CONC2SEQ: A Frama-C Plugin for the Verification of Parallel Compositions of C Programs

Allan Blanchard[3,2] Nikolai Kosmatov[2] Frédéric Loulergue[1,3]

[1]School of Informatics, Computing, and Cyber Systems

[2]CEA LIST

[3]Laboratoire d'Informatique Fondamentale d'Orléans

SCAM, October 2, 2016

Our Goal

Deductive verification of concurrent C programs using Frama-C

Our Goal

Deductive verification of concurrent C programs using Frama-C

Our Assumptions

Concurrent program:

- Parallel composition C functions
- Sequential C + atomic blocks
- Sequentially consistent memory model

Our Goal

Deductive verification of concurrent C programs using Frama-C

Our Assumptions

Concurrent program:

- Parallel composition C functions
- Sequential C + atomic blocks
- Sequentially consistent memory model

Our Proposal

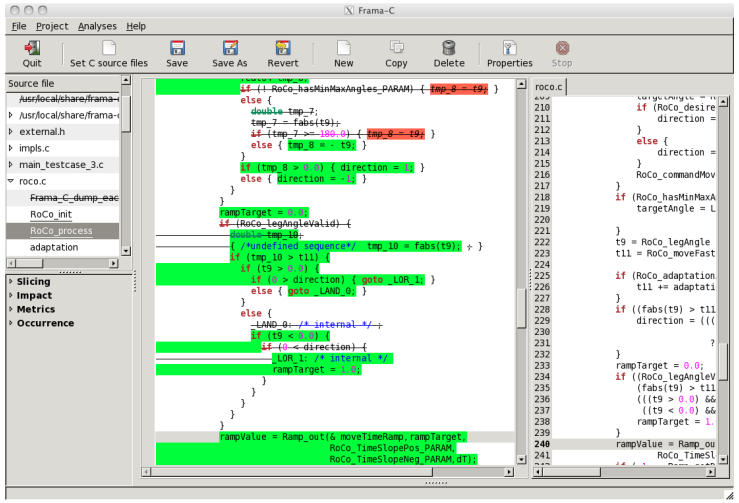
- A Plugin: CONC2SEQ
- Principle: program and annotations transformation
- Transform a concurrent program into an equivalent sequential one to be verified

A Framework for the Analysis of C Programs

- Analysis of C programs
- ACSL: ANSI/ISO C Specification Langage
- Static analyses:
  - Value analysis
  - Deductive verification
- Dynamic analysis:
  - E-ACSL: runtime assertion checking
- Developed by CEA LIST & Inria (2005-)
- Open source and released under LGPL

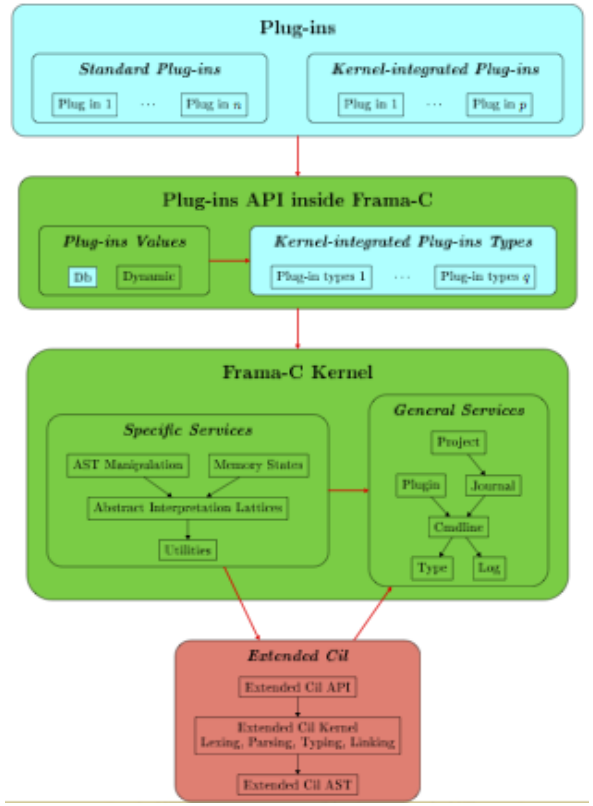
frama C

Software Analyzers



Plugins

- Extensible platform through plugins
- 1 plugin = 1 analyser
- Collaboration between analysers
- Fully written in OCaml (> 100 kloc)
- Minimal number of dependencies (including fork of CIL)
- API for developing analysers



CONC2SEQ: An Example

Single Writer / Multiple Readers

- Shared variable `d`
- Shared variable `acc` to store the access status:

acc	status
-1	write access
0	no access
> 0	number of readers

- Two functions in the API:
  - Read: `int read (int * l)`
  - Write: `int write(int value)`

CONC2SEQ: An Example

Single Writer / Multiple Readers

- Shared variable `d`
- Shared variable `acc` to store the access status:

acc	status
-1	write access
0	no access
> 0	number of readers

- Two functions in the API:
  - Read: `int read (int * l)`
  - Write: `int write(int value)`

Properties to Prove

- Several readers can execute concurrently
- Only one reader allowed to run
- Mutual exclusion betwen reader and writer

CONC2SEQ: An Example

```
int write(int value){
  int r, exp = 0;

  r=cmp_xchg(int,&acc,&exp,-1);

  if (!r) return 0;

  d = value;

  fetch_and_add(int,&acc,1);

  return 1;
}
```

```
int read(int* l){
  int r, a = acc;
  if (a≥0) {

    r=cmp_xchg(int,&acc,&a,a+1);

  } else return 0;
  if (!r) return 0;
  *l = d;

  fetch_and_sub(int,&acc,1);

  return 1;
}
```

CONC2SEQ: An Example

```
int write(ir
  int r, exp

  r=cmp_xchg                                          acc,&a,a+1);

  if (!r) re

  d = value;

  fetch_and_a                                         ,&acc,1);

  return 1;
}
```

r=cmp_xchg(int, ptr, old, new);

```
if (*ptr==*old){
  *ptr=new;
  r=1;
} else {
  *old=*ptr;
  r=0;
}
```

```
  return 1;
}
```

CONC2SEQ: An Example

```
int write(int value){
  int r, exp = 0;
  ATOMIC (
   r=cmp_xchg(int,&acc,&exp,-1);
  );
  if (!r) return 0;

  d = value;
  ATOMIC (
   fetch_and_add(int,&acc,1);
  );
  return 1;
}
```

```
int read(int* l){
  int r, a = acc;
  if (a≥0) {
    ATOMIC (
     r=cmp_xchg(int,&acc,&a,a+1);
    );
  } else return 0;
  if (!r) return 0;
  *l = d;
  ATOMIC (
   fetch_and_sub(int,&acc,1);
  );
  return 1;
}
```

CONC2SEQ: An Example

```
//@ ghost int rd __attribute__ ((thread_local));
//@ ghost int rw __attribute__ ((thread_local));

int write(int value){
  int r, exp = 0;
  ATOMIC (
   r=cmp_xchg(int,&acc,&exp,-1);
   /*@ ghost wr = (r==1); */ );
  if (!r) return 0;

  d = value;
  ATOMIC (
   fetch_and_add(int,&acc,1);
   /*@ ghost wr = 0; */ );
  return 1;
}
```

```
int read(int* l){
  int r, a = acc;
  if (a≥0) {
    ATOMIC (
     r=cmp_xchg(int,&acc,&a,a+1);
     /*@ ghost rd = (r == 1) ;*/ );
  } else return 0;
  if (!r) return 0;
  *l = d;
  ATOMIC (
   fetch_and_sub(int,&acc,1);
   /*@ ghost rd = 0;*/ );
  return 1;
}
```

CONC2SEQ: An Example

```
int write(int value){
  int r, exp = 0;
  ATOMIC (
   r=cmp_xchg(int,&acc,&exp,-1);
   /*@ ghost wr = (r==1); */ );
  if (!r) return 0;

  d = value;
  ATOMIC (
   fetch_and_add(int,&acc,1);
   /*@ ghost wr = 0; */ );
  return 1;
}
```

```
/*@ requires \valid(l)
          /\ \separated(l,&acc,&d); */
int read(int* l){
  int r, a = acc;
  if (a≥0) {
    ATOMIC (
     r=cmp_xchg(int,&acc,&a,a+1);
     /*@ ghost rd = (r == 1) ;*/ );
  } else return 0;
  if (!r) return 0;
  *l = d;
  ATOMIC (
   fetch_and_sub(int,&acc,1);
   /*@ ghost rd = 0;*/ );
  return 1;
}
```

CONC2SEQ: An Example

Properties

```
/*@ logic ℤ sum(ℤ a, ℤ b) = a+b ; */

/*@ predicate inv =
      (acc ≥ −1) ∧ 0≤rd≤1 ∧ 0≤wr≤1
    ∧ (acc == −1 ⟺ (1 == th_redux(sum, wr, 0))
                  ∧ (0 == th_redux(sum, rd, 0)))
    ∧ (acc ≥ 0 ⟺ (acc == th_redux(sum, rd, 0))
                  ∧ (0 == th_redux(sum, wr, 0)));

    global invariant sw_mr: inv; */
```

CONC2SEQ: Design Principles

Threads

`MAX_THREADS`: logic value assumed > 0

Memory

- Global variables: unchanged
- Function parameter, local variable, thread local: global array indexed by thread identifier
- `pct`: global array (thread program counter)

+ axioms stating all is OK (valid addresses, separation)

CONC2SEQ: Design Principles

Statements

- 1 atomic statement or 1 atomic block = 1 function
- Basically:
  - same operations but memory accesses
  - update of the thread program counter

Simulation Loop

- Random selection of a thread
- Call of a “function statement”

CONC2SEQ: Design Principles

Example

```
/*  Transformation of statement:
          d = value;
    of function write. */

/*@ requires valid_th(th) ∧ *(pct+th) == 22 ;
    requires simulation ∧  inv ;

    ensures *(pct+th) == 24;
    ensures  simulation ∧  inv ; */

void write_Instr_22(unsigned th){
  d = *(write_value + th);
  *(pct + th) = 24;
  return;
}
```

CONC2SEQ: Design Principles

Interleaving

```
/*@ requires simulation ∧ inv ; */
void interleave(void) {
  unsigned int th;
  th = some_thread();
  /*@ loop invariant simulation ∧ inv ; */
  while (1) {
    th = some_thread();
    switch (*(pct + th)) {
      case 0   : choose_call(th);    break;
      case -15 : init_write(th);     break;
      case -30 : init_read(th);      break;
      case 22  : write_Instr_22(th); break;
      // ... similar cases for other atomic steps
    }
  }
}
```

Conclusions and Future Work

Conclusions

- CONC2SEQ a plugin for Frama-C
- Verification by transformation to a simulating sequential program

Ongoing and Future Work

- Formal verification of the transformation in Coq
- Additional features for CONC2SEQ
- Coq library to handle proof obligations not proved by SMT solvers