

Combining Static and Dynamic Approaches to Software Verification in Frama-C

Sébastien Bardin, Nikolai Kosmatov

joint work with O.Chebaro, M.Delahaye, A.Giorgetti, A.Jakobsson,
J.Julliand, G.Petiot, J.Signoles, ...



Imperial College, London, July 1st, 2015

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

Detecting runtime errors by static analysis and testing (SANTE)

Deductive verification assisted by testing (STADY)

Optimizing testing by value analysis and weakest precondition (LTest)

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

Conclusion

Outline

Frama-C, a platform for analysis of C code

Detecting runtime errors by static analysis and testing (SANTE)

Deductive verification assisted by testing (STADY)

Optimizing testing by value analysis and weakest precondition (LTest)

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

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)
- ▶ 2008: [First public release](#) of Frama-C (Hydrogen)
- ▶ Today: [Frama-C Sodium](#) (v.11)
 - ▶ [Multiple projects](#) around the platform
 - ▶ A growing community of users. . .
 - ▶ and of developers

Frama-C at a glance



- ▶ A **F**ramework for **M**odular **A**nalysis of **C** code
- ▶ Developed at CEA LIST and INRIA Saclay
- ▶ Released under **GPL** license
- ▶ **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/> [Cuoq et al. SEFM 2012, 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

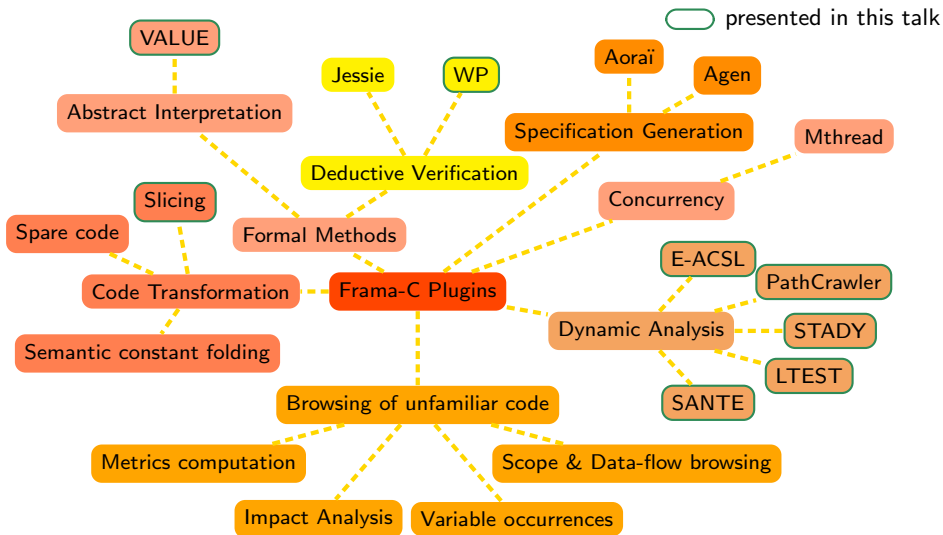
```

/*@ 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



Outline

Frama-C, a platform for analysis of C code

Detecting runtime errors by static analysis and testing (SANTE)

Deductive verification assisted by testing (STADY)

Optimizing testing by value analysis and weakest precondition (LTest)

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

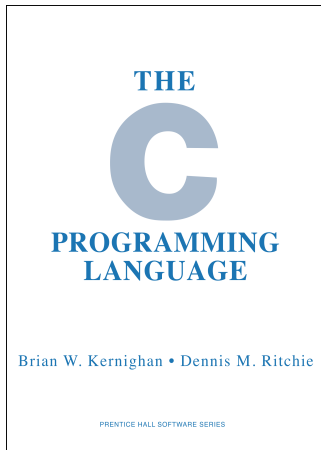
Conclusion

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
- ▶ ...



SANTE: Goals

Detection of runtime errors: two approaches



Static analysis

Issue: leaves unconfirmed errors
that can be safe

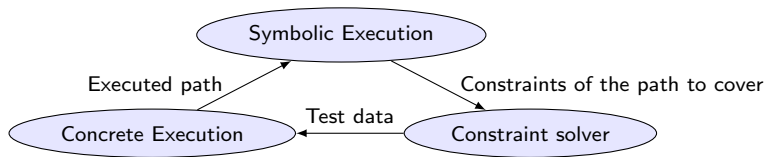


Testing

Issue: cannot detect all errors if
test coverage is partial

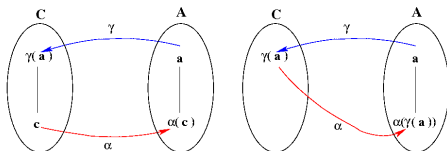
Goal: Combine both techniques to detect runtime errors more efficiently

Plugin PathCrawler for test generation



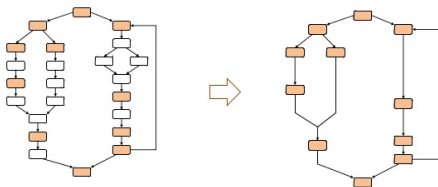
- ▶ Performs **Dynamic Symbolic Execution (DSE)**
- ▶ **Automatically creates test data** to cover program paths (explored in depth-first search, [Botella et al. AST 2009])
- ▶ Uses code instrumentation, concrete and symbolic execution, constraint solving
- ▶ **Exact semantics**: doesn't approximate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

Plugin “VALUE” for value analysis



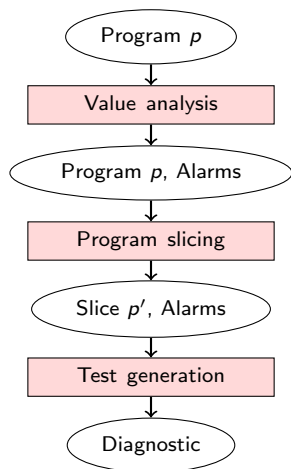
- ▶ Based on **abstract interpretation** [Cousot, POPL 1977]
- ▶ Computes an **overapproximation** of sets of possible values of variables at each instruction
- ▶ Considers **all possible executions**
- ▶ Reports **alarms** when cannot prove absence of errors

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. . .

SANTE: Methodology for detection of runtime errors



- ▶ **Value analysis** detects alarms
- ▶ **Slicing** reduces the program (w.r.t. one or several alarms)
- ▶ **Testing** (PathCrawler) is used to generate tests on a reduced program to diagnose alarms (after adding error branches to trigger errors)
- ▶ **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:

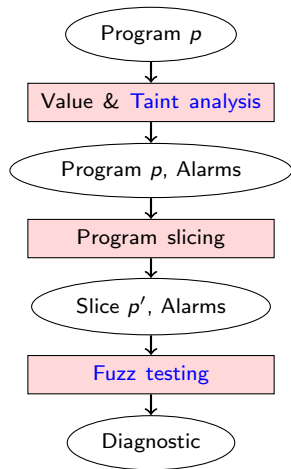
- ▶ 32% in average, up to 89% for some examples
- ▶ program paths in counter-examples are in average 19% shorter

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., Submitted 2015]

Outline

Frama-C, a platform for analysis of C code

Detecting runtime errors by static analysis and testing (SANTE)

Deductive verification assisted by testing (STADY)

Optimizing testing by value analysis and weakest precondition (LTest)

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

Conclusion

Plugin WP for deductive verification

$$\frac{\frac{\{A \wedge b\} c \{B\} \quad \frac{(A \wedge \neg b) \Rightarrow B}{\{(A \wedge \neg b)\} \text{skip} \{B\}}}{\{A\} \text{if } b \text{ then } c \text{ else skip} \{B\}}}{\{A\} \text{if } b \text{ then } c \{B\}}$$

- ▶ 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,...
- ▶ often require **costly manual analysis**

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

Example: An erroneous version

```

/*@ 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 assigns k;
        loop variant n-k;
    */
    for(k = 0; k < n; k++)
        if (t[k] != 0)
            return 0;
    return 1;
}

```

Postcondition
unproven...

... because of a missing
loop invariant.

The reason could also be a
wrong precond, or postcond., or code

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**

STADY: Methodology for diagnosis of proof failures

- ▶ Define **three kinds of proof failures**:
 - ▶ non-compliance (between the code and its specification)
 - ▶ subcontract weakness (for a loop or a called function)
 - ▶ prover incapacity
- ▶ Perform **dedicated instrumentation** allowing to detect non-compliances and subcontract weaknesses
- ▶ Apply **testing** (PathCrawler) 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

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, Submitted 2015]

Outline

Frama-C, a platform for analysis of C code

Detecting runtime errors by static analysis and testing (SANTE)

Deductive verification assisted by testing (STADY)

Optimizing testing by value analysis and weakest precondition (LTest)

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

Conclusion

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

Enemy 1: Lack of support for several coverage criteria

- ▶ symbolic execution is about paths, not coverage criteria
- ▶ instrumentation leads to poor performance

Enemy 2: Uncoverable test objectives

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

A generic approach to coverage criteria: Labels [ICST 2014]

- ▶ Annotate programs with **labels**
 - ▶ predicate attached to specific program instruction
- ▶ Label (loc, φ) is covered if a test execution
 - ▶ reaches the instruction at loc
 - ▶ satisfies the predicate φ
- ▶ Labels may **encode lots of criteria**, e.g.:
 - ▶ DC, CC, MCC, GACC
 - ▶ Function coverage
 - ▶ Weak mutation (side-effect-free operators)
 - ▶ Input domain partition

Example: Condition Coverage

```
statement_1;
if ( x == y && a < b )
{...};
statement_3;
```

→

```
statement_1;
// 11:  x==y
// 12:  !(x==y)
// 13:  a<b
// 14:  !(a<b)
if ( x == y && a < b )
{...};
statement_3;
```

Example: Multiple Condition Coverage

```
statement_1;
if ( x == y && a < b )
{...};
statement_3;
```



```
statement_1;
// 11:  x==y && a<b
// 12:  x==y && a>=b
// 13:  x!=y && a<b
// 14:  x!=y && a>=b
if ( x == y && a < b )
{...};
statement_3;
```

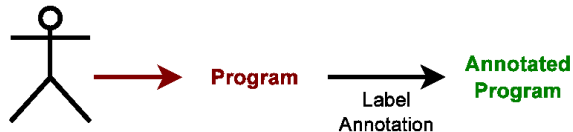
Example: Function Coverage

```
int main() {  
    ...  
}  
int f() {  
    ...  
}
```

```
→ int main() {  
    // 11: true  
    ...  
}  
int f() {  
    // 12: true  
    ...  
}
```

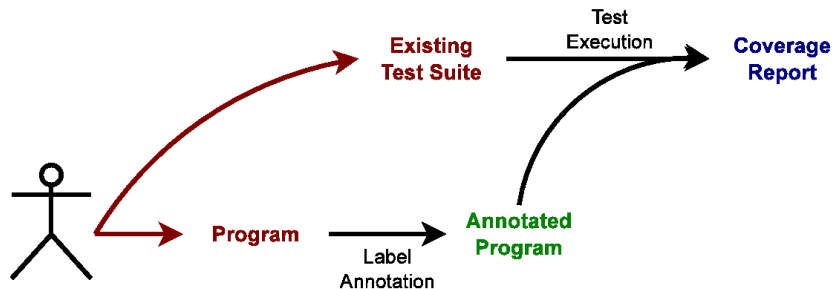
FRAMA-C/LTEST

A label-based automated test framework [TAP 2014]



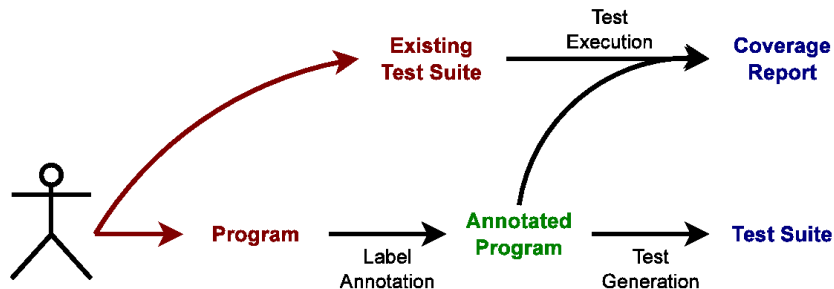
FRAMA-C/LTEST

A label-based automated test framework [TAP 2014]



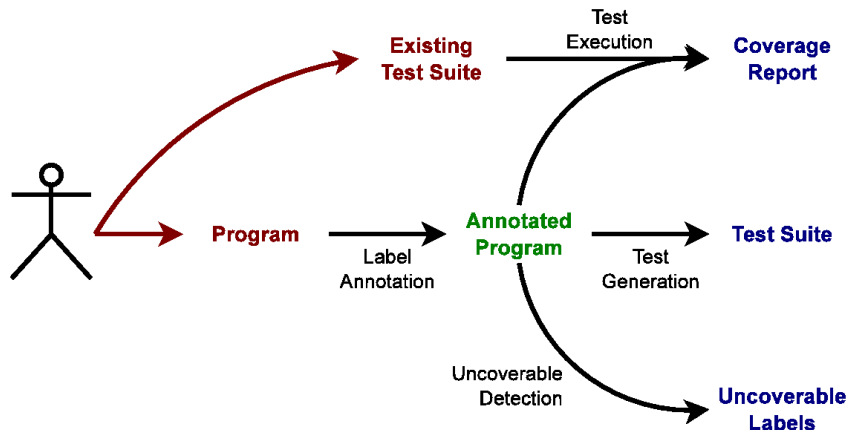
FRAMA-C/LTEST

A label-based automated test framework [TAP 2014]



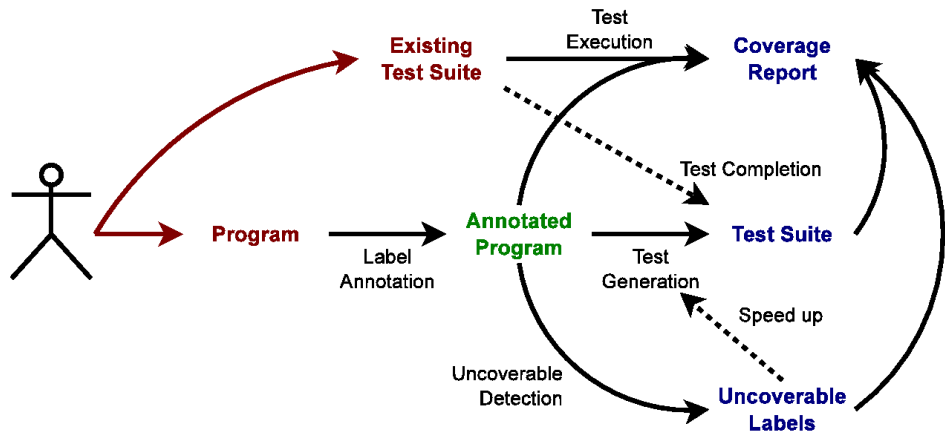
FRAMA-C/LTEST

A label-based automated test framework [TAP 2014]

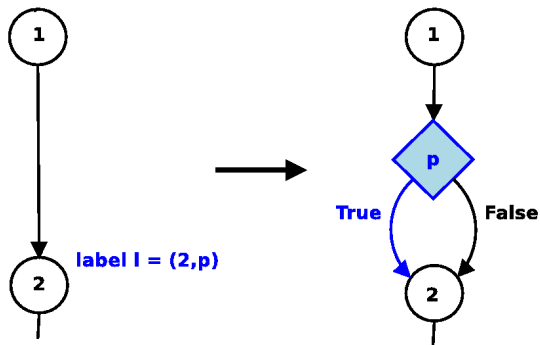


FRAMA-C/LTEST

A label-based automated test framework [TAP 2014]

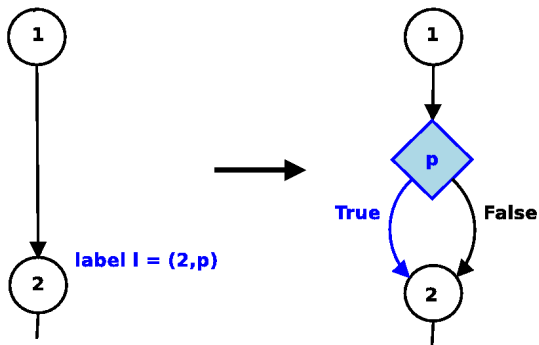


Labels in DSE: Direct Instrumentation



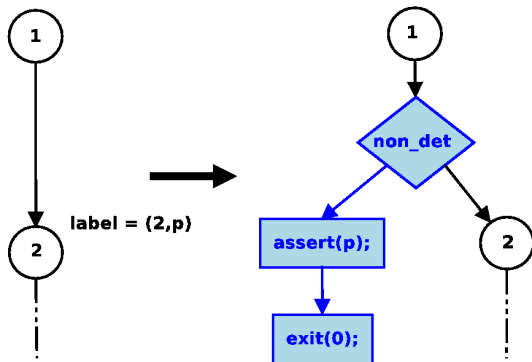
- ▶ covering label $l \iff$ covering true branch
- ▶ sound & complete

Labels in DSE: Direct Instrumentation



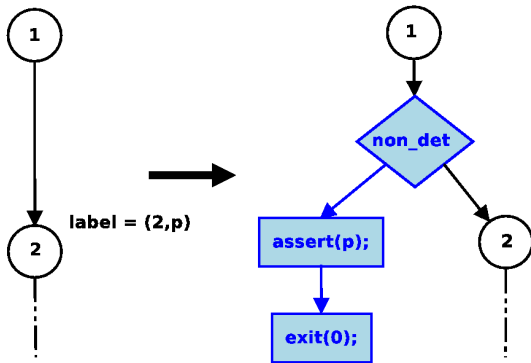
- ▶ covering label $l \iff$ covering true branch
- ▶ sound & complete
- ▶ **dramatic overhead** [APEX 272× time overhead]

Labels in DSE: Tight Instrumentation [ICST2014]



- ▶ covering label $l \iff$ covering assert
- ▶ sound & complete

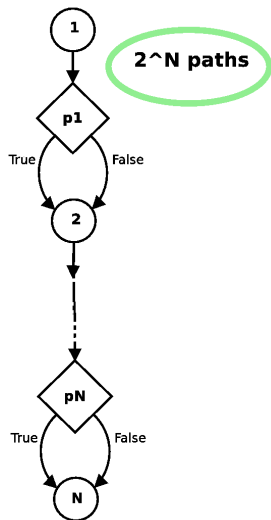
Labels in DSE: Tight Instrumentation [ICST2014]



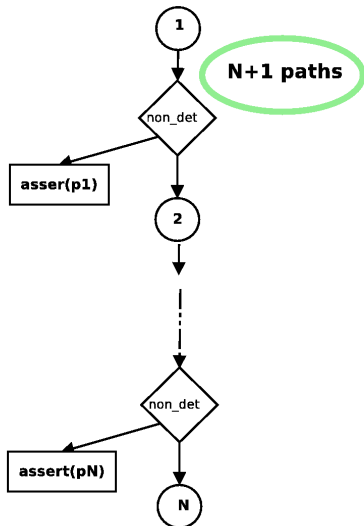
- ▶ covering label $l \iff$ covering assert
- ▶ sound & complete
- ▶ no blow-up of the search space

Labels in DSE: direct vs. tight instrumentation

Direct instrumentation



Tight Instrumentation



LTest: Uncoverable test objectives [ICST 2015]

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
}
```

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
}
```

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
}

```

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
}
```

Example: VALUE \oplus 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
}

```


Example: VALUE \oplus 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!
}

```

LTest: Results and Experiments

- ▶ a **practical and generic formalism** for coverage criteria using labels, suitable for formal reasoning and static analysis
- ▶ an **optimized DSE** technique for labels
- ▶ an **automatic, sound and generic method** for detection of uncoverables
- ▶ 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]

Outline

Frama-C, a platform for analysis of C code

Detecting runtime errors by static analysis and testing (SANTE)

Deductive verification assisted by testing (STADY)

Optimizing testing by value analysis and weakest precondition (LTest)

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

Conclusion

E-ACSL language

The [E-ACSL language](#) is an 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
<code>\valid(p)</code>	p is a valid pointer
<code>\initialized(p)</code>	$*p$ has been initialized
<code>\block_length(p)</code>	Length of p 's memory block
<code>\base_address(p)</code>	Base address of p 's memory block
<code>\offset(p)</code>	Offset of p in its memory block

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

Example: a C program annotated with ACSL

```

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

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)
- ▶ Experiments on a set of 9 annotated programs
- ▶ 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, JFLA 2015]

Outline

Frama-C, a platform for analysis of C code

Detecting runtime errors by static analysis and testing (SANTE)

Deductive verification assisted by testing (STADY)

Optimizing testing by value analysis and weakest precondition (LTest)

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

Conclusion

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