Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs (SCAM 2016)

AFADL 2017

Allan Blanchard, Frédéric Loulergue, Nikolai Kosmatov, Matthieu Lemerre

June 15th, 2017

Table of Contents

Dedicated Analysis

Most concurrent program analyzers are dedicated to this task

- they implement a specific analysis
- they are often hard to design

Sequential Code Analyzers

Sequential code analyzers work well

- How can we bring them to concurrent code analysis?
- Especially when we have many of them

The Frama-C code analysis platform (`frama-c.com`)

Software Analyzers

- Deductive verification (WP)
- Abstract Interpretation (Eva)
- Runtime assertion checking (E-ACSL)
- ...

Simulating Code: Motivation

Idea 1: Intrinsically concurrent analysis tools

- better integration
- but hard to develop

Idea 2: Simulate concurrent programs by sequential ones

- sequential analyzers will be able to treat it

Overview I

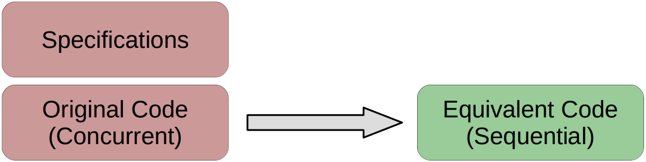
Specifications

Original Code (Concurrent)

Requirements

- Equivalence of code must be proved
- Do not alter specification meaning
- Added specifications must always be automatically proved

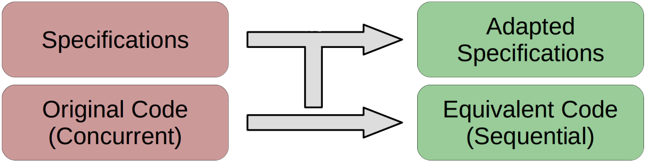
Overview I



Requirements

- Equivalence of code must be proved
- Do not alter specification meaning
- Added specifications must always be automatically proved

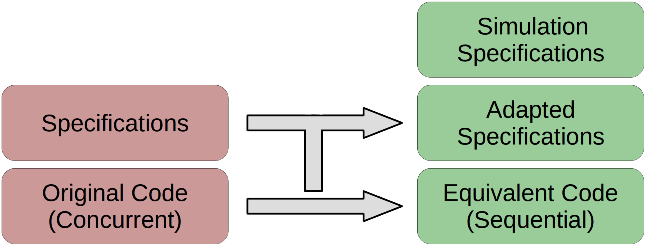
Overview I



Requirements

- Equivalence of code must be proved
- Do not alter specification meaning
- Added specifications must always be automatically proved

Overview I



Requirements

- Equivalence of code must be proved
- Do not alter specification meaning
- Added specifications must always be automatically proved

Overview II

Code transformation

- Each local variable becomes a simulating array
- Each instruction becomes a function
- All functions are interleaved to simulate concurrency

We suppose an interleaving semantics $\Rightarrow$ SC memory model

Specifications transformation

- Invariants are simulating functions pre/post conditions
- Each variable is replaced by its simulation counterpart

Conc2Seq – Features

Conc2Seq role

- Perform code transformation
- Adapt specifications

Supported

- Most C instructions
- Thread local variables
- Atomic operations (`stdatomic.h`)
- Atomic blocks of code
- Global invariants

Code transformation: variables

Original Code

```
int global;
int th_v thread_local;

void foo(){
  int v;
}
```

Generated Code

```
int* pct;
int  global;
int* tl_th_v;
int* foo_v;

/*@ axiomatic Validity_of_sim_vars {
predicate simulation{L} reads <sim ptrs>;

axiom all_simulations_separated{L}:
simulation ==>
\separated( <memory blocks/globals> );
```

simulating variables separation

```
axiom pct_is_valid{L}:
simulation ==>
( \forall integer j; valid_th(j) ==>
    \valid(\at(pct,L)+j));
//...
} */
```

simulating variables validity

Code transformation: atomic instructions

Original Code

```
void foo(){
  int v;
  th_v = atomic_load(&global);


  /*@ atomic \true; */{
    v = 42;
    global += v;
  }
}
```

Generated Code

```
void foo_Call_1(uint th){
  tl_th_v[th] = atomic_load(&global);
  pct[th] = 2;
}

void foo_Atomic_2(uint th){
  foo_v[th] = 42;
  global += foo_v[th];
  pct[th] = 3;
}
```

Code transformation: interleaving loop

Generated Code

```
void interleave(){
  unsigned int th = some_thread();
  /*@ loop invariant: translated_global_invariant ;
      loop invariant: simulation_global_invariant ; */
  while (1) {
    th = some_thread();
    switch (*(pc + th)) {
    case -1: init_formals_foo(th); break;
    case  0: choose_call(th);      break;
    case  1: foo_Call_1(th);       break;
    case  2: foo_Atomic_2(th);     break;
    case  3: foo_Return_3(th);     break;
    }
  }
}
```

Specification Transformation

Global invariants

- set as pre and post-condition of each simulating function
- universal quantification on thread identifiers when needed

(Original) function contracts

- preconditions are used to specify call initialization
- postconditions are verified in return simulation

Simulation specification

- invariant about the program counter

Conc2Seq – ACSL fold

New ACSL built-in to talk about threads

A logic fold operation on the value of a variable for all threads

- Generate an axiomatic definition for each usage ...
- ... according to provided types and logic function.

Idea

```
thread_reduction(func, v, init)
                ~
func(sim_v[0], func(..., func(sim_v[NTH], init)))
```

Let's Sum Up

Concurrent program analysis by sequential code analyzers

- based on a code transformation method
- simulation of a concurrent program by a sequential one
- implemented in the CONC2SEQ plugin of FRAMA-C

We prove that the simulation is sound if the considered program

- is sequentially consistent
- does not contain recursion
- does not allocate memory dynamically

Ongoing & Future Work

About the FRAMA-C plugin itself:

- add function call simulation to Conc2Seq
- add a SP calculus for local variables
- add new specification primitives for concurrent behaviors
- experiment on more case studies

The proof is currently a pen & paper proof

- mechanized proof using Coq

Ongoing & Future Work

About the FRAMA-C plugin itself:

- add function call simulation to Conc2Seq
- add a SP calculus for local variables
- add new specification primitives for concurrent behaviors
- experiment on more case studies

The proof is currently a pen & paper proof

- mechanized proof using Coq

Thank you ! Questions ?

Université d'Orléans
Laboratoire d'Informatique Fondamentale d'Orléans — Bâtiment IIIA
Rue Léonard de Vinci
F-45067 ORLÉANS
http://www.univ-orleans.fr/lifo/