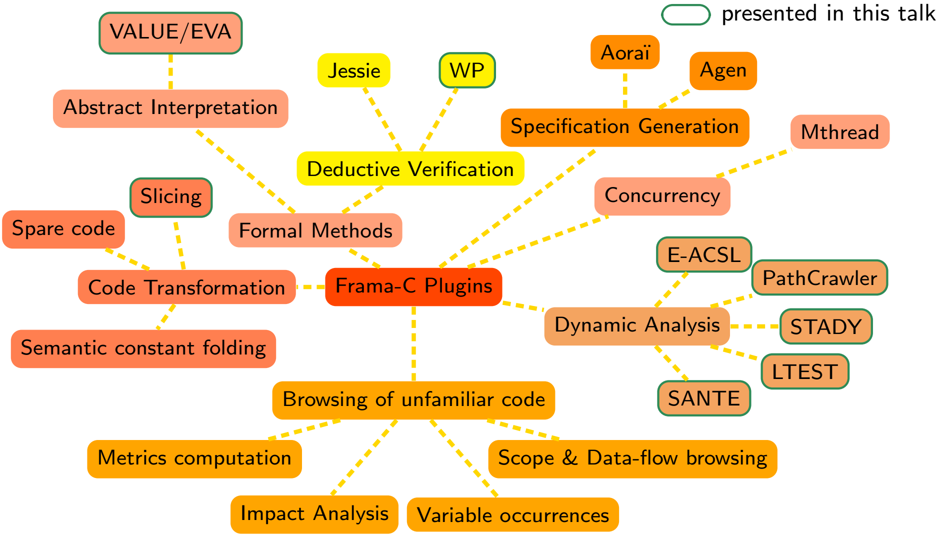
- First-order logic
- Pure C expressions
- C types + $\mathbb{Z}$ (integer) and $\mathbb{R}$ (real)
- Built-in predicates and logic functions particularly over pointers: `\valid(p) \valid(p+0..2)`, `\separated(p+0..2,q+0..5)`, `\block_length(p)`

Example: a C program annotated in ACSL

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Can be proven in Frama-C/WP

Main plugins

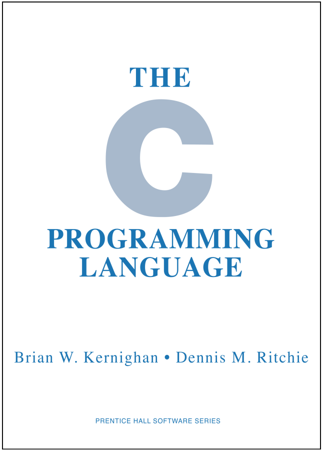


The C language is risky!

- Low-level operations
- Widely used for critical software
- Lack of security mechanisms

Runtime errors are common:

- Division by 0
- Invalid array index
- Invalid pointer
- Non initialized variable
- Out-of-bounds shifting
- Arithmetical overflow
- ...



Outline

From ACSL to E-ACSL

ACSL was designed for static analysis tools only

- based on logic and mathematics
- cannot execute any term/predicate (e.g. unbounded quantification)
- cannot be used by dynamic analysis tools (e.g. testing or monitoring)

E-ACSL: executable subset of ACSL [Delahaye et al, RV'13]

- few restrictions
- one compatible semantics change

E-ACSL Language

Executable subset of ACSL:

- it is verifiable in finite time, suitable for runtime assertion checking
- limitations: only bounded quantification, no axioms, no lemmas
- Includes builtin memory-related predicates, for a pointer *p*:

Builtin predicate	Description
`\valid(p)`	*p* is a valid pointer
`\initialized(p)`	$*p$ has been initialized
`\block_length(p)`	Length of *p*'s memory block
`\base_address(p)`	Base address of *p*'s memory block
`\offset(p)`	Offset of *p* in its memory block

[Delahaye et al. SAC 2013]

E-ACSL Integers

- mathematical integers to preserve ACSL semantics
- 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

E-ACSL plugin

The E-ACSL plugin is a runtime verification tool for E-ACSL specifications:

- it translates annotated program p into another program p'
- p' exits with error message if an annotation is violated
- otherwise p and p' have the same behavior

E-ACSL plug-in at a Glance

http://frama-c.com/eacsl.html

- convert E-ACSL annotations into C code
- implemented as a Frama-C plug-in

```
int sum2(int x, int y) {
  /*@ assert x+y>INT_MAX;*/
  return x + y;
}
```

⟶

```
int sum2(int x, int y) {
  /*@ assert x+y>INT_MAX;*/
  e_acsl_assert(x+y>INT_MAX);
  return x + y;
}
```

E-ACSL plug-in at a Glance

http://frama-c.com/eacsl.html

- convert E-ACSL annotations into C code
- implemented as a Frama-C plug-in

```
int sum2(int x, int y) {
  /*@ assert x+y>INT_MAX;*/
  return x + y;
}
```

$\longrightarrow$

```
int sum2(int x, int y) {
  /*@ assert x+y>INT_MAX;*/
  e_acsl_assert(x+y>INT_MAX);
  return x + y;
}
```

- the general translation is more complex than it may look

Example: an overflow in a C program

Operations in machine integers are bounded:

`x+y` cannot be more than `INT_MAX`

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

int sum2(int x, int y) {
  if( x + y > INT_MAX )
    printf("\n Overflow!! \n\n");
  int sum = x + y;
  return sum;
}

int main(){
  int x, y;
  x = INT_MAX; y = INT_MAX;
  printf("\n Sum of %d and %d is %d \n\n", x, y, sum2(x,y));
}
```

Cannot detect the overflow

Example: an overflow in a C program, cont'd

Operations in ACSL annotations have unbounded (mathematical) integer semantics: x+y can be more than INT_MAX

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

int sum2(int x, int y) {
  //@ assert x + y <= INT_MAX;
  int sum = x + y;
  return sum;
}

int main(){
  int x, y;
  x = INT_MAX; y = INT_MAX;
  printf("\n Sum of %d and %d is %d \n\n", x, y, sum2(x,y));
}
```

Should detect the overflow (unbounded integers)

E-ACSL and Unbounded Integer Support

- use GMP library for mathematical integers

```
/*@ assert x+y <= INT_MAX; */
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, y);              // e_acsl_2 = y
mpz_init(e_acsl_3);
mpz_sum(e_acsl_3, e_acsl_1, e_acsl_2);   // e_acsl_3 = x+y
mpz_init_set_si(e_acsl_4, INT_MAX);        // e_acsl_4=INT_MAX
e_acsl_5 = mpz_cmp(e_acsl_3, e_acsl_4); // (x+y) <= INT_MAX
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);
```

E-ACSL and Unbounded Integer Support

- use GMP library for mathematical integers

```
/*@ assert x+y <= INT_MAX; */
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, y);                // e_acsl_2 = y
mpz_init(e_acsl_3);
mpz_sum(e_acsl_3, e_acsl_1, e_acsl_2);       // e_acsl_3 = x+y
mpz_init_set_si(e_acsl_4, INT_MAX);          // e_acsl_4=INT_MAX
e_acsl_5 = mpz_cmp(e_acsl_3, e_acsl_4);      // (x+y) <= INT_MAX
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);
```

- how to restrict GMPs as much as possible? Use on-the-fly typing and longer types when possible
- almost no GMP in practice [Jakobsson et al, JFLA'15]

E-ACSL and Runtime Error (RTE) Detection

Translation should not introduce runtime errors

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  return u/v;
}
```

E-ACSL and Runtime Error (RTE) Detection

Translation should not introduce runtime errors

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  return u/v;
}
```

$\xrightarrow{\text{E-ACSL}}$

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

E-ACSL and Runtime Error (RTE) Detection

Translation should not introduce runtime errors

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  return u/v;
}
```

E-ACSL →

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

↓ RTE plug-in

```
int foo(int u, int v) {
  /*@ assert v != 0; */
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

E-ACSL and Runtime Error (RTE) Detection

Translation should not introduce runtime errors

```c
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  return u/v;
}
```

E-ACSL →

```c
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

↓ RTE plug-in

```c
int foo(int u, int v) {
  /*@ assert v != 0; */
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

← E-ACSL

```c
int foo(int u, int v) {
  /*@ assert v != 0; */
  e_acsl_assert(v != 0);
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

E-ACSL and Memory Monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;




/*@ assert len > 0 ; */

a_inv = malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */


  a_inv[len - i - 1] = a[i];  // array a inversed
}
free(a_inv);
```

Instrumented program (simplified)

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;




/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i, sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
free(a_inv);
```

Memory monitoring in E-ACSL

- The memory model of E-ACSL should contain all live allocations of the input program, with the necessary metadata
- E-ACSL runtime support library offers primitives to store and query such metadata
- All (de)allocations are instrumented with a call to the library

[Kosmatov et al. RV 2013; Jakobsson et al. SAC 2015; Vorobyov et al. ISMM 2017]

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(a);
```

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
• __store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(a);
```

Memory model: {(a, 16)}

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
• __store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(a);
```

Memory model: {(a, 16), (&len, 4)}

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(&len,4);
__store_block(&i,4);
__store_block(&a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(&a_inv);
__delete_block(&i);
__delete_block(&len);
__delete_block(a);
```

Memory model: {(a, 16), (&len, 4), (&i, 4)}

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(a);
```

Memory model: {(a, 16), (&len, 4), (&i, 4), (&a_inv, 4)}

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
• a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(a);
```

Memory model: {(a, 16), (&len, 4), (&i, 4), (&a_inv, 4), (a_inv, 16)}

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
```

- `__e_acsl_free(a_inv);`

```
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(a);
```

Memory model: {(a, 16), (&len, 4), (&i, 4), (&a_inv, 4), ~~(a_inv, 16)~~}

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
• __delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(a);
```

Memory model: {(a, 16), (&len, 4), (&i, 4), ~~(&a_inv, 4), (a_inv, 16)~~}

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(& a_inv);
• __delete_block(& i);
__delete_block(& len);
__delete_block(a);
```

Memory model: {(a, 16), (&len, 4), ~~(&i, 4), (&a_inv, 4), (a_inv, 16)~~}

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
• __delete_block(& len);
__delete_block(a);
```

Memory model: {(a, 16), ~~(&len, 4), (&i, 4), (&a_inv, 4), (a_inv, 16)~~}

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
• __delete_block(a);
```

Memory model: {~~(a, 16), (&len, 4), (&i, 4), (&a_inv, 4), (a_inv, 16)~~}

E-ACSL: Goal and Solution

Goal: avoid the monitoring of irrelevant statements

- Annotations do not necessarily evaluate memory-related properties for all memory locations
- Full memory monitoring is costly and seldom necessary

Solution: E-ACSL performs a pre-analysis of the input program, which:

- Consists in a backward data-flow analysis
- Over-approximates the set of variables that must be monitored to verify memory related annotations
- Identified irrelevant memory locations are not monitored

Instrumented program with full memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(a);
```

Instrumented program after pre-analysis

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(a);
```

E-ACSL: Experiments and Results

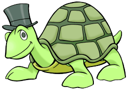
- In E-ACSL plugin, static analysis helps to avoid
  - irrelevant memory monitoring
  - systematic usage of an unbounded integer library (GMP)
- Static analysis provides a significant speedup (55% in average, going up to 98% in some examples)

[Delahaye et al. SAC 2013; Kosmatov et al. RV 2013; Jakobsson et al. SAC 2015, SCP 2016; Vorobyov et al. ISMM 2017]

Outline

SANTE: Goals

Detection of runtime errors: two approaches



Static analysis

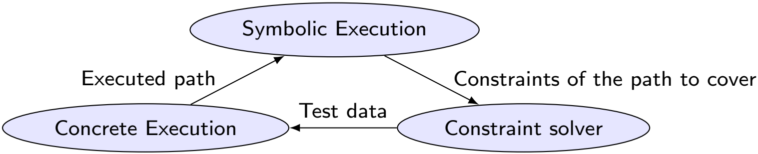
Issue: leaves unconfirmed errors that can be safe

Dynamic Analysis

Issue: cannot detect all errors if test coverage is partial

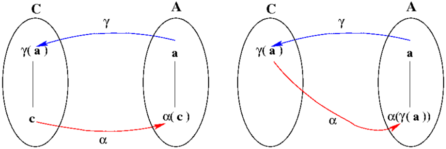
Goal: Combine both techniques to detect runtime errors more efficiently

Plugin PathCrawler for test generation



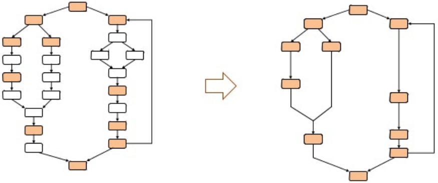
- Performs Dynamic Symbolic Execution (DSE)
- Automatically creates test data to cover program paths (explored in depth-first search, [Botella et al. AST 2009])
- Uses code instrumentation, concrete and symbolic execution, constraint solving
- Exact semantics: doesn't approximate path constraints
- Similar to PEX, DART/CUTE, KLEE, SAGE, etc.
- Online version: `pathcrawler-online.com`

Plugin “VALUE” for value analysis



- Based on abstract interpretation [Cousot, POPL 1977]
- Computes an overapproximation of sets of possible values of variables at each instruction
- Considers all possible executions
- Reports alarms when cannot prove absence of errors

Plugin Slicing



- Simplifies the program using control and data dependencies
- Preserves the executions reaching a point of interest (*slicing criterion*) with the same behavior
- Example of slicing criteria: instructions, annotations (alarms), function calls and returns, read and write accesses to selected variables. . .

Example: Value Analysis and a True Positive

```c
int divide (int a)
{
  int x,y,z,t;
  if( a )
    { x = 0; y = 0; }
  else
    { x = 10; y = 10; }
  z = x + y;
  //@ assert z != 0;
  t = 20 / z;
  return t;
}
```

EVA detects a risk of division by zero

Dynamic analysis can help to confirm the error!

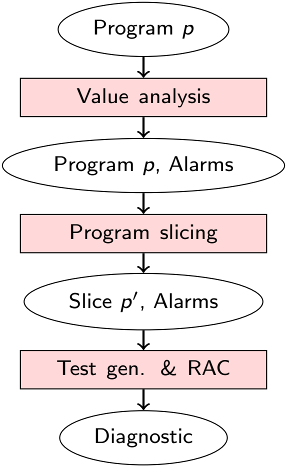
Example: Value Analysis and a False Alarm

```
int divide (int a)
{
  int x,y,z,t;
  if( a )
    { x = 10; y = 0; }
  else
    { x = 0; y = 10; }
  z = x + y;
  //@ assert z != 0;
  t = 20 / z;
  return t;
}
```

EVA detects a risk of division by zero

Dynamic analysis cannot confirm the error!

SANTE: Methodology for detection of runtime errors



- Value analysis detects alarms
- Slicing reduces the program (w.r.t. one or several alarms)
- Test generation (PathCrawler) on a reduced program to diagnose alarms (after adding error branches to trigger errors)
- Runtime Assertion Checking checks for failures
- Diagnostic
  - bug if a counter-example is generated
  - if not, and all paths were explored, the alarm is safe
  - otherwise, unknown

SANTE: Experiments

- 9 benchmarks with known errors (from Apache, libgd, . . .)

Alarm classification:

- all known errors found by SANTE
- SANTE leaves less unclassified alarms than VALUE (by 88%) or PathCrawler (by 91%) alone

Program reduction:

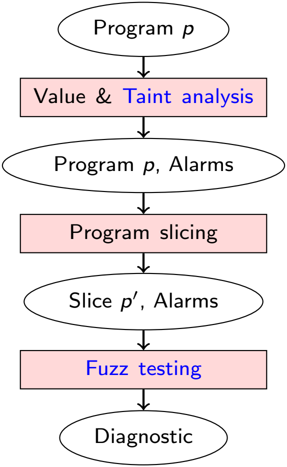
- 32% in average, up to 89% for some examples
- program paths in counter-examples are in average 19% shorter

Execution time:

- Average speedup w.r.t. testing alone is 43% (up to 98% for some examples)

[Chebaro et al. TAP 2009, TAP 2010, SAC 2012, ASEJ 2014]

Application to security



- Reused in EU FP7 project STANCE (CEA LIST, Dassault, Search Lab, FOKUS,...)
- Taint analysis to identify most security-relevant alarms
- Fuzz testing (Flinder tool) for efficient detection of vulnerabilities
- Applied to the recent Heartbleed security flaw (2014) in OpenSSL, other case studies in progress
- [Kiss et al., HVC 2015]

Outline

Plugin WP for deductive verification

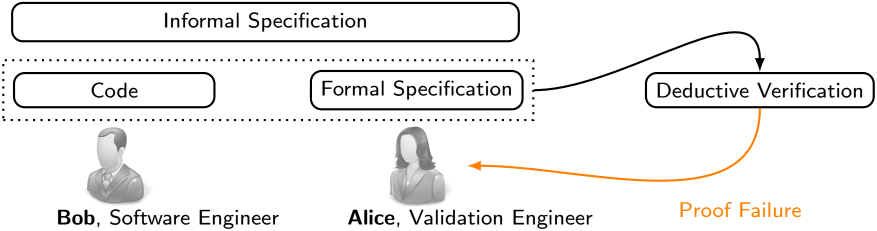
$$\dfrac{\dfrac{\{A \wedge b\}\ c\ \{B\} \qquad \dfrac{(A \wedge \neg b) \Rightarrow B}{\{(A \wedge \neg b)\}\ \mathtt{skip}\ \{B\}}}{\{A\}\ \mathtt{if}\ b\ \mathtt{then}\ c\ \mathtt{else}\ \mathtt{skip}\ \{B\}}}{\{A\}\ \mathtt{if}\ b\ \mathtt{then}\ c\ \{B\}}$$

- Based on Weakest Precondition calculus [Dijkstra, 1976]
- Proves that a given program respects its specification

The enemy: proof failures, i.e. unproven properties

- can result from very different reasons
  - an error in the code,
  - an insufficient precondition,
  - a too weak subcontract (e.g. loop invariant, callee's contract),
  - a too strong postcondition,. . .

Global Motivation: Facilitate Software Verification



Why does my proof fail?

Analysis of proof failures is costly and often requires

- deep knowledge of provers
- careful review of code / specification
- interactive proof in a proof assistant

Modular Deductive Verification in a Nutshell

```
// Pre_f assumed
f(<args>){
  code1;
// Pre_g to be proved
  g(<args>);
// Post_g assumed
  code2;
}
// Post_f to be proved
```

```
// Pre_f assumed
f(<args>){
  code1;
// I to be proved
  while(C){
// I ∧ C assumed
    code3;
// I to be proved
  }
// I ∧ ¬C assumed
  code2;
}
// Post_f to be proved
```

A proof failure can be due to various reasons!

For convenience, we say:
A *subcontract* of f is the contract of a called function or loop in f.

Example: Several reasons for the same proof failure

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Can be proven with Frama-C/WP

Example: Several reasons for the same proof failure

Postcondition unproven. . .

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result == 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

. . . because it is incorrect.

Example: Several reasons for the same proof failure

Postcondition unproven. . .

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 0;
}
```

. . . because the code is incorrect.

Example: Several reasons for the same proof failure

Postcondition unproven...

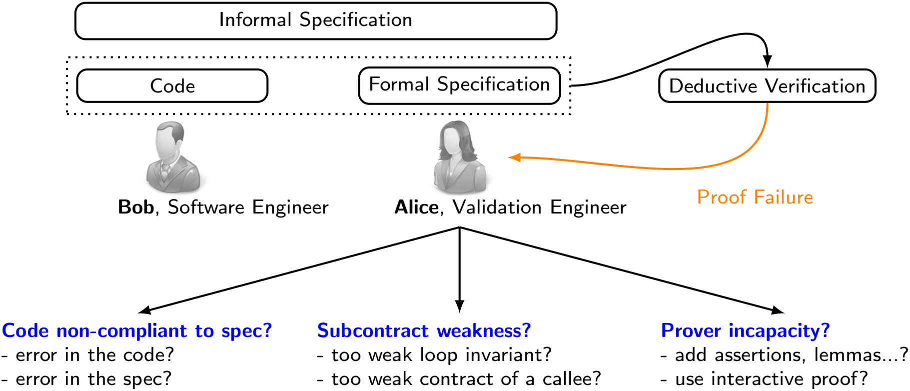
```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

...because a loop invariant is missing.

STADY: Goals

- Help the validation engineer to understand and fix the proof failures
- Provide a counter-example to illustrate the issue
- Do it automatically and efficiently

What is the right way to go?



Our main goals: a complete verification methodology to

- automatically and precisely diagnose proof failures,
- provide a counter-example to illustrate the issue

STADY: Methodology for diagnosis of proof failures

- Define three kinds of proof failures:
  - non-compliance (direct conflict betw. code and spec)
  - subcontract weakness (too weak contract for some loop or callee)
  - prover incapacity (the property holds, but is not proven)
- Perform dedicated instrumentation allowing to detect non-compliances and subcontract weaknesses
- Apply testing (PathCrawler) and RAC to try to find a counter-example and to classify the proof failure
- Indicate a more precise feedback (if possible, with a counter-example) to help the user to understand and to fix the proof failure

Instrumentation for Non-Compliance Detection: A Function Contract

```
/*@ requires Pre_g;
    ensures Post_g;       */

Type_f g(...) {

  code1;

}
```

$\rightarrow$

```
Type_f g(...) {
  fassert(Pre_g);

  code1;


  fassert(Post_g);
}
```

Principle:

- translate annotations into C code, similarly to runtime assertion checking, but in a way that DSE can trigger errors
- details in [Petiot, SCAM'14]

Instrumentation for Subcontract Weakness Detection:

```
/*@ assigns x1,..,xN;
    ensures Post_g;        */

Type_g g(...) {
  code3;
}

Type_f f(...) {
  code1;
  g(Args);
  code2;
}
```

→

```
Type_g g_sw(...) {
  x1 = NonDet();
  ...
  xN = NonDet();
  Type_g ret=NonDet();
  fassume(Post_g);
  return ret;
} //respects Post_g
Type_g f(...) {
  code1
  g_sw(Args);
  code2;
}
```

- **Principle:** Replace the callee/loop code by the most general code respecting its contract, then try to trigger errors with DSE
- requires (`loop`) `assigns` clauses

STADY: Initial experiments

- 20 annotated (provable) programs (from [Burghardt, Gerlach])
- 928 mutants generated (erroneous code, erroneous or missing annotation)
- STADY is applied to classify proof failures

Alarm classification:

- STADY classified 97% proof failures

Execution time: comparable to WP

- WP takes in average 2.6 sec. per mutant (13 sec. per unproven mutant)
- STADY takes in average 2.7 sec. per unproven mutant

Partial coverage:

- Testing with partial coverage remains efficient in STADY

[Petiot et al. TAP 2014, SCAM 2014, TAP 2016]

Outline

Context: white-box testing

- Generate a test input
- Run it and check for errors
- Estimate coverage: if enough, then stop, else loop

Coverage criteria (decision, mcdc, mutants, etc.) play a major role

- generate tests, decide when to stop, assess quality of testing

The enemy: Uncoverable test objectives

- waste generation effort, imprecise coverage ratios
- cause: structural coverage criteria are ... structural
- detecting uncoverable test objectives is undecidable

Recognized as a hard and important issue in testing

- no practical solution, not so much work (compared to test gen.)
- **real pain** (e.g. aeronautics, mutation testing)

LTest: Goals

We focus on white-box (structural) coverage criteria

Automatic detection of uncoverable test objectives

- a *sound* method
- applicable to a large class of coverage criteria
- strong detection power, reasonable speed
- rely as much as possible on existing verification methods

Note. The test objective "reach location *loc* and satisfy predicate p" is uncoverable $\Leftrightarrow$ the assertion `assert` $(\neg p)$; at location *loc* is valid

Example: program with two uncoverable test objectives

```
int main() {
  int a = nondet(0 .. 20);
  int x = nondet(0 .. 1000);
  return g(x,a);
}
int g(int x, int a) {
  int res;
  if(x+a >= x)
    res = 1;    // the only possible outcome
  else
    res = 0;
 // l1: res == 0
 // l2: res == 2
  return res;
}
```

Example: program with two valid assertions

```
int main() {
  int a = nondet(0 .. 20);
  int x = nondet(0 .. 1000);
  return g(x,a);
}
int g(int x, int a) {
  int res;
  if(x+a >= x)
    res = 1;     // the only possible outcome
  else
    res = 0;
 //@ assert res != 0
 //@ assert res != 2
  return res;
}
```

Example: program with two valid assertions

```
int main() {
  int a = nondet(0 .. 20);
  int x = nondet(0 .. 1000);
  return g(x,a);
}
int g(int x, int a) {
  int res;
  if(x+a >= x)
    res = 1;     // the only possible outcome
  else
    res = 0;
 //@ assert res != 0     // both VALUE and WP fail
 //@ assert res != 2     // detected as valid
  return res;
}
```

LTest Methodology: Combine VALUE $\oplus$ WP

Goal: get the best of the two worlds

- Idea: VALUE passes to WP the global information that WP needs

Which information, and how to transfer it?

- VALUE computes variable domains
- WP naturally takes into account assumptions (`assume`)

Proposed solution:

- **VALUE exports computed variable domains in the form of WP-assumptions**

Example: alone, both VALUE and WP fail

```
int main() {
  int a = nondet(0 .. 20);
  int x = nondet(0 .. 1000);
  return g(x,a);
}
int g(int x, int a) {



  int res;
  if(x+a >= x)
    res = 1;     // the only possible outcome
  else
    res = 0;
 //@ assert res != 0     // both VALUE and WP fail

  return res;
}
```

Example: VALUE⊕WP

```
int main() {
  int a = nondet(0 .. 20);
  int x = nondet(0 .. 1000);
  return g(x,a);
}
int g(int x, int a) {
 //@ assume 0 <= a <= 20
 //@ assume 0 <= x <= 1000 // VALUE inserts domains...
  int res;
  if(x+a >= x)
    res = 1;     // the only possible outcome
  else
    res = 0;
 //@ assert res != 0

  return res;
}
```

Example: VALUE⊕WP

```
int main() {
  int a = nondet(0 .. 20);
  int x = nondet(0 .. 1000);
  return g(x,a);
}
int g(int x, int a) {
 //@ assume 0 <= a <= 20
 //@ assume 0 <= x <= 1000 // VALUE inserts domains...
  int res;
  if(x+a >= x)
    res = 1;     // the only possible outcome
  else
    res = 0;
 //@ assert res != 0       // ...  and WP succeeds!

  return res;
}
```

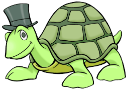
LTest: Results and Experiments

- automatic, sound and generic method
- new combination of existing verification techniques
- experiments for 12 programs and 3 criteria (CC, MCC, WM):
  - strong detection power (95%),
  - reasonable detection speed ($\leq$ 1s/obj.),
  - test generation speedup (3.8x in average),
  - more accurate coverage ratios (99.2% instead of 91.1% in average, 91.6% instead of 61.5% minimum)

[Bardin et al. ICST 2014, TAP 2014, ICST 2015, ICSE 2018]

Outline

Conclusion



- Combining Static and Dynamic analyses can be beneficial for various domains of software verification:
  - detection of runtime errors and security vulnerabilities,
  - deductive verification,
  - runtime assertion checking,
  - test generation, ...
- Both ways: static helps dynamic and dynamic helps static
- Frama-C provides a rich and extensible framework for combined analyses