Practical introduction to Frama-C
(without Mathematical notations ;-))

David MENTRÉ <d.mentre@fr.merce.mee.com>

Using content of Jochen Burghardt (Fraunhofer First), Virgile Prevosto (CEA), Julien Signoles (CEA), Nikolay Kosmatov (CEA) and Pascal Cuoq (TrustInSoft)

Content of this introduction to Frama-C

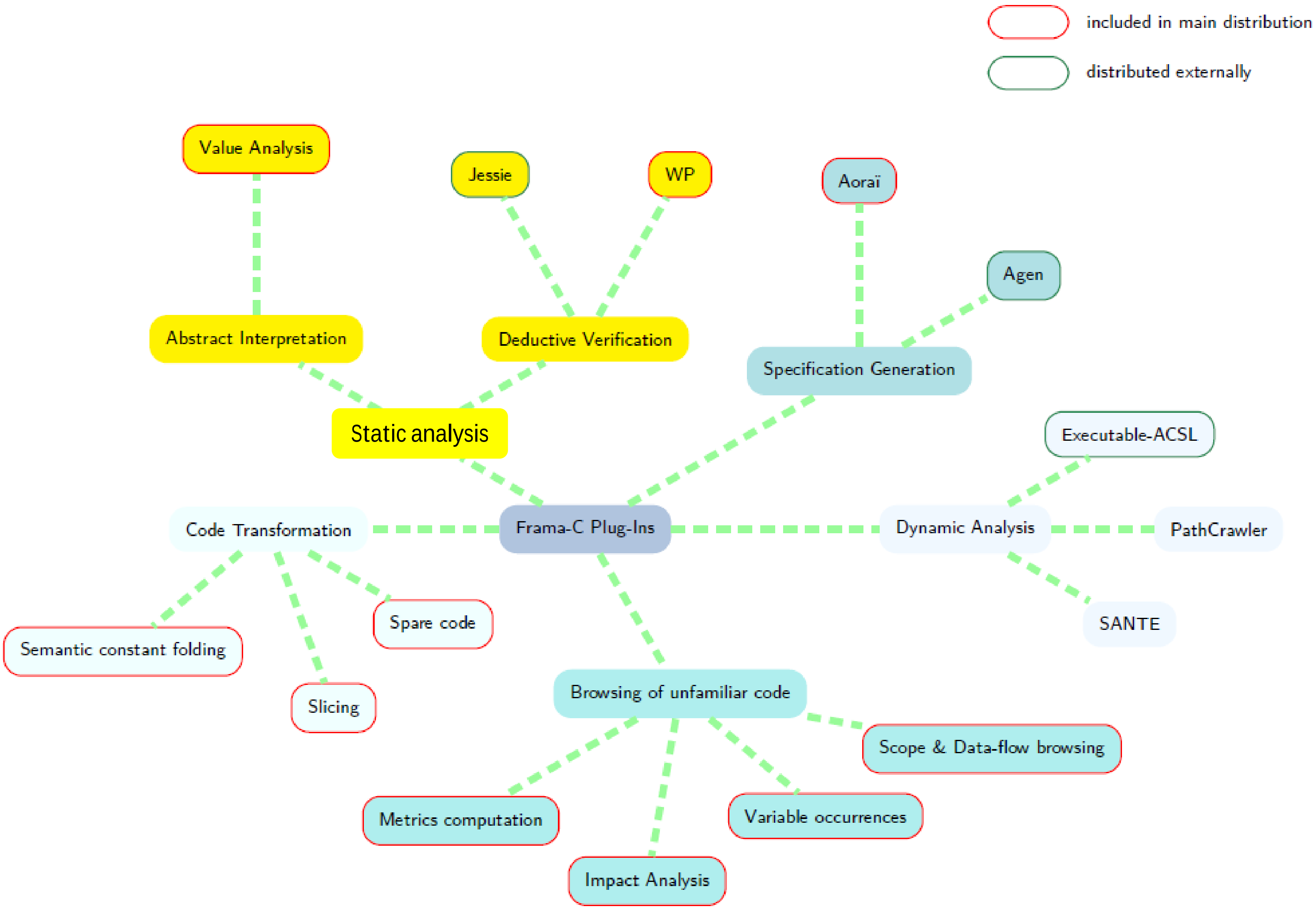
- **What** is Frama-C?
- Interlude: **why** doing formal verification
- The notion of “**contract**”
- **First** use of Frama-C tool
- Basic use of Frama-C/**WP** through examples
- A more **complex** example with WP: find()
- **Behaviors**: clean contracts
- find() example with Frama-C/**Value analysis**
- **E-ACSL**
- **Conclusion**

WHAT IS FRAMA-C?

What is Frama-C?

- **Frama-C** is **FRAM**ework for St**A**tic of **C** language
- Build upon
  - A **core** to read C files and build **Abstract Syntax Trees**
  - A set of **plug-ins** to do **static analyses** and **annotate** those syntax trees
  - **Collaboration** of plug-ins
    - A plug-in can **use** the analysis of **another** plug-in
- Purposes
  - **Static analyses** of C code
  - **Transformation** of C code
  - Framework to **build tools** analyzing and manipulating C code
    - New plug-ins programmed in **OCaml** language

Frama-C plugins



Some plug-ins developed around Frama-C

- **Taster**
  - coding rules, Atos/Airbus, Delmas &al., ERTS 2010
- **Dassault**'s internal plug-ins
  - Automatic annotation, call of external symbolic tool to validate lemmas, interval input subdivision, ...
  - Pariente & Ledinot, FoVeOOs 2010
- **Fan-C**
  - flow dependencies, Atos/Airbus, Duprat &al., ERTS 2012
- Various **academic** experiments, mostly **security**-related

What are main plug-ins of Frama-C?

- **Value analysis**
  - Static verification of C code using **Abstract Interpretation** techniques
- **WP**
  - Static verification of C code using **Weakest Precondition** calculus
  - **Jessie** similar tool

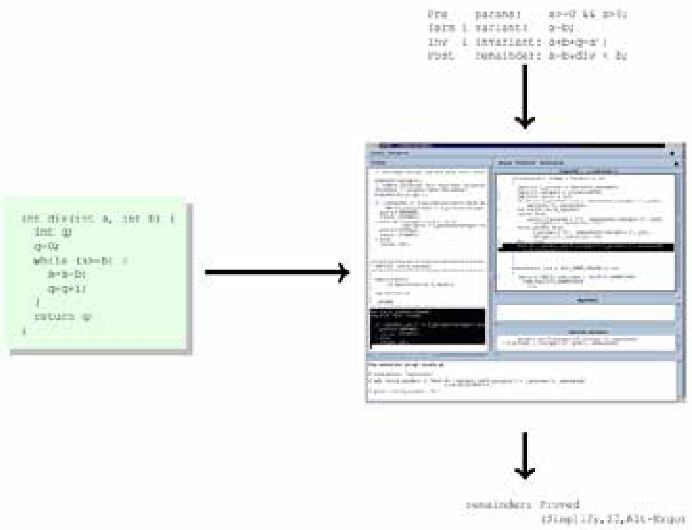
- A lot of **other** plug-ins useful in specific cases
  - **InOut** (computation of outputs from inputs), **Metrics** (analyze code complexity), **Aoraï** (temporal verification), **PathCrawler** (test generation), **Spare code** (remove spare code), ...

Frama-C specification language

- Frama-C is using its own formal **specification language**: **ACSL**
  - ANSI/ISO C Specification Language
- ACSL annotations as special C **comments** `/*@        */`
- ACSL has a lot of **features**
  - **Not** all of them **understood** by all plug-ins!!
    - See each plug-in **documentation** to check the supported features
- E-ACSL: "**Executable**" ACSL variant
  - Annotations can be **compiled** and executed
  - Compatible with ACSL
  - **Mix** test and formal verification!
  - More details later

History of Frama-C

- 90’s: ***CAVEAT***, an Hoare logic-based tool for C programs at CEA
- 2000’s: *CAVEAT* used by Airbus during certification process of the **A380** (DO-178 level A qualification)
- 2001: ***Why*** and (2004) its C front-end ***Caduceus*** (at INRIA)
- 2006: Joint project to write a successor to *CAVEAT* and *Caduceus*
- **2008**: First public release of ***Frama-C*** (Hydrogen version)
- 2010: start of Device-Soft project between Fraunhofer FIRST (now FOKUS) and CEA LIST
- Today (**2013**):
  - Frama-C **Fluorine** (v9.3)
  - Multiple projects around the platform
  - A growing **community** of **users**...
  - ... and of plug-ins **developers**



Frama-C main documentation

- One needs **several** manuals to work
  - **User** manual: manual covering Frama-C main interface, GUI, ...
  - **ACSL** manual: all details of ACSL specification language
  - **Value Analysis** manual: tutorial and detail use of Value Analysis plug-in
  - **WP** manual: detail use of WP plug-in
  - **RTE** manual: detail use of RTE (Run Time Error) plug-in
    - Use with WP
- It can need some time to **find** the searched information ;-)
  - **Ask me** or Frama-C mailing list for information

More information on Frama-C

- Developed at **CEA** and **INRIA** Saclay
- Frama-C is an **Open Source** project (GNU LGPL v2 license)
- **Code** & **documentation** http://frama-c.com
- **Support**
  - Mailing list http://lists.gforge.inria.fr/cgi-bin/mailman/listinfo/frama-c-discuss
    - Very helpful if questions are asked with complete C code
  - StackOverflow with “frama-c” tag http://stackoverflow.com/tags/frama-c/
- **Bug tracking** system http://bts.frama-c.com/
- **Wiki** http://bts.frama-c.com/dokuwiki/doku.php?id=mantis:frama-c:start
  - Papers, tutorials, external plug-ins, …
- **Blog** http://blog.frama-c.com/

INTERLUDE: WHY DOING FORMAL VERIFICATION?

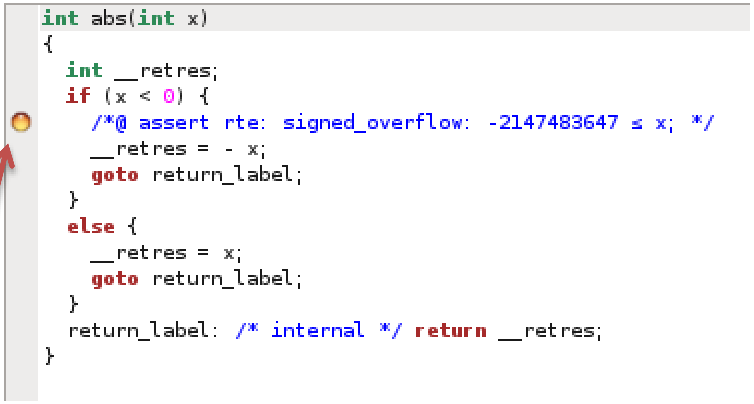
Questions on a simple program

- What **does** the following program?
- Is it **correct**?

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

Answers on a simple program

- The program computes the **absolute value** of x
- It is **buggy**!
  - If **x == -2^{31}**, 2^{31} cannot be represented in binary two's complement!
    - C's int goes from -2^{31} (-2147483648) to 2^{31} -1 (2147483647)
- A formal tool (like Frama-C) can **catch** it
  - "**frama-c-gui** -wp -wp-rte abs.c"
  - **Systematically**!!
    - Of course a programmer **knows** about such issues…
    - … but he might **forget** it while doing more complex things



```
int abs(int x)
{
  int __retres;
  if (x < 0) {
    /*@ assert rte: signed_overflow: -2147483647 ≤ x; */
    __retres = - x;
    goto return_label;
  }
  else {
    __retres = x;
    goto return_label;
  }
  return_label: /* internal */ return __retres;
}
```

Cannot be proved

Question on a little more complex program

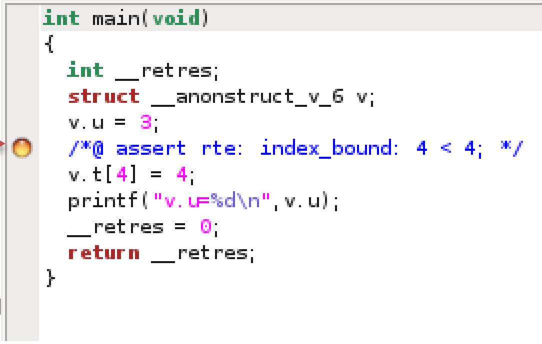
- What **prints** this program?

```
#include <stdio.h>

int main(){
  struct {
    int t[4];
    int u;
  }
 v;


  v.u = 3;
  v.t[4] = 4;
  printf("v.u=%d\n", v.u);
  return 0;
}
```

- **Both** v.u=**3** and v.u=**4**!

```
$ gcc struct-undefined.c && ./a.out
v.u=4
$ gcc -O2 struct-undefined.c && ./a.out
v.u=3
```

- This program uses **undefined** behavior of C99
  - Access **out of bound** of v.t object: optimized in -O2
  - Issue **identified** by Frama-C



```
int main(void)
{
  int __retres;
  struct __anonstruct_v_6 v;
  v.u = 3;
  /*@ assert rte: index_bound: 4 < 4; */
  v.t[4] = 4;
  printf("v.u=%d\n",v.u);
  __retres = 0;
  return __retres;
}
```

THE NOTION OF “CONTRACT”

The notion of “contract”

- **Contract** of a function defines
  - What the function **requires** from the outside world
  - What the function **ensures** to the outside world
    - Provided the “requires” part is fulfilled!
- Similar to **business** contract
- Going back to our **abs()** function
  - abs() requires that **$x > -2^{31}$**: `requires x >= - 2147483647;`
  - abs() ensures that
    - Its result is **positive**: `ensures \result >= 0;`
    - Its result is **–x if x is negative**, x otherwise:
      - `ensures x < 0 ==> \result == -x;`
      - `ensures x >= 0 ==> \result == x;`
  - “**\result**” denotes function result
  - Using Frama-C **notation**:

Formal annotation

```
/*@ requires x >= -2147483647;
    ensures \result >= 0;
    ensures x < 0 ==> \result == -x;
    ensures x >= 0 ==> \result == x;
 */
```

Version of abs() with contract

- Full **Frama-C** version of abs()
  - Contract is put **before** first line of abs()

```
/*@ requires x >= -2147483647;
    ensures \result >= 0;
    ensures x < 0 ==> \result == -x;
    ensures x >= 0 ==> \result == x;
 */
int abs(int x){
  if (x < 0)
    return -x;
  else
    return x;
}
```

- Contracts can be more elaborated (see later)

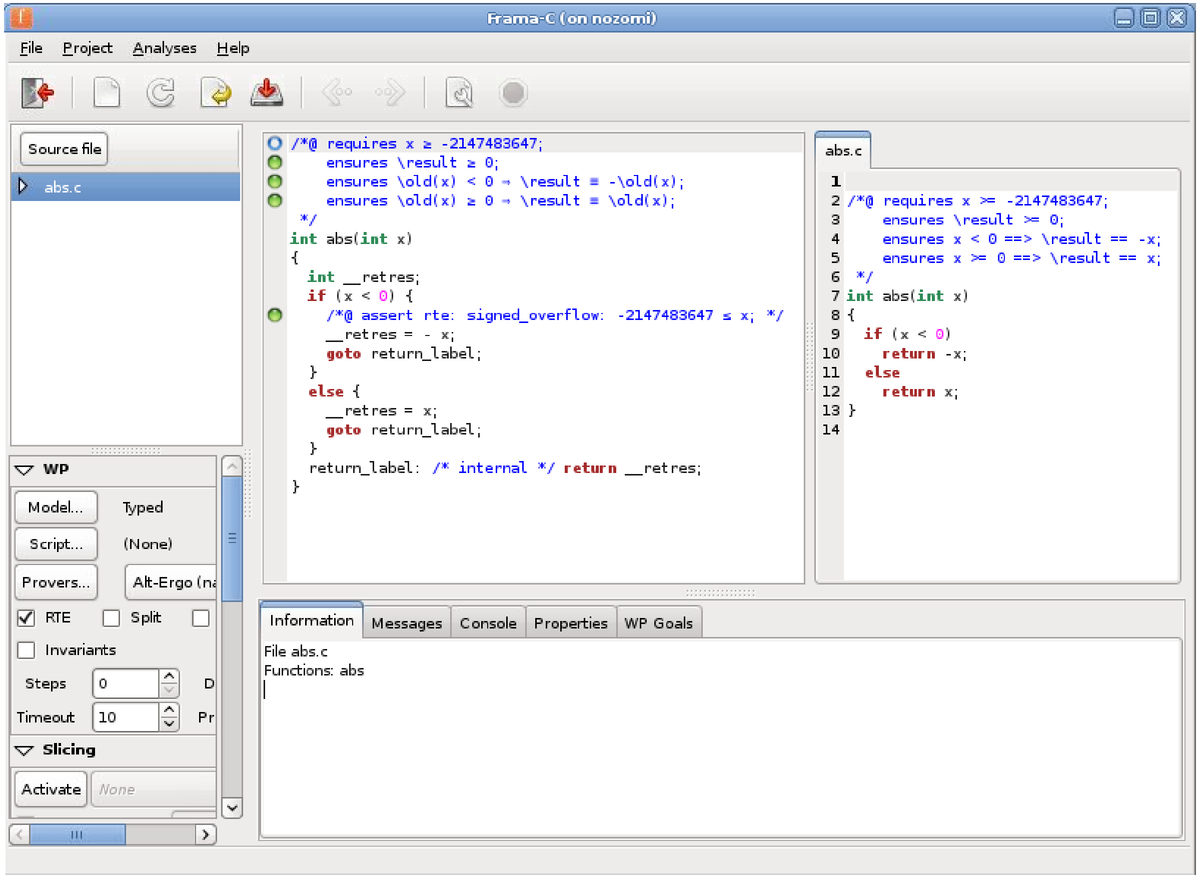
- Note: one can do the same with **assert()** and **test** it
  - But this is more **cumbersome**!

```
#include <assert.h>

int abs(int x){
  int old_x = x;
  int returned_x;

  assert(x >= -2147483647);

  if (x < 0)
    returned_x = -x;
  else
    returned_x = x;

  assert(old_x < 0 ?
           returned_x == -old_x : 1);
  assert(old_x >= 0 ?
           returned_x == old_x : 1);

  return returned_x;
}
```

■ FIRST USE OF FRAMA-C TOOL

Use of Frama-C/WP tool on abs()

- Call with “**frama-c-gui -wp -wp-rte** abs.c”
  - **-wp**: call WP plug-in
  - **-wp-rte**: call RTE plug-in that inserts additional checks for Run Time Errors
- **DEMO**!
  - Start without contract
  - Add progressively contract parts
  - Show how Alt-Ergo is called



Typed

(None)

Alt-Ergo (na

☑ RTE ☐ Split ☐

☐ Invariants

Steps 0

Timeout 10

```
/*@ requires x ≥ -2147483647;
    ensures \result ≥ 0;
    ensures \old(x) < 0 ⇒ \result ≡ -\old(x);
    ensures \old(x) ≥ 0 ⇒ \result ≡ \old(x);
 */
int abs(int x)
{
  int __retres;
  if (x < 0) {
    /*@ assert rte: signed_overflow: -2147483647 ≤ x; */
    __retres = - x;
    goto return_label;
  }
  else {
    __retres = x;
    goto return_label;
  }
  return_label: /* internal */ return __retres;
}
```

abs.c

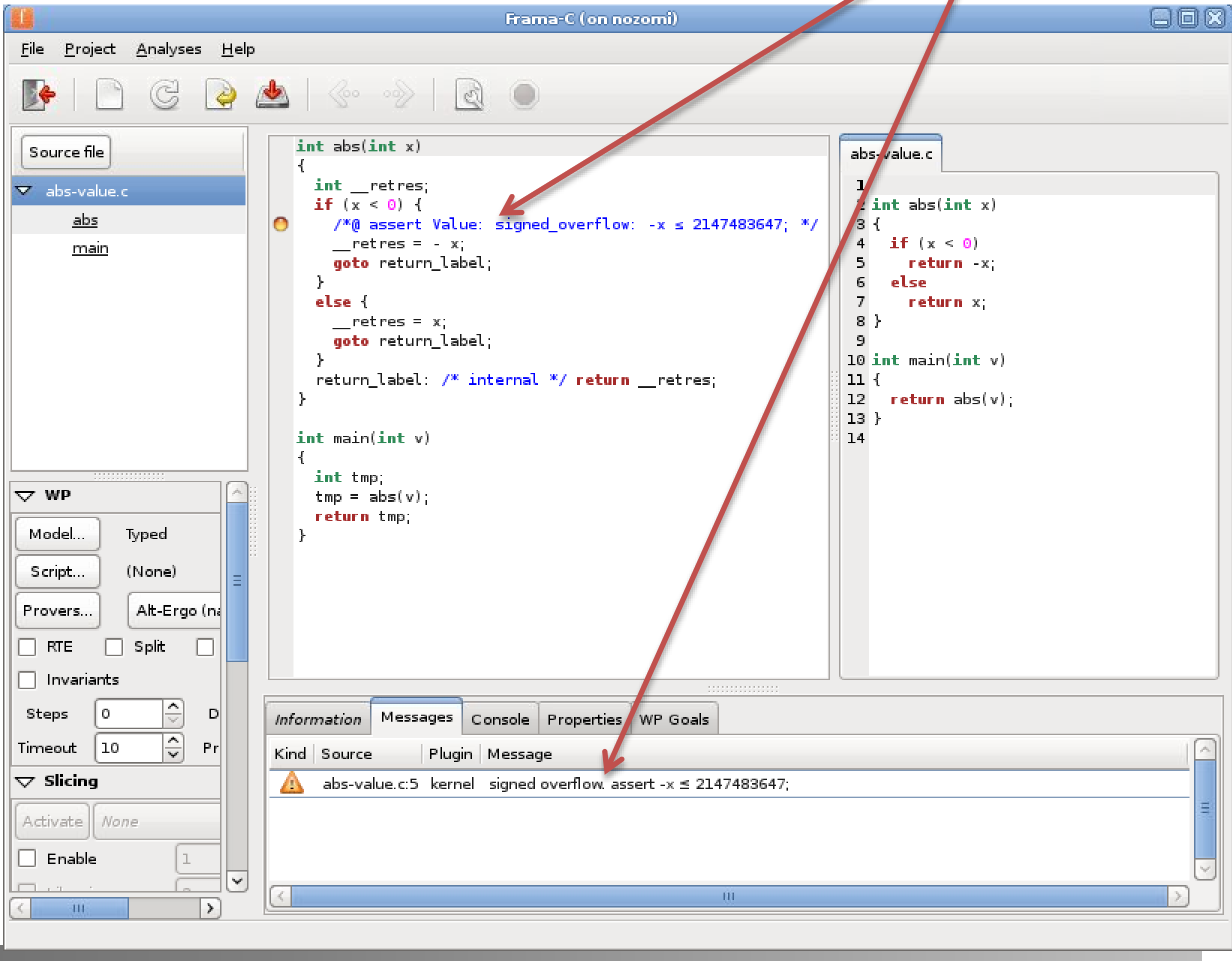
```

/*@ requires x >= -2147483647;
    ensures \result >= 0;
    ensures x < 0 ==> \result == -x;
    ensures x >= 0 ==> \result == x;
 */
int abs(int x)
{
  if (x < 0)
    return -x;
  else
    return x;
}

```

File abs.c
Functions: abs

Use of Frama-C/Value tool on abs()

- Call with “frama-c-gui **-val** abs-value.c”
  - **-val**: call Value analysis plug-in
- Need to write a “**driver**”
  - **call** the function in all possible contexts
- **DEMO**!
  - Start with driver only
  - Add correction code

Overflow is seen



```c
int abs(int x)
{
  int __retres;
  if (x < 0) {
    /*@ assert Value: signed_overflow: -x ≤ 2147483647; */
    __retres = - x;
    goto return_label;
  }
  else {
    __retres = x;
    goto return_label;
  }
  return_label: /* internal */ return __retres;
}

int main(int v)
{
  int tmp;
  tmp = abs(v);
  return tmp;
}
```

```c

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

int main(int v)
{
  return abs(v);
}

```

Kind	Source	Plugin	Message
⚠	abs-value.c:5	kernel	signed overflow. assert -x ≤ 2147483647;

Comparison of WP vs. Value analysis

- **Value analysis**
  - Need **less** annotations
  - Need to write a proper **driver** and used function **contracts**
    - Possible **incorrect** analysis if incorrect driver
  - **Limited** set of proved properties
    - Mainly absence of Run Time Error
- **WP**
  - Need to add **more** annotations: more work
  - More **complex** properties can be proved
- No definitive tool
- Both tools can be **combined**
  - Advantage of Frama-C framework over other tools!

BASIC USE OF FRAMA-C/WP THROUGH EXAMPLES

Function call and contract

- A contract is an "**opaque**" specification of function behavior
  - Function **callers** only see the **contract**
    - Contract **considered correct** even if not proved
  - If **no** contract… unknown behavior! (default contract)
- **DEMO** on call.c: "frama-c-gui -wp -wp-rte call.c"
  - Initial state: **all** proved
  - Show farenheit_to_celsius() "**requires**" not fulfilled
    - farenheit_to_celsius() and main() "**ensures**" still **proved**
  - Show farenheit_to_celsius() "**ensures**" not fulfilled
    - main() "**ensures**" still **proved**
- **Everything** should be proved to guarantee the program correct !

Old and new values, pointers: swap()

- In a contract, need to express:
  - **Validity** of pointers
  - For a variable x, value of x at function **entrance** and **exit**
- **Informal** specification
  - “Exchange two integer values pointed by pointers”
  - **Prototype**: `void swap(int *a, int *b)`
- What is swap() **formal** specification?
  - **Requires**: the pointers need to be **valid**
    - “**\valid(a)**”: pointer a is valid
  - **Ensures**: the pointed values are **swapped**
    - “**\old(a)**”: value of a at function **entrance** (in function contract ensures)
    - “**a**”: value of a at function **exit**

swap() contract and code

- **Contract** and **code**

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

  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

- **DEMO**: “frama-c-gui -wp -wp-rte swap.c”

Side note: Frama-C operators in specification

- Several **operators** useful in specification
  - Similar to **C** notation

Operator	Informal meaning	Formal meaning (C notation)
!p	**NOT** p	!p
p && q	p **AND** q	p && q
p || q	p **OR** q	p || q
p ==> q	**IF** p **THEN** q	(p ? q : 1)
p <==> q	p **IF AND ONLY IF** q	p == q

- No logical “IF p THEN q1 **ELSE** q2”
  - Use “(**p ==>** q1) **&&** (**!p ==>** q2)” instead
  - Or more simply “p **?** q1 **:** q2”

swap() variation: two elements in an array

- **Informal** specification
  - “In array a[] of size n, exchange array elements indexed by n1 and n2”
- **Prototype**:
  - `void array_swap(int n, int a[], int n1, int n2)`
- What is its **formal** specification?
  - The indexes are within array **bounds**
    - `requires n >= 0 && 0 <= n1 < n && 0 <= n2 < n;`
  - The array a[] is **valid** memory area up to cell number n
    - `requires \valid(a+(0..n-1));` (similar to `&a[0]` valid, ..., `&a[n]` valid)
  - The indexed values are **swapped**
    - `ensures (a[n1] == \old(a[n2]) && a[n2] == \old(a[n1]));`

array_swap() contract and code

- **Contract** and **code**

```
/*@ requires n >= 0 && 0 <= n1 < n && 0 <= n2 < n;
    requires \valid(a+(0..n-1));
    ensures (a[n1] == \old(a[n2]) && a[n2] == \old(a[n1]));
 */
void array_swap(int n, int a[], int n1, int n2){
  int tmp;

  tmp = a[n1];
  a[n1] = a[n2];
  a[n2] = tmp;
}
```

- **DEMO**: “frama-c-gui -wp -wp-rte array_swap.c”

A MORE COMPLEX EXAMPLE WITH WP: FIND()

find() specification

- **Informal** specification
  - “Return the index of an occurrence of v in a[]”
  - “Array a[] is of size n, value v and n are integers”

- **Prototype**:

```
int find(int n, const int a[], int v)
```

- What is its **formal** specification**?**
  - We will elaborate it through some unit **tests**

Case 1: find() finds v in a[]

- **Informal** specification
  - “Return the index of an occurrence of v in a[]”
  - “Array a[] is of size n, value v and n are integers”
- **Prototype**:

```
int find(int n, const int a[], int v)
```

- find() **finds v** in a[]

```
int a[5] = { 9, 7, 8, 9, 6 };

int const f1 = find(5, a, 8);
assert(f1 == 2);
```

- **Formally**

```
ensures 0 <= \result < n ==> a[\result] == v;
```

Case 2: find() does not find v in a[]

- **Informal** specification
  - “Return the index of an occurrence of v in a[]”
  - “Array a[] is of size n, value v and n are integers”
  - “**Returns -1 if v is not found**”
- Prototype:

```
int find(int n, const int a[], int v)
```

- find() **does not find v** in a[]

```
int a[5] = { 9, 7, 8, 9, 6 };

int const f2 = find(5, a, 15);
assert(f2 == -1 );
```

- **Formally**
  - If find() returns -1, then
    - for all index i, if i is in a[] bounds then a[i] != v

```
ensures \result == -1
              ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
```

Side note: types used in ACSL annotations

- In ACSL, **distinction** between C program and mathematical **types**

C program type	Mathematical type
`int, short`	`integer` (Z)
`float, double`	`real` (R)

- Usually one uses mathematical types for annotations
  - "`\forall integer i; ...`"
    - And not "`\forall int i; ...`"
    - It simplifies generated Verification Condition (not need to add restrictions on `int` range)

Case 3: find() does not modify a[]

- Would it be a **valid** find()**?**

```
int find(int n, int a[], int v){
 if (n > 0) {
    a[0] = v;
    return 0;
  } else
    return -1;
}
```

- We can express it formally
  - **`assigns \nothing`**;
  - Note: "**const**" expressed it formally but Frama-C does **not understand** "const"

Case 4: valid input and returned values

- **Informal** specification
  - “Array a[] is of size n, value v and n are integers”
- **Formal** specification?
  - `requires 0 <= n && \valid(a+(0..n-1));`
- **Informal** specification
  - “find() result is between -1 and n (excluded)
- **Formal** specification?
  - `ensures -1 <= \result < n;`

Wrap-up: find() formal contract

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;
    ensures \result == -1
           ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
    ensures 0 <= \result < n ==> a[\result] == v;
    ensures -1 <= \result < n;
 */
```

find() code

- **DEMO**: how to **prove** find() code?
  - “frama-c-gui -wp -wp-rte find.c”

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;
    ensures \result == -1
           ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
    ensures 0 <= \result < n ==> a[\result] == v;
    ensures -1 <= \result < n;
 */
int find(int n, const int a[], int v){
  int i;

  for (i=0; i < n; i++) {
    if (a[i] == v) {
      return i;    }
  }

  return -1;
}
```

Loops: how to handle them?

- Main rule: **loops** are “**opaque**”
  - So one needs to **add** needed **annotations** to help automatic provers prove desired properties
  - loop **invariant**, loop **assigns**, loop **variant**

- Loop **invariant**: property always true in a loop
  - Should be **true** at loop **entry**
  - Should be **true** at each loop **iteration**
    - Even if **no** iterations are possible
  - Should be true at loop **exit**

Example of loop invariant (1/2)

- “Loop index is between **0** and **n** (inclusive)”

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;
    ensures \result == -1
           ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
    ensures 0 <= \result < n ==> a[\result] == v;
    ensures -1 <= \result < n;
 */

int find(int n, const int a[], int v){
  int i;

/*@
    loop invariant 0 <= i <= n;



*/
  for (i=0; i < n; i++) {
    if (a[i] == v) {
      return i;    }
  }

  return -1;
}
```

Example of loop invariant (2/2)

- “**Up to** index i, value v is still **not found**”

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;
    ensures \result == -1
           ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
    ensures 0 <= \result < n ==> a[\result] == v;
    ensures -1 <= \result < n;
 */

int find(int n, const int a[], int v){
  int i;

/*@
    loop invariant 0 <= i <= n;
    loop invariant \forall integer j; 0 <= j < i ==> a[j] != v;

*/
  for (i=0; i < n; i++) {
    if (a[i] == v) {
      return i;    }
  }

  return -1;
}
```

We build progressively the desired property

Loop assigns and loop variant

- Loop **assigns**: what is assigned within the loop

- Loop **variant**: to prove **termination**
  - Show a metric **strictly decreasing** at each loop iteration and **bounded** by 0

```
int find(int n, const int a[], int v){
  int i;

/*@ loop invariant 0 <= i <= n;
    loop invariant \forall integer j; 0 <= j < i ==> a[j] != v;
    loop assigns i;
    loop variant n - i;
 */
  for (i=0; i < n; i++) {
    if (a[i] == v) {
      return i;    }
  }

  return -1;
}
```

find() final proved code

- “frama-c-gui -wp -wp-rte find-proved.c”

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;
    ensures \result == -1
           ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
    ensures 0 <= \result < n ==> a[\result] == v;
    ensures -1 <= \result < n;
 */

int find(int n, const int a[], int v){
  int i;

/*@ loop invariant 0 <= i <= n;
    loop invariant \forall integer j; 0 <= j < i ==> a[j] != v;
    loop assigns i;
    loop variant n - i; */
  for (i=0; i < n; i++) {
    if (a[i] == v) {
      return i;    }
  }

  return -1;
}
```

A note on proof with WP

- **More** annotations than code!
  - **8** lines of **code**
  - **10** lines of **annotations**
- Because what we prove is **complicated**
  - A loop, in **all** possible cases!
- It corresponds to **exhaustive** test!

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
   assigns \nothing;
   ensures \result == -1
        ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
   ensures 0 <= \result < n ==> a[\result] == v;
   ensures -1 <= \result < n;
 */

int find(int n, const int a[], int v){
  int i;

/*@ loop invariant 0 <= i <= n;
   loop invariant \forall integer j; 0 <= j < i ==> a[j] != v;
   loop assigns i;
   loop variant n - i; */
  for (i=0; i < n; i++) {
   if (a[i] == v) {
     return i;   }
  }

  return -1;
}
```

BEHAVIORS: CLEAN CONTRACTS

How to write clean contracts?

- Important to write **clean** contracts
  - Improve **readability**: contract is a readable **specification**
    - Help **understand** the code (e.g. in code review)
    - But such specification can be **mechanically** checked!
      - **No** more out-dated comments
  - Help proofs
- “**Behaviors**” can be use to separate several cases
  - **Name** each behavior
  - Give a “**sub-contract**” for each behavior
    - assumes, requires, ensures
- **Bonus**: one can additionally **check** that all behaviors...
  - ...Cover **all** possible inputs (**complete** behaviors)
  - ...Cover **different** cases (**disjoint** behaviors)

find() contract using behaviors

- “frama-c-gui -wp -wp-rte find-behavior.c”

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;
```

Array contains v at an index i

```
    behavior found:
      assumes \exists integer i; 0 <= i < n && a[i] == v;
      ensures a[\result] == v;
```

In that case return the correct index

Array does not contain v for all possible indexes i

```
    behavior not_found:
      assumes \forall integer i; 0 <= i < n ==> a[i] != v;
      ensures \result == -1;
```

In that case return -1

```
    complete behaviors;
    disjoint behaviors;
 */
```

We cover all behaviors

All behaviors consider different cases

Side note: \exists and \forall operators

- To express something over a **range** of values
- Examples
  - `int a[5] = {1, 5, 3, 2, 1};`
  - `\exists integer i; 0 <= i < 5 && a[i] == 1;`

i	-1	0	1	2	3	4	5
a[i]	?	1	5	3	2	1	?
0 <= i < 5	û	ü	ü	ü	ü	ü	û
a[i] == 1	û	ü	û	û	û	ü	û

  - `\forall integer i; 0 <= i < 5 ==> a[i] != 4;`

i	-1	0	1	2	3	4	5
a[i]	?	1	5	3	2	1	?
0 <= i < 5	û	ü	ü	ü	ü	ü	û
a[i] != 4	û	ü	ü	ü	ü	ü	û

Side note: opposite expressions

- **Opposite** expressions: 1st example
  - `int a[5] = {1, 5, 3, 2, 1};`

`\exists index i; a[i] == 1`

`\forall index i; a[i] != 1`

i	0	1	2	3	4
a[i]	1	5	3	2	1
a[i] == 1	ü ü	û	û	û	ü ü
a[i] != 1	û	ü	û ü	ü	û

True ü

False û

- Still **opposite** expressions (with proper indexing)
  - `\exists integer i; 0 <= i < n && a[i] == v;`

  vs.
  - `\forall integer i; 0 <= i < n ==> a[i] != v;`

FIND() EXAMPLE WITH FRAMA-C/VALUE ANALYSIS

Value analysis on find() example

- Is it possible to prove properties with **less** annotations?
  - **Yes**, on a specific program with **Value analysis** plug-in

- We need to define a **driver** calling find()

```
#define N 10

int main(void){
  int a[N] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  int i, n, result;

  for (i=0; i < N; i++) {
    result = find(N, a, i);
    //@ assert result == i;
  }

  return 0;
}
```

Calling Value analysis with proper parameters

- “frama-c-gui **-val** find-value-constant.c”
  - assert is **not** proved, 2 ensures of find() **not** proved
  - We need to augment the **precision** of the analysis
    - Use “**-slevel *n***” parameter: semantic unrolling
      - compute up to ***n* states** from different execution path before computing the union of states

- “frama-c-gui -val **-slevel 10** find-value-constant.c”
  - Now everything is **proved**!
  - **Except** “\assigns nothing”
    - Value analysis doesn’t look at it!

- Rule of thumb: **increase -slevel** parameter
  - But analysis take **longer time**... up to being unusable!
  - è **balance** -slevel precision with needs

A more generic verification

- We can use a more **generic** driver

```
#define N 10

int main(void){
  int a[N];
  int i, n, result;

  for (i = 0; i < N; i++)
    a[i] = Frama_C_interval(-2147483647, 2147483648);

  while (1) {
    n = Frama_C_interval(0, N);
    result = find(n, a, 0);
  }

  return 0;
}
```

Return random value between min and max

Call find() with a[] size between 0 and N elements

Result of Value analysis with generic driver

- “frama-c-gui -val **-slevel 10** find-value-generic.c”
  - All ensures clauses of find() **not** proved
  - A **check added** in find()’s for loop

- “frama-c-gui -val **-slevel 100** find-value-generic.c”
  - One ensures clause **proved**
  - **No** more **check** in find()’s for loop
  - And still no proof attempt on “assigns \nothing”

- Value analysis is similar to a set of **symbolic tests**
  - Exhaustive testing is **not always** possible

E-ACSL

E-ACSL

- **E-ACSL** is **Executable** ACSL
  - Logic of E-ACSL modified to make all annotations **compilable**
    - Partial logic (failure can occur) instead of total logic
  - Compatible: all E-ACSL expressions are **valid ACSL** expressions

- DEMO: first-eacsl.c

```
int main(void){
  int x = 0;

  /*@ assert x == 0; */
  /*@ assert x == 1; */

  return 0;
}
```

This assertion is invalid

Calling E-ACSL

- Annotate C code
  - “frama-c -e-acsl first-eacsl.c -then-on e-acsl -print -ocode monitored.c”

  - **-e-acsl**: call E-ACSL **plug-in** to generate annotated code in new Frama-C project named “e-acsl”
  - **-then-on e-acsl**: switch to Frama-C project named “e-acsl”
  - **-print**: print code of current project
  - **-ocode monitored.c**: output printed code in “monitored.c” file

E-ACSL annotated code

- **Generated** by e-acsl plug-in

```
/* Generated by Frama-C */
struct __anonstruct___mpz_struct_1 {
   int _mp_alloc ;
   int _mp_size ;
   unsigned long *_mp_d ;
};
typedef struct __anonstruct___mpz_struct_1 __mpz_struct;
typedef __mpz_struct ( __attribute__((__FC_BUILTIN__)) mpz_t)[1];
typedef unsigned int size_t;
/*@ requires predicate ≢ 0;
    assigns \nothing; */
extern  __attribute__((__FC_BUILTIN__)) void e_acsl_assert(int predicate,
                                                           char *kind,
                                                           char *fct,
                                                           char *pred_txt,
                                                           int line);

/*@
model __mpz_struct { ℤ n };
*/
int __fc_random_counter __attribute__((__unused__));
unsigned long const __fc_rand_max = (unsigned long)32767;
/*@ ghost extern int __fc_heap_status; */

/*@
axiomatic
  dynamic_allocation {
  predicate is_allocable{L}(size_t n)
    reads __fc_heap_status;

  }
 */
extern size_t __memory_size;

/*@
predicate diffSize{L1, L2}(ℤ i) =
  \at(__memory_size,L1)-\at(__memory_size,L2) ≡ i;
 */
int main(void)
{
  int __retres;
  int x;
  x = 0;
  /*@ assert x ≡ 0; */
  e_acsl_assert(x == 0,(char *)"Assertion",(char *)"main",(char *)"x == 0",6);
  /*@ assert x ≡ 1; */
  e_acsl_assert(x == 1,(char *)"Assertion",(char *)"main",(char *)"x == 1",7);
  __retres = 0;
  return __retres;
}
```

```
int main(void){
  int __retres;
  int x;
  x = 0;
  /*@ assert x ≡ 0; */
  e_acsl_assert(x == 0,(char *)"Assertion",(char *)"main",(char *)"x == 0",6);
  /*@ assert x ≡ 1; */
  e_acsl_assert(x == 1,(char *)"Assertion",(char *)"main",(char *)"x == 1",7);
  __retres = 0;
  return __retres;
}
```

Compiling and executing annotated code

- **Compile** annotated code
  - “gcc `frama-c -print-share-path`/e-acsl/e_acsl.c monitored.c”
  - **`frama-c -print-share-path`/e-acsl/e_acsl.c**: compile with e_acsl.c support library

- **Execute** annotated code

```
 $ ./a.out
Assertion failed at line 7 in function main.
The failing predicate is:
x == 1.
```

Test and proof with E-ACSL

- E-ACSL allows to **mix** test and proof
  - Use E-ACSL annotation on code
  - **Test** it!
  - For safety critical code: **prove** it!

- **Documentation** on E-ACSL
  - **E-ACSL** manual: documentation for E-ACSL specification language
  - **E-ACSL implementation** manual: what is currently implemented by E-ACSL plug-in
  - **E-ACSL user** manual: how to use the plug-in

■ CONCLUSION

Not addressed in this presentation

- **Axiomatization** in specification language
  - To write more complex specifications and proofs
- Plug-in **development** using OCaml API
  - To develop one’s own analyses, to automate manual review
- **Ghost** variables and code
- All **plug-ins** in detail (InOut, PathCrawler, Aoraï, ...)
- ...

To conclude

- Frama-C is a generic **framework** for **static** analysis of **C code**
  - Set of **plug-ins** for code discovery and analysis
  - Two **main** plug-ins: **WP** and **Value analysis**
  - All plug-in use a single **specification** language: **ACSL** (in comments)

- **WP**: proof of complete **properties** possible
  - But a lot (and sometimes complex) **annotations** are needed
- **Value analysis**: needs **less** annotations
  - But a proper **driver** and called function **contracts** are needed
  - Prove **less** properties
    - Mainly absence of **Run Time error**

- Both tools (and others) can be **combined**
  - Tailor the analysis to the user **needs**

?