

Towards an Executable ACSL

Virgile Prevosto

CEA List

July 13th, 2010

if (long ra
t for 0 <=
C1) if (m
tmp2 =
of the

tmp2[0] = 1 << (Nb1 - 1) else if (tmp1[0]) >= 1 << (Nb1 - 1) tmp2[0] = 1 << (Nb1 - 1) + abs(tmp2[0] - tmp1[0]); /* Then the second part takes the first one. */
tmp1[0] = 0; k = 0; k <= 1; tmp1[0][k] += 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
l = 1; tmp1[0][l] >= 1; /* Final rounding. tmp2[0][l] is now represented on 9 bits. */ if (tmp1[0][l] < -255) m2[0][l] = -255; else if (tmp1[0][l] > 255) m2[0][l] = 255; else m2[0][l] = tmp1[0][l];



Introduction

Context

A first ACSL specification

Basic expressions

Code Annotations

Function Contracts

Loops

Logic Definitions

long n;
for (i = 0; i < n; i++)
 C[i] = m;
tmp2 =
of the

tmp2[i] = (i < (n-1) ? m : tmp1[i]) >= (i < (n-1) ? tmp2[i] : (i < (n-1) ? 0 : tmp2[i] + tmp1[i])); /* Then the second part looks like the first one: */
tmp1[i+k] = 0; k = k-1; tmp1[i+k] += m2[i][k] * tmp2[i+k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
i = i-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];



Introduction

Context

A first ACSL specification

Basic expressions

Code Annotations

Function Contracts

Loops

Logic Definitions

```
long n;  
for (i = 0; i < n; i++)  
  C[i] = i * i;  
tmp2 =  
... of the
```

```
tmp2[i] = (i < (n-1) ? else if (tmp1[i]) >= (i < (n-1) ? tmp2[i] : (i < (n-1) ? else tmp2[i] = tmp1[i]); /* Then the second part looks like the first one: */  
tmp1[i][k] = 0; k = 0; k++) tmp1[i][k] += mc2[i][k] * tmp2[k][j]; /* 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][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
```



Frama-C in a nutshell

- ▶ <http://frama-c.com>
- ▶ 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 verification plug-ins

- ▶ Value analysis
- ▶ Jessie plug-in
- ▶ Wp
- ▶ Sante and link to the PathCrawler tool



Frama-C in a nutshell

- ▶ <http://frama-c.com>
- ▶ 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 verification plug-ins

- ▶ Value analysis
- ▶ Jessie plug-in
- ▶ Wp
- ▶ Sante and link to the PathCrawler tool



The ACSL Language

- ▶ <http://frama-c.com/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



long n;
for (i = 0; i < n; i++)
C[i] = 0;
tmp2 = ...
of the

$tmp2[i] = 0; i < (n-k); i++) tmp1[i] += C[i-k];$ Then the second part takes the first part.
 $tmp1[i] = 0; k < i < n; i++) tmp1[i] += C[i-k];$ The $[i]$ coefficient of the matrix product $MC2 * TMP2$, that is, $*MC2[i](TMP1) = MC2[i](MC1 * M1) = MC2 * M1 * MC1$
 $i = 1; tmp1[i] += C[i];$ Final rounding: $tmp2[i] = (tmp1[i] < 256 ? tmp1[i] : 256) / 256$ else if $(tmp1[i] > 255) / 256$ else $tmp2[i] = tmp1[i]$

The ACSL Language

- ▶ <http://frama-c.com/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



This presentation

A first contact with ACSL

- ▶ Presentation of ACSL
- ▶ Simple specification examples
- ▶ Covering most constructions of the language

Executable subset of ACSL

- ▶ Identifying constructions that are relevant to run-time assertion checking
- ▶ Discussing how they could be implemented/compiled in C.

```

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

```

```

tmp2[i] = (i < (n/2) ? tmp1[i] : tmp1[n-1-i]); // Then the second part takes the first part
tmp1[i] = 0; k = 5; 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 = 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

```



This presentation

A first contact with ACSL

- ▶ Presentation of ACSL
- ▶ Simple specification examples
- ▶ Covering most constructions of the language

Executable subset of ACSL

- ▶ Identifying constructions that are relevant to run-time assertion checking
- ▶ Discussing how they could be implemented/compiled in C.

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

```
tmp2[i] = (i < (n-1) ? tmp1[i] : 0); // First rounding: tmp2[i] is now represented on 3 bits.
tmp1[i] = 0; k = 5; k = k + 1; tmp1[i] = mc2[i][k] * tmp2[i]; // The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1.
i = i + 1; tmp1[i] >>= 1; // Final rounding: tmp2[i] is now represented on 3 bits. if (tmp1[i] < -255) m2[i] = -255; else if (tmp1[i] > 255) m2[i] = 255; else m2[i] = tmp1[i];
```



```

/*@
requires \valid(root) && finite_list(root);
assigns \nothing;
ensures mem(\result, root);
ensures
  \forall integer n; mem(n, root) ==>
    \result >= 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 Invariants

```

/*@ loop invariant \valid(root) &&
    reachable(\at(root,Pre),root) &&
    mem(max, \at(root,Pre)) &&
    \forall int n;
        mem_sub(n, \at(root,Pre),root) ==>
            max >= n;
*/
while(root->next) {
    ...
}

```



Introduction

Context

A first ACSL specification

Basic expressions

Code Annotations

Function Contracts

Loops

Logic Definitions

```
long n;  
for (i = 0; i < n; i++)  
  tmp2[i] = 1;  
// End of the
```

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



Description

- ▶ All pure C expressions are valid ACSL expressions
- ▶ Integer operations are performed in \mathbb{Z} , real in \mathbb{R} .

Executable Setting

- ▶ Translation is trivial...
- ▶ ... Modulo the representation of mathematical datatypes
 - ▶ For integers, we could use some BigNum library (GMP)
 - ▶ For reals: ?



Description

- ▶ Propositional connectors
- ▶ Quantifiers

Executable Setting

- ▶ Propositional level can be mapped directly to booleans
- ▶ Possible to support some limited form of quantification over a finite set, e.g.

```
\forall integer x; 0 <= x <= 10 ==> a[x] <= a[10]
```



Built-in constructions

Description

- ▶ Predicates and logic functions about pointers (`\valid`, `\baseaddr`,...).
- ▶ Evaluation of an expression at a given program point (`\at`, `\old`).

Executable Setting

- ▶ Would require a dedicated memory management (allocation table).
- ▶ Store needed results in intermediate variables as required.

```
long n;  
for (i = 0; i < n; i++)  
  C[i] = 0;  
tmp2 = ...  
... of the
```

```
tmp2[0] = 1; // (n-1) * 0 + tmp2[0] = 1 // (n-1) * 0 + tmp2[0] = 1 // (n-1) * 0 + tmp2[0] = 1 // (n-1) * 0 + tmp2[0] = 1  
Then the second pass: for (i = 0; i < n; i++)  
  tmp1[i] = 0; k = 5; k = i; tmp1[i] = m2[0][k] * tmp2[k]; // The [i][k] coefficient of the matrix product M2 * TMP2, that is, *M2[i][TMP1] = M2[i][M1] * M1  
i = 1; tmp1[1] >>= 2; // First operand tmp2[1] is now represented on 3 bits: 2^1 * tmp1[1] <= 2^0 * 2^1 = 2^0 else if (tmp1[1] < 2^0) tmp2[1] = 2^0 * tmp1[1] + 2^0 * tmp2[1] = 2^0 * (tmp1[1] + tmp2[1])
```



Introduction

Context

A first ACSL specification

Basic expressions

Code Annotations

Function Contracts

Loops

Logic Definitions

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

```
tmp2[i] = (i < (n-1) ? else { tmp1[i] } >= (i < (n-1) ? tmp2[i] : (i < (n-1) ? else { tmp2[i] : tmp1[i] }); /* Then the second part looks like the first one: */  
tmp1[i][k] = 0; k = 0; k++ } tmp1[i][k] += mc2[i][k] * tmp2[k][j]; /* 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][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
```



Pre/Post conditions

Description

- ▶ **requires** supposed to hold when entering the function.
- ▶ **ensures** supposed to hold at function exit.
- ▶ Notion of **behavior** with **assumes** clause.

Executable Setting

- ▶ Use assertions (when the underlying predicate can be evaluated at run time).



Assigns clause

Description

- ▶ **assigns** describes the locations that may be modified during one run of the function.
- ▶ In other words, every location not in an **assigns** clause must retain the same value between pre and post state.

Executable Setting

- ▶ Using allocation table and type information, might be possible to check the exact property.
- ▶ Other possibility: Decorate each assignment with an assertion that the lval is in **assigns** (overapproximation but might be easier to check).



Inductive invariant

Description

- ▶ Must be true before entering the loop.
- ▶ If true when entering, must stay true at the end of a loop step.

Executable Setting

- ▶ Use plain assertions (property a bit weaker).

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

tmp2[i] = (i < (n-1)) ? tmp1[i] : (i < (n-1)) ? tmp2[i] : tmp1[i];
Then the second part takes for the first part
tmp1[i] = 0; k = 8; 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 3 bits: if (tmp1[i] < 256) tmp2[i] = 255; else if (tmp1[i] > 255) tmp2[i] = 256; else tmp2[i] = tmp1[i];



Description

- ▶ **loop variant**: integer expression which:
 - ▶ is always non negative;
 - ▶ is strictly decreasing between two loop steps

Executable Setting

- ▶ Use assertions
- ▶ Store the preceding value of the variant

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

tmp2[i] = (i < (n-1) ? tmp2[i+1] : 0);
tmp1[i] = 0; k = 0; k++ tmp1[i][k] += m2[i][k] * tmp2[k];
Final rounding: tmp2[i] is now represented on 9 bits: if (tmp1[i] < -256) tmp1[i] = -256; else if (tmp1[i] > 255) tmp1[i] = 255; else tmp1[i] = tmp1[i];



Introduction

Context

A first ACSL specification

Basic expressions

Code Annotations

Function Contracts

Loops

Logic Definitions

```
if (long n)
  for (i = 0; i < n; i++)
    tmp2 =
      ...
  end of the
```

```
tmp2[0] = (i < (n-1) ? tmp1[0] : 0) + (i < (n-1) ? tmp2[0] : 0); /* Then the second part looks like the first one:
tmp1[0] = 0; k = 0; k++ tmp1[0] += mc2[0][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[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];
```



Direct definition

```

/*@ predicate
    hd_max{L}(list* root, integer m) =
        \valid(root) && root->element <= m;
*/

```

Executable setting

- ▶ Inline the definition.
- ▶ Use C function.



Axiomatic definition

```

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

Executable setting

- ▶ Not translatable in the general case
- ▶ Might identify some patterns in axioms?

