

# Frama-C

## A Collaborative Framework for C Code Verification

Tutorial at RV 2016

Nikolai Kosmatov, Julien Signoles



Madrid, September 27<sup>th</sup>, 2016

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

Combinations of Analyses

Conclusion

# Outline

## Frama-C Overview

Formal Specification and Deductive Verification with WP

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

Combinations of Analyses

Conclusion

## Frama-C Historical Context

- ▶ 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)

## Frama-C Historical Context

- ▶ 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)

## Frama-C Historical Context

- ▶ 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)
- ▶ 2004: start of Frama-C project as a successor to CAVEAT and Caduceus
- ▶ 2008: [First public release](#) of Frama-C (Hydrogen)

## Frama-C Historical Context

- ▶ 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)
- ▶ 2004: start of Frama-C project as a successor to CAVEAT and Caduceus
- ▶ 2008: [First public release](#) of Frama-C (Hydrogen)
- ▶ 2012: [WP](#): Weakest-precondition based plugin
- ▶ 2012: [E-ACSL](#): Runtime Verification plugin
- ▶ 2013: CEA Spin-off [TrustInSoft](#)
- ▶ 2016: [Eva](#): Evolved Value Analysis
- ▶ 2016: [Frama-Clang](#): C++ extension
- ▶ Today: [Frama-C Aluminium](#) (v.13)
- ▶ Upcoming: [Frama-C Silicium](#) (v.14, expected in November)

# Frama-C Open Source Distribution

Framework for analyses of source code written in ISO 99 C

[Kirchner & al @FAC'15]

- ▶ analyze C++ code extended with **ACSL** annotations
- ▶ **ACSL**
  - ▶ ISO/ANSI C Specification Language
  - ▶ *langua franca* of analyzers
- ▶ almost **open source** (LGPL 2.1)  
<http://frama-c.com>
- ▶ also proprietary extensions and distributions
- ▶ targets both **academic** and **industrial** usage



# 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;
}

```

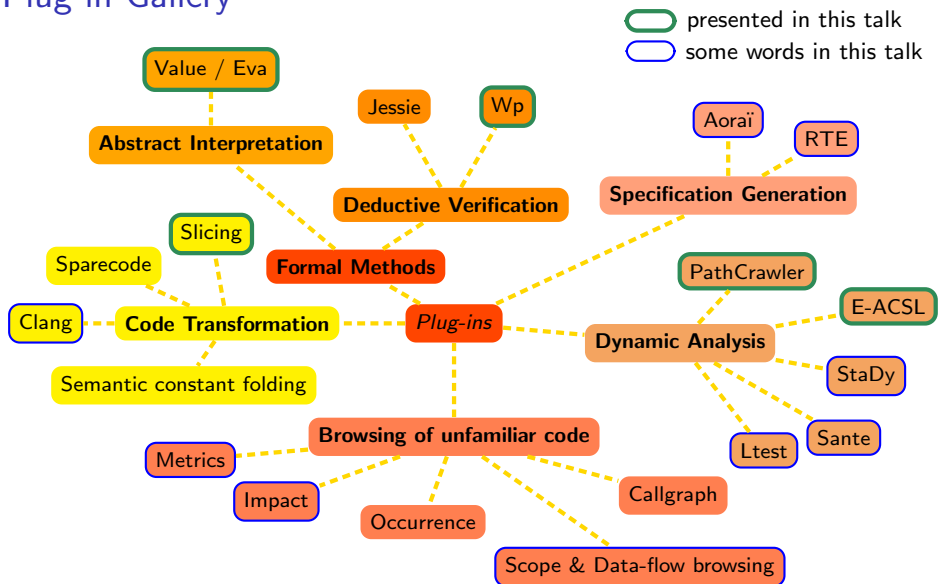
Can be proven  
with Frama-C/WP

# Frama-C, a Collection of Tools

## Several tools inside a single platform

- ▶ **plug-in architecture** *à la* Eclipse [S. @F-IDE'15]
- ▶ tools provided as plug-ins
  - ▶ 21 plug-ins in the open source distribution
  - ▶ outside open source plug-ins (E-ACSL & Frama-Clang, a few others)
  - ▶ close source plug-ins, either at CEA (about 20) or outside
- ▶ plug-ins connected to a **kernel**
  - ▶ provides an uniform setting
  - ▶ provides general services
  - ▶ synthesizes useful information

# Plug-in Gallery



# Frama-C, a Development Platform

- ▶ developed in **OCaml** ( $\approx$  180 kloc in the open source distribution,  $\approx$  300 kloc with proprietary extensions)
- ▶ was based on **Cil** [Necula & al @CC'02]
- ▶ **library** dedicated to analysis of C code

development of plug-ins by third party

- ▶ **powerful low-cost** analyser
- ▶ dedicated plug-in for **specific task** (verifying your coding rules)
- ▶ dedicated plug-in for fine-grain parameterization
- ▶ **extension** of existing analysers

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

Overview of ACSL and WP

Function contracts

Programs with loops

My proof fails... What to do?

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

Combinations of Analyses

Conclusion

# Goal

In this part, we will see

- ▶ how to specify a C program using ACSL
- ▶ how to prove it with an automatic tool using Frama-C/WP
- ▶ how to understand and fix proof failures

# Objectives of Deductive Verification

Rigorous, mathematical proof of semantic properties of a program

- ▶ functional properties
- ▶ safety:
  - ▶ all memory accesses are valid,
  - ▶ no arithmetic overflow,
  - ▶ no division by zero, ...
- ▶ termination

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

Overview of ACSL and WP

Function contracts

Programs with loops

My proof fails... What to do?

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

Combinations of Analyses

Conclusion



# ACSL: ANSI/ISO C Specification Language

## Presentation

- ▶ 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

- ▶ Typed first-order logic
- ▶ Pure C 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)`

# WP plugin

- ▶ Hoare-logic based plugin, developed at CEA List
- ▶ Proof of semantic properties of the program
- ▶ Modular verification (function by function)
- ▶ Input: a program and its specification in ACSL
- ▶ WP generates verification conditions (VCs)
- ▶ Relies on Automatic Theorem Provers to discharge the VCs
  - ▶ Alt-Ergo, Simplify, Z3, Yices, CVC3, CVC4 ...
- ▶ WP manual at <http://frama-c.com/wp.html>
- ▶ If all VCs are proved, the program respects the given specification
  - ▶ Does it mean that the program is correct?

# WP plugin

- ▶ Hoare-logic based plugin, developed at CEA List
- ▶ Proof of semantic properties of the program
- ▶ Modular verification (function by function)
- ▶ Input: a program and its specification in ACSL
- ▶ WP generates verification conditions (VCs)
- ▶ Relies on Automatic Theorem Provers to discharge the VCs
  - ▶ Alt-Ergo, Simplify, Z3, Yices, CVC3, CVC4 ...
- ▶ WP manual at <http://frama-c.com/wp.html>
- ▶ If all VCs are proved, the program respects the given specification
  - ▶ Does it mean that the program is correct?
  - ▶ NO! If the specification is wrong, the program can be wrong!

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

Overview of ACSL and WP

**Function contracts**

Programs with loops

My proof fails... What to do?

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

Combinations of Analyses

Conclusion

# Contracts

- ▶ **Goal:** specification of imperative functions
- ▶ **Approach:** give assertions (i.e. properties) about the functions
  - ▶ **Precondition** is supposed to be true on entry (ensured by the caller)
  - ▶ **Postcondition** must be true on exit (ensured by the function)
- ▶ Nothing is guaranteed when the precondition is not satisfied
- ▶ **Termination** may be guaranteed or not (total or partial correctness)

## Primary role of contracts

- ▶ Must reflect the informal specification
- ▶ Should not be modified just to suit the verification tasks

## Example 1

Specify and prove the following program:

```
// returns the absolute value of x
int abs ( int x ) {
    if ( x >=0 )
        return x ;
    return -x ;
}
```

Try to prove with Frama-C/WP using the basic command

► `frama-c-gui -wp file.c`

## Example 1 (Continued)

The basic proof succeeds for the following program:

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

- ▶ The returned value is not always as expected.

## Example 1 (Continued)

The basic proof succeeds for the following program:

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

- ▶ The returned value is not always as expected.
- ▶ For  $x = \text{INT\_MIN}$ ,  $-x$  cannot be represented by an `int` and overflows
- ▶ Example: on 32-bit,  $\text{INT\_MIN} = -2^{31}$  while  $\text{INT\_MAX} = 2^{31} - 1$



## Safety warnings: arithmetic overflows

Absence of arithmetic overflows can be important to check

- ▶ A sad example: crash of Ariane 5 in 1996

WP can automatically check the absence of runtime errors

- ▶ Use the command `frama-c-gui -wp -wp-rte file.c`
- ▶ It generates VCs to ensure that runtime errors do not occur
  - ▶ in particular, arithmetic operations do not overflow
- ▶ If not proved, an error may occur.

## Example 1 (Continued) - Solution

This is the completely specified program:

```
#include <limits.h>
/*@ requires x > INT_MIN;
    ensures (x >= 0 ==> \result == x) &&
        (x < 0 ==> \result == -x);
    assigns \nothing;
*/
int abs ( int x ) {
    if ( x > 0 )
        return x ;
    return -x ;
}
```

## Example 2

Specify and prove the following program:

```
// returns the maximum of x and y
int max ( int x, int y ) {
    if ( x >=y )
        return x ;
    return y ;
}
```

## Example 2 (Continued) - Find the error

The following program is proved. Do you see any error?

```
/*@ ensures \result >= x && \result >= y;  
*/  
int max ( int x, int y ) {  
    if ( x >=y )  
        return x ;  
    return y ;  
}
```

## Example 2 (Continued) - a wrong version

This is a wrong implementation that is also proved. Why?

```
#include <limits.h>
/*@ ensures \result >= x && \result >= y;
*/
int max ( int x, int y ) {
    return INT_MAX ;
}
```

## Example 2 (Continued) - a wrong version

This is a wrong implementation that is also proved. Why?

```
#include <limits.h>
/*@ ensures \result >= x && \result >= y;
*/
int max ( int x, int y ) {
    return INT_MAX ;
}
```

- ▶ Our specification is incomplete
- ▶ Should say that the returned value is one of the arguments

## Example 2 (Continued) - Solution

This is the completely specified program:

```
/*@ ensures \result >= x && \result >= y;  
    ensures \result == x || \result == y;  
    assigns \nothing;  
*/  
int max ( int x, int y ) {  
    if ( x >=y )  
        return x ;  
    return y ;  
}
```

## Example 3

Specify and prove the following program:

```
// returns the maximum of *p and *q
int max_ptr ( int *p, int *q ) {
    if ( *p >= *q )
        return *p ;
    return *q ;
}
```



## Example 3 (Continued) - Explain the proof failure

Explain the proof failure with the option `-wp-rte` for the program:

```
/*@ ensures \result >= *p && \result >= *q;  
    ensures \result == *p || \result == *q;  
*/  
int max_ptr ( int *p, int *q ) {  
    if ( *p >= *q )  
        return *p ;  
    return *q ;  
}
```

## Example 3 (Continued) - Explain the proof failure

Explain the proof failure with the option `-wp-rte` for the program:

```
/*@ ensures \result >= *p && \result >= *q;  
    ensures \result == *p || \result == *q;  
*/  
int max_ptr ( int *p, int *q ) {  
    if ( *p >= *q )  
        return *p ;  
    return *q ;  
}
```

- ▶ Nothing ensures that pointers `p`, `q` are valid
- ▶ It must be ensured either by the function, or by its precondition

## Safety warnings: invalid memory accesses

An invalid pointer or array access may result in a [segmentation fault or memory corruption](#).

- ▶ WP can automatically generate VCs to check memory access validity
  - ▶ use the command `frama-c-gui -wp -wp-rte file.c`
- ▶ They ensure that each pointer (array) access has a [valid offset \(index\)](#)
- ▶ If the function assumes that an input pointer is valid, it must be [stated in its precondition](#), e.g.
  - ▶ `\valid(p)` for one pointer `p`
  - ▶ `\valid(p+0..2)` for a range of offsets `p`, `p+1`, `p+2`

## Example 3 (Continued) - Find the error

The following program is proved. Do you see any error?

```
/*@ requires \valid(p) && \valid(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
*/
int max_ptr ( int *p, int *q ) {
    if ( *p >= *q )
        return *p ;
    return *q ;
}
```

## Example 3 (Continued) - a wrong version

This is a wrong implementation that is also proved. Why?

```
/*@ requires \valid(p) && \valid(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
*/
int max_ptr ( int *p, int *q ) {
    *p = 0;
    *q = 0;
    return 0 ;
}
```

## Example 3 (Continued) - a wrong version

This is a wrong implementation that is also proved. Why?

```
/*@ requires \valid(p) && \valid(q);
   ensures \result >= *p && \result >= *q;
   ensures \result == *p || \result == *q;
*/
int max_ptr ( int *p, int *q ) {
  *p = 0;
  *q = 0;
  return 0 ;
}
```

- ▶ Our specification is incomplete
- ▶ Should say that the function cannot modify \*p and \*q

## Assigns clause

The clause `assigns v1, v2, ... , vN;`

- ▶ Part of the postcondition
- ▶ Specifies which (non local) variables can be modified by the function
- ▶ Avoids to state for all unchanged global variables  $v$ :  
`ensures \old(v) == v;`
- ▶ Avoids to forget one of them: explicit permission is required
- ▶ If nothing can be modified, specify `assigns \nothing`

## Example 3 (Continued) - Solution

This is the completely specified program:

```
/*@ requires \valid(p) && \valid(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
    assigns \nothing;
*/
int max_ptr ( int *p, int *q ) {
    if ( *p >= *q )
        return *p ;
    return *q ;
}
```



# Behaviors

## Specification by cases

- ▶ Global precondition (**requires**) applies to all cases
- ▶ Global postcondition (**ensures**, **assigns**) applies to all cases
- ▶ Behaviors define contracts (refine global contract) in particular cases
- ▶ For each case (each **behavior**)
  - ▶ the subdomain is defined by **assumes** clause
  - ▶ the behavior's precondition is defined by **requires** clauses
    - ▶ it is supposed to be true whenever **assumes** condition is true
  - ▶ the behavior's postcondition is defined by **ensures**, **assigns** clauses
    - ▶ it must be ensured whenever **assumes** condition is true
- ▶ **complete behaviors** states that given behaviors cover all cases
- ▶ **disjoint behaviors** states that given behaviors do not overlap

## Example 4

Specify using behaviors and prove the function abs:

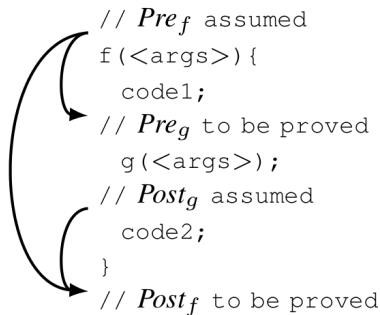
```
// returns the absolute value of x
int abs ( int x ) {
    if ( x >=0 )
        return x ;
    return -x ;
}
```

## Example 4 (Continued) - Solution

```
#include <limits.h>
/*@ requires x > INT_MIN;
    assigns \nothing;
    behavior pos:
        assumes x >= 0;
        ensures \result == x;
    behavior neg:
        assumes x < 0;
        ensures \result == -x;
    complete behaviors;
    disjoint behaviors;
*/
int abs ( int x ) {
    if ( x >= 0 )
        return x ;
    return -x ;
}
```

# Contracts and function calls

```
// Pref assumed
f(<args>){
  code1;
// Preg to be proved
  g(<args>);
// Postg assumed
  code2;
}
// Postf to be proved
```



Pre/post of the caller and of the callee have **dual roles** in the caller's proof

- ▶ Pre of the caller **is assumed**, Post of the caller **must be ensured**
- ▶ Pre of the callee **must be ensured**, Post of the callee **is assumed**

## Example 5

Specify and prove the function `max_abs`

```
int abs ( int x );
int max ( int x, int y );

// returns maximum of absolute values of x and y
int max_abs( int x, int y ) {
    x=abs(x);
    y=abs(y);
    return max(x,y);
}
```

## Example 5 (Continued) - Explain the proof failure for

```

#include <limits.h>
/*@ requires x > INT_MIN;
    ensures (x >= 0 ==> \result == x) &&
           (x < 0 ==> \result == -x);
    assigns \nothing; */
int abs ( int x );

/*@ ensures \result >= x && \result >= y;
    ensures \result == x || \result == y;
    assigns \nothing; */
int max ( int x, int y );

/*@ ensures \result >= x && \result >= -x &&
           \result >= y && \result >= -y;
    ensures \result == x || \result == -x ||
           \result == y || \result == -y;
    assigns \nothing; */
int max_abs( int x, int y ) {
    x=abs(x);
    y=abs(y);
    return max(x,y);
}

```

## Example 5 (Continued) - Explain the proof failure for

```

#include <limits.h>
/*@ requires x > INT_MIN;
    ensures (x >= 0 ==> \result == x) &&
           (x < 0 ==> \result == -x);
    assigns \nothing; */
int abs ( int x );

/*@ ensures \result >= x && \result >= y;
    assigns \nothing; */
int max ( int x, int y );

/*@ requires x > INT_MIN;
    requires y > INT_MIN;
    ensures \result >= x && \result >= -x &&
           \result >= y && \result >= -y;
    ensures \result == x || \result == -x ||
           \result == y || \result == -y;
    assigns \nothing; */
int max_abs( int x, int y ) {
    x=abs(x);
    y=abs(y);
    return max(x,y);
}

```

## Example 5 (Continued) - Solution

```

#include <limits.h>
/*@ requires x > INT_MIN;
    ensures (x >= 0 ==> \result == x) &&
           (x < 0 ==> \result == -x);
    assigns \nothing; */
int abs ( int x );

/*@ ensures \result >= x && \result >= y;
    ensures \result == x || \result == y;
    assigns \nothing; */
int max ( int x, int y );

/*@ requires x > INT_MIN;
    requires y > INT_MIN;
    ensures \result >= x && \result >= -x &&
           \result >= y && \result >= -y;
    ensures \result == x || \result == -x ||
           \result == y || \result == -y;
    assigns \nothing; */
int max_abs( int x, int y ) {
    x=abs(x);
    y=abs(y);
    return max(x,y);
}

```



# Outline

Frama-C Overview

**Formal Specification and Deductive Verification with WP**

Overview of ACSL and WP

Function contracts

**Programs with loops**

My proof fails... What to do?

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

Combinations of Analyses

Conclusion

# Loops and automatic proof

- ▶ What is the issue with loops? Unknown, **variable number of iterations**
- ▶ The only possible way to handle loops: **proof by induction**
- ▶ Induction needs a suitable **inductive property**, that is proved to be
  - ▶ satisfied just before the loop, and
  - ▶ satisfied after  $k + 1$  iterations whenever it is satisfied after  $k \geq 0$  iterations
- ▶ Such inductive property is called **loop invariant**
- ▶ The verification conditions for a loop invariant include two parts
  - ▶ **loop invariant initially holds**
  - ▶ **loop invariant is preserved** by any iteration

## Loop invariants - some hints

How to find a suitable loop invariant? Consider two aspects:

- ▶ identify **variables modified in the loop**
  - ▶ variable number of iterations prevents from deducing their values (relationships with other variables)
  - ▶ define their possible value intervals (relationships) after  $k$  iterations
  - ▶ use **loop assigns** clause to list variables that (might) have been assigned so far after  $k$  iterations
- ▶ identify realized actions, or **properties already ensured by the loop**
  - ▶ what **part of the job** already realized after  $k$  iterations?
  - ▶ what **part of the expected loop results** already ensured after  $k$  iterations?
  - ▶ why the next iteration can proceed as it does? ...

A **stronger property** on each iteration may be required to prove the final result of the loop

Some experience may be necessary to find appropriate loop invariants

## Loop invariants - more hints

Remember: a loop invariant must be true

- ▶ before (the first iteration of) the loop, even if no iteration is possible
- ▶ after any complete iteration even if no more iterations are possible
- ▶ in other words, any time before the loop condition check

In particular, a **for** loop

```
for (i=0; i<n; i++) { /* body */ }
```

should be seen as

```
i=0;           // action before the first iteration
while ( i<n ) // an iteration starts by the condition check
{
    /* body */
    i++;       // last action in an iteration
}
```

# Loop termination

- ▶ Program termination is undecidable
- ▶ A tool cannot deduce neither the exact number of iterations, nor even an upper bound
- ▶ If an upper bound is given, a tool can check it by induction
- ▶ An upper bound on the number of remaining loop iterations is the key idea behind the loop variant

## Terminology

- ▶ **Partial correctness:** if the function terminates, it respects its specification
- ▶ **Total correctness:** the function terminates, and it respects its specification

## Loop variants - some hints

- ▶ Unlike an invariant, a loop variant is an **integer expression**, not a predicate
- ▶ Loop variant is **not unique**: if  $V$  works,  $V + 1$  works as well
- ▶ No need to find a precise bound, any working loop variant is OK
- ▶ To find a variant, **look at the loop condition**
  - ▶ For the loop **while**( $\text{exp1} > \text{exp2}$  ), try **loop variant**  $\text{exp1} - \text{exp2}$ ;
- ▶ In more complex cases: ask yourself why the loop terminates, and try to give an integer upper bound on the number of remaining loop iterations

## Example 6

Specify and prove the function `all_zeros`:

```
// returns a non-zero value iff all elements
// in a given array t of n integers are zeros
int all_zeros(int t[], int n) {
    int k;
    for(k = 0; k < n; k++)
        if (t[k] != 0)
            return 0;
    return 1;
}
```

## Example 6 (Continued) - Solution

```

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

```



## Example 7

Specify and prove the function `find_min`:

```
// returns the index of the minimal element
// of the given array a of size length
int find_min(int* a, int length) {
    int min, min_idx;
    min_idx = 0;
    min = a[0];
    for (int i = 1; i < length; i++) {
        if (a[i] < min) {
            min_idx = i;
            min = a[i];
        }
    }
    return min_idx;
}
```

## Example 7 (Continued) - Solution

```

/*@ requires length > 0 && \valid(a+(0..length-1));
   assigns \nothing;
   ensures 0<=\result<length &&
      (\forall integer j; 0<=j<length ==> a[\result]<=a[j]);*/
int find_min(int* a, int length) {
  int min, min_idx;
  min_idx = 0;
  min = a[0];
  /*@ loop invariant 0<=i<=length && 0<=min_idx<length;
     loop invariant \forall integer j; 0<=j<i ==> min<=a[j];
     loop invariant a[min_idx]==min;
     loop assigns min, min_idx, i;
     loop variant length - i; */
  for (int i = 1; i<length; i++) {
    if (a[i] < min) {
      min_idx = i;
      min = a[i];
    }
  }
  return min_idx;
}

```

# Outline

Frama-C Overview

**Formal Specification and Deductive Verification with WP**

Overview of ACSL and WP

Function contracts

Programs with loops

**My proof fails... What to do?**

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

Combinations of Analyses

Conclusion

# Proof failures

A proof of a VC for some annotation can fail for **various reasons**:

- ▶ incorrect implementation (→ check your code)
- ▶ incorrect annotation (→ check your spec)
- ▶ missing or erroneous (previous) annotation (→ check your spec)
- ▶ insufficient timeout (→ try longer timeout)
- ▶ complex property that automatic provers cannot handle.

## Analysis of proof failures

When a proof failure is due to the specification, the erroneous annotation may be **not obvious to find**. For example:

- ▶ proof of a “**loop invariant preserved**” may fail in case of
  - ▶ incorrect loop invariant
  - ▶ incorrect loop invariant in a previous, or inner, or outer loop
  - ▶ missing **assumes** or **loop assumes** clause
  - ▶ too weak precondition
  - ▶ ...
- ▶ proof of a **postcondition** may fail in case of
  - ▶ incorrect loop invariant (too weak, too strong, or inappropriate)
  - ▶ missing **assumes** or **loop assumes** clause
  - ▶ inappropriate postcondition in a called function
  - ▶ too weak precondition
  - ▶ ...

## Analysis of proof failures (Continued)

- ▶ Additional statements (`assert`, `lemma`, ...) may help the prover
  - ▶ They can be provable by the same (or another) prover or checked elsewhere
- ▶ Separating independent properties (e.g. in separate, non disjoint behaviors) may help
  - ▶ The prover may get lost with a bigger set of hypotheses (some of which are irrelevant)

### When nothing else helps to finish the proof:

- ▶ an `interactive proof assistant` can be used
- ▶ Coq, Isabelle, PVS, are not that scary: we may need only a small portion of the underlying theory

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

## Value Analysis

- Value

- Eva

- Derived analyses

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

Combinations of Analyses

Conclusion

# Goal

In this part, we will see

- ▶ how Value Analysis works
- ▶ how evolved is the new reimplementaion
- ▶ how useful are derived analyses



# Value Analysis Overview

## Domain of variations of variables of the program

- ▶ abstract interpretation
- ▶ automatic analysis
- ▶ correct over-approximation
- ▶ alarms for potential invalid operations
- ▶ alarms for potential invalid ACSL annotations
- ▶ ensures the absence of runtime errors
- ▶ graphical interface: display the domain of each variable at each program point

# Value Historical Domains

- ▶ **One hard-wired non-relation domain**
  - ▶ **small sets** of integers, e.g.  $\{5, 18, 42\}$
  - ▶ reduced product of **intervals**: quick to compute, e.g.  $[1..41]$
  - ▶ **modulo**: pretty good for arrays of structures, e.g.  $[1..41], 1\%2$
  - ▶ precise representation of **pointers**, e.g. *32-bit aligned offset from  $\&t[0]$*
  - ▶ **initialization** information
- ▶ **ad-hoc trace partitioning**
- ▶ **alarms** on potential RTE and invalid annotations
- ▶ **highly optimized**
  - ▶ excellent results on embedded code
  - ▶ possible usage in low-level C code

demo

# Value Parameterization

- ▶ Value is **automatic**
- ▶ but requires **fine-tuned parameterization** to be precise/efficient
- ▶ **trade-off** between time efficiency vs memory efficiency vs precision
- ▶ **stubbing**: `main` function and missing library function
  - ▶ either provide C code or ACSL specification (usually, **assigns**)
  - ▶ similar to testing
- ▶ lots of parameters, but a few almost always useful

# Value Parameterization

Cont'd

- ▶ **level  $n$** : superpose up to  $n$  states during the analysis

```
int main(void) {  
    int t[10];  
    for(int i = 0; i < 10; i++) t[i] = i;  
}
```

# Value Parameterization

Cont'd

- ▶ **level  $n$** : superpose up to  $n$  states during the analysis

```
int main(void) {
    int t[10];
    for(int i = 0; i < 10; i++) t[i] = i;
}
```

- ▶ case splitting through **ACSL disjunctions**

```
int gcd(int x, int y) {
    int a = x, b = y;
    while(b!=0) {
        int tmp = a % b;
        a = b; b = tmp;
    }
    return a;
}
```

# Value Parameterization

Cont'd

- ▶ **level  $n$** : superpose up to  $n$  states during the analysis

```
int main(void) {
    int t[10];
    for(int i = 0; i < 10; i++) t[i] = i;
}
```

- ▶ case splitting through **ACSL disjunctions**

```
int gcd(int x, int y) {
    int a = x, b = y;
    /*@ assert b < 0 || b == 0 || b > 0; */
    while(b!=0) {
        int tmp = a % b;
        a = b; b = tmp;
        /*@ assert b < 0 || b == 0 || b > 0; */
    }
    return a;
}
```

# Eva, Evolved Value Analysis

## Major reimplementation in Frama-C Aluminium

- ▶ 100% compatible
- ▶ generic analysis on the abstract domain
- ▶ allow combination of abstract domains and some inter-reductions of their states
- ▶ comparable analysis time for better results
- ▶ should be easy to add new domain
  - ▶ Apron
  - ▶ conditional predicates [Blazy, Bühler & Yakobowski @SCP'16]

# Eva Domains

the new design relies on the separation between:

- ▶ **values**
  - ▶ abstraction of the possible C **values** of an expression
  - ▶ **abstract** transformers for arithmetic **operators** on expressions
  - ▶ **communication interface** for abstract domains
- ▶ **domains**
  - ▶ abstraction of the set of **reachable states** at a program point
  - ▶ **abstract** transformers of **states** through statements
  - ▶ can be queried for the values of some C expressions
- ▶ **everyone can implement new domains easily**

D. Bühler's PhD works



# Derived analyses

- ▶ results from Value/Eva are useful for other plug-ins
  - ▶ domains of variations
  - ▶ **aliasing** information
  - ▶ **dependency** information
- ▶ program dependency graph (PDG)
  - ▶ slicing
  - ▶ impact analysis
- ▶ domain specific analysis
  - ▶ information flow analysis [Assaf & al @SEC'13]
  - ▶ concurrency analysis

example

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

Value Analysis

**Structural Unit Testing with PathCrawler**

Runtime Verification with E-ACSL

Combinations of Analyses

Conclusion

# Goal

In this part, we will see

- ▶ how to generate test cases using Frama-C/PathCrawler,
- ▶ how to specify test parameters,
- ▶ how to specify an oracle.

## Outline

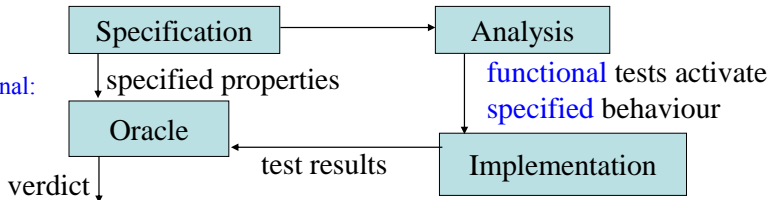
1. **Structural testing: a brief introduction**
2. **PathCrawler tool**
3. **Test parameters**
4. **Oracle and program debugging**
5. **Structural test for other properties/purposes**
6. **Strengths and limits of structural testing**
7. **Bypassing the limits**

## Outline

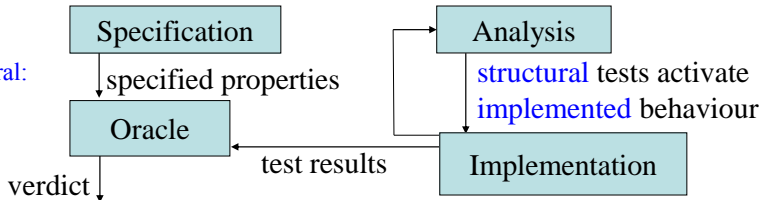
- 1. Structural testing: a brief introduction**
2. PathCrawler tool
3. Test parameters
4. Oracle and program debugging
5. Structural test for other properties/purposes
6. Strengths and limits of structural testing
7. Bypassing the limits

## Structural vs. functional testing

Functional:



Structural:





list

## Unit structural testing is useful

Manually created functional test cases do not cover all the code

- Certain « functional » test cases can be missed
- Certain parts of code can depend on implementation choices and cannot be properly covered by the specification

Evaluation of structural coverage

Adding test cases to complete structural coverage



## Unit structural testing can be mandatory

### List

Development, evaluation and certification standards

- Common Criteria for IT Security Evaluation
- DO-178B (avionics)
- ECCS-E-ST-40C (space)
- IEC/EN 61508 (*Electronic Safety-related Systems*) & derived standards:
  - ♦ ISO 26262 (automotive)
  - ♦ IEC/EN 50128 (rail)
  - ♦ IEC/EN 60601 (medical)
  - ♦ EC/EN 61513 (nuclear)
  - ♦ IEC/EN 60880 (nuclear safety-critical)
  - ♦ IEC/EN 61511 (process e.g. petrochemical, pharmaceutical)



## CFG and code coverage by example

## list

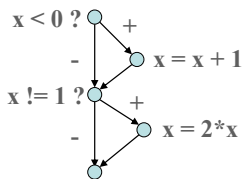
C code

```

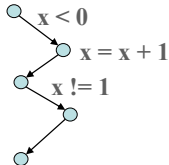
1 int f(int x){
2   if(x < 0)
3     x = x + 1;
4   if(x != 1)
5     x = 2*x;
6   return x; }

```

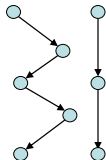
control-flow graph (CFG)



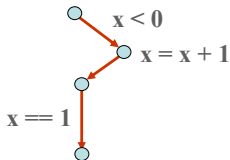
statement coverage



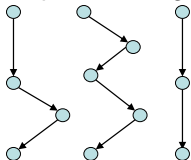
branch coverage



infeasible path



all-path coverage



# Path predicate (path condition) by example

## List

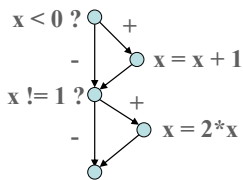
C code

```

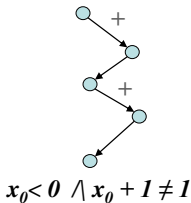
1 int f(int x){
2   if(x < 0)
3     x = x + 1;
4   if(x != 1)
5     x = 2*x;
6   return x; }

```

control-flow graph (CFG)



path predicate



$x_0 \geq 0 \wedge x_0 = 1$



$x_0 < 0 \wedge x_0 + 1 = 1$

infeasible path



unsatisfiable path predicate



list

## Automated structural testing... Why?

Achieving desired test coverage manually is costly

Must be done again after any code modification

Infeasibility of a test objective can be difficult to show manually

**Automated structural testing tools can be used**

- to reach the uncovered objectives,
- to determine that some of them are unreachable,
- with a low cost overhead

## Outline

1. Structural testing: a brief introduction
- 2. PathCrawler tool**
3. Test parameters
4. Oracle and program debugging
5. Strengths and limits of structural testing
6. Structural test for other properties/purposes
7. Bypassing the limits

## PathCrawler tool

- Concolic testing tool for C developed at CEA LIST
- Input: a complete compilable source code
- Automatically creates test cases to cover program paths (explored in depth-first search)
- Uses code instrumentation, concrete and symbolic execution, constraint solving
- Exact semantics: don't rely on concrete values to approximate the path predicate
- Similar to PEX, DART/CUTE, KLEE, SAGE etc.



## PathCrawler explores the tree of feasible paths

List

*depth-first search with non-deterministic choice of suffix*

$$\text{test1: } x = -5 \quad x_0 < 0 \xrightarrow[x_1 = x_0 + 1]{+2} x_1 \neq 1 \xrightarrow[x_2 = 2x_1]{+4}$$

```
1 int f(int x){
2   if(x < 0)
3     x = x + 1;
4   if(x != 1)
5     x = 2*x;
6   return x; }
```





## PathCrawler explores the tree of feasible paths

List

*depth-first search with non-deterministic choice of suffix*

$$\text{test1: } x = -5 \quad x_0 < 0 \xrightarrow[x_1 = x_0 + 1]{+2} x_1 \neq 1 \xrightarrow[x_2 = 2x_1]{+4} x_0 < 0 \wedge (x_0 + 1) \neq 1$$

```
1 int f(int x){
2   if(x < 0)
3     x = x + 1;
4   if(x != 1)
5     x = 2*x;
6   return x; }
```





## PathCrawler explores the tree of feasible paths

list

*depth-first search with non-deterministic choice of suffix*

$$\text{test1: } x = -5 \quad x_0 < 0 \xrightarrow[\substack{+2 \\ x_1 = x_0 + 1}]{\quad} x_1 \neq 1 \xrightarrow[\substack{+4 \\ x_2 = 2x_1}]{\quad} x_0 < 0 \wedge (x_0 + 1) \neq 1$$

-4 →

$x_0 < 0 \wedge (x_0 + 1) = 1$  **infeas.**

```
1 int f(int x) {
2   if(x < 0)
3     x = x + 1;
4   if(x != 1)
5     x = 2*x;
6   return x; }
```





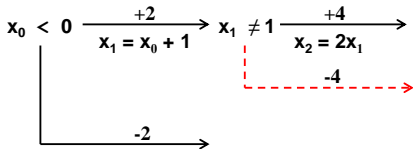


# PathCrawler explores the tree of feasible paths

list

*depth-first search with non-deterministic choice of suffix*

test1:  $x = -5$



$$x_0 < 0 \wedge (x_0 + 1) \neq 1$$

$x_0 < 0 \wedge (x_0 + 1) = 1$  **infeas.**

$$x_0 \geq 0$$

```

1 int f(int x) {
2   if(x < 0)
3     x = x + 1;
4   if(x != 1)
5     x = 2*x;
6   return x; }

```

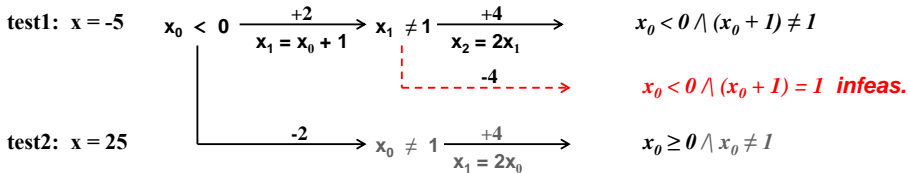




## PathCrawler explores the tree of feasible paths

List

*depth-first search with non-deterministic choice of suffix*



```

1 int f(int x) {
2   if(x < 0)
3     x = x + 1;
4   if(x != 1)
5     x = 2*x;
6   return x; }

```



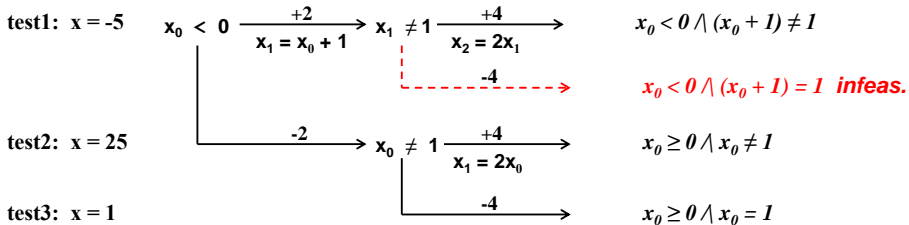




# PathCrawler explores the tree of feasible paths

List

*depth-first search with non-deterministic choice of suffix*



## pathcrawler-online.com

### Freely available test-case generation web service

- Instead of open-source or demonstration version
- No porting, no installation, universal user interface
- Well adapted to
  - Teaching
  - Use by project partners
  - Evaluation, understanding of Precondition and Oracle
- **Limited version (contact us for unlimited access)**

### During the tutorial

- **Browser: no cache recommended**
- **Do not start several test generation sessions in parallel**



## Example 1. Robust implementation of `TriType`

### list

Simple program `TriType`

- inputs: three floating-point numbers  $i, j, k$
- returns the type of the triangle with sides  $i, j, k$ :  
3 (not a triangle), 2 (equilateral), 1 (isosceles), 0 (other)

**Robust : validity of inputs is tested (“not a triangle”)**

⇒ **Any test case can be interesting and useful**

**“Test with predefined params” on [pathcrawler-online.com](http://pathcrawler-online.com)**

**Observe the number of test cases. Check the results.**

## PathCrawler outputs

- **A suite of test cases including**
  - ♦ Input values (check these for Example 1)
  - ♦ Concrete outputs (check these for Example 1)
  - ♦ Symbolic outputs (better illustrated by Example 5)
  - ♦ Path predicate (better illustrated by Example 5)
  - ♦ Test driver
  - ♦ Oracle verdict (better illustrated by Example 10)
- **Explored program paths with**
  - ♦ their status (covered, infeasible, assume violated ...)
  - ♦ path predicate (only for covered paths in online version)

## Outline

1. Structural testing: a brief introduction
2. PathCrawler tool
- 3. Test parameters**
4. Oracle and program debugging
5. Strengths and limits of structural testing
6. Structural test for other properties/purposes
7. Bypassing the limits



## Example 2. Non robust implementation of Tritype

**No validity check lines 10-13, no “not a triangle” answer  
⇒ Are the test cases still interesting?**

**“Test with predefined params” on [pathcrawler-online.com](https://pathcrawler-online.com)  
Observe the number of test cases. Check the results.**

**Where is the problem?  
Do we really want such input values in this case?**

## Exercise 3. Customize test parameters for Tritype

How to generate appropriate test cases only ?  
⇒ define a precondition!

Exercise. Start from Example 2. “Customize test parameters”

- Restrict the domains of inputs  $i, j, k$  to non negative values:

$[0 .. 1.7976931348623157e+308]$

- Add 3 unquantified preconditions:

$$i + j > k$$

$$j + k > i$$

$$i + k > j$$

- Confirm parameters and check the results.



## Example 4. C Precondition for Tritype

### List

**Another way to define a precondition  
⇒ in a C function**

`Tritype_precond` returns 1 iff the precondition is verified

**“Customize test parameters” on [pathcrawler-online.com](https://pathcrawler-online.com)  
to check that Pathcrawler has activated the C precondition.**

**Confirm & observe the number of test cases & results.**

## Test parameters

- **Define admissible inputs (precondition)**
  - ♦ Domains of input variables
  - ♦ Relations between variables...
- **Wrong test parameters may**
  - ♦ Indicate inexistent bugs (the bug is in the input)
  - ♦ Provoke runtime errors

## Example 5. Merge with default parameters

Merge of two sorted arrays  $t_1$ ,  $t_2$  into a sorted array  $t_3$

- inputs: arrays  $t_1[3]$ ,  $t_2[3]$ ,  $t_3[6]$  of fixed size

“Test with predefined params” on [pathcrawler-online.com](https://pathcrawler-online.com)

Check the concrete outputs.

**What is wrong with the concrete outputs?**

This example also illustrates well the information on array inputs, symbolic outputs and path predicate included in a test-case





## Example 7. Merge with pointer inputs

### List

Merge of two sorted arrays  $t_1$ ,  $t_2$  into a sorted array  $t_3$

- inputs: arrays  $t_1[ ]$ ,  $t_2[ ]$ ,  $t_3[ ]$  of variable size, 11 the size of  $t_1$ , 12 the size of  $t_2$ , 11+12 the size of  $t_3$
- precondition  $t_1, t_2$  ordered arrays predefined
- reduced domains of elements  $[-100,100]$  predefined

**“Test with predefined params” on [pathcrawler-online.com](http://pathcrawler-online.com)  
Check the results.**

**Why are there errors?**

## Exercise 8. Input arrays (pointers) size

### List

**t1, t2, t3 should contain resp. l1, l2, l1+l2 allocated elements.  
Wrong input array size => Runtime errors while executing tests!**

**Exercise. Start from Example 7. “Customize test parameters”**

- **Specify domains for  $\dim(t1)$ ,  $\dim(t2)$ ,  $\dim(t3)$**

$$0 \leq \dim(t3) \leq 6$$

$$0 \leq \dim(t2) \leq 3$$

$$0 \leq \dim(t1) \leq 3$$

- **Add three unquantified preconditions:**

$$\dim(t1) = l1$$

$$\dim(t2) = l1$$

$$\dim(t3) = l1 + l2$$

- **Confirm parameters and check the results.**

**Are there errors? Why? How many test cases are generated?**



- In presence of loops, all-path criterion may generate too many test cases
- The user may want to limit their number
- **k-path coverage restricts the all-path criterion to paths with at most k consecutive iterations of each loop (k=0,1,2...)**



## Exercise 9. Merge with partial test coverage: k-path

### List

To reduce the number of test cases, modify test criterion.

**Exercise.** Continue Exercise 8 with the same test parameters you defined. “Customize test parameters”

- Set “Path selection strategy” to 2 (for k-path with  $k=2$ )
- Confirm parameters and check the results.

How many test cases are generated now?

## Outline

1. Structural testing: a brief introduction
2. PathCrawler tool
3. Test parameters
- 4. Oracle and program debugging**
5. Strengths and limits of structural testing
6. Structural test for other properties/purposes
7. Bypassing the limits

# Oracle

Role of an oracle:

- examines the inputs and outputs of each test
- decides whether the implementation has given the expected results
- provides a verdict (success, failure)

An oracle can be provided by

- another, or previous implementation
- checking the results without implementing the algorithm



## Exercise 10a. Oracle and debugging

---

### List

Start from Example 10a, “Customize test parameters” to see an example of an oracle

**Is this oracle complete ?**



## Exercise 10b. Oracle and debugging

---

### List

Start from Example 10b, “Customize test parameters” to see another example of an oracle

**Is this oracle complete ?**



## Exercise 10c. Oracle and debugging

---

### List

Start from Example 10a, “Customize test parameters” to see the predefined oracle

**Exercise. Confirm parameters and check the results.  
Can you find an error in the implementation?**

**Hint: The paths of failed test cases have a common part...**

## Outline

1. Structural testing: a brief introduction
2. PathCrawler tool
3. Test parameters
4. Oracle and program debugging
- 5. Structural test for other properties/purposes**
6. Strengths and limits of structural testing
7. Bypassing the limits





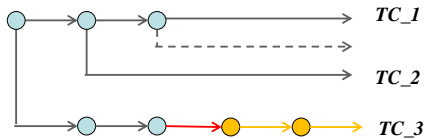
## Structural test for other properties or purposes

### List

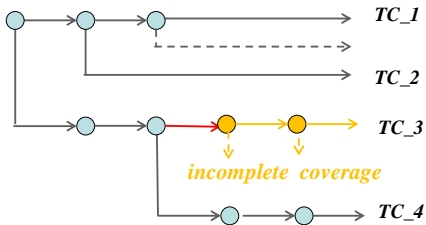
PathCrawler explores the implementation and can also be used to check:

- for **runtime errors** during program execution (seen in Ex.7)
- for **anomalies** detected during analysis of the covered paths:
  - uninitialised variables
  - buffer overflow
  - integer overflow
  - ...
- whether the implementation performs **unnecessary computation**
- the effective **execution time** of each path (at least for one set of inputs), by running the generated tests on a platform which can measure execution time
- for unreachable or **“dead” code**: check infeasible partial paths.  
If all paths leading to the code are infeasible then the code is unreachable (for the given precondition): is this intentional ?

## Runtime error or anomaly: search space is pruned



*error during test execution or  
anomaly detected by analysis*





## Example Uninit. Uninitialised variable

### List

In this example, the local variables are not always initialised before their value is read. This is a typical “anomaly”: probably a bug but does not cause a run-time error.

**“Test with predefined parameters” and check the results.**

**Are there any errors or warnings? Why?  
Are all feasible paths covered?**



## Example UC. Unnecessary computation

### List

`Bsearch` is an implementation of dichotomic search for value `x` in sorted array `A`.

**“Customize test parameters” to see the predefined oracle and parameters. Confirm them and check the results.**

**Examine the predicates and input values of the cases where `x` is present. Is this an efficient implementation?**

## Outline

1. Structural testing: a brief introduction
2. PathCrawler tool
3. Test parameters
4. Oracle and program debugging
5. Structural test for other properties/purposes
- 6. Strengths and limits of structural testing**
7. Bypassing the limits



## Dichotomic search: structural vs. other strategies

### list

**Example:** dichotomic search for a value  $x$  in a sorted array  $A[10]$ .

**Random testing:** Unlikely to construct cases in which  $x$  equals one of the elements of  $A$  and to detect false negatives ( $x$  not detected when present)

**Functional testing:** Constructs

- many cases in which  $x$  is present (probably from 1 to 10?) and
- fewer cases in which  $x$  is absent (1 or 2 ?)

**Structural testing:** Constructs a case

- for each position in  $A$  for which  $x$  can be detected and
- for each relation to elements of  $A$  for which absence of  $x$  is detected.

Structural test. constructs more presence cases than random, more absence cases than functional, rarely constructs cases where  $x$  is present by chance.



## Example Chance. Failures by chance?

### list

`Bsearch` is another implementation of dichotomic search for value  $x$  in sorted array  $A$ . It contains a bug which can result in false positives ( $x$  present but not detected).

The parameters are the same as in the previous example. Confirm them and check the results.

Is the presence or absence of  $x$  in  $A$  always determined by the path predicate?

Hint: look at failing cases or those where  $x$  is present.



## Example 11. Limitations of structural testing

### list

`Bsearch` is another erroneous implementation of dichotomic search for value  $x$  in sorted array `A`.

The parameters are the same as in the previous example.  
Confirm them and check the results.

Are there any failures?



## Limitations of structural testing

### List

Structural testing is

- **effective** when a bug is **always revealed** by a path,
- **less so** when **only some of the values** which activate the path cause the bug to be revealed

**PathCrawler chooses arbitrary values to test each path**

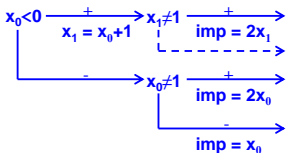
**They may not be the values which will reveal a bug**

**We can make PathCrawler *go looking for bugs*  
by sub-dividing the paths**

## Outline

1. Structural testing: a brief introduction
2. PathCrawler tool
3. Test parameters
4. Oracle and program debugging
5. Structural test for other properties/purposes
6. Strengths and limits of structural testing
7. **Bypassing the limits**

list



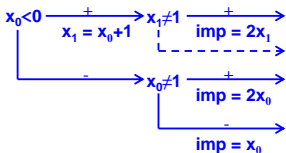
## implementation

```

int f(int x){
  if(x < 0)
    x = x + 1;
  if(x != 1)
    x = 2*x;
  return x; }

```

list



implementation

```
int f(int x){
  if(x < 0)
    x = x + 1;
  if(x != 1)
    x = 2*x;
  return x; }
```

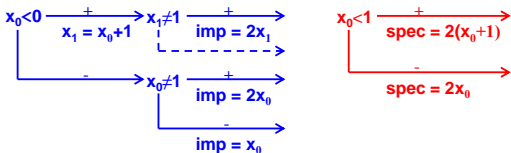
specification

*If  $x$  is less than 1 then  
the result should be  $2(x + 1)$   
else the result should be  $2x$*



## Cross-checking conformity with a specification

List



implementation

specification

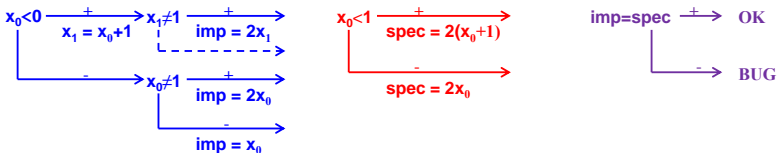
```
int f(int x){  
  if(x < 0)  
    x = x + 1;  
  if(x != 1)  
    x = 2*x;  
  return x; }
```

```
int spec_f(int x){  
  if(x < 1)  
    x = 2*(x + 1);  
  else  
    x = 2*x;  
  return x; }
```



## Cross-checking conformity with a specification

list



implementation

```

int f(int x) {
  if(x < 0)
    x = x + 1;
  if(x != 1)
    x = 2*x;
  return x; }
  
```

specification

```

int spec_f(int x) {
  if(x < 1)
    x = 2*(x + 1);
  else
    x = 2*x;
  return x; }
  
```

comparison

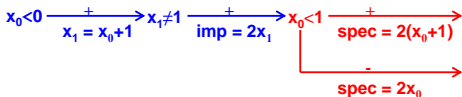
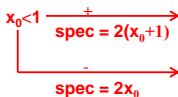
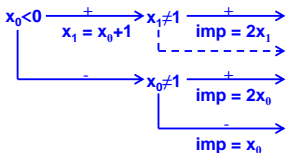
```

int cross_f(int x) {
  int imp = f(x);
  int spec = spec_f(x);
  if(imp != spec)
    return 0;
  else return 1; }
  
```



## Cross-checking conformity with a specification

List

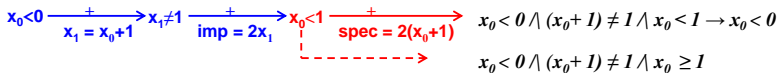
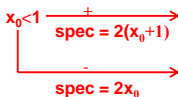
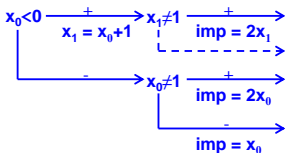


```
int cross_f(int x){
    int imp = f(x);
    int spec=spec_f(x);
    if(imp!=spec)
        return 0;
    else return 1; }
```



## Cross-checking conformity with a specification

list



```

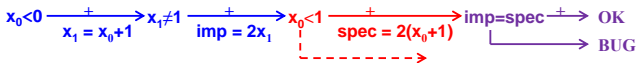
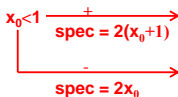
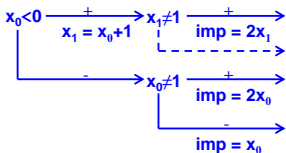
int cross_f(int x) {
    int imp = f(x);
    int spec = spec_f(x);
    if (imp != spec)
        return 0;
    else return 1; }
  
```





## Cross-checking conformity with a specification

List



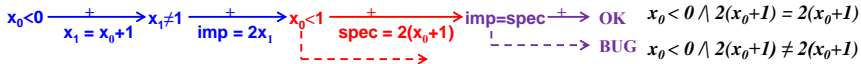
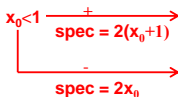
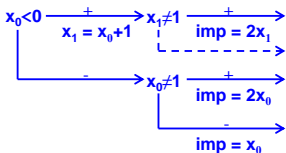
```

int cross_f(int x){
    int imp = f(x);
    int spec=spec_f(x);
    if(imp!=spec)
        return 0;
    else return 1; }
  
```



# Cross-checking conformity with a specification

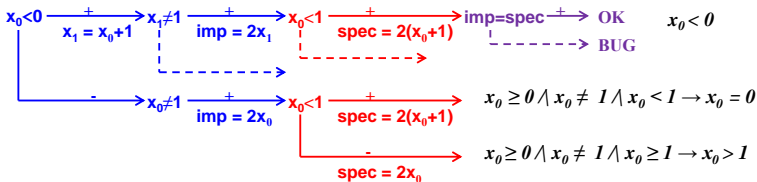
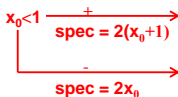
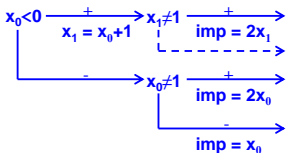
list





# Cross-checking conformity with a specification

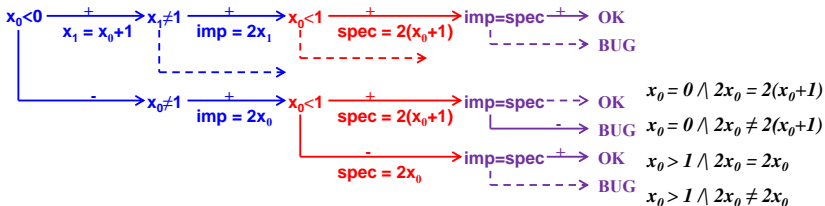
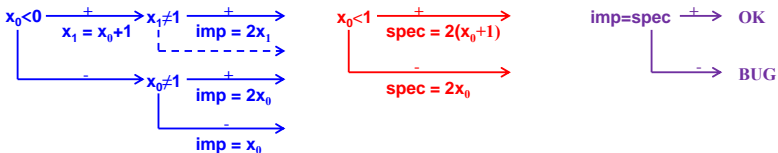
list





# Cross-checking conformity with a specification

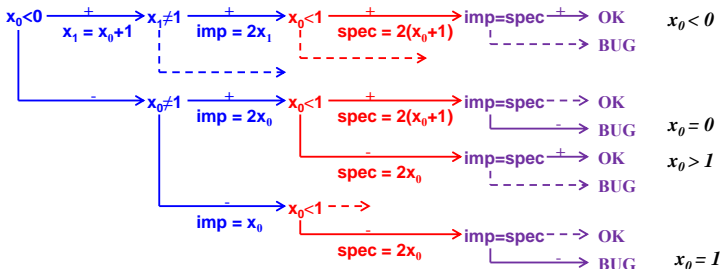
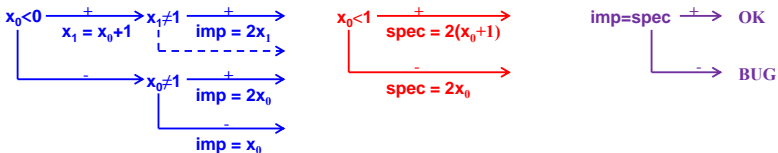
list





# Cross-checking conformity with a specification

list





## Example 12. Testing conformity with a specification

### List

`Spec_Bsearch` is a specification for `Bsearch`, similar to the oracle. Test function `CompareBsearchSpec` that

- stores inputs, calls `Bsearch`,
- calls `Spec_Bsearch` to provide a verdict.

All-path testing will try cover all combinations of paths in `Bsearch` and `Spec_Bsearch`.

**“Customize test parameters” to see the predefined oracle and parameters. Confirm them and check the results.**

**Why are failures reported this time? Can you find the bug?**

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

Value Analysis

Structural Unit Testing with PathCrawler

**Runtime Verification with E-ACSL**

E-ACSL Specification Language

E-ACSL Plug-in

Combinations of Analyses

Conclusion

# Goal

In this part, we will see

- ▶ the differences between ACSL and E-ACSL
- ▶ how to check E-ACSL properties at runtime
- ▶ how static analyses improve efficiency of the monitor



# 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, K. & S. @RV'13]
  - ▶ few restrictions
  - ▶ one compatible semantics change

# E-ACSL Restrictions

- ▶ **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
```

- ▶ **sets** must be finite
- ▶ no lemmas nor axiomatics
- ▶ no way to express **termination** properties

# 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**

## Error in annotations?

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

# E-ACSL 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 div(int x, int y) {
  /*@ assert y-1 != 0; */
  return x / (y-1);
}

int div(int x, int y) {
  /*@ assert y-1 != 0; */
  e_acsl_assert(y-1 != 0);
  return x / (y-1);
}
```

E-ACSL  $\longrightarrow$

# 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 div(int x, int y) {
  /*@ assert y-1 != 0; */
  return x / (y-1);
}

```

E-ACSL  $\longrightarrow$

```

int div(int x, int y) {
  /*@ assert y-1 != 0; */
  e_acsl_assert(y-1 != 0);
  return x / (y-1);
}

```

- ▶ the general translation is more complex than it may look

## E-ACSL Integer Support

- ▶ use **GMP library** for mathematical integers

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

## E-ACSL Integer Support

- ▶ use **GMP library** for mathematical integers

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

- ▶ how to **restrict GMPs** as most as possible? on-the-fly **typing**

almost no GMP in practice

[Jakobsson, K. & S. @JFLA'15]



# E-ACSL RTE Detection

must prevent introducing RTE when translating annotations

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

# E-ACSL RTE Detection

must prevent introducing RTE when translating annotations

```
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;
}
```

# E-ACSL RTE Detection

must prevent introducing RTE when translating annotations

```

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

```

E-ACSL  $\longrightarrow$

```

int foo(int u, int v) {
  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 RTE Detection

must prevent introducing RTE when translating annotations

```

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

```

E-ACSL  $\longrightarrow$

```

int foo(int u, int v) {
  e_acsl_assert(u/v == 2);
  return u/v;
}

```

RTE plug-in  
↓

```

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  $\longleftarrow$

```

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

```

# E-ACSL Memory Observation

- ▶ memory-related constructs like `\valid`, `\initialized`, `\block_length`, `\base_addr`, `\offset` require to know the memory structure at runtime
- ▶ C library for memory observation
  - ▶ Patricia-trie based implementation [K., Petiot & S. @SAC'13]
  - ▶ New shadow-memory based implementation [with K. Vorobyov]
- ▶ once again the translation is quite heavy
- ▶ `dataflow analysis` to instrument the code only when required
  - ▶ `backward`
  - ▶ `over-approximating`
  - ▶ parameterized by an `alias analysis`
  - ▶ [Jakobsson, K. & S. @JFLA'15 & @SAC'15]

# E-ACSL Memory Observation

```
void f(void) {  
    int x, y, z, *p;  
  
    p = &x;  
    x = 0;  
    y = 1;  
    z = 2;  
  
    /*@ assert \valid(p); */  
  
    *p = 3;  
  
}
```

# E-ACSL Memory Observation

```
void f(void) {
  int x, y, z, *p;
  // allocations of local variables
  __store_block((void *)&p, 4U); __store_block((void *)&z, 4U);
  __store_block((void *)&y, 4U); __store_block((void *)&x, 4U);
  __full_init((void *)&p); p = &x; // initialization of p
  __full_init((void *)&x); x = 0; // initialization of x
  __full_init((void *)&y); y = 1; // initialization of y
  __full_init((void *)&z); z = 2; // initialization of z
  // validity check
  /*@ assert \valid(p); */
  { int __e_acsl_initialized, __e_acsl_and;
    __e_acsl_initialized = __initialized((void *)&p, sizeof(int *));
    if (__e_acsl_initialized) { int __e_acsl_valid;
      __e_acsl_valid = __valid((void *)p, sizeof(int));
      __e_acsl_and = __e_acsl_valid;
    } else __e_acsl_and = 0;
    e_acsl_assert(__e_acsl_and); }
  *p = 3;
  // free allocated variables
  __delete_block((void *)&p); __delete_block((void *)&z);
  __delete_block((void *)&y); __delete_block((void *)&x);
}
```

# E-ACSL Memory Observation

```

void f(void) {
  int x, y, z, *p;
  // allocations of local variables
  __store_block((void *)&p, 4U); __store_block((void *)&z, 4U);
  __store_block((void *)&y, 4U); __store_block((void *)&x, 4U);
  __full_init((void *)&p); p = &x; // initialization of p
  __full_init((void *)&x); x = 0; // initialization of x
  __full_init((void *)&y); y = 1; // initialization of y
  __full_init((void *)&z); z = 2; // initialization of z
  // validity check
  /*@ assert \valid(p); */
  { int __e_acsl_initialized, __e_acsl_and;
    __e_acsl_initialized = __initialized((void *)&p, sizeof(int *));
    if (__e_acsl_initialized) { int __e_acsl_valid;
      __e_acsl_valid = __valid((void *)p, sizeof(int));
      __e_acsl_and = __e_acsl_valid;
    } else __e_acsl_and = 0;
    e_acsl_assert(__e_acsl_and); }
  *p = 3;
  // free allocated variables
  __delete_block((void *)&p); __delete_block((void *)&z);
  __delete_block((void *)&y); __delete_block((void *)&x);
}

```



## Possible Usage in Combination with Other Tools

- ▶ check unproved properties of **static analyzers** (e.g. Value, WP)
- ▶ check the absence of **runtime error** in combination with RTE
- ▶ check **memory consumption** and **violations** (use-after-free)
- ▶ help **testing tools** by checking properties which are not easy to observe
- ▶ complement program transformation tools
  - ▶ **temporal properties** (Aorai)
  - ▶ **information flow properties** (SecureFlow)

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

## Combinations of Analyses

- Detecting runtime errors by static analysis and testing (SANTE)

- Deductive verification assisted by testing (STADY)

- Optimizing testing by value analysis and weakest precondition (LTest)

## Conclusion

# Goal

In this part, we

- ▶ describe some combinations of static and dynamic analyses,
- ▶ illustrate their implementation as plugins of Frama-C.

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

## Combinations of Analyses

Detecting runtime errors by static analysis and testing (SANTE)

Deductive verification assisted by testing (STADY)

Optimizing testing by value analysis and weakest precondition (LTest)

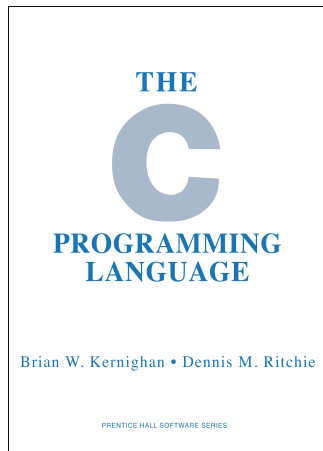
Conclusion

# 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

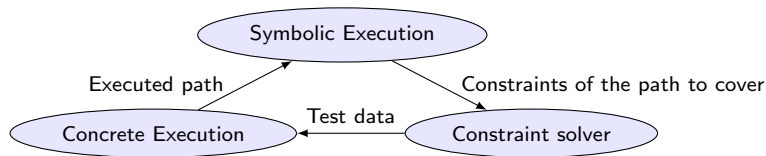


Testing

**Issue:** cannot detect all errors if  
test coverage is partial

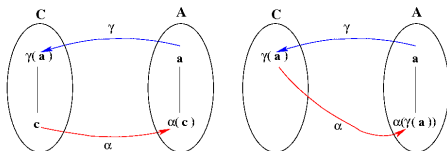
**Goal:** Combine both techniques to detect runtime errors more efficiently

## 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](http://pathcrawler-online.com)

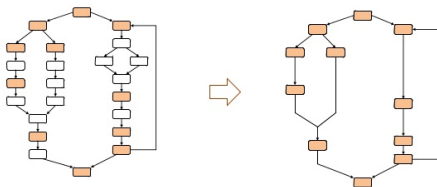
## Plugin “VALUE” for value analysis



- ▶ 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

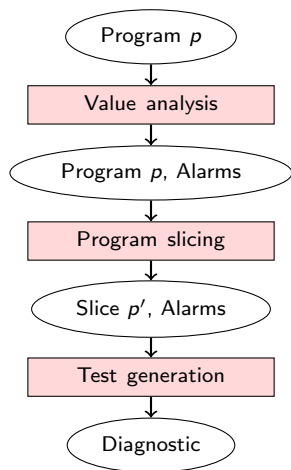


# Plugin Slicing



- ▶ **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:

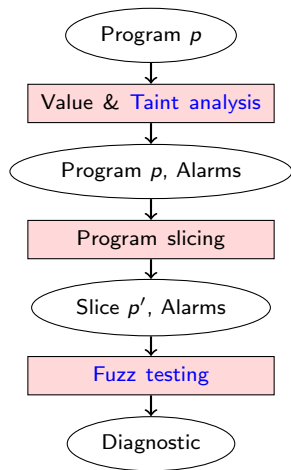
- ▶ 32% in average, up to 89% for some examples
- ▶ program paths in counter-examples are in average 19% shorter

### 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., HVC 2015]

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

## Combinations of Analyses

Detecting runtime errors by static analysis and testing (SANTE)

**Deductive verification assisted by testing (STADY)**

Optimizing testing by value analysis and weakest precondition (LTest)

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, TAP 2016]

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

## Combinations of Analyses

Detecting runtime errors by static analysis and testing (SANTE)

Deductive verification assisted by testing (STADY)

Optimizing testing by value analysis and weakest precondition (LTest)

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

## 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
}
```

## 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

- ▶ 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]

# Outline

Frama-C Overview

Formal Specification and Deductive Verification with WP

Value Analysis

Structural Unit Testing with PathCrawler

Runtime Verification with E-ACSL

Combinations of Analyses

Conclusion

## Conclusion

We have presented how to:

- ▶ formally specify C code with **ACSL**
- ▶ prove programs with **WP**
- ▶ verify the absence of runtime errors with **Value**
- ▶ generate test cases with **PathCrawler**
- ▶ verify annotations at runtime with **E-ACSL**
- ▶ combine analyses in different ways

All of these and much more inside **Frama-C**

May be used for:

- ▶ **teaching**
- ▶ **academic** prototyping
- ▶ **industrial** applications

<http://frama-c.com>



## Some Industrial Applications

- ▶ **Airbus & Atos**: WP and home-made plug-ins for avionic applications
- ▶ **EDF & Areva**: Value for nuclear applications
- ▶ **IRSN**: WP for nuclear applications
- ▶ **Bureau Veritas**: normative activities and Frama-Clang
- ▶ **TrustInSoft** and their customers: Value and Frama-Clang for security applications
- ▶ **Dassault Aviation**: home-made plug-ins + Value + Slicing + E-ACSL for security counter-measures
- ▶ **Mitsubishi Electric**: experimenting PathCrawler
- ▶ ...

