

# Frama-C Training Session

## Tips and Tricks in Software Verification

Virgile Prevosto

CEA List

October 22<sup>nd</sup>, 2010

long n  
for i < n  
C[i] = m  
tmp2 =  
of the

tmp2[i] = 0; if (i < (n-1)) also if (tmp1[i]) >= 1) << (n-1) - i; else tmp2[i] = tmp1[i]; /\* Then the second part takes like the first one: \*/  
tmp1[i] = 0; k = 5; k--> tmp1[i][k] += mc2[i][k] \* tmp2[k]; /\* The [i][j] coefficient of the matrix product MC2\*TMP2, that is: \*MC2\*(TMP1) = MC2\*(MC1\*M1) = MC2\*M1 \*MC1  
i = 1; tmp1[i][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] = tmp1[i];





## Verification Environment

### Initial Specification

### Intermediate Contracts

### Loop Invariants

### Auxiliary Annotations

```
(long n)
for (i = 0; i < n; i++)
  tmp2 = ...
// ...
// ...
```

```
tmp2[0] = 1; // (N-1) else if (tmp1[0]) >= 1; // (N-1) tmp2[0] = (1 << (N-1)) - 1; else tmp2[0] = tmp1[0]; /* Then the second part looks like the first one:
tmp1[0][0] = 0; k = 0; 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
l = 1; tmp1[0][l] >>= 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];
```





## Decide which tool to use

Jessie	Value analysis
<ul style="list-style-type: none"> <li>✓ Arbitrary functional properties</li> <li>✓ Modular</li> <li>✗ Write formal specification</li> <li>✗ Needs user interaction</li> </ul>	<ul style="list-style-type: none"> <li>✗ Properties that can be encoded in the abstraction used</li> <li>✗ Complete program</li> <li>✓ Just need the code</li> <li>✓ Automated</li> </ul>



Verification Environment

Initial Specification

Intermediate Contracts

Loop Invariants

Auxiliary Annotations

long ra  
for (i = 0;  
i < n; i++)  
tmp2 =  
of the

tmp2[i] = (i < (Nb1 - 1)) ? tmp1[i] : (i < (Nb1 - 1)) ? tmp2[i] : (i < (Nb1 - 1)) ? tmp1[i] : 1; /\* Then the second part looks like the first one: "Nb1 - 1" is replaced by "Nb1 - 2" and "Nb1 - 1" is replaced by "Nb1 - 1". \*/  
tmp1[i] = 0; k = 0; k++ tmp1[i] += mc2[i][k] \* tmp2[k]; /\* The [i][j] coefficient of the matrix product MC2\*TMP2, that is, \*MC2\*(TMP1) = MC2\*(MC1\*M1) = MC2\*M1\*MC1 - 1 = 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] = tmp1[i];



# From informal to formal specification

## Example

Function `sort` should take an array of `int` and its length and sort it in place. It does not return anything.

## High level vs low level specification

- ▶ ACSL is essentially a low-level specification language
- ▶ Needs more link with modelization languages
- ▶ A good goal for future research projects?



## Using predicate and logic functions

- ▶ Use them for notions that come up frequently in the development.
- ▶ Pay attention to axiomatics: nothing prevents you to write inconsistencies
  - ▶ Proof-reading
  - ▶ Quick check: write **lemma** that are obviously false and see if they can be proved.
  - ▶ Complete check: write (in Coq, Isabelle) a model of the axiomatics.

▶ Inductive definitions provide syntactic safeguards

▶ Pay attention to generality:

`sorted_whole(int* a, integer length) or`

`sorted_slice(int* a, integer first, integer last)`





# Contract of Entry Points

## Primary role

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

## What to put in it

- ▶ Functional properties
- ▶ Safety requirements (`\valid`)



Verification Environment

Initial Specification

Intermediate Contracts

Loop Invariants

Auxiliary Annotations

long ra  
for (i = 0;  
i < n; i++)  
tmp2 =  
of the

tmp2[i] = (i < (Nb1 - 1)) ? else { tmp1[i] >= (i < (Nb1 - 1)) ? tmp2[i] : (i < (Nb1 - 1) - 1) ? else { tmp2[i] = tmp1[i]; } /\* Then the second part looks like the first one: "else {  
tmp1[i][k] = 0; k = 0; k++} tmp1[i][j] += mc2[i][k] \* tmp2[k][j];" /\* The [i][j] coefficient of the matrix product MC2\*TMP2, that is, \*MC2\*(TMP1) = MC2\*(M1)\*M1) = MC2\*(M1)\*M1)  
i = 1; tmp1[i][i] >= 1; /\* Final rounding: tmp2[i][i] is now represented on 9 bits. \*if (tmp1[i][i] < -256) m2[i][i] = -256; else if (tmp1[i][i] > 255) m2[i][i] = 255; else m2[i][i] = tm



## Contracts for internal functions

- ▶ Specification is here only to help the whole proof
- ▶ **requires** can be over-specified (if called only in particular contexts)
- ▶ **ensures** can be under-specified (if result does not impact directly the properties of interest in main functions)
- ▶ Pay attention to **assigns**
- ▶ Specification can evolve according to needs of provers



Verification Environment

Initial Specification

Intermediate Contracts

Loop Invariants

Auxiliary Annotations

long n;  
for (i = 0; i < n; i++)  
 C1; if (i % 2 == 0)  
 tmp2 = i;  
 // ...  
 // ...  
 // ...

tmp2[0] = 0; for (k = 0; k < n; k++) tmp1[0][k] += mc2[0][k] \* tmp2[k]; /\* The [i][j] coefficient of the matrix product MC2\*TMP2, that is, \*MC2\*(TMP1) = MC2\*(M1)\*M1 = MC2\*(M1)\*M1 - 1. tmp1[0][i] >= 1; \*/ Final rounding: tmp2[0][i] is now represented on 9 bits: \*if (tmp1[0][i] < -256) m2[0][i] = -256; else if (tmp1[0][i] > 255) m2[0][i] = 255; else m2[0][i] = tmp1[0][i];



## Traversing Loops

- ▶ The only thing known on the state after the loop is what is in the invariants. Make them strong enough
- ▶ But not too strong (or it won't be possible to prove them inductively).
- ▶ Each assigned location should appear somewhere in an invariant
- ▶ Use **loop assigns** give a precise idea of what has been assigned so far.

```

long n;
for (i = 0; i < n; i++)
    tmp2 =
    ...

```

```

tmp2[0] = 1; // (n-1) else if (tmp1[0] <= 0) tmp2[0] = 1; else if (tmp1[0] > 0) tmp2[0] = tmp1[0] + 1; Then the second pass takes the first pass
tmp1[0] = 0; k = 5; k++; tmp1[0] = mc2[0][k] * tmp2[0][k]; // The [i][j] 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 3 bits: if (tmp1[0] <= 255) tmp2[0] = 255; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] =

```



Verification Environment

Initial Specification

Intermediate Contracts

Loop Invariants

Auxiliary Annotations

long ra  
for (i = 0;  
i < n; i++)  
tmp2 =  
... of the

tmp2[i] = (i < (n-1) ? tmp1[i] : 0) \* (i < (n-1) ? tmp2[i] : 1) \* (i < (n-1) ? 1 : tmp1[i]); /\* Then the second part looks like the first one: \*/  
tmp1[i] = 0; k = 0; k++ tmp1[i][k] += 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][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...



## Helping automated provers

- ▶ Should be introduced carefully, when it appears that provers definitely go in the wrong direction on a given property
- ▶ Completely tied to a particular verification task
- ▶ **assert**: Give a cut point on a particular program state
- ▶ **lemma**: Some general property that the provers can not discover by themselves.



```

long n;
for (i = 0; i < n; i++) {
  tmp1[i] = 0;
  tmp2[i] = 0;
  for (k = 0; k < n; k++) {
    tmp1[i][k] = 0;
    tmp2[i][k] = 0;
  }
  for (j = 0; j < n; j++) {
    tmp1[i][j] = 0;
    tmp2[i][j] = 0;
  }
  for (k = 0; k < n; k++) {
    tmp1[i][k] = m2[i][k];
    tmp2[i][k] = m2[i][k];
  }
  for (j = 0; j < n; j++) {
    tmp1[i][j] = m2[i][j];
    tmp2[i][j] = m2[i][j];
  }
  for (k = 0; k < n; k++) {
    tmp1[i][k] = m2[i][k];
    tmp2[i][k] = m2[i][k];
  }
}

```

# Introducing Proof Assistant

- ▶ When everything else fails
- ▶ Coq, Isabelle, PVS, are not that scary: we need only a tiny portion of the underlying theory.
- ▶ When used to them, helps understanding where the issue lies.
- ▶ Plans for an interactive theorem prover à la Caveat.

