Using Frama-C

Frama-C 29.0
Coq 8.15.2
Alt-ergo 2.4.2

Ming-Hsien Tsai, 2024/11/13

Frama-C

- A suite of tools for the analysis of source code written in C
  - A modified version of CIL (C Intermediate Language) as the kernel
  - Static and dynamic analysis techniques
  - Extensible architecture
  - Collaborations across analyzers
  - Bug free versus bug finding

A Simple Program

```c
int abs(int x) {
  if (x < 0) return -x;
  else return x;
}
```

abs.c

Is this program correct?

A Simple Program

```c
int abs(int x) {
  if (x < 0) return -x;
  else return x;
}
```

abs.c

Is this program correct? No

Range of 32-bit int: $[-2^{31}, 2^{31} - 1]$

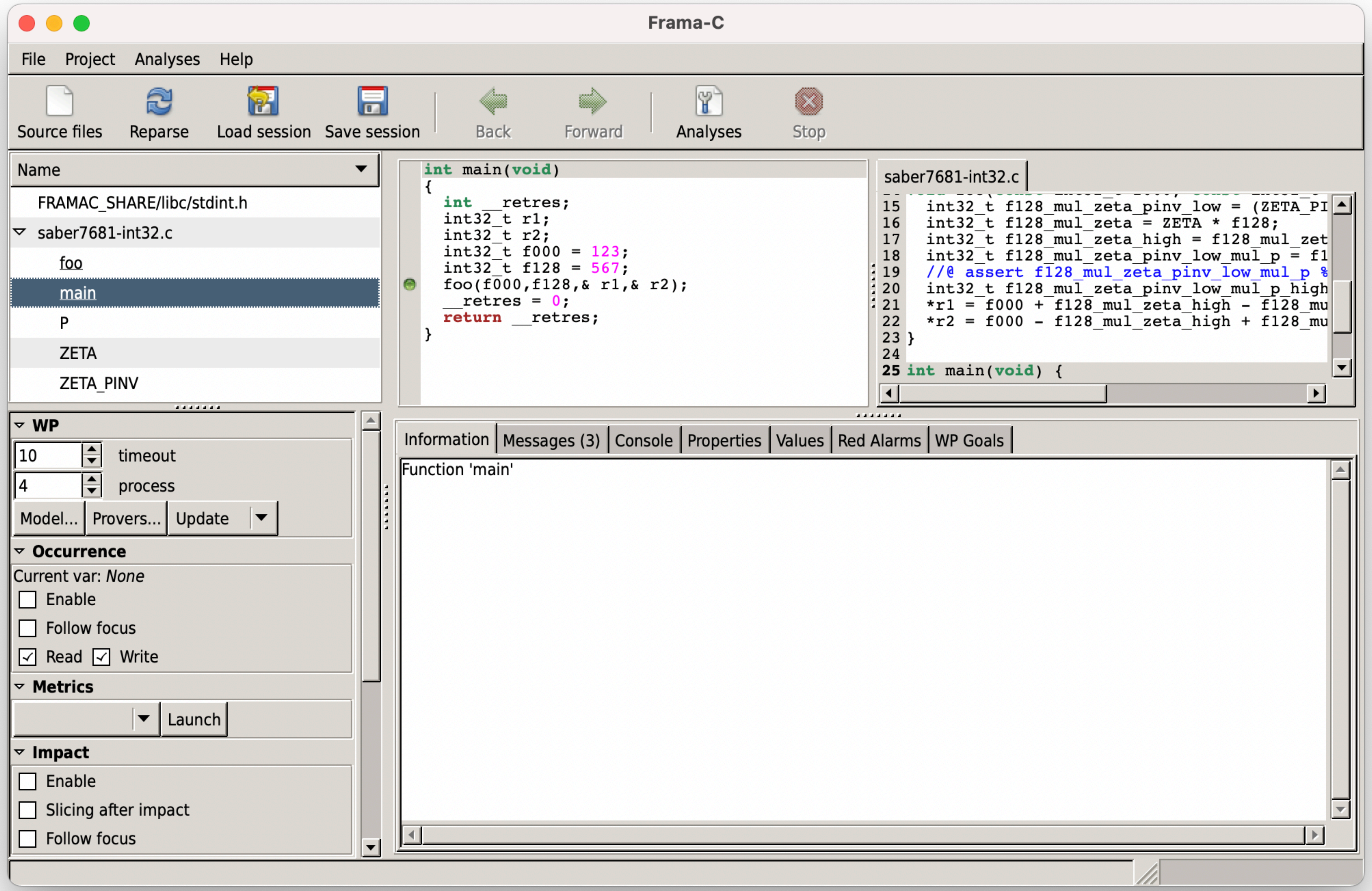
Installation

- Installation instructions: https://frama-c.com/html/get-frama-c.html
- It is recommended to install Frama-C via opam (https://opam.ocaml.org)
  - frama-c
  - why3
  - why3-coq
  - coq
  - coqide
  - alt-ergo

Basic Usage

$ frama-c -PLUGIN -OPTION$_1$ -OPTION$_2$ … file.c -OPTION$_i$ …

$ frama-c-gui -PLUGIN -OPTION$_1$ -OPTION$_2$ … file.c -OPTION$_i$ …



Action Order

- Actions are applied in order according to -then.
  - $ frama-c ARGS-ACT-1 -then ARGS-ACT-2 -then ARGS-ACT-3 ...
- The action after -then-on PROJECT is applied after PROJECT
- The action specified after -then-last is applied on the last project created by a program transformer

Value Analysis via EVA

- Based on abstract interpretation
- Compute variation domains for variables
- Can detect runtime errors such as overflow problems
- Byproducts:
  - Possible ranges of values and addresses for variables
  - Code reachability
  - Call graphs
- Recursive calls are not supported

EVA Example 1

```c
int dbl(int n) {
  return n * 2;
}

int main(void) {
  int n, m = 0;
  printf("Enter an integer: ");
  scanf("%d", &n);
  if (0 <= n && n <= 3)
    m = dbl(n);
  return 0;
}
```

dbl-1.c

...
[eva:final-states] Values at end of function main:
  n ∈ [--..--]
  m ∈ {0; 2; 4; 6}
  __retres ∈ {0}
  S___fc_stdin[0..1] ∈ [--..--]
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
...

[--..--]: the set of all integers that fit within the type of the variable or expression

$ frama-c -eva dbl-1.c

EVA Example 2
-Wider Range-

```c
int dbl(int n) {
  return n * 2;
}

int main(void) {
  int n, m = 0;
  printf("Enter an integer: ");
  scanf("%d", &n);
  if (0 <= n && n <= 9)
    m = dbl(n);
  return 0;
}
```

dbl-2.c

```
[eva:final-states] Values at end of function dbl:
  __retres ∈ [0..18],0%2
[eva:final-states] Values at end of function main:
  n ∈ [--..--]
  m ∈ [0..18],0%2
  __retres ∈ {0}
...
```

[L..H]: { n | L ≤ n ≤ H }

[L..H],r%m: { n | L ≤ n ≤ H, and n % m = r }

-eva-ilevel <n>: controls the maximal number of integers that should be precisely represented as a set

EVA Example 3

-Loops-

```c
int main(void) {
  int x = 0, y = 1;
  for (int i = 0; i < 10; i++) {
    int tmp = x;
    x = y;
    y = tmp + 2 * y;
  }
  int a = x;
  int b = y;
  return 0;
}
```

dbl-3.c

[eva:final-states] Values at end of function main:

$x \in$ [0..2147483647]
$y \in$ [1..2147483647]
$a \in$ [0..2147483647]
$b \in$ [1..2147483647]
__retres $\in$ {0}
...

EVA Example 4
-Precision Improvement-

```c
int main(void) {
  int x = 0, y = 1;
  //@ loop unroll 10;
  for (int i = 0; i < 10; i++) {
    int tmp = x;
    x = y;
    y = tmp + 2 * y;
  }
  int a = x;
  int b = y;
  return 0;
}
```

dbl-4.c

```
[eva:final-states] Values at end of function main:
  x ∈ {2378}
  y ∈ {5741}
  a ∈ {2378}
  b ∈ {5741}
  __retres ∈ {0}
...
```

-eva-auto-loop-unroll <n>: loops with less than <n> iterations will be completely unrolled
-eva-min-loop-unroll <n>: specify the number of iterations to unroll in each loop

Catch Overflow Bugs

abs.c

```
int abs(int x) {
  if (x < 0) return -x;
  else return x;
}
```

Range of 32-bit int: $[-2^{31}, 2^{31} - 1]$

$2^{31} = 2147483648$

```
$ frama-c -eva -main abs abs.c
[kernel] Parsing abs.c (with preprocessing)
[eva] Analyzing a complete application starting at abs
[eva:initial-state] Values of globals at initialization

[eva:alarm] abs.c:5: Warning: signed overflow. assert -x ≤ 2147483647;
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function abs:
  __retres ∈ [0..2147483647]
[eva:summary] ====== ANALYSIS SUMMARY ======
...
```

False Alarms

```c
#define MAX 100

int compute_next(int x) {
  return (x % 2 == 0) ?
         x / 2 : 3*x + 1;
}

void collatz(int n) {
  int t[MAX];
  t[0] = n;
  for (int i = 0; i < MAX; i++) {
    t[i + 1] = compute_next(t[i]);
  }
}
```

```
$ frama-c -eva -main collatz collatz.c
...
[eva:summary] ====== ANALYSIS SUMMARY ======
  ----------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 12 statements reached (out of 14): 85% coverage.
  ----------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ----------------------------------------------------------------------------
  4 alarms generated by the analysis:
     2 integer overflows
     1 access out of bounds index
     1 access to uninitialized left-values
  ----------------------------------------------------------------------------
  No logical properties have been reached by the analysis.
  ----------------------------------------------------------------------------
```

Example from Guide to Software Verification with Frama-C

False Alarms (cont'd)

```c
3 #define MAX 100

5 int compute_next(int x) {
6   return (x % 2 == 0) ?
          x / 2 : 3*x + 1;
7 }

9 void collatz(int n) {
10  int t[MAX];
11  t[0] = n;
12  for (int i = 0; i < MAX; i++) {
13    t[i + 1] = compute_next(t[i]);
14  }
15}
```

```
$ frama-c -eva -main collatz collatz.c
...
[eva:alarm] collatz.c:6: Warning: signed overflow.
assert -2147483648 ≤ 3 * x;
[eva:alarm] collatz.c:6: Warning: signed overflow.
assert 3 * x ≤ 2147483647;
[eva] collatz.c:12: starting to merge loop iterations
[eva:alarm] collatz.c:13: Warning:
  accessing uninitialized left-value. assert
\initialized(&t[i]);
[eva:alarm] collatz.c:13: Warning:
  accessing out of bounds index. assert (int)(i + 1) <
100;
...
[eva:summary] ====== ANALYSIS SUMMARY ======
...
```

Example from Guide to Software Verification with Frama-C

A Corrected Version

```c
#include <limits.h>
#define MAX 100

int compute_next(int x) {
  return (x % 2 == 0) ? x / 2 : 3*x + 1;
}

int collatz(int n) {
  int i, t[MAX];
  t[0] = n;
  if (n < 0) return -1;
  for (i = 0; i < MAX - 1; i++) {
    int v = t[i];
    if (v > INT_MAX / 3) return -1;
    t[i + 1] = compute_next(v);
  }
  return i;
}
```

```
$ frama-c -eva -main collatz collatz-corrected.c
...
----------------------------------------------------------------------------
2 alarms generated by the analysis:
    1 integer overflow
    1 access to uninitialized left-values
----------------------------------------------------------------------------
No logical properties have been reached by the analysis.
----------------------------------------------------------------------------
```

There are still false alarms

⇒ We need to increase the precision of EVA

Example from Guide to Software Verification with Frama-C with one modification

Increase EVA Prevision

```c
#include <limits.h>
#define MAX 100

int compute_next(int x) {
  return (x % 2 == 0) ? x / 2 : 3*x + 1;
}

int collatz(int n) {
  int i, t[MAX];
  t[0] = n;
  if (n < 0) return -1;
  for (i = 0; i < MAX - 1; i++) {
    int v = t[i];
    if (v > INT_MAX / 3) return -1;
    t[i + 1] = compute_next(v);
  }
  return i;
}
```

```
$ frama-c -eva -main collatz -eva-precision 5 collatz-corrected.c
...
----------------------------------------------------------------------------
No errors or warnings raised during the analysis.
----------------------------------------------------------------------------
0 alarms generated by the analysis.
----------------------------------------------------------------------------
No logical properties have been reached by the analysis.
----------------------------------------------------------------------------
```

-eva-precision <N>: <N> from 0 to 11

Example from Guide to Software Verification with Frama-C with one modification

Limitations: Recursive Functions

```c
int loop(int x) {
  if (x < 0)
    return loop(x + 1);
  else if (x > 0)
    return loop(x - 1);
  else
    return 0;
}
```

```
$ frama-c -eva -eva-precision 11 -main loop eva-limitations-1.c
...
[eva:final-states] Values at end of function loop:
  __retres ∈ [--..--]
...
```

The function loop always returns 0

The result of EVA can be further improved with proper ACSL specifications

Limitations: Value Domains

```
int divi(int x, int y) {
  if (y > 10000) return 0;
  if (y < -10000) return 0;
  if (x > 10000) return 0;
  if (x < -10000) return 0;
  if (y != 0) return x / y;
  return 0;
}
```

The abstract domain [-10000, 10000] of y after y != 0 is still [-10000, 10000]

This can be improved by case splitting

```
$ frama-c -eva -eva-precision 11 -main divi eva-limitations-2.c
...
[eva:final-states] Values at end of function divi:
  __retres ∈ [-10000..10000]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ----------------------------------------------------------------------------
  1 function analyzed (out of 1): 100% coverage.
  In this function, 22 statements reached (out of 22): 100% coverage.
  ----------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ----------------------------------------------------------------------------
  1 alarm generated by the analysis:
      1 division by zero
  ----------------------------------------------------------------------------
  No logical properties have been reached by the analysis.
  ----------------------------------------------------------------------------
```

Limitations: Value Domains (cont'd)

```c
int divi(int x, int y) {
  if (y > 10000) return 0;
  if (y < -10000) return 0;
  if (x > 10000) return 0;
  if (x < -10000) return 0;
  //@ split y >= 0;
  if (y != 0) return x / y;
  return 0;
}
```

Split into two cases by //@ split e;

- e is satisfied ⇒ in this case, [0, 10000]
- e is violated ⇒ in this case, [-10000, -1]

```
$ frama-c -eva -eva-precision 11 -main divi eva-limitations-3.c
...
[eva:summary] ====== ANALYSIS SUMMARY ======
  ----------------------------------------------------------------------
  1 function analyzed (out of 1): 100% coverage.
  In this function, 23 statements reached (out of 23): 100% coverage.
  ----------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ----------------------------------------------------------------------
  0 alarms generated by the analysis.
  ----------------------------------------------------------------------
  No logical properties have been reached by the analysis.
  ----------------------------------------------------------------------
```

Runtime Assertions via E-ACSL

- E-ACSL is used for runtime annotation checking (RAC)
- Translate an annotated C program into another program with runtime assertions
  - Both programs have the same behavior if no annotation is violated
- Possible usage:
  - Detect undefined behaviors (+RTE)
  - Verification of linear temporal properties (+Aoraï)
  - Verification of security properties (+SecureFlow)

E-ACSL Example 1

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

int main(void) {
  int x, y, z;
  z = max(x, y);
  return 0;
}
```

max.c

$ frama-c -e-acsl max.c -then-last -print

E-ACSL Example 1

```
/*@
      esult;
      == y;
```

```
int __gen_e_acsl_max(int x, int y)
{
  int __gen_e_acsl_at_2;
  int __gen_e_acsl_at;
  int __retres;
  __gen_e_acsl_at = x;
  __gen_e_acsl_at_2 = y;
  __retres = max(x,y);
  {
   …
  }
}
```

```
    return 0;
  }
```

max.c

$ frama-c -e-acsl max.c -then-last -print

Every call to max is replaced by a call to __gen_e_acsl_max.

E-ACSL Example 1

```
/*@
```

```
int __gen_e_acsl_max(int
{
  int __gen_e_acsl_at_2;
  int __gen_e_acsl_at;
  int __retres;
  __gen_e_acsl_at = x;
  __gen_e_acsl_at_2 = y;
  __retres = max(x,y);
  {
   …
  }
}
```

```
    return 0;
  }
```

```
int __gen_e_acsl_and;
int __gen_e_acsl_or;
__e_acsl_assert_data_t __gen_e_acsl_assert_data = {.values = (void *)0};
__e_acsl_assert_register_int(& __gen_e_acsl_assert_data,"\\old(x)",0,
                             __gen_e_acsl_at);
__e_acsl_assert_register_int(& __gen_e_acsl_assert_data,"\\result",0,
                             __retres);
if (__gen_e_acsl_at <= __retres) {
  __e_acsl_assert_register_int(& __gen_e_acsl_assert_data,"\\old(y)",0,
                               __gen_e_acsl_at_2);
  __e_acsl_assert_register_int(& __gen_e_acsl_assert_data,"\\result",0,
                               __retres);
  __gen_e_acsl_and = __gen_e_acsl_at_2 <= __retres;
}
else __gen_e_acsl_and = 0;
…
```

__gen_e_acsl_and == 1 iff x <= \result && y <= \result

max.c

$ frama-c -e-acsl max.c -then-last -print

Every call to max is replaced by a call to __gen_e_acsl_max.

E-ACSL Example 2
-With RTE-

```c
int main(void) {
  int x = 0xffff;
  int y = 0xfff;
  int z = x + y;
  return 0;
}
```

eacsl.c

$ frama-c -rte eacsl.c -then -print

```c
int main(void)
{
  int __retres;
  int x = 0xffff;
  int y = 0xfff;
  /*@ assert rte: signed_overflow: -2147483648 ≤ x + y; */
  /*@ assert rte: signed_overflow: x + y ≤ 2147483647; */
  int z = x + y;
  __retres = 0;
  return __retres;
}
```

E-ACSL Example 2

-With RTE+E-ACSL-

```
int main(void) {
  int x = 0xffff;
  int y = 0xfff;
  int z = x + y;
  return 0;
}
```

eacsl.c

$ frama-c -rte eacsl.c -then -e-acsl -then-last -print

```
int main(void)
{
  int __retres;
  __e_acsl_memory_init((int *)0,(char ***)0,8UL);
  int x = 0xffff;
  int y = 0xfff;
  {
    …
    /*@ assert rte: signed_overflow: -9223372036854775808 ≤ x + (long)y; */
    /*@ assert rte: signed_overflow: x + (long)y ≤ 9223372036854775807; */
    __e_acsl_assert(x + (long)y <= 2147483647L,& __gen_e_acsl_assert_data);
    …
    /*@ assert rte: signed_overflow: -9223372036854775808 ≤ x + (long)y; */
    /*@ assert rte: signed_overflow: x + (long)y ≤ 9223372036854775807; */
    __e_acsl_assert(-2147483648L <= x + (long)y,& __gen_e_acsl_assert_data_2);
    __e_acsl_assert_clean(& __gen_e_acsl_assert_data_2);
  }
  /*@ assert rte: signed_overflow: -2147483648 ≤ x + y; */
  /*@ assert rte: signed_overflow: x + y ≤ 2147483647; */
  int z = x + y;
  __retres = 0;
  __e_acsl_memory_clean();
  return __retres;
}
```

Limitations of E-ACSL

- Uninitialized values
  - Runtime error may not occur depending on the compiler
- Incomplete programs
- Recursive functions
- Variadic functions
- Function pointers

```
int main(void) {
  int x;
  /*@ assert x == 0; */
  return 0;
}
```

Test Cases Generation via PathCrawler

- Generate test inputs
- Cover all feasible execution paths
- Based on constraint resolution
- Try it online at http://pathcrawler-online.com:8080/

A Simple Example for PathCrawler

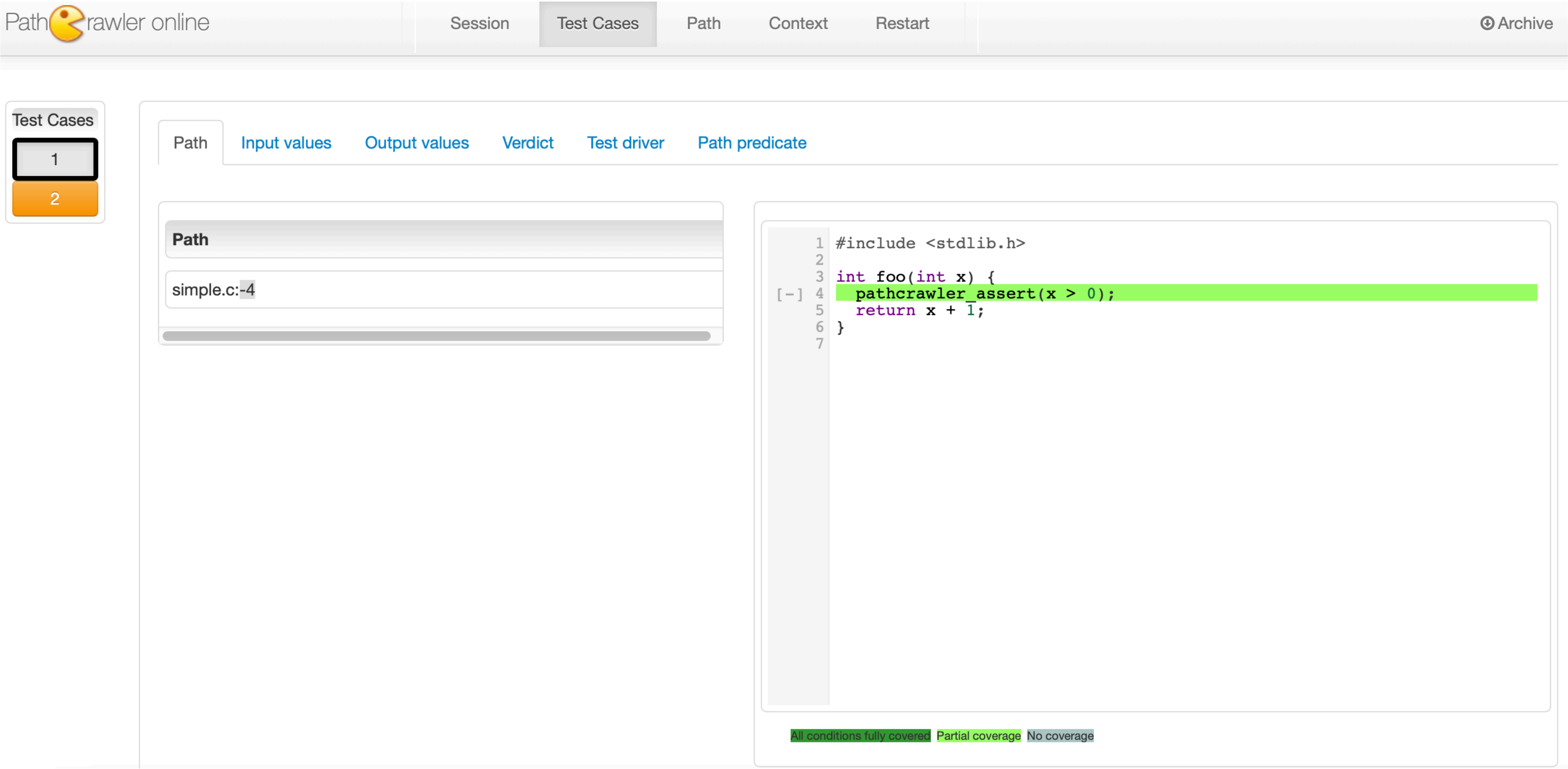
```c
int foo(int x) {
  pathcrawler_assert(x > 0);
  return x + 1;
}
```

simple.c

pathcrawler_assert(condition)

- May be used at any location in the program under test
- Will force the tool to generate test cases to cover both
  - the case condition is true, and
  - condition is false

A Simple Example for PathCrawler (cont'd)



Program Slicing

- Program slicing computes a subset of program statements that may affect a given set of values called slicing criterion
    - control dependency
    - data dependency

```
…
if (i <= j)
  x = y * 2;
else
  y = y + 3;
return x;
```

→

```
…
if (i <= j)
  x = y * 2;


return x;
```

slicing criterion: x at the end of the program

Program Slicing

-Example 1-

```c
int max(int m, int n) {
  if (m <= n) return n;
  else return m;
}

int dbl(int m) {
  return m * 2;
}

int f(void) {
  int m = 3, n = 5;
  int a = max(m, n);
  int b = dbl(m);
  /*@ assert b <= 10; */
  int c = b + 100;
  return c;
}

void main(void) { f(); … }
```

slicing-1.c

```c
/* Generated by Frama-C */
int dbl_slice_1(int m)
{
  int __retres;
  __retres = m * 2;
  return __retres;
}

void f_slice_1(void)
{
  int m = 3;
  int b = dbl_slice_1(m);
  /*@ assert b ≤ 10; */ ;
  return;
}

void main(void) { f_slice_1(); … }
```

$ frama-c slicing-1.c -slice-assert f -then-last -print

Slicing Criteria
-Code Observation-

- -slice-calls f1,...,fn: calls to functions f1,...,fn
- -slice-return f1,...,fn: returned values of functions f1,...,fn
- -slice-value v1,...,vn: left-values at the end of the entry function (specified by -main)
- -slice-wr v1,...,vn: write accesses to left-values
- -slice-rd v1,...,vn: read accesses to left-values
- -slice-pragma f1,...,fn: slicing pragmas in the code of functions f1,...,fn

Ref: https://frama-c.com/fc-plugins/slicing.html

Slicing Criteria

-Proving Properties-

- -slice-assert f1,…,fn: assertions of functions f1,…,fn
- -slice-loop-inv f1,…,fn: loop invariants of functions f1,…,fn
- -slice-loop-var f1,…,fn: loop variants of functions f1,…,fn
- -slice-threat f1,…,fn: threats (emitted by Eva) of functions f1,…,fn

Program Slicing

-Example 2: Case 1 of -slice-rd-

```c
int max(int m, int n) { … }

int dbl(int m) { … }

int f(void) {
  int m = 3, n = 5;
  int a = max(m, n);
  int b = dbl(m);
  /*@ assert b <= 10; */
  int c = b + 100;
  return c;
}

void main(void) { f(); … }
```

slicing-1.c

```c
/* Generated by Frama-C */
void f(void)
{
  return;
}
```

$ frama-c slicing-1.c -slice-rd c -main f -then-last -print

Program Slicing

-Example 3: Case 2 of -slice-rd-

```
int max(int m, int n) { … }

int dbl(int m) { … }

int f(void) {
  int m = 3, n = 5;
  int a = max(m, n);
  int b = dbl(m);
  /*@ assert b <= 10; */
  int c = b + 100;
  return c + a;
}

void main(void) { f(); … }
```

slicing-2.c

$ frama-c slicing-2.c -slice-rd c -main f -then-last -print

Program Slicing
-Example 3: Case 2 of -slice-rd-

```c
int max(int m, int n) { … }

int dbl(int m) { … }

int f(void) {
  int m = 3, n = 5;
  int a = max(m, n);
  int b = dbl(m);
  /*@ assert b <= 10; */
  int c = b + 100;
  return c + a;
}

void main(void) { f(); … }
```

slicing-2.c

```c
/* Generated by Frama-C */
int max_slice_1(int n)
{ … }

int dbl_slice_1(int m)
{ … }

void f(void)
{
  int __retres;
  int m = 3;
  int n = 5;
  int a = max_slice_1(n);
  int b = dbl_slice_1(m);
  /*@ assert b ≤ 10; */ ;
  int c = b + 100;
  __retres = c + a;
  return;
}
```

$ frama-c slicing-2.c -slice-rd c -main f -then-last -print

Program Slicing

-Example 4: -slice-pragma-

```c
int max(int m, int n) { … }

int dbl(int m) { … }

int f(void) {
  int m = 3, n = 5;
  int a = max(m, n);
  /*@ slice pragma expr a; */
  /*@ slice pragma stmt; */
  int b = dbl(m);
  /*@ assert b <= 10; */
  int c = b + 100;
  return c + a;
}

void main(void) { f(); … }
```

slicing-3.c

$ frama-c slicing-3.c -slice-pragma f -main f -then-last -print

Program Slicing

-Example 4: -slice-pragma-

```
int max(int m, int n) { … }

int dbl(int m) { … }

int f(void) {
  int m = 3, n = 5;
  int a = max(m, n);
  /*@ slice pragma expr a; */
  /*@ slice pragma stmt; */
  int b = dbl(m);
  /*@ assert b <= 10; */
  int c = b + 100;
  return c + a;
}

void main(void) { f(); … }
```

slicing-3.c

```
/* Generated by Frama-C */
int max_slice_1(int n)
{ … }

int dbl_slice_1(int m)
{ … }

void f(void)
{
  int __retres;
  int m = 3;
  int n = 5;
  int a = max_slice_1(n);
  /*@ slice pragma expr a; */ ;
  /*@ slice pragma stmt; */
  int b = dbl_slice_1(m);
  /*@ assert b ≤ 10; */ ;
  return;
}
```

$ frama-c slicing-3.c -slice-pragma f -main f -then-last -print

Deductive Verification via WP

- Based on weakest-precondition calculus
- Relies on external automated provers and proof assistants
- Provers are invoked via Why3 (http://why3.lri.fr)
  - Alt-Ergo
  - CVC4
  - Gappa
  - Princess
  - Vampire
  - Z3
  - Coq
  - PVS
  - Isabelle/HOL

After installation of why3 and external provers, run command `why3 config detect` to detect available provers.

WP Example 1

```
/*@
  @ ensures \result == x + y;
  @ assigns \nothing;
  */
int add(int x, int y) {
  return x + y;
}
```

add.c

$ frama-c -wp add.c -then -report

```
[kernel] Parsing add.c (with preprocessing)
[wp] Warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] [Cache] not used
[wp] Proved goals:    2 / 2
  Qed:             2
[report] Computing properties status...

--------------------------------------------------------------------------------
--- Properties of Function 'add'
--------------------------------------------------------------------------------

[  Valid  ] Post-condition (file add.c, line 2)
            by Wp.typed.
[  Valid  ] Assigns nothing
            by Wp.typed.
[  Valid  ] Default behavior
            by Frama-C kernel.
...
```

WP Example 2

-With RTE-

```
/*@
  @ ensures \result == x + y;
  @ assigns \nothing;
  */
int add(int x, int y) {
  return x + y;
}
```

add.c

$ frama-c -wp -wp-rte add.c -then -report

Refine the specification such that the absence of runtime errors can be proven

```
[kernel] Parsing add.c (with preprocessing)
[rte:annot] annotating function add
[wp] 4 goals scheduled
[wp] [Timeout] typed_add_assert_rte_signed_overflow_2 (Alt-Ergo) (Cached)
[wp] [Timeout] typed_add_assert_rte_signed_overflow (Alt-Ergo) (Cached)
[wp] [Cache] updated:2
[wp] Proved goals:    2 / 4
  Qed:           2
  Timeout:       2
[report] Computing properties status...
...
[ Partial ] Post-condition (file add.c, line 2)
            By Wp.typed, with pending:
             - Assertion 'rte,signed_overflow' (file add.c, line 6)
             - Assertion 'rte,signed_overflow' (file add.c, line 6)
...
[    -    ] Assertion 'rte,signed_overflow' (file add.c, line 6)
            tried with Wp.typed.
[    -    ] Assertion 'rte,signed_overflow' (file add.c, line 6)
            tried with Wp.typed.
...
```

WP Example 3

```
/*@ requires \valid(a) && \valid(b);
  @ ensures *a == \old(*b) && *b == \old(*a);
  @ assigns *a, *b;
  @*/
void swap(int *a, int *b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

void order3(int *a, int *b, int *c) {
  if (*a > *b) swap(a, b);
  if (*a > *c) swap(a, c);
  if (*b > *c) swap(b, c);
}
```

order3.c

Write a specification for order3

Source: A. Blanchard. Introduction to C program proof with Frama-C and its WP plugin, Creative Commons, 2020.

WP Example 3

```c
/*@ requires \valid(a) && \valid(b);
  @ ensures *a == \old(*b) && *b == \old(*a);
  @ assigns *a,
  @*/
void swap(int *
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

void order3(int
  if (*a > *b)
  if (*a > *c)
  if (*b > *c) swap(b, c);
}
```

order3.c

```c
/*@
  @ requires \valid(a) && \valid(b) && \valid(c) && \separated(a, b, c);
  @ ensures *a <= *b <= *c;
  @ ensures { *a, *b, *c } == { \old(*a), \old(*b), \old(*c) };
  @ assigns *a, *b, *c;
  @*/
void order3(int *a, int *b, int *c) {
  if (*a > *b) swap(a, b);
  if (*a > *c) swap(a, c);
  if (*b > *c) swap(b, c);
}
```

sol/order3-annotated-1.c

Source: A. Blanchard. Introduction to C program proof with Frama-C and its WP plugin, Creative Commons, 2020.

WP Example 3
-Additional Assertions-

```c
/*@ requires \valid(a) && \valid(b);
  @ ensures *a == \old(*b) && *b == \old(*a);
  @ assigns *a, *b;
  @*/
void swap(int *a, int *b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

void order3(int *a, int *b, int *c) {
  if (*a > *b) swap(a, b);
  if (*a > *c) swap(a, c);
  if (*b > *c) swap(b, c);
}
```

sol/order3-annotated-2.c

```c
void test() {
  int a1 = 5, b1 = 3, c1 = 4;

  order3(&a1, &b1, &c1);
  //@ assert a1 == 3 && b1 == 4 && c1 == 5;

  int a2 = 2, b2 = 2, c2 = 2;
  order3(&a2, &b2, &c2);
  //@ assert a2 == 2 && b2 == 2 && c2 == 2;

  int a3 = 4, b3 = 3, c3 = 4;
  order3(&a3, &b3, &c3);
  //@ assert a3 == 3 && b3 == 4 && c3 == 4;

  int a4 = 4, b4 = 5, c4 = 4;
  order3(&a4, &b4, &c4);
  //@ assert a4 == 4 && b4 == 4 && c4 == 5;
}
```

WP Example 3

-Additional Assertions-

```c
/*@ requires \valid(a) && \valid(b);
  @ ensures *a == \old(*b) && *b == \old(*a);
  @ assigns *a, *b;
  @*/
void swap(int *a, int *b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

void order3(int *a, int *b, int *c) {
  if (*a > *b) swap(a, b);
  if (*a > *c) swap(a, c);
  if (*b > *c) swap(b, c);
}
```

With the previous annotation

sol/order3-annotated-2.c

```c
void test() {
  int a1 = 5, b1 = 3, c1 = 4;

  order3(&a1, &b1, &c1);
○ //@ assert a1 == 3 && b1 == 4 && c1 == 5;

  int a2 = 2, b2 = 2, c2 = 2;
  order3(&a2, &b2, &c2);
○ //@ assert a2 == 2 && b2 == 2 && c2 == 2;

  int a3 = 4, b3 = 3, c3 = 4;
  order3(&a3, &b3, &c3);
✗ //@ assert a3 == 3 && b3 == 4 && c3 == 4;

  int a4 = 4, b4 = 5, c4 = 4;
  order3(&a4, &b4, &c4);
✗ //@ assert a4 == 4 && b4 == 4 && c4 == 5;
}
```

Write a specification for order3 such that all assertions are verified

WP Example 3

-Refined Annotations I-

```
/*@
  @ requires \valid(a) && \valid(b) && \valid(c) && \separated(a, b, c);
  @ ensures *a <= *b <= *c;
  @ ensures { *a, *b, *c } == { \old(*a), \old(*b), \old(*c) };
  @ ensures (\old(*a) == \old(*b)) ==>
            (*a == *b == \old(*a) || *b == *c == \old(*a) || *c == *a == \old(*a));
  @ ensures (\old(*b) == \old(*c)) ==>
            (*a == *b == \old(*b) || *b == *c == \old(*b) || *c == *a == \old(*b));
  @ ensures (\old(*c) == \old(*a)) ==>
            (*a == *b == \old(*c) || *b == *c == \old(*c) || *c == *a == \old(*c));
  @ assigns *a, *b, *c;
  @*/
void order3(int *a, int *b, int *c) {
  if (*a > *b) swap(a, b);
  if (*a > *c) swap(a, c);
  if (*b > *c) swap(b, c);
}
```

sol/order3-annotated-3.c

WP Example 3

-Refined Annotations II-

```
/*@
  @ requires \valid(a) && \valid(b) && \valid(c) && \separated(a, b, c);
  @ ensures *a <= *b <= *c;
  @ ensures { *a, *b, *c } == { \old(*a), \old(*b), \old(*c) };
  @ ensures \forall int* x, int* y;
              \subset({ x, y }, { a, b, c }) && \separated(x, y) && \old(*x) == \old(*y) ==>
                \exists int* u, int* v;
                  \subset({ u, v }, { a, b, c }) && \separated(u, v) && *u == *v == \old(*x);
  @ assigns *a, *b, *c;
  @*/
  void order3(int *a, int *b, int *c) {
  if (*a > *b) swap(a, b);
  if (*a > *c) swap(a, c);
  if (*b > *c) swap(b, c);
}
```

sol/order3-annotated-5.c

WP Exercise

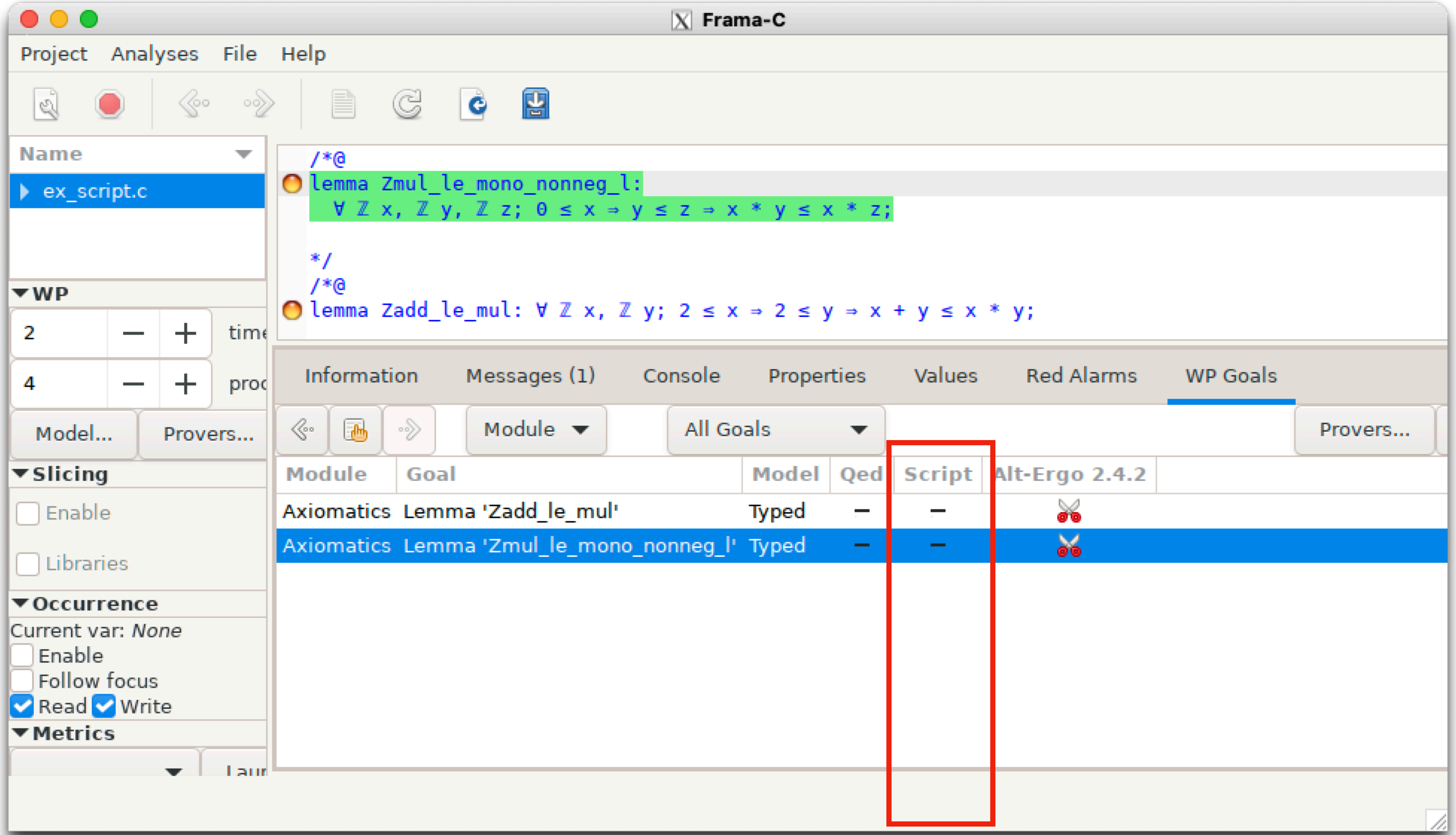
```c
#include <limits.h>

/*@
  @ requires 0 <= x <= INT_MAX / 2;
  @ assigns \nothing;
  @ ensures \result == 2 * x;
  @*/
int times2 (int x) {
  int r = 0 ;
  /*@
    @ loop invariant …;
    @ loop assigns …;
    @ loop variant …;
    @*/
  while (x > 0) {
    r += 2;
    x --;
  }
  return r;
}
```

times2.c

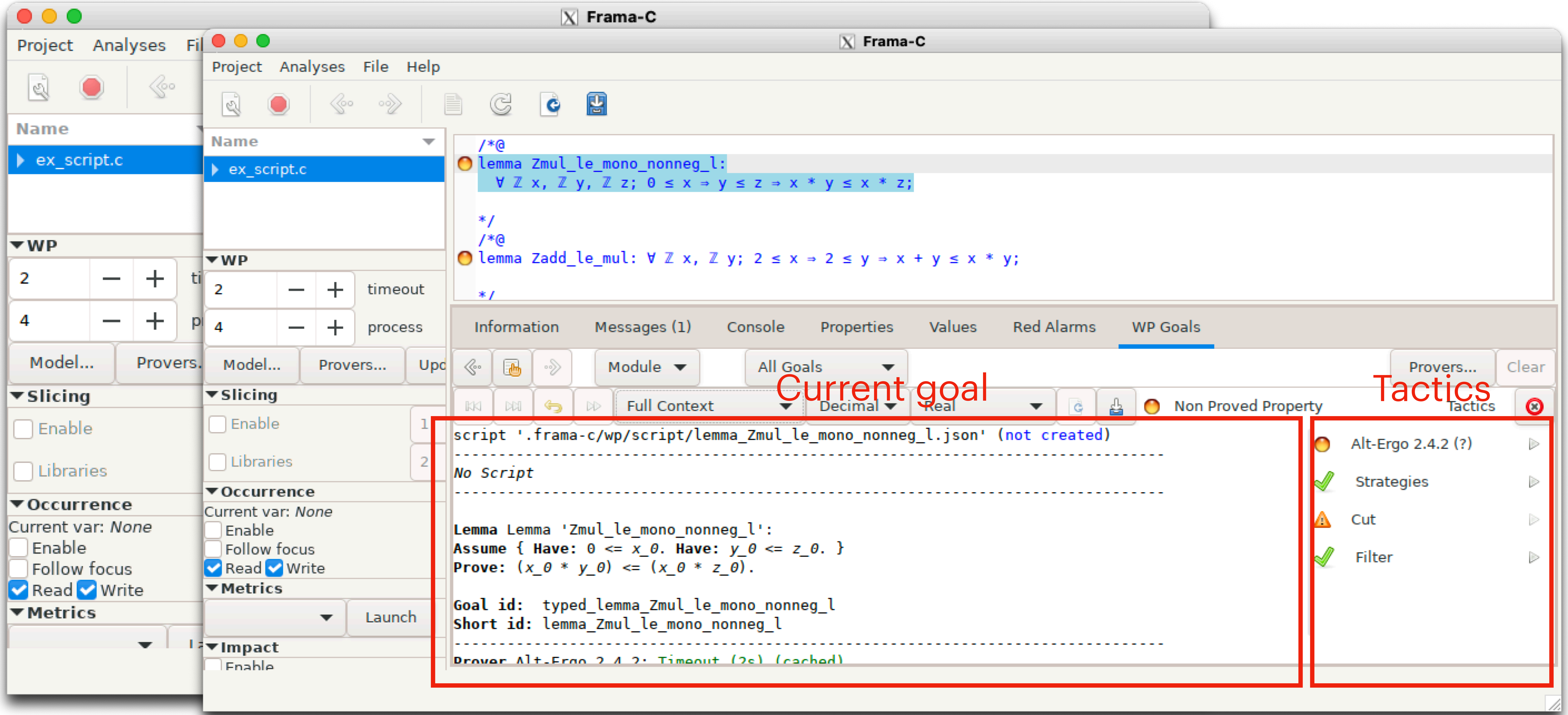
Deductive Verification with Interactive Prover

- For proof obligations that cannot be discharged by automatic provers, an interactive prover such as Coq can be used
- We show how to use interactive provers in the following examples

Prover: Script



Prover: Script



Script Prover
-Example-

```
/*@
  @ lemma Zmul_le_mono_nonneg_l :
  @          \forall integer x, y, z;
  @            0 <= x ==> y <= z ==> x * y <= x * z;
  @
  @ lemma Zadd_le_mul :
  @          \forall integer x, y;
  @            2 <= x ==> 2 <= y ==> x + y <= x * y;
  @*/

/*@
  @ requires 2 <= x;
  @ requires 2 <= y;
  @ requires x * (x * y) <= INT_MAX;
  @ ensures x * (x + y) <= x * (x * y);
  @ assigns \nothing;
  @*/
void foo(int x, int y) {
}
```

ex_script.c

Script Prover
-Applying Tactics-

- Zmul_le_mono_nonneg_l:
  - Apply induction on x
- Zadd_le_mul
  - Apply induction on x
  - Case analysis on <=
- Scripts can be saved and replayed with `tip` added to `-wp-prover`

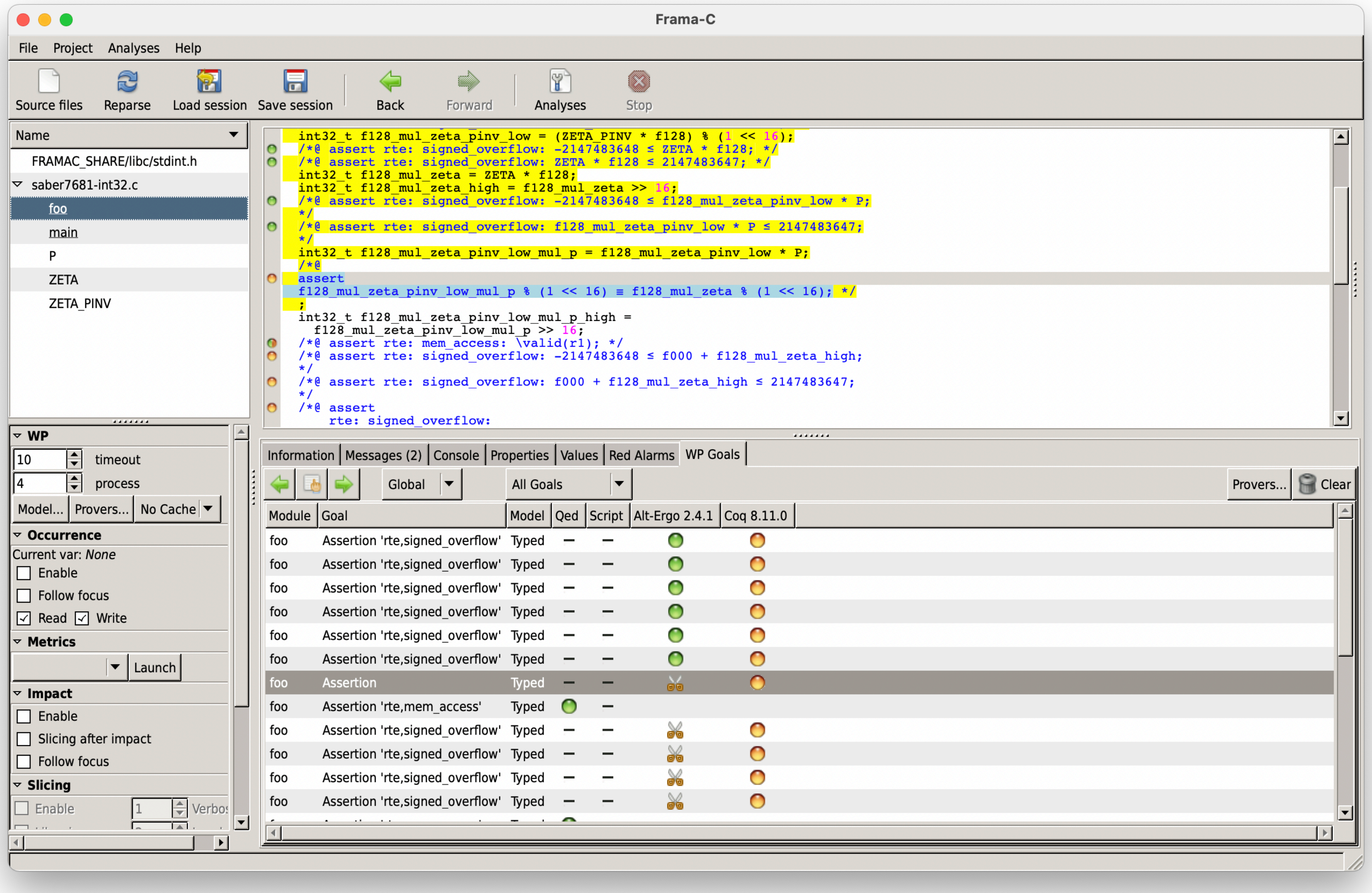
Prover: Coq

- Run the following command to invoke Frama-C

  $ frama-c-gui -wp -wp-rte -wp-prover alt-ergo,coq EXAMPLE.c

- Double click the orange circle of the assertion on the Coq column to edit the Coq proof script

- unknown
- surely valid
- surely invalid
- valid under hypothesis
- invalid under hypothesis



Field Operations

-Annotated C Code-

```
const int32_t P = 7681;
const int32_t ZETA = 3777;
const int32_t ZETA_PINV = 28865;

/*@
  @ requires \valid(r1) && \valid(r2);
  @ requires \separated(r1, r2, &P, &ZETA, &ZETA_PINV);
  @ requires (-4096 < f000 < 4096);
  @ requires (-4096 < f128 < 4096);
  @ assigns *r1, *r2;
  @*/
void foo(const int32_t f000, const int32_t f128, int32_t *r1, int32_t *r2) {
  int32_t f128_mul_zeta_pinv_low = (ZETA_PINV * f128) % (1 << 16);
  int32_t f128_mul_zeta = ZETA * f128;
  int32_t f128_mul_zeta_high = f128_mul_zeta >> 16;
  int32_t f128_mul_zeta_pinv_low_mul_p = f128_mul_zeta_pinv_low * P;
  //@ assert f128_mul_zeta_pinv_low_mul_p % (1 << 16) == f128_mul_zeta % (1 << 16);
  int32_t f128_mul_zeta_pinv_low_mul_p_high = f128_mul_zeta_pinv_low_mul_p >> 16;
  *r1 = f000 + f128_mul_zeta_high - f128_mul_zeta_pinv_low_mul_p_high;
  *r2 = f000 - f128_mul_zeta_high + f128_mul_zeta_pinv_low_mul_p_high;
}
```

saber7681-int32.c

alt-ergo fails to prove the assertion

Field Operations

-Proof Obligations-

```
/*@
  @ requires \valid(r1) && \valid(r2);
  @ requires \separated(r1, r2, &P, &ZETA, &ZETA_PINV);
  @ requires (-4096 < f000 < 4096);
  @ requires (-4096 < f128 < 4096);
  @ assigns *r1, *r2;
  @*/
void foo(const int32_t f000, const int32_t f128, int32_t *r1, int32_t *r2) {
  // (((ZETA_PINV * f128) % (1 << 16)) * P) % (1 << 16) == (ZETA * f128) % (1 << 16)
  int32_t f128_mul_zeta_pinv_low = (ZETA_PINV * f128) % (1 << 16);
  // (f128_mul_zeta_pinv_low * P) % (1 << 16) == (ZETA * f128) % (1 << 16)
  int32_t f128_mul_zeta = ZETA * f128;
  // (f128_mul_zeta_pinv_low * P) % (1 << 16) == f128_mul_zeta % (1 << 16)
  int32_t f128_mul_zeta_high = f128_mul_zeta >> 16;
  // (f128_mul_zeta_pinv_low * P) % (1 << 16) == f128_mul_zeta % (1 << 16)
  int32_t f128_mul_zeta_pinv_low_mul_p = f128_mul_zeta_pinv_low * P;
  //@ assert f128_mul_zeta_pinv_low_mul_p % (1 << 16) == f128_mul_zeta % (1 << 16);
  …
}
```

```
\valid(r1) && \valid(r2) -> \separated(r1, r2, &P, &ZETA, &ZETA_PINV) -> (-4096 < f000 < 4096) -> (-4096 < f128 < 4096) ->
(((ZETA_PINV * f128) % (1 << 16)) * P) % (1 << 16) == (ZETA * f128) % (1 << 16)
```

Field Operations

-Prove by Hand-

```
  (((ZETA_PINV * f128) % (1 << 16)) * P) % (1 << 16)
= (((28865 * f128) % 65536) * 7681) % 65536
= ((28865 * f128) * 7681) % 65536               # ((a%n) * b)%n = (a * b)%n
= (7681 * (28865 * f128)) % 65536               # a * b = b * a
= ((7681 * 28865) * f128) % 65536               # a * (b * c) = (a * b) * c
= ((7681 * 28865) % 65536 * f128) % 65536       # ((a%n) * b)%n = (a * b)%n
= (3777 * f128) % 65536
= (ZETA * f128) % (1 << 16)
```

```
const int32_t P = 7681;
const int32_t ZETA = 3777;
const int32_t ZETA_PINV = 28865;
```

Field Operations

-Prove Goals using Coq-

- Assertion
  - .frama-c/wp/interactive/foo_assert.v
  - Print definitions
  - Search for lemmas
  - Basic tactics

Multiplication by Addition

-Annotated C Code-

```c
/*@
  @ requires INT_MIN <= x * y <= INT_MAX;
  @ ensures \result == x * y;
  @*/
int mul(int x, int y) {
  int r = 0;
  /*@
    @ loop assigns r, y;
    @ loop invariant r + x * y == \at(x, Pre) * \at(y, Pre);
    @ loop variant \abs(y);
    @*/
  while (y != 0) {
    if (0 < y) { r += x; y -= 1; }
    else       { r -= x; y += 1; }
  }
  return r;
}
```

mul_by_add.c

Multiplication by Addition

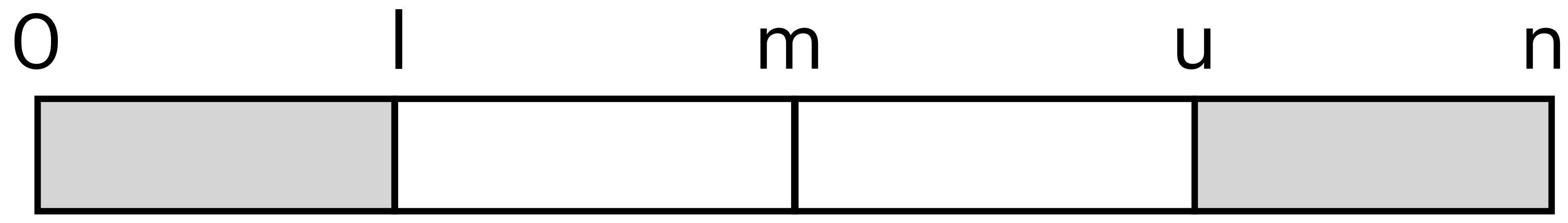
-Prove Goals using Coq-

- Invariant (preserved)
  - .frama-c/wp/interactive/mul_loop_invariant_preserved.v
  - Basic tactics
  - Proof automation
- Loop variant at loop (decrease)
  - .frama-c/wp/interactive/mul_loop_variant_decrease.v
  - Apply lemmas

Binary Search
-C Code-

```c
int binary_search(long t[], int n, long v) {
  int l = 0, u = n - 1;
  while (l <= u) {
    int m = (l + u) / 2;
    if (t[m] < v)       l = m + 1;
    else if (t[m] > v) u = m - 1;
    else               return m;
  }
  return -1;
}
```

binary_search.c



Source: http://proval.lri.fr/gallery/BinarySearchACSL.en.html (invalid now)

Binary Search

-Function Contract-

```
/*@ requires 0 <= n <= (INT_MAX / 2) && \valid(t + (0..n-1));
  @ ensures -1 <= \result < n;
  @ assigns \nothing;
  @ behavior success:
  @   ensures \result >= 0 ==> t[\result] == v;
  @ behavior failure:
  @   assumes sorted(t,0,n-1);
  @   ensures \result == -1 ==>
  @     \forall integer k; 0 <= k < n ==> t[k] != v;
  @*/
```

```
int binary_search(long t[], int n, long v) {
  int l = 0, u = n - 1;
  while (l <= u) {
    int m = (l + u) / 2;
    if (t[m] < v) l = m + 1;
    else if (t[m] > v) u = m - 1;
    else return m;
  }
  return -1;
}
```

Binary Search

-Loop Annotations-

```
/*@ loop invariant 0 <= l <= u + 1 <= n;
  @ loop assigns l, u;
  @ for failure:
  @   loop invariant
  @   \forall integer k;
  @     0 <= k < n && t[k] == v ==> l <= k <= u;
  @ loop variant u-l;
  @*/
```

```
int binary_search(long t[], int n, long v) {
  int l = 0, u = n - 1;
  while (l <= u) {
    int m = (l + u) / 2;
    if (t[m] < v) l = m + 1;
    else if (t[m] > v) u = m - 1;
    else return m;
  }
  return -1;
}
```

Binary Search

-Prove Goals using Coq-

- Invariant (preserved)
  - Contradiction
  - Focus mode
- Loop variant at loop (decrease)
- Post-condition
- Post-condition for `failure′
- Invariant for `failure′ (preserved)

Nistonacci Numbers

-Axiomatic Definition-

$$\text{nist}(n) = \begin{cases} n & \text{if } n < 2 \\ \text{nist}(n - 2) + 2 * \text{nist}(n - 1) & \text{otherwise} \end{cases}$$

```
/*@
  @ axiomatic Nist {
  @   logic integer nist(integer n);
  @   axiom nist1 : \forall integer n; n < 2 ==> nist(n) == n;
  @   axiom nist2 : \forall integer n; !(n < 2) ==> nist(n) == nist(n - 2) + 2 * nist(n - 1);
  @ }
  @*/
```

nistonacci.c

Source: http://toccata.lri.fr/gallery/nistonacci.fr.html

Nistonacci Numbers

-Goal-

```
/*@
  @ requires 0 <= n < m;
  @ ensures n < nist(m);
  @ assigns \nothing;
  @*/
void foo(int n, int m) {
  …
}
```

nistonacci.c

alt-ergo fails to prove the postcondition

Nistonacci Numbers

-Lemma-

```
//@ lemma nist_geN : \forall integer n; 0 <= n ==> n <= nist(n);

/*@
  @ requires 0 <= n < m;
  @ ensures n < nist(m);
  @ assigns \nothing;
  @*/
void foo(int n, int m) {
  …
}
```

nistonacci-lemma.c

alt-ergo proves the postcondition but fails to prove the lemma

Nistonacci Numbers

-Implement nist-

```c
/*@
  @ requires 0 <= n;
  @ ensures n <= \result;
  @ assigns \nothing;
  @*/
int nist_impl(int n) {
  int x = 0, y = 1, i = 0;
  /*@
    @ loop invariant 0 <= i <= n;
    @ loop invariant x == nist(i);
    @ loop invariant y == nist(i + 1);
    @ loop assigns x, y, i;
    @*/
  for (i = 0; i < n; i++) {
    int tmp = x;
    x = y;
    y = tmp + 2 * y;
  }
  return x;
}
```

nistonacci-impl.c

Nistonacci Numbers

-Prove Goals using Coq-

- Invariant (preserved) (`y = nist(n + 1)`)
  - Order of operands
  - Replace terms
- Post-condition
  - Induction

Nistonacci Numbers

-Prove Postcondition by Alt-ergo-

- What can be added to the annotations so that alt-ergo can prove the postcondition (n `<= \result`)?

```
/*@
  @ requires 0 <= n;
  @ ensures n <= \result;
  @ ensures \result == nist(n);
  @ assigns \nothing;
  @*/
int nist_impl(int n) {
  int x = 0, y = 1, i = 0;
  /*@
    @ loop invariant 0 <= i <= n;
    @ loop invariant x == nist(i);
    @ loop invariant y == nist(i + 1);
    @ loop assigns x, y, i;
    @*/
  for (i = 0; i < n; i++) {
    int tmp = x;
    x = y;
    y = tmp + 2 * y;
  }
  return x;
}
```

nistonacci-impl.c

Nistonacci Numbers

-Lemma Function-

```
/*@ ghost
 /@
  @ requires 0 <= n;
  @ ensures n <= \result;
  @ ensures \result == nist(n);
  @ assigns \nothing;
  @/
int nist_impl(int n) {
  …
}
@*/

/*@
  @ requires 0 <= n < m;
  @ ensures n < nist(m);
  @ assigns \nothing;
  @*/
void foo(int n, int m) {
  //@ ghost nist_impl(m);
}
```

nistonacci-ghost.c

Nistonacci Numbers

-Simpler Lemma Function-

```
/*@ ghost
  /@
   @ requires 0 <= n;
   @ ensures n <= nist(n);
   @ assigns \nothing;
   @/
void nist_geN(int n) {
  if (n >= 2) {
    nist_geN(n-1);
    nist_geN(n-2);
  }
  return;
}
@*/
```

nistonacci-rec.c

Inconsistent Annotations

```c
/*@ logic integer last0(integer n) = n%10 == 0 ? 1 + last0(n / 10) : 0; */

/*@
  ensures \false;
  assigns \nothing;
 */
int foo(void) {
  /*@ assert last0(0) == 1; */
  return 0;
}
```

inconsistent1.c

Output:

...

[wp] Proved goals: 3 / 3

Qed: 1

Alt-Ergo 2.4.2: 2 (5ms)

Summary

- Frama-C is a powerful and flexible tool for deductive program verification
- There are still the following challenges:
  - Write a correct specification
  - Write a strong enough loop invariant
  - Analyze proof failures

Reference:Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles. A Lesson on Proof of Programs with Frama-C. Invited Tutorial Paper. International Conference on Tests and Proofs. 2013.