Specification and Proof of C Programs using ACSL and Frama-C

Virgile Prevosto

CEA List

February 5, 2009
Introduction

Function Contracts

Verification activities

Advanced Specification

Advanced Verification Activities
Introduction

Function Contracts

Verification activities

Advanced Specification

Advanced Verification Activities
Frama-C in a nutshell

▶ Developed at CEA List and INRIA Saclay-Île de France
▶ Framework for multiple analyses of C
▶ a kernel providing basic functionalities, based on CIL
▶ a set of plug-ins performing various static analyses

Some existing plug-ins

▶ Value analysis
▶ “semantic browsing” (dependencies, slicing, impact)
▶ Deductive verification (Jessie plug-in)
▶ Security analysis
The ACSL Language

- [http://frama-c.cea.fr/acsl.html](http://frama-c.cea.fr/acsl.html)
- ANSI/ISO C Specification Language
- Formal specification language dedicated to C programs
- Based on function contracts and code assertions

Main objectives

- Syntax and concepts close to C
- Independence from a particular analysis
- Allow communication between analyses
The ACSL Language

- ANSI/ISO C Specification Language
- Formal specification language dedicated to C programs
- Based on function contracts and code assertions

Main objectives

- Syntax and concepts close to C
- Independence from a particular analysis
- Allow communication between analyses
A first contact with ACSL
▶ Presentation of ACSL to non-specialists
▶ Simple specification examples
▶ Covering all the constructions of the language

More on the Frama-C training session this spring
A first contact with ACSL

- Presentation of ACSL to non-specialists
- Simple specification examples
- Covering all the constructions of the language

More on the Frama-C training session this spring
Introduction

Function Contracts

Verification activities

Advanced Specification

Advanced Verification Activities
Behavior of a function

- Pre-conditions (requires)
- Post-conditions (ensures)
- Pointer validity (� תהן)
- Overflows

Example

```c
int M;
/*@ requires \valid(p) \land \valid(q);
 requires *p \geq 0 \land *q \geq 0;
 ensures M \equiv (*p + *q) / 2; */
void mean(int* p, int* q) {
  if (*p \geq *q) { M = (*p - *q) / 2 + *q; }
  else { M = (*q - *p) / 2 + *p; }
}
```
Behavior of a function

- Pre-conditions (requires)
- Post-conditions (ensures)
- Pointer validity (valid)
- Overflows

Example

```c
int M;
/*@ requires valid(p) ∧ valid(q);
   requires *p ≥ 0 ∧ *q ≥ 0;
   ensures M ≡ (*p + *q) / 2; */
void mean(int* p, int* q) {
    if (*p ≥ *q) { M = (*p - *q) / 2 + *q; }
    else { M = (*q - *p) / 2 + *p; }
}
```
Behavior of a function

- Pre-conditions (requires)
- Post-conditions (ensures)
- Pointer validity (valid)
- Overflows

Example

```c
int M;
/*@ requires valid(p) ∧ valid(q);
  requires *p ≥ 0 ∧ *q ≥ 0;
  ensures M ≡ (*p + *q) / 2; */
void mean(int* p, int* q) {
  if (*p ≥ *q) { M = (*p − *q) / 2 + *q; }
  else { M = (*q − *p) / 2 + *p; }
}
```
Behavior of a function

- Pre-conditions (requires)
- Post-conditions (ensures)
- Pointer validity (\valid)
- Overflows

Example

```c
int M;
/*@ requires \valid(p) \land \valid(q);
    requires *p \geq 0 \land *q \geq 0;
    ensures M \equiv (*p + *q) / 2; */
void mean(int* p, int* q) {
    if (*p \geq *q) { M = (*p - *q) / 2 + *q; } 
    else { M = (*q - *p) / 2 + *p; } }
```
Behavior of a function

- Pre-conditions (requires)
- Post-conditions (ensures)
- Pointer validity (\valid)
- Overflows

Example

```c
int M;
/*@ requires \valid(p) \land \valid(q);
  requires *p \geq 0 \land *q \geq 0;
  ensures M \equiv (*p + *q) / 2; */
void mean(int* p, int* q) {
  if (*p \geq *q) { M = (*p - *q) / 2 + *q; }
  else { M = (*q - *p) / 2 + *p; }
}
```
The specification:

```c
/*@ 
   requires \valid(p) \land \valid(q);
   requires *p \geq 0 \land *q \geq 0;
   ensures M \equiv (*p + *q) / 2;
*/

void mean(int* p, int* q);
```

An admissible implementation:
The specification:

```c
/*@ 
  requires \valid(p) \land \valid(q);
  requires *p \geq 0 \land *q \geq 0;
  ensures M \equiv (*p + *q) / 2;
*/

void mean(int* p, int* q);
```

An admissible implementation:

```c
void mean(int *p, int* q) {
  *p = *q = M = 0; }
```
The specification:

```c
/*@ 
requires valid(p) ∧ valid(q);
requires *p ≥ 0 ∧ *q ≥ 0;
ensures M ≡ (*p + *q) / 2;
ensures *p ≡ old(*p) ∧ *q ≡ old(*q);
*/
void mean(int* p, int* q);
```

An admissible implementation:
The specification:

```c
/*@ 
requires \valid(p) \land \valid(q);
requires *p \geq 0 \land *q \geq 0;
ensures M \equiv (*p + *q) / 2;
ensures *p \equiv \text{old}(*p) \land *q \equiv \text{old}(*q);
*/
void mean(int* p, int* q);
```

An admissible implementation:

```c
void mean(int *p, int* q) {
  if (*p \geq *q) ...
  else ...
  A = 0;
}
```
The specification:

```c
/*@ 
requires \valid(p) \land \valid(q);
requires *p \geq 0 \land *q \geq 0;
ensures M \equiv (*p + *q) / 2;
assigns M;
*/
void mean(int* p, int* q);
```

An admissible implementation:
The specification:

/*@ 
requires valid(p) ∧ valid(q);
requires *p ≥ 0 ∧ *q ≥ 0;
ensures M ≡ (*p + *q) / 2;
assigns M;
*/

void mean(int* p, int* q);

An admissible implementation:

void mean(int *p, int* q) {
    if (*p ≥ *q) { M = (*p − *q) / 2 + *q; } 
    else { M = (*q − *p) / 2 + *p; }
}
Termination

- Post condition true when the function exits normally.
- By default, a function always terminates.
- ... as long as its pre-condition holds.

```c
/*@ 
requires \valid(p) \land \valid(q);
ensures ... 
assigns M;
*/

void mean(int* p, int* q) {
    if (*p \geq *q) ...
}
```
Post condition true \textit{when the function exits normally.}

By default, a function always terminates.

... as long as its pre-condition holds.

/*@ 
  \textbf{requires valid}(p) \land \textbf{valid}(q);
  \textbf{ensures} ...
  \textbf{assigns} M;
*/

void mean(int* p, int* q) {
    while(1);
    if (*p ≥ *q) ...
}
Post condition true *when the function exits normally.*

By default, a function always terminates.

... as long as its pre-condition holds.

```c
/*@ 
  requires \valid(p) \land \valid(q);
  ensures ...
  assigns M;
  terminates \true;
*/
void mean(int* p, int* q) {
  if (*p ≥ *q) ...
}
```
Termination

- Post condition true *when the function exits normally*.
- By default, a function always terminates...
- ... as long as its pre-condition holds.

```c
/*@ 
requires \valid(p) \land \valid(q);
ensures ...
assigns M;
terminates \true;
*/
void mean(int* p, int* q) {
  if (p == NULL \lor q == NULL) while(1);
  if (*p >= *q) ...
}
```
Introduction

Function Contracts

Verification activities

Advanced Specification

Advanced Verification Activities
Jessie Plugin

- Hoare-logic based
- generates some proof obligations...
- ...that are discharged by automated provers
Hoare-logic based

generates some proof obligations...

...that are discharged by automated provers
Jessie Plugin

- Hoare-logic based
- generates some proof obligations...
- ...that are discharged by automated provers
- `frama-c -jessie-analysis -jessie-gui mean.c`
Lemmas

- Property that can be derived from definitions
- Proof obligation
- Development of a standard library of lemmas

```c
/*@ lemma div_def: \forall integer i; 0 \leq i - 2*(i/2) \leq 1;
lemma div_mod:
  \forall integer i, j;
  i + 2*j - 2*(((i + 2*j)/2)) \equiv i - 2*(i/2);
lemma div_le: \forall integer i; 0 \leq i \implies 0 \leq i/2 \leq i; */
```
Lemmas

- Property that can be derived from definitions
- Proof obligation
- Development of a standard library of lemmas

```c
/*@ lemma div_def: \forall integer i; 0 \leq i - 2*(i/2) \leq 1;
  lemma div_mod:
  \forall integer i,j;
  i+2*j - 2*((i+2*j)/2) \equiv i - 2*(i/2);
  lemma div_le: \forall integer i; 0 \leq i \implies 0 \leq i/2 \leq i;
*/
```
Property which holds at a given point

Can be used to guide theorem provers

```c
if (*p ≥ *q) {
    M = (*p − *q) / 2 + *q;
    //@ assert *p − *q + 2(*q) ≡ *p+*q;
}
```
Introduction

Function Contracts

Verification activities

Advanced Specification

Advanced Verification Activities
Inductive Predicates

- Naming properties
- Case definition
- Might be tied to a program point

typedef struct _list
{ int element; struct _list* next; } list;
/*@ inductive reachable{L}(list* root, list* node) {
case reachable_hd{L}:
 ∀ list* l1; reachable(l1,l1);
case reachable_next{L}:
 ∀ list* l1, *l2;
\valid(l1) ⇒ reachable(l1->next,l2) ⇒
 reachable(l1,l2);
}*/
Inductive Predicates

- Naming properties
- Case definition
- Might be tied to a program point

```c
typedef struct _list
    { int element; struct _list* next; } list;
/*@ inductive reachable{L}(list* root, list* node) {
    case reachable_hd{L}:
        ∀ list* l1; reachable(l1,l1);
    case reachable_next{L}:
        ∀ list* l1, *l2;
        \valid(l1) ⇒ reachable(l1->next,l2) ⇒
        reachable(l1,l2);
}*/
```
Inductive Predicates

- Naming properties
- Case definition
- Might be tied to a program point

```c
typedef struct _list
{
    int element;
    struct _list* next;
} list;

/*@ inductive reachable{L}(list* root, list* node) {
    case reachable_hd{L}:
        ∀ list* l1; reachable(l1,l1);
    case reachable_next{L}:
        ∀ list* l1, *l2;
        \valid(l1) ⇒ reachable(l1->next,l2) ⇒
        reachable(l1,l2);
}*/
```
Inductive Predicates

- Naming properties
- Case definition
- Might be tied to a program point

```c
typedef struct _list
{ int element; struct _list* next; } list;
/*@ inductive reachable\{L\}(list* root, list* node) {
    case reachable_hd\{L\}:
        ∀ list* l1; reachable(l1,l1);
    case reachable_next\{L\}:
        ∀ list* l1, *l2;
        \valid(l1) \implies reachable(l1->next,l2) \implies reachable(l1,l2);
}@*/
```
Define properties of a given type
Must hold when entering and exiting a function
Based on the static type of a variable

/*@ type invariant lists(list* l) =
∀ list* l1;
    reachable(l, l1) ∧ \valid(l1) ⇒
        l1→next ≡ \null ∨ \valid(l1→next);
*/
Define properties of a given type
Must hold when entering and exiting a function
Based on the static type of a variable

/*@ type invariant lists(list* l) =
\forall list* l1;
    reachable(l, l1) \land \text{valid}(l1) \Rightarrow
    l1\rightarrow next \equiv \text{null} \lor \text{valid}(l1\rightarrow next);
*/

/*@ requires lists(root);
ensures lists(root);*/
void f(list* root);
Type invariant

- Define properties of a given type
- Must hold when entering and exiting a function
- Based on the static type of a variable

```c
/*@ type invariant lists(list* l) = 
∀ list* l1;
    reachable(l, l1) ∧ \valid(l1) ⇒
    l1->next ≡ \null ∨ \valid(l1->next);
*/
```

```c
void f(struct _list* root);
```
Introduction

Function Contracts

Verification activities

Advanced Specification

Advanced Verification Activities
A complete example: Specification

Compute the max element of a non-empty finite list

/*@ 
requires ;
assigns ;
ensures ;
*/

int max_list(list* root);
Compute the max element of a non-empty finite list

/*@
requires \valid(root) \land finite_list(root);
assigns ;
ensures ;
*/

int max_list(list* root);
A complete example: Specification

Compute the max element of a non-empty finite list

/*@ 
requires \valid(root) \land finite_list(root);
assigns \nothing;
ensures ;
*/

int max_list(list* root);
Compute the max element of a non-empty finite list

```c
/*@ 
requires valid(root) ∧ finite_list(root);
assigns nothing;
ensures mem(\result,root);
*/

int max_list(list* root);
```
A complete example: Specification

Compute the max element of a non-empty finite list

/*@
requires \valid(root) \land finite_list(root);
assigns \nothing;
ensures mem(\result,root);
ensures \forall \text{int } n; \text{mem}(n,root) \implies \result \geq n;
*/

int max_list(list* root);
int max_list(list* root) {
    int max = root->element;
    while(root->next) {
        root = root->next;
        if (root->element > max) max = root->element;
    }
    return max;
}
Loop Invariant

- Inductive property
- Allows to specify one loop step

```c
/*@ loop invariant valid(root) ∧ 
   reachable(at(root,Pre),root) ∧ 
   mem(max,at(root,Pre)) ∧ 
   ∀ int n; 
   mem_sub(n, at(root,Pre),root) ⇒ max ≥ n; */

while(root->next) {
   ...
}
```

Proof?