Deductive Verification of C Programs

Kalyan Krishnamani

NVIDIA Corporation

USA

at

CSI National Conference on Formal Methods

Bangalore

October 2014

Plan of the Talk

Goal of the talk

Verification

- Formal Verification of C
- Motivation
- Deductive Verification

Frama-C

- ACSL
- Verification using Frama-C

Other details

- Interesting Frama-C plugins
- Other interesting tools

Goal of the Talk

What will you learn ? ☺

- Formal specification, verification of C programs
- Motivation to write cleaner code (very useful !)
- A simple tool that you can readily experiment with - Frama-C
- More tools that might address some of your problems

Goal of the Talk

What will you learn ? ☺

- Formal specification, verification of C programs
- Motivation to write cleaner code (very useful !)
- A simple tool that you can readily experiment with - Frama-C
- More tools that might address some of your problems

What you will not learn ? ☹

- Developing custom plugins using Frama-C
- OCaml, Why/Why3, Coq, Boogie, etc.

Goal of the Talk

What will you learn ? ☺

- Formal specification, verification of C programs
- Motivation to write cleaner code (very useful !)
- A simple tool that you can readily experiment with - Frama-C
- More tools that might address some of your problems

What you will not learn ? ☹

- Developing custom plugins using Frama-C
- OCaml, Why/Why3, Coq, Boogie, etc.

How can you get maximum benefit ?

- Start with simple examples and understand them completely
- Incrementally apply to more involved examples (from your own work ?)
- Use / get involved in Frama-C mailing list discussions

Let's start...

Is this a correct C function ?

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

Let's start...

Is this a correct C function ?

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

- If $x == -2^{31}$, 2^{31} can not be represented in binary 2's complement
  - C integers go from -2^{31} to $2^{31} - 1$

Let's start...

Is this a correct C function ?

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

- If $x == -2^{31}$, 2^{31} can not be represented in binary 2's complement
  - C integers go from -2^{31} to $2^{31} - 1$
- Let's use Frama-C demo

Let's start...

Is this a correct C function ?

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

- If $x == -2^{31}$, 2^{31} can not be represented in binary 2's complement
  - C integers go from -2^{31} to $2^{31} - 1$
- Let's use Frama-C demo
- You might think of it as a silly example and a sillier verification task. But ...

Some Motivation

- June 4, 1996. Ariane 5 launch
- 37 seconds after lift-off, loses control, breaks off and self-destructs

What happened ?

Some Motivation

- June 4, 1996. Ariane 5 launch
- 37 seconds after lift-off, loses control, breaks off and self-destructs

What happened ?

- Software failed when an attempt to convert a 64-bit floating point number to a signed 16-bit integer caused the number to overflow
- No exception handler. software shuts down
- Back up software was a copy of this. That behaved exactly the same way
- Code re-use without verification

Proving Programs

Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Consider the analogy of checking an addition. If it is given as:

1374
5906
6719
4337
7768

26104

one must check the whole at one sitting, because of the carries.

But if the totals for the various columns are given, as below:

1374

Another quick example

Is this program correct ?

```
// bounds.c
int main() {
  struct {
   int u[4];
   int v;
  } s;
  s.v = 3;
  s.u[4] = 4;
  printf ("s.v=%d\n", s.v);
  return 0;
}
```

Another quick example

Is this program correct ?

```
// bounds.c
int main() {
  struct {
   int u[4];
   int v;
  } s;
  s.v = 3;
  s.u[4] = 4;
  printf ("s.v=%d\n", s.v);
  return 0;
}
```

- `gcc` will still compile this

Another quick example

Is this program correct ?

```
// bounds.c
int main() {
  struct {
   int u[4];
   int v;
  } s;
  s.v = 3;
  s.u[4] = 4;
  printf ("s.v=%d\n", s.v);
  return 0;
}
```

- `gcc` will still compile this
- `gcc bounds.c` and `gcc -O2 bounds.c` might produce different results

Another quick example

Is this program correct ?

```c
// bounds.c
int main() {
  struct {
   int u[4];
   int v;
  } s;
  s.v = 3;
  s.u[4] = 4;
  printf ("s.v=%d\n", s.v);
  return 0;
}
```

- `gcc` will still compile this
- `gcc bounds.c` and `gcc -O2 bounds.c` might produce different results
- This is because `-O2` turns off out of bounds check

Another quick example

Is this program correct ?

```
// bounds.c
int main() {
  struct {
   int u[4];
   int v;
  } s;
  s.v = 3;
  s.u[4] = 4;
  printf ("s.v=%d\n", s.v);
  return 0;
}
```

- `gcc` will still compile this
- `gcc bounds.c` and `gcc -O2 bounds.c` might produce different results
- This is because `-O2` turns off out of bounds check
- Let's use Frama-C demo

Frama-C : History

- Jointly developed by C.E.A and INRIA
- A successor to CAVEAT tool (Hoare logic for C) and Caduceus (Why+C front end)
- CAVEAT was being tested to certify certain critical code of A380 by Airbus
- Also a successor to Krakatoa (similar concepts for Java)

Frama-C

Frama-C : History

- Jointly developed by C.E.A and INRIA
- A successor to CAVEAT tool (Hoare logic for C) and Caduceus (Why+C front end)
- CAVEAT was being tested to certify certain critical code of A380 by Airbus
- Also a successor to Krakatoa (similar concepts for Java)

Frama-C : Now

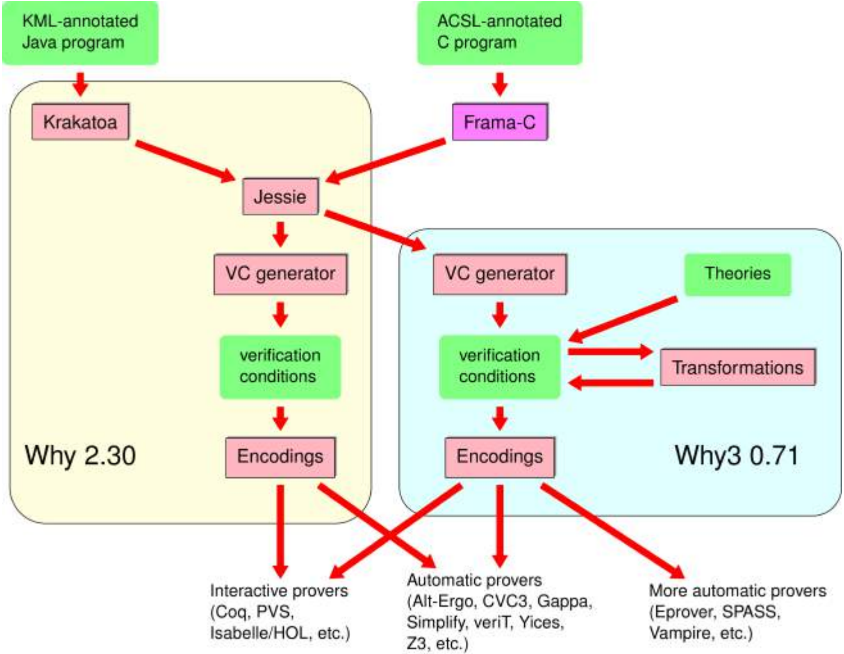
- Several versions of Frama-C released (several interesting plugins)
- An active, growing community of users
- Avionics companies like Airbus, Dassault Aviation are active users

Frama-C : Ingredients

Frama-C : A **Fra**mework for **M**odular **A**nalysis of **C** Programs

- Modular Architecture
- Uses CIL (UC Berkeley's) for C program representation
- ACSL front-end as the specification
- Several plugins :
  - Value Analysis
  - Impact Analysis
  - Jessie
  - Slicing
  - WP

Frama-C : Architecture



Deductive Verification : Hoare Triple

- Introduced by Floyd and Hoare
- Hoare triple :

$$\{P\}\ s\ \{Q\}$$

Deductive Verification : Hoare Triple

- Introduced by Floyd and Hoare
- Hoare triple :

$$\{P\}\ s\ \{Q\}$$

  - s is any program fragment, P is the pre-condition and Q, the post-condition
  - If P holds, Q will hold after the execution of statement s
  - Deduction rules based on Hoare triple

Deductive Verification : Ingredients

- Specification language
- Deduction rules
- Verification Condition generator

Deduction Rules : Specification

Specification :

- *Specification Language* : mathematical language to write the components of the specification - function contracts, invariants, etc.
- What language could be used ?

Deduction Rules : Specification

Specification :

- *Specification Language* : mathematical language to write the components of the specification - function contracts, invariants, etc.
- What language could be used ?
  - A natural choice is first order predicate calculus with equality

Specification :

- *Specification Language* : mathematical language to write the components of the specification - function contracts, invariants, etc.
- What language could be used ?
  - A natural choice is first order predicate calculus with equality
  - + other theories : linear arithmetic, bit-vectors, purely applicative arrays, etc.

Deduction Rules : Specification

Specification :

- *Specification Language* : mathematical language to write the components of the specification - function contracts, invariants, etc.
- What language could be used ?
  - A natural choice is first order predicate calculus with equality
  - + other theories : linear arithmetic, bit-vectors, purely applicative arrays, etc.
  - + recursive functions, algebraic data-types (Dafny), polymorphism and inductive predicates (Why)

Deduction Rules : Specification

Specification :

- *Specification Language* : mathematical language to write the components of the specification - function contracts, invariants, etc.
- What language could be used ?
  - A natural choice is first order predicate calculus with equality
  - + other theories : linear arithmetic, bit-vectors, purely applicative arrays, etc.
  - + recursive functions, algebraic data-types (Dafny), polymorphism and inductive predicates (Why)
- Alternative - use the logic of an existing general-purpose proof assistant (Coq, PVS, Isabelle, ACL2, etc.)
  - Provides a very rich higher-order logic specification language ☺
  - Well-developed libraries makes specifications easier ☺
  - Proof automation is difficult to achieve ☹

Deduction Rules : Hoare Triple

A way of binding together our specifications with the programming language

- *Hoare triple* provides a way of integrating pre, post-conditions with program fragments

$$\{P\}\ f(x_1, \cdots, x_n)\ \{Q\}$$

- Enables modular reasoning

- Correctness of the program will amount to correctness of the statement

$$\forall x, P(x) \implies Q(x, f(x))$$

Deduction Rules : Hoare Triple

A way of binding together our specifications with the programming language

- *Hoare triple* provides a way of integrating pre, post-conditions with program fragments

$$\{P\}\ f(x_1, \cdots, x_n)\ \{Q\}$$

- Enables modular reasoning
- Correctness of the program will amount to correctness of the statement

$$\forall x, P(x) \implies Q(x, f(x))$$

- One can derive the correctness of programs by using the Deduction rules of Hoare Triple, but intermediate assertions may not compose nicely

Deduction Rules : Hoare Triple

A way of binding together our specifications with the programming language

- *Hoare triple* provides a way of integrating pre, post-conditions with program fragments

$$\{P\}\ f(x_1, \cdots, x_n)\ \{Q\}$$

- Enables modular reasoning
- Correctness of the program will amount to correctness of the statement

$$\forall x, P(x) \implies Q(x, f(x))$$

- One can derive the correctness of programs by using the Deduction rules of Hoare Triple, but intermediate assertions may not compose nicely
- One alternative is to fill the program with known assertions and let the system figure out the rest of them

Weakest Preconditions & Verification Condition Generation

Hoare Triple $\{P\}\ s\ \{Q\}$

Weakest Precondition $wp(s, Q)$

$wp(s, Q)$) captures the requirements over the initial state such that execution of program fragment s will result in a final state satisfying Q

Weakest Preconditions & Verification Condition Generation

Hoare Triple $\{P\}\ s\ \{Q\}$

Weakest Precondition $wp(s, Q)$

$wp(s, Q)$) captures the requirements over the initial state such that execution of program fragment s will result in a final state satisfying Q

- Most VC generators compute *weakest preconditions*
  - Dijkstra's calculus of weakest preconditions

Weakest Preconditions & Verification Condition Generation

Hoare Triple $\{P\}\ s\ \{Q\}$

Weakest Precondition $wp(s, Q)$

$wp(s, Q)$) captures the requirements over the initial state such that execution of program fragment s will result in a final state satisfying Q

- Most VC generators compute *weakest preconditions*
  - Dijkstra's calculus of weakest preconditions
- The validity of Hoare Triple is now equivalent to

$$P \implies wp(s, Q)$$

- Programs are required to be free of aliasing

Weakest Preconditions & Verification Condition Generation

Hoare Triple $\{P\}\ s\ \{Q\}$

Weakest Precondition $wp(s, Q)$

$wp(s, Q)$) captures the requirements over the initial state such that execution of program fragment s will result in a final state satisfying Q

- Most VC generators compute *weakest preconditions*
  - Dijkstra's calculus of weakest preconditions
- The validity of Hoare Triple is now equivalent to

$$P \implies wp(s, Q)$$

- Programs are required to be free of aliasing
  - The semantics of the programming language is encoded as a set of types, symbols and axioms known as the *Memory Model* and used with a version of the program free of aliasing, performing operations on this memory model

Weakest Preconditions & Verification Condition Generation

Hoare Triple $\{P\}\ s\ \{Q\}$

Weakest Precondition $wp(s, Q)$

$wp(s, Q)$) captures the requirements over the initial state such that execution of program fragment s will result in a final state satisfying Q

- Most VC generators compute *weakest preconditions*
  - Dijkstra's calculus of weakest preconditions
- The validity of Hoare Triple is now equivalent to

$$P \implies wp(s, Q)$$

- Programs are required to be free of aliasing
  - The semantics of the programming language is encoded as a set of types, symbols and axioms known as the *Memory Model* and used with a version of the program free of aliasing, performing operations on this memory model
  - The common strategy is to come up with the memory model and use an intermediate language (Why, BoogiePL, etc.) for encoding the program

Deduction Rules

$$\frac{}{\{P\}\{P\}}$$

$$\frac{P \Rightarrow P' \qquad \{P'\}\mathsf{s}\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\mathsf{s}\{Q\}}$$

$$\frac{\{P\}\mathsf{s\_1}\{R\} \qquad \{R\}\mathsf{s\_2}\{Q\}}{\{P\}\mathsf{s\_1;s\_2}\{Q\}}$$

$$\frac{e \text{ evaluates without error}}{\{P[x \leftarrow e]\}\mathsf{x{=}e;}\{P\}}$$

$$\frac{\{P \wedge \mathsf{e}\}\mathsf{s\_1}\{Q\} \qquad \{P \wedge \mathsf{!e}\}\mathsf{s\_2}\{Q\}}{\{P\}\,\mathsf{if}\ \ (\mathsf{e})\ \ \mathsf{s\_1}\ \mathsf{else}\ \ \mathsf{s\_2}\{Q\}}$$

$$\frac{\{I \wedge \mathsf{e}\}\mathsf{s}\{I\}}{\{I\}\mathsf{while}\ \ (\mathsf{e})\ \ \mathsf{s}\{I \wedge \mathsf{!e}\}}$$

Deduction Rules - Examples

```
//@ assert P { x |-> e };
x = e;
//@ assert P;
```

```
//@ assert y+1 >0 && a[2*(y+1)] == 0;
x = y+1;
//@ assert (x > 0) && a[2*x] == 0;
```

Deduction Rules - Examples

```
//@ assert P && B;
 Q;
//@ assert S;
```

```
//@ assert P &&!B;
 R;
//@ assert S;
```

```
//@ assert P;
if (B) {
  Q;
} else {
  R;
//@ assert S;
```

Deduction Rules - Examples

- Find `P` which is preserved by each execution of the loop body

```
//@ assert P && B;

 S;

 //@ assert P;
```

Deduction Rules - Examples

- Find `P` which is preserved by each execution of the loop body

```
//@ assert P && B;

S;

//@ assert P;
```

```
//@ assert P;
while (B)
{
   S;
}
//@ assert !B && P
```

Deduction Rules - Examples

- Find `P` which is preserved by each execution of the loop body

```
//@ assert P && B;

 S;

 //@ assert P;
```

```
//@ assert P;
while (B)
{
   S;
}
//@ assert !B && P
```

P is the loop invariant

Rules for Programming Languages

- Transform C statements into one of these constructs

Rules for Programming Languages

- Transform C statements into one of these constructs
- **`switch`** ,**`case`** statements re-written as **`if`** –**`else`** form

Rules for Programming Languages

- Transform C statements into one of these constructs
- **`switch`** ,**`case`** statements re-written as **`if`** -**`else`** form
- **`for`** `{ P; Q; R }` re-written as `P;` **`while`** `(Q) { S; R }`

Rules for Programming Languages

- Transform C statements into one of these constructs
- **`switch`** , **`case`** statements re-written as **`if`** -**`else`** form
- **`for`** `{ P; Q; R }` re-written as `P;` **`while`** `(Q) { S; R }`
- What about other programming languages ?

Rules for Programming Languages

- Transform C statements into one of these constructs
- **`switch`** , **`case`** statements re-written as **`if`** –**`else`** form
- **`for`** `{ P; Q; R }` re-written as `P;` **`while`** `(Q) { S; R }`
- What about other programming languages ?

We develop an intermediate language for transform programming constructs into

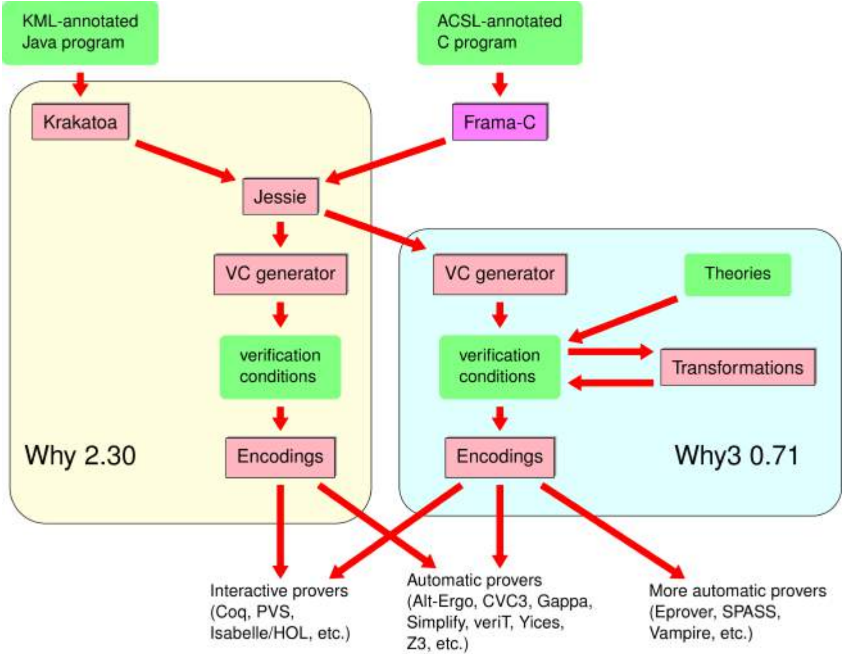
Rules for Programming Languages

- Transform C statements into one of these constructs
- **`switch`** , **`case`** statements re-written as **`if`** -**`else`** form
- **`for`** `{ P; Q; R }` re-written as `P;` **`while`** `(Q) { S; R }`
- What about other programming languages ?

We develop an intermediate language for transform programming constructs into

- Frama-C uses why (or jessie)
- Boogie, VCC, Dafny, Chalice (BoogiePL) KeY, etc., all use intermediate languages

Verification Condition Generation



ACSL

Frama-C + ACSL (**ACSL : A**NSI/ISO - **C S**pecification **L**anguage)

- Based on the notion of *Contracts*
- Users specify the properties of interest as ACSL annotations
- The analysis engines of Frama-C verify the properties
- The plugins (for different analyses) can be combined (they communicate info back and forth)
- Plugins are extensible
  - One can write a new plugin that uses the information computed by other plugins

ACSL : Ingredients

Function Contracts :

- What the function **requires** from its environment
- What the function **ensures** to its environment

ACSL : Ingredients

Function Contracts :

- What the function **requires** from its environment
- What the function **ensures** to its environment
- `abs()` function : `\result` denotes the return result of the function

ACSL : Ingredients

Function Contracts :

- What the function **requires** from its environment
- What the function **ensures** to its environment
- `abs()` function : `\result` denotes the return result of the function
  - `requires x >= -2147483647`

ACSL : Ingredients

Function Contracts :

- What the function **requires** from its environment
- What the function **ensures** to its environment
- `abs()` function : `\result` denotes the return result of the function
  - `requires x >= -2147483647`
  - `ensures \result >= 0`

ACSL : Ingredients

Function Contracts :

- What the function **requires** from its environment
- What the function **ensures** to its environment
- `abs()` function : `\result` denotes the return result of the function
  - `requires x >= -2147483647`
  - `ensures \result >= 0`
  - `ensures x < 0 ==> \result == -x`

ACSL : Ingredients

Function Contracts :

- What the function **requires** from its environment
- What the function **ensures** to its environment
- `abs()` function : `\result` denotes the return result of the function
  - `requires x >= -2147483647`
  - `ensures \result >= 0`
  - `ensures x < 0 ==> \result == -x`
  - `ensures x >= 0 ==> \result == x`

Annotated abs() function

```
/*@ 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;
}
```

More Annotations : `\valid`, `\old`, etc.

`swap()`:

- Swap the integer values in two memory locations
- Signature : `void swap (int *a , int *b)`

More Annotations : `\valid`, `\old`, etc.

`swap()`:

- Swap the integer values in two memory locations
- Signature : `void swap (int *a , int *b)`
- Properties :

More Annotations : `\valid`, `\old`, etc.

`swap()`:

- Swap the integer values in two memory locations
- Signature : `void swap (int *a , int *b)`
- Properties :
  - The pointers are valid

More Annotations : `\valid`, `\old`, etc.

`swap()`:

- Swap the integer values in two memory locations
- Signature : `void swap (int *a , int *b)`
- Properties :
  - The pointers are valid
  - When the function exits, the values are actually swapped

More Annotations : `\valid`, `\old`, etc.

`swap()`:

- Swap the integer values in two memory locations
- Signature : `void swap (int *a , int *b)`
- Properties :
  - The pointers are valid - **`\valid`**
  - When the function exits, the values are actually swapped - we'll use **`\old`**

Annotated `swap(..)` function

```c
/*@ 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

One more example...

- Input : array `a` of size `n`
- Output : array `a` with items at `n1` and `n2` exchanged
- Signature : `void array_swap (int n, int a[], int n1, int n2)`
- Properties :

One more example...

- Input : array `a` of size `n`
- Output : array `a` with items at `n1` and `n2` exchanged
- Signature : `void array_swap (int n, int a[], int n1, int n2)`
- Properties :
  - The indices `n1` and `n2` are within bounds

One more example...

- Input : array `a` of size `n`
- Output : array `a` with items at `n1` and `n2` exchanged
- Signature : `void array_swap (int n, int a[], int n1, int n2)`
- Properties :
  - The indices `n1` and `n2` are within bounds
  - The array itself is a valid memory location

One more example...

- Input : array `a` of size `n`
- Output : array `a` with items at `n1` and `n2` exchanged
- Signature : `void array_swap (int n, int a[], int n1, int n2)`
- Properties :
  - The indices `n1` and `n2` are within bounds
  - The array itself is a valid memory location
  - When the function returns, the values at `n1` and `n2` are actually swapped

`array_swap(..)` Property Annotations

- Input : array `a` of size `n`
- Output : array `a` with items at `n1` and `n2` exchanged
- Signature : `void array_swap (int n, int a[], int n1, int n2)`
- Properties :

`array_swap(..)` Property Annotations

- Input : array `a` of size `n`
- Output : array `a` with items at `n1` and `n2` exchanged
- Signature : `void array_swap (int n, int a[], int n1, int n2)`
- Properties :
  - `requires n >= 0 && 0 <= n1 <= n && 0 <= n2 <= n`

`array_swap(..)` Property Annotations

- Input : array `a` of size `n`
- Output : array `a` with items at `n1` and `n2` exchanged
- Signature : `void array_swap (int n, int a[], int n1, int n2)`
- Properties :
  - `requires n >= 0 && 0 <= n1 <= n && 0 <= n2 <= n`
  - `requires \valid(a+(0..n-1))`

`array_swap(..)` Property Annotations

- Input : array `a` of size `n`
- Output : array `a` with items at `n1` and `n2` exchanged
- Signature : `void array_swap (int n, int a[], int n1, int n2)`
- Properties :
  - `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]))`

Annotated `array_swap(..)` function

```
/*@ 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

Yet another example !

- Input : an array `a`, its size `n` and an integer `v`
- Output : if `v` is found in `a`, return the index of `v`, else return `-1`
- Signature : `int find(const int a[], int n, int v)`
- Properties :

Yet another example !

- Input : an array `a`, its size `n` and an integer `v`
- Output : if `v` is found in `a`, return the index of `v`, else return `-1`
- Signature : `int find(const int a[], int n, int v)`
- Properties :
  - Array bounds

Yet another example !

- Input : an array `a`, its size `n` and an integer `v`
- Output : if `v` is found in `a`, return the index of `v`, else return `-1`
- Signature : `int find(const int a[], int n, int v)`
- Properties :
  - Array bounds
  - Validity of array locations

Yet another example !

- Input : an array `a`, its size `n` and an integer `v`
- Output : if `v` is found in `a`, return the index of `v`, else return `-1`
- Signature : `int find(const int a[], int n, int v)`
- Properties :
  - Array bounds
  - Validity of array locations
  - Array `a[]` is not modified by `find(..)`

Yet another example !

- Input : an array `a`, its size `n` and an integer `v`
- Output : if `v` is found in `a`, return the index of `v`, else return `-1`
- Signature : `int find(const int a[], int n, int v)`
- Properties :
  - Array bounds
  - Validity of array locations
  - Array `a[]` is not modified by `find(..)`
  - When the function returns `-1`

Yet another example !

- Input : an array `a`, its size `n` and an integer `v`
- Output : if `v` is found in `a`, return the index of `v`, else return `-1`
- Signature : `int find(const int a[], int n, int v)`
- Properties :
  - Array bounds
  - Validity of array locations
  - Array `a[]` is not modified by `find(..)`
  - When the function returns `-1`
  - When the function returns a positive value

Yet another example !

- Input : an array `a`, its size `n` and an integer `v`
- Output : if `v` is found in `a`, return the index of `v`, else return `-1`
- Signature : `int find(const int a[], int n, int v)`
- Properties :
  - Array bounds
  - Validity of array locations
  - Array `a[]` is not modified by `find(..)`
  - When the function returns `-1`
  - When the function returns a positive value
  - Loop invariants

`find(..)` : Annotations

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

`find(..)` : Annotations

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

- the return value is either -1 or one of the indices of the array

`find(..)` : Annotations

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

- the return value is either -1 or one of the indices of the array
- `ensures -1 <= \result < n;`

`find(..)` : Annotations

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

- the return value is either -1 or one of the indices of the array
- `ensures -1 <= \result < n;`

When the function returns `-1` :

`find(..)` : Annotations

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

- the return value is either -1 or one of the indices of the array
- `ensures -1 <= \result < n;`

When the function returns `−1` :

- for all the locations `i` in the array (within the bounds), `a[i] != v`

`find(..)` : Annotations

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

- the return value is either -1 or one of the indices of the array
- `ensures -1 <= \result < n;`

When the function returns `-1` :

- for all the locations `i` in the array (within the bounds), `a[i] != v`
- ```
  ensures \result == -1 ==>
     (\forall integer i; 0 <= i < n ==> a[i]!= v);
  ```
  ```

`find(..)` : Annotations

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

- the return value is either -1 or one of the indices of the array
- `ensures -1 <= \result < n;`

When the function returns `-1` :

- for all the locations `i` in the array (within the bounds), `a[i] != v`
- ```
  ensures \result == -1 ==>
     (\forall integer i; 0 <= i < n ==> a[i]!= v);
  ```

When the function returns an index from the array :
  ```

`find(..)` : Annotations

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

- the return value is either -1 or one of the indices of the array
- `ensures -1 <= \result < n;`

When the function returns `-1` :

- for all the locations `i` in the array (within the bounds), `a[i] != v`
- ```
  ensures \result == -1 ==>
     (\forall integer i; 0 <= i < n ==> a[i]!= v);
  ```

When the function returns an index from the array :

- the value in that index `i` equals the value `v`
  ```

`find(..)` : Annotations

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

- the return value is either -1 or one of the indices of the array
- `ensures -1 <= \result < n;`

When the function returns `-1` :

- for all the locations `i` in the array (within the bounds), `a[i] != v`
- ```
  ensures \result == -1 ==>
     (\forall integer i; 0 <= i < n ==> a[i]!= v);
  ```

When the function returns an index from the array :

- the value in that index `i` equals the value `v`
- `ensures 0 <= \result < n ==> a[\result] == v;`
  ```

`find(..)` : Annotations

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

Loop invariants :

- `i` is within bounds `loop invariant 0 <= i <= n;`

`find(..)` : Annotations

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

Loop invariants :

- `i` is within bounds `loop invariant 0 <= i <= n;`
- Till our last iteration, we have not found `v`
  `loop invariant \forall integer k; 0 <= k < i ==> a[k] != v;`

`find(..)` : Annotations

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

Loop invariants :

- `i` is within bounds `loop invariant 0 <= i <= n;`
- Till our last iteration, we have not found `v`
  `loop invariant \forall integer k; 0 <= k < i ==> a[k] != v;`
- A loop variant - a measure that decreases every iteration so that the loop terminates
  `loop variant n-i;`

`find(..)` : Annotations

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

- the array `a[]` is not modified. Frama-C does not understand `const`

`find(..)` : Annotations

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

- the array `a[]` is not modified. Frama-C does not understand `const`
- `assigns \nothing`

`find(..)` : Annotations

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

- the array `a[]` is not modified. Frama-C does not understand `const`
- `assigns \nothing`
- the usual validity and bounds
- `requires n >= 0 && \valid(a+(0..n-1));`

demo

Other Annotations

- Named behaviours

```
behavior success:
   ensures \result >= 0 ==> a[\result] == v;
```

Other Annotations

- Named behaviours

```
behavior success:
  ensures \result >= 0 ==> a[\result] == v;
```

- `\allocable` and `\freeable`

Other Annotations

- Named behaviours

```
behavior success:
   ensures \result >= 0 ==> a[\result] == v;
```

- `\allocable` and `\freeable`
- `ghost` variables

Other Annotations

- Named behaviours

```
behavior success:
  ensures \result >= 0 ==> a[\result] == v;
```

- `\allocable` and `\freeable`
- `ghost` variables
- `invariant`, `variant`, `inductive` predicates,

Other Annotations

- Named behaviours

```
behavior success:
  ensures \result >= 0 ==> a[\result] == v;
```

- `\allocable` and `\freeable`
- `ghost` variables
- `invariant`, `variant`, `inductive` predicates,
- `lemma`, `axiomatic` definitions

Other Annotations

- Named behaviours

```
behavior success:
   ensures \result >= 0 ==> a[\result] == v;
```

- `\allocable` and `\freeable`
- `ghost` variables
- `invariant`, `variant`, `inductive` predicates,
- `lemma`, `axiomatic` definitions
- higher-order logic constructions using `\lambda`

Other Annotations

- Named behaviours

```
behavior success:
  ensures \result >= 0 ==> a[\result] == v;
```

- `\allocable` and `\freeable`
- `ghost` variables
- `invariant`, `variant`, `inductive` predicates,
- `lemma`, `axiomatic` definitions
- higher-order logic constructions using `\lambda`
- and many more $\cdots$

Interesting Frama-C Plugins

The 3 main plugins :

- Jessie
- WP
- Value Analysis

Interesting Frama-C Plugins

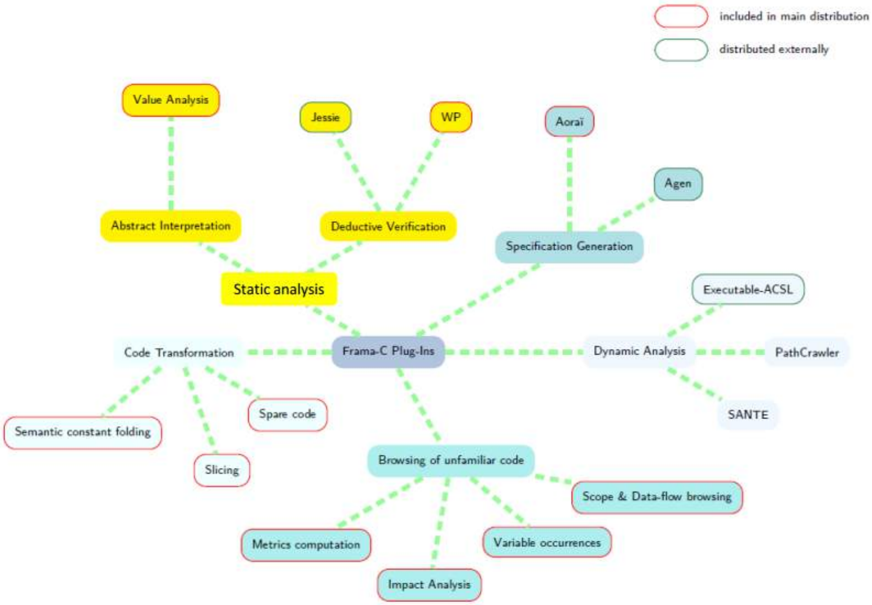
The 3 main plugins :

- Jessie
- WP
- Value Analysis

Other plugins :

- Impact Analysis
- Scope & Data-flow analysis
- Metrics Computation
- E-ACSL, RTE
- Aorai
- PathCrawler
- Mthread

Other Tools


included in main distribution
distributed externally
Value Analysis
Jessie
WP
Aoraï
Agen
Abstract Interpretation
Deductive Verification
Specification Generation
Static analysis
Executable-ACSL
Code Transformation
Frama-C Plug-Ins
Dynamic Analysis
PathCrawler
Spare code
SANTE
Semantic constant folding
Slicing
Browsing of unfamiliar code
Scope & Data-flow browsing
Metrics computation
Variable occurrences
Impact Analysis

Other Tools

- Why/Why3 is an excellent modelling/verification environment
  - It is also a part of Frama-C (VC generator)

Other Tools

- Why/Why3 is an excellent modelling/verification environment
  - It is also a part of Frama-C (VC generator)
- B-method is yet another good verification system
  - Event-B (Rodin) and Atelier-B are the tools supporting the B method
  - Widely used in safety-critical systems verification (eg. Paris Metro Line 14)

Other Tools

- Why/Why3 is an excellent modelling/verification environment
  - It is also a part of Frama-C (VC generator)
- B-method is yet another good verification system
  - Event-B (Rodin) and Atelier-B are the tools supporting the B method
  - Widely used in safety-critical systems verification (eg. Paris Metro Line 14)
- KeY tool is another deductive verification environment (for Java)

Other Tools

- Why/Why3 is an excellent modelling/verification environment
  - It is also a part of Frama-C (VC generator)
- B-method is yet another good verification system
  - Event-B (Rodin) and Atelier-B are the tools supporting the B method
  - Widely used in safety-critical systems verification (eg. Paris Metro Line 14)
- KeY tool is another deductive verification environment (for Java)
- Boogie, Dafny, Chalice (for concurrent programs), etc. from Microsoft Research use some form of deductive reasoning

Other Tools

- Why/Why3 is an excellent modelling/verification environment
  - It is also a part of Frama-C (VC generator)
- B-method is yet another good verification system
  - Event-B (Rodin) and Atelier-B are the tools supporting the B method
  - Widely used in safety-critical systems verification (eg. Paris Metro Line 14)
- KeY tool is another deductive verification environment (for Java)
- Boogie, Dafny, Chalice (for concurrent programs), etc. from Microsoft Research use some form of deductive reasoning
- COMPCERT - Certified C Compiler

Thank you

Thank you.