

1st Asian-Pacific Summer School on Formal Methods

Course 12: Deductive verification of C programs with Frama-C and Jessie

Virgile Prevosto

CEA List

August 30, 2009



Jessie Usage

Function Contracts

Advanced Specification

Example 1: Searching

Example 2: Sorting

(long no
[for i < 0
C1); if (0
tmp2 =
st of the

tmp2[0] = (t < 0 ? (n1 - t)) else if (tmp1[0] >= (t < 0 ? (n1 - t) : tmp2[0]) : (t < 0 ? (n1 - t) : 0); else tmp2[0] = tmp1[0]; /* Then the second pass looks like the first one: */ for (k = 0; k < 8; k++) tmp1[0][k] = mc2[0][k] * tmp2[k][0] /* The [i,j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*(MC1 + I) = 1 * tmp1[0][0] >= 1. */ Final rounding: tmp2[0][0] is now represented on 9 bits. *If (tmp1[0][0] < -256) m2[0][0] = -256; else if (tmp1[0][0] > 255) m2[0][0] = 255; else m2[0][0] = tmp1[0][0];



Jessie Usage

Function Contracts

Advanced Specification

Example 1: Searching

Example 2: Sorting

```
(long n)
{ for (i = 0; i < n; i++)
  C[i] = 0;
  tmp2 = 0;
  // ...
}
```

```
tmp2[i] = (i < n) ? tmp1[i] : 0; else if (tmp1[i] >= 0) { if (i < (n-1) - 1) else tmp2[i] = tmp1[i]; } // Then the second pass looks like the first one:
tmp1[0] = 0; k = 0; k++ tmp1[i] = mc2[i][k] * tmp2[k]; // The [i][k] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
i = 1; tmp1[0] >= 1; // Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] < -256) m2[0] = -256; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = tmp1[0];
// ...
}
```



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq)



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care



Usage

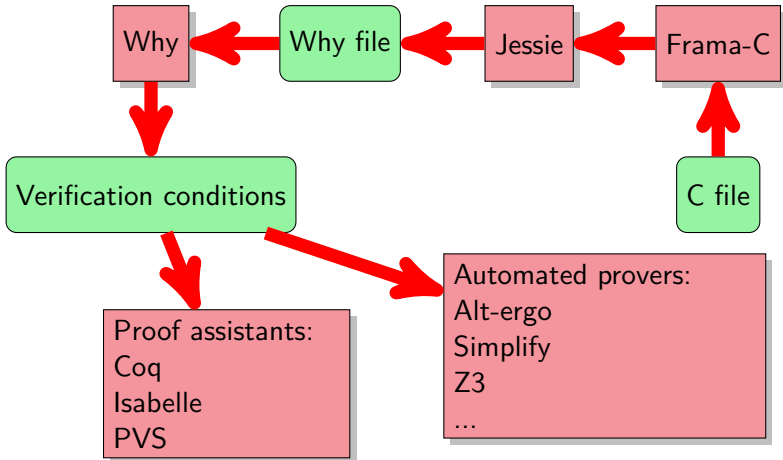
- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care



From Frama-C to Theorem Provers



Check safety of a function

- ▶ Pointer accesses
- ▶ Arithmetic overflow
- ▶ Division

```
unsigned int M;
```

```
void mean(unsigned int* p, unsigned int* q) {
    M = (*p + *q) / 2;
}
```





Safety of a program is important, but this is not sufficient: We also want it to do “the right thing”...

But in order for jessie to verify that, we need to explain it what “the right thing” is, and to explain it formally

This is the purpose of **ACSL**, ANSI/ISO C Specification Language.

- ▶ Behavioral specification language à la JML and Eiffel
- ▶ Function contracts
- ▶ Logic models
- ▶ Independent from any plug-in



Safety of a program is important, but this is not sufficient: We also want it to do “the right thing”...

But in order for jessie to verify that, we need to explain it what “the right thing” is, and to explain it formally

This is the purpose of ACSL, ANSI/ISO C Specification Language.

- ▶ Behavioral specification language à la JML and Eiffel
- ▶ Function contracts
- ▶ Logic models
- ▶ Independent from any plug-in



Safety of a program is important, but this is not sufficient: We also want it to do “the right thing”...

But in order for jessie to verify that, we need to explain it what “the right thing” is, and to explain it formally

This is the purpose of **ACSL**, ANSI/ISO C Specification Language.

- ▶ Behavioral specification language à la JML and Eiffel
- ▶ Function contracts
- ▶ Logic models
- ▶ Independent from any plug-in



Safety of a program is important, but this is not sufficient: We also want it to do “the right thing”...

But in order for jessie to verify that, we need to explain it what “the right thing” is, and to explain it formally

This is the purpose of **ACSL**, ANSI/ISO C Specification Language.

- ▶ Behavioral specification language à la JML and Eiffel
- ▶ Function contracts
- ▶ Logic models
- ▶ Independent from any plug-in



Safety of a program is important, but this is not sufficient: We also want it to do “the right thing”...

But in order for jessie to verify that, we need to explain it what “the right thing” is, and to explain it formally

This is the purpose of ACSL, ANSI/ISO C Specification Language.

- ▶ Behavioral specification language à la JML and Eiffel
- ▶ Function contracts
- ▶ Logic models
- ▶ Independent from any plug-in



Safety of a program is important, but this is not sufficient: We also want it to do “the right thing”...

But in order for jessie to verify that, we need to explain it what “the right thing” is, and to explain it formally

This is the purpose of **ACSL**, ANSI/ISO C Specification Language.

- ▶ Behavioral specification language à la JML and Eiffel
- ▶ Function contracts
- ▶ Logic models
- ▶ Independent from any plug-in



Safety of a program is important, but this is not sufficient: We also want it to do “the right thing”...

But in order for jessie to verify that, we need to explain it what “the right thing” is, and to explain it formally

This is the purpose of **ACSL**, ANSI/ISO C Specification Language.

- ▶ Behavioral specification language à la JML and Eiffel
- ▶ Function contracts
- ▶ Logic models
- ▶ Independent from any plug-in



Safety of a program is important, but this is not sufficient: We also want it to do “the right thing”...

But in order for jessie to verify that, we need to explain it what “the right thing” is, and to explain it formally

This is the purpose of **ACSL**, ANSI/ISO C Specification Language.

- ▶ Behavioral specification language à la JML and Eiffel
- ▶ Function contracts
- ▶ Logic models
- ▶ Independent from any plug-in



- Functional specification
- Pre-conditions (requires)
- Post-conditions (ensures)

Example

```
unsigned int M;
```

```
/*@
```

```
    requires \valid(p) ^ \valid(q);
```

```
    ensures M  $\equiv$  (*p + *q) / 2;
```

```
*/
```

```
void mean(unsigned int* p, unsigned int* q) {
```

```
    if (*p  $\geq$  *q) { M = (*p - *q) / 2 + *q; }
```

```
    else { M = (*q - *p) / 2 + *p; }
```

```
}
```



- Functional specification
- Pre-conditions (requires)
- Post-conditions (ensures)

Example

```
unsigned int M;
```

```
/*@
```

```
  requires \valid(p) ^ \valid(q);
```

```
  ensures M ≡ (*p + *q) / 2;
```

```
*/
```

```
void mean(unsigned int* p, unsigned int* q) {
```

```
  if (*p ≥ *q) { M = (*p - *q) / 2 + *q; }
```

```
  else { M = (*q - *p) / 2 + *p; }
```

```
}
```



- Functional specification
- Pre-conditions (requires)
- Post-conditions (ensures)

Example

```
unsigned int M;
```

```
/*@
```

```
  requires \valid(p) ^ \valid(q);
```

```
  ensures M ≡ (*p + *q) / 2;
```

```
*/
```

```
void mean(unsigned int* p, unsigned int* q) {
```

```
  if (*p ≥ *q) { M = (*p - *q) / 2 + *q; }
```

```
  else { M = (*q - *p) / 2 + *p; }
```

```
}
```



The specification:

```
/*  
    requires \valid(p) ∧ \valid(q);  
    ensures M ≡ (*p + *q) / 2;  
    assigns M;  
*/  
void mean(unsigned int* p, unsigned int* q);
```



Jessie Usage

Function Contracts

Advanced Specification

Example 1: Searching

Example 2: Sorting

```
(long n)
{ for (i = 0; i < n; i++)
  C1; if (i == n)
    tmp2 = ...
  // rest of the
```

```
tmp2[i] = (i <= n-1) ? tmp[i] : 0; else if (tmp[i] >= 0) { i <= n-1 ? tmp2[i] = (i <= (n-1) ? 0 : else tmp2[i] = tmp[i]; } /* Then the second pass looks like the first one:
tmp1[0] = 0; k = 0; k++ } tmp1[i] = mc2[i][k] * tmp2[k]; /* The [i][k] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*(MC1
i = 1; tmp1[i] >= 1; /* Final rounding: tmp2[i] is now represented on 9 bits. *if (tmp1[i] < -256) m2[i] = -256; else if (tmp1[i] > 255) m2[i] = 255; else m2[i] = tm
```



Informal spec

- Input: a **sorted** array and its length, an element to search.
- Output: index of the element or -1 if not found

Implementation

```
int find_array(int* arr, int length, int query) {
    int low = 0;
    int high = length - 1;
    while (low ≤ high) {
        int mean = low + (high - low) / 2;
        if (arr[mean] ≡ query) return mean;
        if (arr[mean] < query) low = mean + 1;
        else high = mean - 1;
    }
    return -1;
}
```



Informal specification

- ▶ Input: an array and its length
- ▶ Output: the array is sorted in ascending order

```
int index_min(int* a, int low, int high);
```

```
void swap(int* arr, int i, int j);
```

```
void min_sort(int* arr, int length) {
    for(int i = 0; i < length; i++) {
        int min = index_min(arr,i,length);
        swap(arr,i,min);
    }
}
```

(long no
for it
C13.11.01
tmp2 =
se of the

tmp2[0][i] = (i < 2 ? (int) -1 : tmp2[0][i-1]) >= (i < 2 ? (int) -1 : tmp2[0][i-1]) ? (i < 2 ? (int) -1 : tmp2[0][i-1]) : tmp2[0][i-1]; // Then the second pass looks like the first one. (MC1
tmp1[0][i] = 0; k = 0; k++ tmp1[0][i] = mc2[0][k] * tmp2[0][i] / 7. The [i,j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*(MC1
i = 1; tmp1[0][i] >= 1; // Final rounding: tmp2[0][i] is now represented on 9 bits. *if (tmp1[0][i] < -255) tmp2[0][i] = -255; else if (tmp1[0][i] > 255) tmp2[0][i] = 255; else tmp2[0][i] = tmp1[0][i];

