Centre Val de Loire UNIVERSITÉ Spirit of the Loire Valley

UNIVERSITE D'ORLEANS

From Concurrent Programs to Simulating Sequential Programs: Correctness of a Transformation

VPT 2017

Allan Blanchard, Frédéric Loulergue, Nikolai Kosmatov

April 29th, 2017

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS LIFO

NORTHERN ARIZONA UNIVERSITY

list cea tech

Table of Contents

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`)

- 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

A Simple Imperative Language

$$
\begin{array}{lcll}
proc & ::= & m(\overline{x})c & m \in Name \\
instr & ::= & x := e & \text{local assignment} \\
 & | & x[y] := e & \text{writing to the heap} \\
 & | & x := y[e] & \text{reading from the heap} \\
 & | & \textbf{while } e \textbf{ do } c & \\
 & | & \textbf{if } e \textbf{ then } c \textbf{ else } c & \\
 & | & m(\overline{e}) & \text{procedure call} \\
 & | & \textbf{atomic}(c) & \text{atomic block} \\
\mathcal{C} \ni c & ::= & \{\} \mid instr; c &
\end{array}
$$

$$
memory \quad ::= \quad [(l_1, size_{l_1}); \ldots; (l_m, size_{l_m})]
$$

$$
\begin{array}{lcl}
prog_{seq} & ::= & \overline{proc}\ \ memory \\
prog_{par} & ::= & \overline{proc}\ \ memory\ \ mains \quad (\text{where } mains : \mathbb{T} \to Name)
\end{array}
$$

A Simple Imperative Language with Concurrency

$$
\begin{array}{lcll}
\mathit{proc} & ::= & m(\overline{x})c & m \in \mathit{Name} \\
\mathit{instr} & ::= & x := e & \text{local assignment} \\
 & \mid & x[y] := e & \text{writing to the heap} \\
 & \mid & x := y[e] & \text{reading from the heap} \\
 & \mid & \textbf{while } e \textbf{ do } c & \\
 & \mid & \textbf{if } e \textbf{ then } c \textbf{ else } c & \\
 & \mid & m(\overline{e}) & \text{procedure call} \\
 & \mid & \textbf{atomic}(c) & \text{atomic block} \\
\mathcal{C} \ni c & ::= & \{\} \mid \mathit{instr}; c &
\end{array}
$$

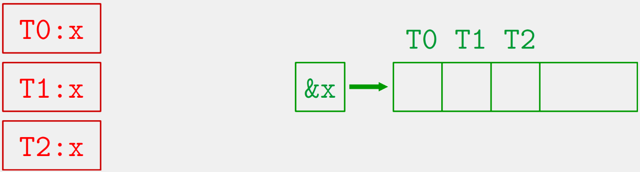
$$
\mathit{memory} \quad ::= \quad [(l_1, \mathit{size}_{l_1}); \ldots; (l_m, \mathit{size}_{l_m})]
$$

$$
\begin{array}{lcl}
\mathit{prog}_{\mathit{seq}} & ::= & \overline{\mathit{proc}} \ \ \mathit{memory} \\
\mathit{prog}_{\mathit{par}} & ::= & \overline{\mathit{proc}} \ \ \mathit{memory} \ \ \mathit{mains} \quad (\text{where } \mathit{mains} : \mathbb{T} \to \mathit{Name})
\end{array}
$$

Overview of the Transformation I

Variables

- Original heap is kept
- Each local variable `x` is simulated by a heap location `&x`



Assumption

- Static memory allocation

Overview of the Transformation II

Statements

- Maintain a program counter (`pct`) for each thread
- Each statement is simulated by a procedure that
  - Recieves in parameter the thread (`tid`) to execute
  - Executes the same action using simulating variables
  - Updates the program counter

Overview of the Transformation III

Procedure calls and returns

- For each procedure p, we add a heap location $\texttt{from}(p)$. It records the program point to return to from p.
- Simulating a call of p_2 from p_1:
  - Update $\texttt{from}(p_2)$ with the next instruction of p_1
  - Place the program counter on the first instruction of p_2
- Simulating a return from p_2 to p_1:
  - Put the program counter on the instruction $\texttt{from}(p_2)$

Assumption

- No recursive call

Overview of the Transformation IV

Main procedure

- Initialize program counters
- Initialize `from()` for each *main* procedure
- Loop until each thread has executed all its instructions:
  - Choose a thread that still has instructions to execute
  - Resolve its program counter
  - Execute the corresponding simulating procedure

Assumption

- Interleaving semantics

Statements

Original code

```
if(x > 0) then
   p[0] := x
else
   p[0] := 42
```

Simulating conditional

```
sim_1(tid){
   ptr := &x ;
   x := ptr[tid] ;
   if(x > 0) then {
      ptr := pct ;
      ptr[tid] := 2
   } else {
      ptr := pct ;
      ptr[tid] := 4
   }
}
```

Statements

Original code

```
if(x > 0) then
    p[0] := x
else
    p[0] := 42
```

Simulating memory write

```
sim_2(tid){
    ptr := &x ;
    x := ptr[tid] ;
    ptr := &p ;
    p := ptr[tid] ;
    p[0] := x ;

    ptr := pct ;
    ptr[tid] := 5
}
```

Table of Contents

Bi-simulation Property I

Theorem

Let $prog_{par}$ be a safe parallel program, $prog_{sim}$ its simulating program, σ^{init}_{par} (resp. σ^{init}_{sim}) an initial state of $prog_{par}$ (resp. $prog_{sim}$).

1. From σ^{init}_{sim}, we can reach, by the initialization sequence, σ^{0}_{sim} equivalent to σ^{init}_{par}.
2. For all σ_{par} reachable from σ^{init}_{par}, there exists an equivalent σ_{sim} reachable from σ^{0}_{sim} with an equivalent trace (Forward simulation).
3. For all σ_{sim} reachable from σ^{0}_{sim}, there exists an equivalent σ_{par} reachable from σ^{init}_{par} with an equivalent trace (Backward simulation).

Bi-simulation Property I

Theorem

Let $prog_{par}$ be a safe parallel program, $prog_{sim}$ its simulating program, σ_{par}^{init} (resp. σ_{sim}^{init}) an initial state of $prog_{par}$ (resp. $prog_{sim}$).

1. From σ_{sim}^{init}, we can reach, by the initialization sequence, σ_{sim}^{0} equivalent to σ_{par}^{init}.
2. For all σ … reachable from σ^{init}, there exists an equivalent σ_{sim} rea… …ce (Forward simulation).
3. For all σ_{sim} reachable from σ_{sim}^{0}, there exists an equivalent σ_{par} reachable from σ_{par}^{init} with an equivalent trace (Backward simulation).

Initialization establishes equivalence

Bi-simulation Property I

Theorem

Let $prog_{par}$ be a safe parallel program, $prog_{sim}$ its simulating program, σ_{par}^{init} (resp. σ_{sim}^{init}) an initial state of $prog_{par}$ (resp. $prog_{sim}$).

1. From σ_{sim}^{init}, we can reach, by the initialization sequence, σ_{sim}^{0} equivalent to σ_{par}^{init}.
2. For all σ_{par} reachable from σ_{par}^{init}, there exists an equivalent σ_{sim} reachable from σ_{sim}^{0} with an equivalent trace (Forward simulation).
3. For all σ reachable from σ^{0}, there exists an equivalent σ_{p} … (Backward simulation).

All existing parallel executions are simulated

Bi-simulation Property I

Theorem

Let $prog_{par}$ be a safe parallel program, $prog_{sim}$ its simulating program, σ_{par}^{init} (resp. σ_{sim}^{init}) an initial state of $prog_{par}$ (resp. $prog_{sim}$).

1. From σ_{sim}^{init}, we can reach, by the initialization sequence, σ_{sim}^{0} equivalent to σ_{par}^{init}.
2. For all σ_{par} reachable from σ_{par}^{init}, there exists an equivalent σ … orward simulation).
3. For all σ_{sim} reachable from σ_{sim}^{0}, there exists an equivalent σ_{par} reachable from σ_{par}^{init} with an equivalent trace (Backward simulation).

Only existing parallel executions are simulated

Bi-simulation Property II

Original Program

Simulating Program

$$
\begin{tikzcd}
 & \sigma_{sim}^{init} \arrow[d] \\
\sigma_{par}^{init} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma_{sim}^{0} \arrow[d, dashed] \\
\sigma_{par} \arrow[r, leftrightarrow] \arrow[d] & \sigma_{sim} \arrow[d] \\
\sigma'_{par} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma'_{sim} \arrow[d, dashed] \\
{} & {}
\end{tikzcd}
$$

Bi-simulation Property II

Original Program

Simulating Program

From σ_{sim}^{init} ...

$$
\begin{tikzcd}
 & \sigma_{sim}^{init} \arrow[d] \\
\sigma_{par}^{init} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma_{sim}^{0} \arrow[d, dashed] \\
\sigma_{par} \arrow[r, leftrightarrow] \arrow[d] & \sigma_{sim} \arrow[d] \\
\sigma'_{par} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma'_{sim} \arrow[d, dashed] \\
{} & {}
\end{tikzcd}
$$

Bi-simulation Property II

Original Program

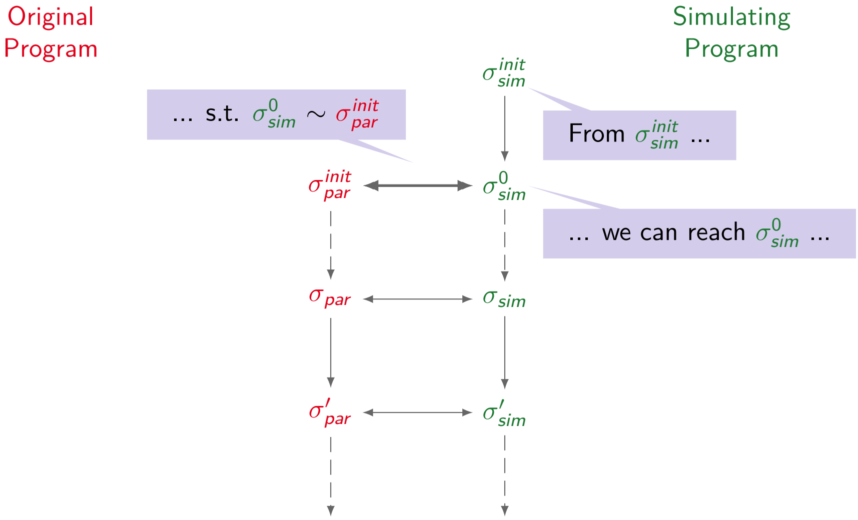
Simulating Program

$$
\begin{tikzcd}
 & \sigma_{sim}^{init} \arrow[d] \\
\sigma_{par}^{init} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma_{sim}^{0} \arrow[d, dashed] \\
\sigma_{par} \arrow[r, leftrightarrow] \arrow[d] & \sigma_{sim} \arrow[d] \\
\sigma'_{par} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma'_{sim} \arrow[d, dashed] \\
{} & {}
\end{tikzcd}
$$

From σ_{sim}^{init} ...

... we can reach σ_{sim}^{0} ...

Bi-simulation Property II



Bi-simulation Property II

Original Program

Simulating Program

$$
\begin{tikzcd}
 & \sigma_{sim}^{init} \arrow[d] \\
\sigma_{par}^{init} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma_{sim}^{0} \arrow[d, dashed] \\
\sigma_{par} \arrow[r, leftrightarrow] \arrow[d] & \sigma_{sim} \arrow[d] \\
\sigma'_{par} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma'_{sim} \arrow[d, dashed] \\
{} & {}
\end{tikzcd}
$$

Bi-simulation Property II

Original Program

Simulating Program

$$
\begin{tikzcd}
 & \sigma_{sim}^{init} \arrow[d] \\
\sigma_{par}^{init} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma_{sim}^{0} \arrow[d, dashed] \\
\sigma_{par} \arrow[r, leftrightarrow] \arrow[d] & \sigma_{sim} \arrow[d] \\
\sigma'_{par} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma'_{sim} \arrow[d, dashed] \\
  &
\end{tikzcd}
$$

Forall σ_{sim} ...

Bi-simulation Property II

Original Program

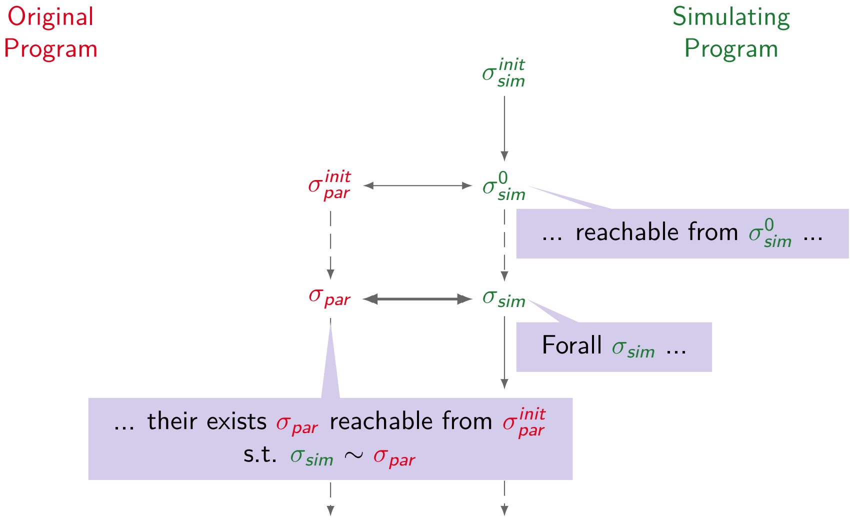
Simulating Program

$$
\begin{tikzcd}
 & \sigma_{sim}^{init} \arrow[d] \\
\sigma_{par}^{init} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma_{sim}^{0} \arrow[d, dashed] \\
\sigma_{par} \arrow[r, leftrightarrow, thick] \arrow[d] & \sigma_{sim} \arrow[d] \\
\sigma'_{par} \arrow[r, leftrightarrow] \arrow[d, dashed] & \sigma'_{sim} \arrow[d, dashed] \\
  &
\end{tikzcd}
$$

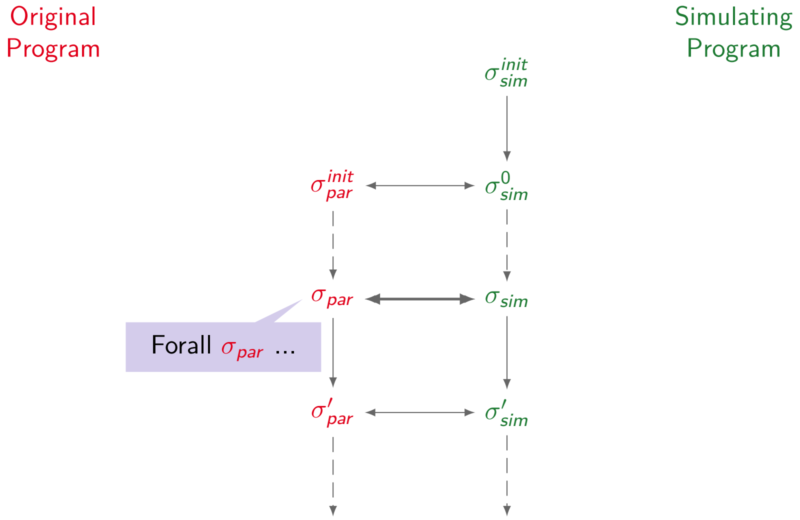
... reachable from σ_{sim}^{0} ...

Forall σ_{sim} ...

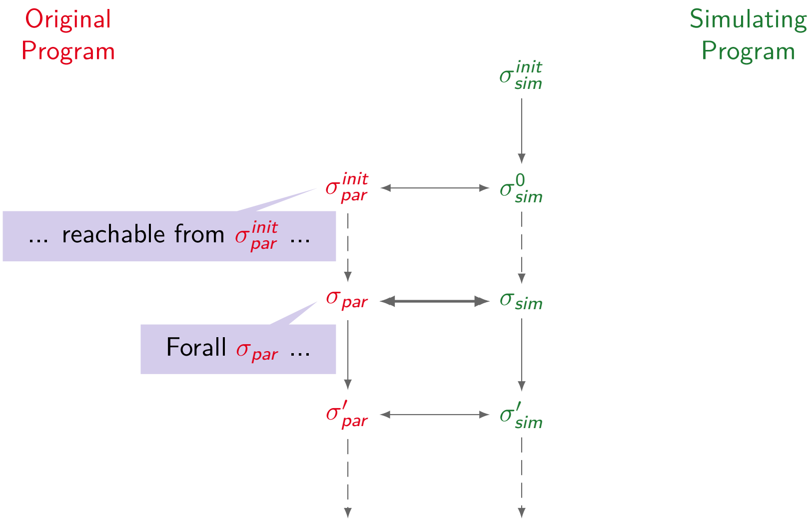
Bi-simulation Property II



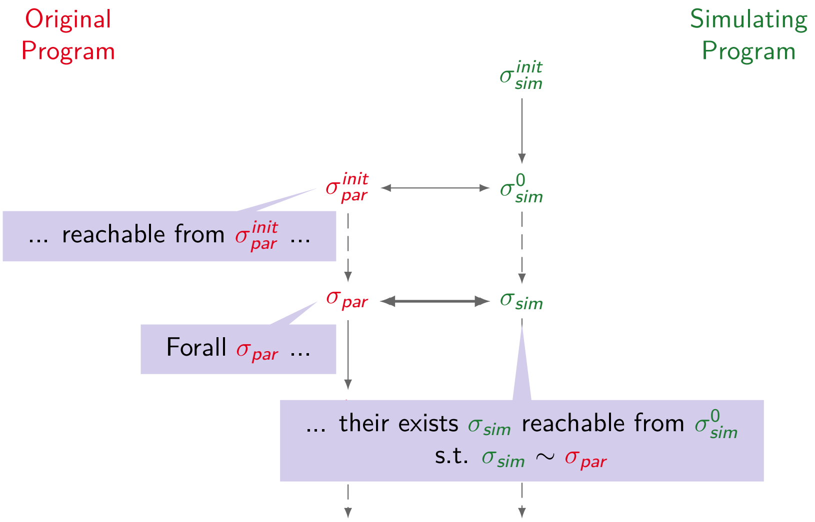
Bi-simulation Property II



Bi-simulation Property II



Bi-simulation Property II



Equivalence Relations

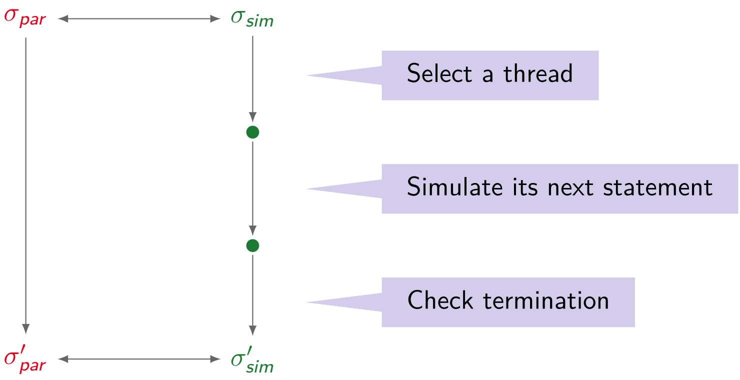
States

- The original heap is a subheap of the simulating one
- The simulating heap correctly models local variables
- The simulating heap correctly models stacks

Traces

All actions involving global memory in the original program must happen in the same order in the simulation. We also check procedure calls and return's.

Bi-simulation Property III



Basic Ideas of the Proof

By induction

- on instructions for the forward simulation
- on loop iterations for the backward simulation

What we can notice

- Sequential actions are deterministic, their translation too
- Thread selection is not determinist,
- but we can choose the same thread

Table of Contents

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 Conq2Seq 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

Our formalization is more general than our Frama-C plugin

- add function call simulation to Conc2Seq

The proof is currently a pen & paper proof

- mechanized proof using Coq is ongoing

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