Centre Val de Loire
UNIVERSITÉ
Spirit of the Loire Valley

UNIVERSITE D'ORLEANS

VERIFICATION OF CONCURRENT PROGRAMS ASSISTED BY CODE AND SPECIFICATION TRANSFORMATION

PHD THESIS DEFENCE

ALLAN BLANCHARD

December 6th, 2016

list
ceatech

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS
LIFO

DGA

Table of Contents

Table of Contents

Programs

Complex software

- Software is composed of millions of lines of code
- More multi-core architectures
- More concurrent programs

Concurrent Programs

Complex software

- Software is composed of millions of lines of code
- More multi-core architectures
- More concurrent programs

Concurrent Programs

Complex software

- Software is composed of millions of lines of code
- More multi-core architectures
- More concurrent programs

Concurrent Programs

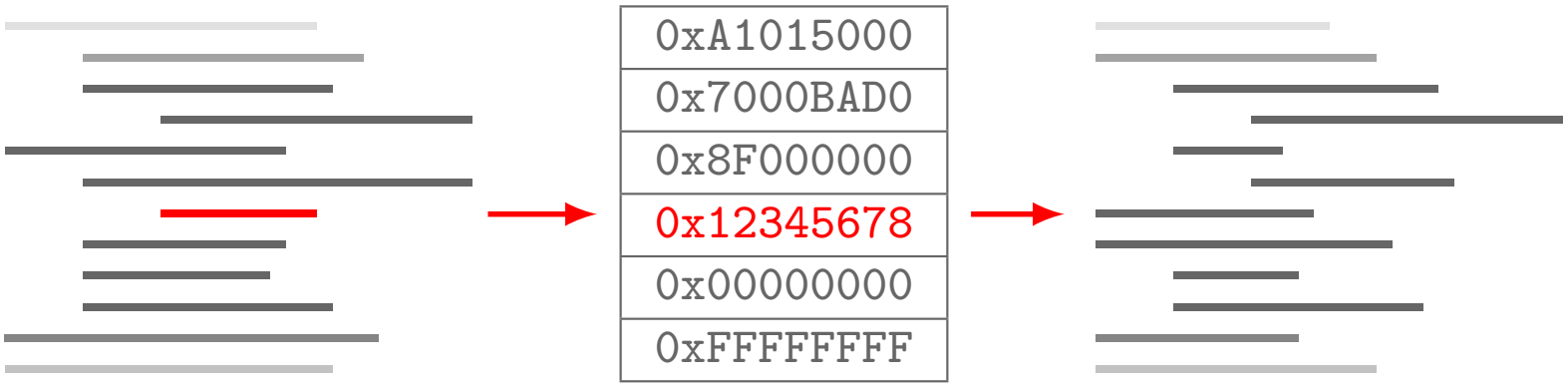
Complex software

- Software is composed of millions of lines of code
- More multi-core architectures
- More concurrent programs

Concurrent Programs

Complex software

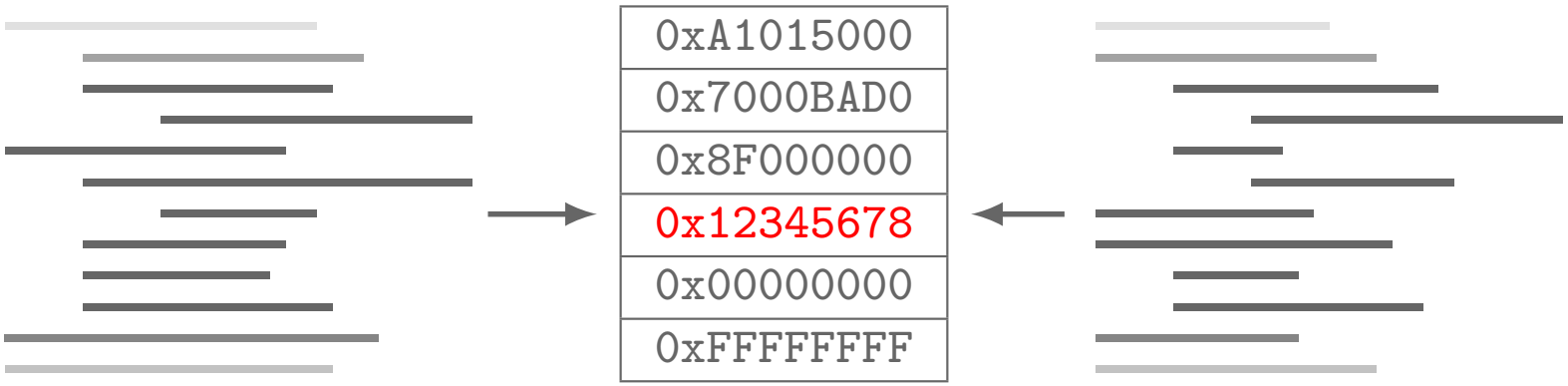
- Software is composed of millions of lines of code
- More multi-core architectures
- More concurrent programs



Concurrent Programs

Complex software

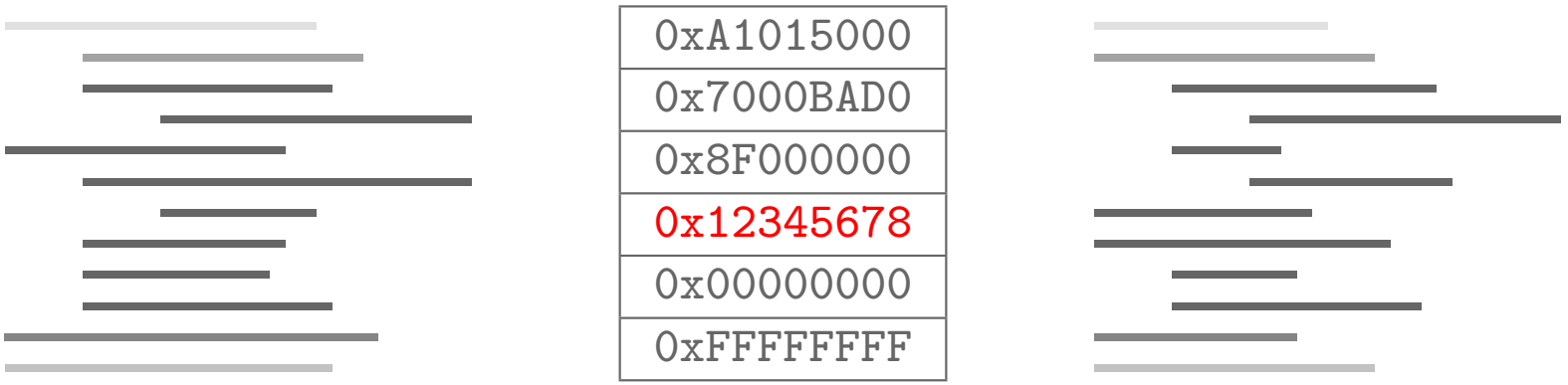
- Software is composed of millions of lines of code
- More multi-core architectures
- More concurrent programs



Concurrent Programs

Complex software

- Software is composed of millions of lines of code
- More multi-core architectures
- More concurrent programs



How can we ensure the validity of our software ?

Existing Methods

Most commonly used

- Program testing

Formal Methods

- Abstract interpretation
- Model-checking
- Theorem proving

Existing Methods

Most commonly used

- Program testing

Formal Methods

- Abstract interpretation
- Model-checking
- Theorem proving

Concurrent Programs Logics

Assuming sequentially consistent memory model

- HOARE 1975. Critical sections, abstract from time
- OWICKI-GRIES 1976. Resource invariant, ghost variables
- Rely-Guarantee, JONES 1983. Interferences
- Concurrent Separation Logic, O'HEARN et al 2007
- Combination of RG and CSL, VAFEIADIS et al 2007

Assuming weak memory models

- Relaxed separation logic, VAFEIADIS et al 2013
- Ghost, Procotols and Separation, VAFEIADIS et al 2014

Existing Verification Tools

Concurrent C code verification

- VCC (Owicki-Gries + $\sim$ Rely Guarantee + Ownership)
- Verifast (Separation logic)

FRAMA-C

- C code analysis
- Modular architecture based on plugins
- Most analyses do not handle concurrency

Contribution

Code and specification transformation

Transform a concurrent code into a (simulating) sequential one

- Experimented on a use case
- Automated in a FRAMA-C plugin
- Proved sound

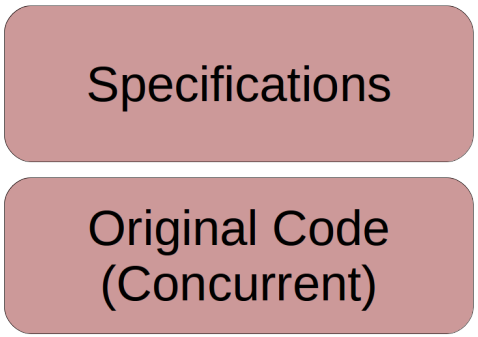
Weak memory models

A constraint based solver for weak memory behaviors

- Generate all candidate executions of a program
- Determine for each one if it is allowed by a given model

Table of Contents

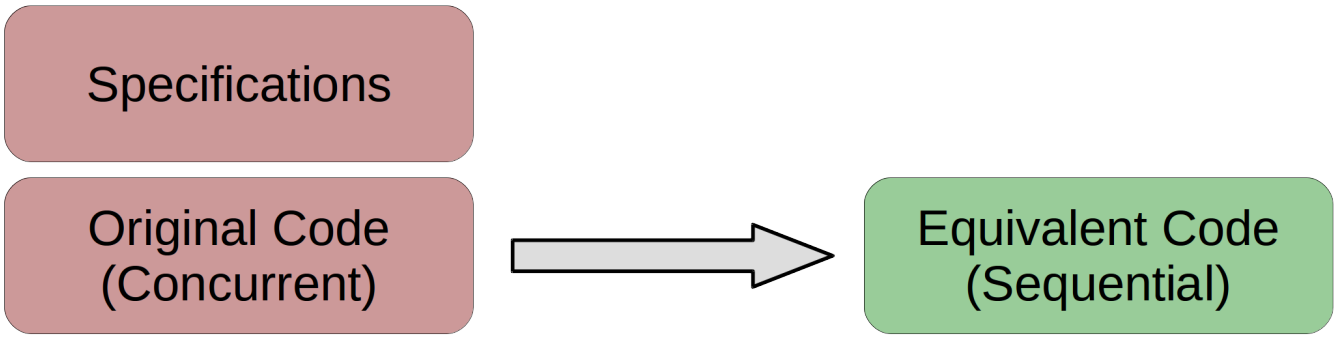
Overview I



Requirements

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

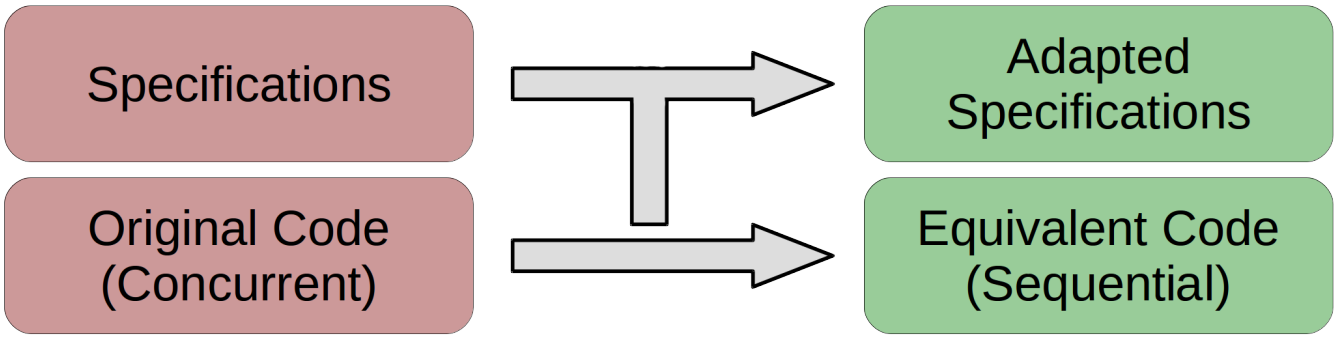
Overview I



Requirements

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

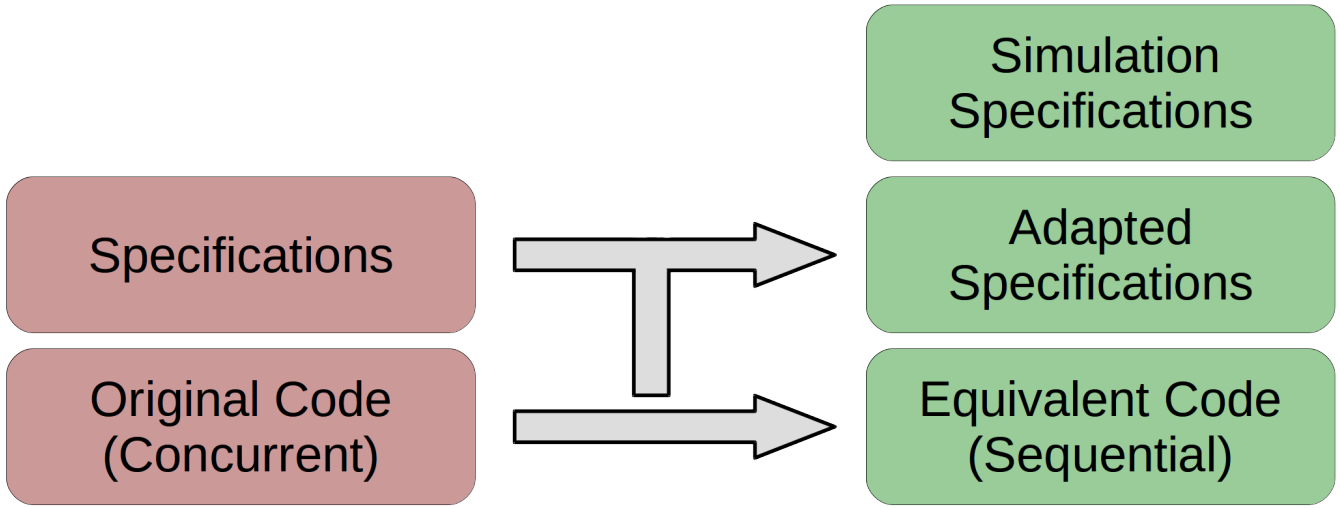
Overview I



Requirements

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

Overview 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

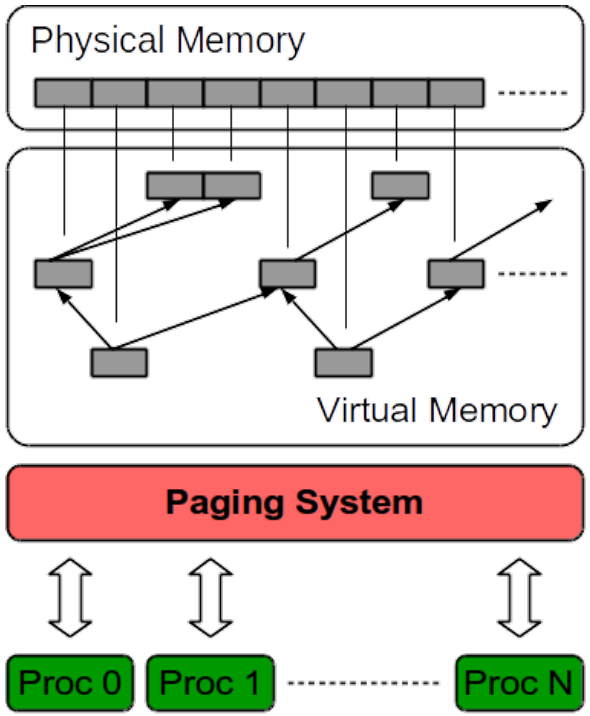
We suppose an interleaving semantics $\Rightarrow$ SC memory model

Specifications transformation

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

Case study – Micro-kernel code

- Anaxagoros Microkernel Virtual Memory Subsystem
  - Manages memory pages
  - Organized as hierarchy
  - **Counts mappings to pages**
- We want to verify the function used to update page tables
  - For any page, the indicated number of mappings must be greater or equal to the reality



Case study – Verification results

For atomic low-level functions

- ACSL specifications
- Proof with FRAMA-C and WP

For the page update function (concurrent)

- Specification and proof for sequential version
- Simulation of concurrent executions
- Weakening of the specification and proof:
  - Mostly automatic with FRAMA-C and WP
  - Auxiliary lemmas proved with COQ

Case study – Simulation

Original Code

```
#define NOF 2048
#define MAX  256

uint mappings[NOF];

int set_entry(uint fn, uint idx, uint new){
  uint c_n = mappings[new];
  if(c_n >= MAX) return 1;
  if(!CAS(&mappings[new], c_n, c_n+1))
    return 1;

  page_t p = get_frame(fn);
  uint old = atomic_exchange(&p[idx], new);

  if(!old) return 0;

  fetch_and_sub(&mappings[old], 1);
  return 0;
}
```

Simulating Code

```
#define THD 16

uint pct[THD];

uint fn [THD];
uint idx[THD];
uint new[THD];
uint c_n[THD];
uint old[THD];

//@ghost uint ref[THD]

...
```

Case study – Simulation

Original Code

```
#define NOF 2048
#define MAX  256

uint mappings[NOF];

int set_entry(uint fn, uint idx, uint new){
  uint c_n = mappings[new];
  if(c_n >= MAX) return 1;
  if(!CAS(&mappings[new], c_n, c_n+1))
    return 1;

  page_t p = get_frame(fn);
  uint old = atomic_exchange(&p[idx], new);

  if(!old) return 0;

  fetch_and_sub(&mappings[old], 1);
  return 0;
}
```

Simulating Code

```
...

void gen_args(uint th){
  fn[th]  = random_page();
  idx[th] = random_idx();
  new[th] = random_page();
  pct[th] = 1;
}

...
```

Case study – Simulation

Original Code

```
#define NOF 2048
#define MAX  256

uint mappings[NOF];

int set_entry(uint fn, uint idx, uint new){
  uint c_n = mappings[new];
  if(c_n >= MAX) return 1;
  if(!CAS(&mappings[new], c_n, c_n+1))
    return 1;

  page_t p = get_frame(fn);
  uint old = atomic_exchange(&p[idx], new);

  if(!old) return 0;

  fetch_and_sub(&mappings[old], 1);
  return 0;
}
```

Simulating Code

```
...

void read_map_new(uint th){
  c_n[th] = mappings[new[th]];
  pct[th] = 2;
}

...
```

Case study – Simulation

Original Code

```
#define NOF 2048
#define MAX  256

uint mappings[NOF];

int set_entry(uint fn, uint idx, uint new){
  uint c_n = mappings[new];
  if(c_n >= MAX) return 1;
  if(!CAS(&mappings[new], c_n, c_n+1))
    return 1;

  page_t p = get_frame(fn);
  uint old = atomic_exchange(&p[idx], new);

  if(!old) return 0;

  fetch_and_sub(&mappings[old], 1);
  return 0;
}
```

Simulating Code

```
...

void interleaving(){
  while(true){
    uint th = choose_a_thread();

    switch(pct[th]){
    case 0 : gen_args(th);      break;
    case 1 : read_map_new(th); break;
    case 2 : test_map_new(th); break;
    case 3 : cas_map_new(th);  break;
    case 4 : exch_entry(th);   break;
    case 5 : test_old(th);      break;
    case 6 : fas_map_old(th);  break;
    }
  }
}
```

Case study – Benefits and Limitations

Verification results

- Using FRAMA-C with WP
- Mostly automatic thanks to SMT solvers (CVC4, Z3)
- Completed by interactive proof using COQ

Limitations

- Most of the work could have been done automatically
- Known fixed-size arrays : fixed number of threads
- Limited scaling expected

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

CONC2SEQ – 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> );

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

simulating variables separation

simulating variables validity

CONC2SEQ – 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;
}
```

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)))
```

Soundness proof – Simplified language

mth	::=	$m(\mathit{list}\ x)\mathit{block}$	$m \in \mathit{Names}$
main	::=	$m(\mathit{list}\ e)$	
c	::=	$x := e$	local assignment
	|	$x[y] := e$	memory store
	|	$x := y[e]$	memory load
	|	**while** e **do** *block*	
	|	**if** e **then** *block* **else** *block*	
	|	$m(\mathit{list}\ e)$	call
	|	**atomic** *block*	
v	::=	$n \mid l \mid b$	$n \in \mathbb{Z}, l \in \mathbb{L}, b \in \mathbb{B},$
e	::=	$v \mid x \mid op(\bar{e})$	$x \in \mathcal{X}$

Soundness proof – Definitions: States and Traces

States

local environments	$P = \{\rho : \mathcal{X} \rightharpoonup \mathcal{V}\}$
heaps	$\Sigma = \{\sigma : \mathbb{L} \rightharpoonup \mathbb{N} \rightharpoonup \mathcal{V}\}$
global state (seq)	$\gamma_{seq} : \Sigma \times \overline{Names \times P \times \mathcal{C}}$
global state ($\|\|$)	$\gamma_{\|\|} : \Sigma \times (\mathbb{T} \to \overline{Names \times P \times \mathcal{C}})$

Traces

Actions:

- $a ::= \tau \mid \texttt{call}\ m \mid \texttt{return}\ m \mid \texttt{read}\ l\ n\ v \mid \texttt{write}\ l\ n\ v$
- $\texttt{atomic}\ a*$

Soundness proof – Hypotheses

Memory

- Statically allocated
- Does not initially contain addresses

Forbidden actions

- Nested atomic blocks
- Thread spawning
- Recursive functions

Soundness proof – Function call/return

New simulating memory blocks

- For each function we add an address *from* in the heap
- It indicates the next instruction to perform in the caller

Call/Return simulation

- Call simulation updates *from* for the executed thread
- Return simulation puts the program counter to this value

Soundness proof – Equivalences

State equivalence

- $\sigma_{sim} = \sigma_{||} \sqcup \sigma_{sim,\mathcal{X}}$
- $\sigma_{sim,\mathcal{X}}$ correctly represents every local environment
- $\sigma_{sim,\mathcal{X}}$ correctly models every stacks

Trace equivalence

- Forall action (t, a), we have a list of actions $(\texttt{write}\ \ ptid\ 0\ t) :: l$, where:
  - $l = []$ if a is τ
  - $l = [a]$ if a is a memory access
  - $l = [\texttt{call}\ \ m]$ if a is a call/return and m is its simulation
  - l is the list of actions of a, if a is an atomic block action.

Soundness proof – Main result

Theorem

Forall (safe) $p_{||}$, p_s its simulation, from states $\gamma_{||,init}$ and $\gamma_{s,init}$

- by simulation initialization, we reach $\gamma_{s,0}$ equivalent to $\gamma_{||,init}$
- forall γ_s reachable from $\gamma_{s,0}$ with a trace t_s, there exists $\gamma_{||}$ reachable from $\gamma_{||,init}$ with a trace $t_{||}$ equivalent to t_s.
- forall $\gamma_{||}$ reachable from $\gamma_{||,init}$ with a trace $t_{||}$, there exists γ_s reachable from $\gamma_{s,0}$ with a trace t_s equivalent to $t_{||}$.

Proof ideas

- Sequential actions are deterministic, their translation too
- Thread selection: just choose the same thread

Formalization – Toward a Coq proof

Implemented up to known

- Language and semantics
- Transformation function

Next steps

- Write equivalence in Coq
- Perform the Coq proof

Table of Contents

Sequential Consistency (Lamport 1979)

A simple program

$$\mathtt{x} := 0;\ \mathtt{y} := 0$$

Thread 0	Thread 1
$\mathtt{x} := 1;$	$\mathtt{y} := 1;$
$\mathtt{r}_0 := \mathtt{y};$	$\mathtt{r}_1 := \mathtt{x};$

Possible results :

- $\mathtt{r}_0 = 1 \wedge \mathtt{r}_1 = 1$
- $\mathtt{r}_0 = 0 \wedge \mathtt{r}_1 = 1$
- $\mathtt{r}_0 = 1 \wedge \mathtt{r}_1 = 0$

Impossible result :

- $\mathtt{r}_0 = 0 \wedge \mathtt{r}_1 = 0$

Existing dedicated tools

Herding cats (Alglave et al. 2014)

Generic framework for weak behaviors

- Written in OCaml
- Provides a language to specify memory models

JMMSolve (Schrijvers 2004) based on CCMM (Saraswat 2004)

Program executions under Java Memory Model

- Based on Concurrent Constraint-based Memory Machines
- Written using Constraint Handling Rules (CHR)

Prolog and CHR

Prolog

Declarative language for logic programming

Constraint Handling Rules

Declarative language for constraint programming

- Maintains a store of constraints ($\sim$ terms)
- Handled by rules that will add or remove constraints

Goals of our solver

To identify allowed executions

- For a given parallel program
- According to a given memory model

Additional goals

- Possibility to add new memory models
- Support of specific instructions

Chosen approach

Generate all candidate executions

An execution is represented by ordering relations :

- For each location *l*, a total ordering of every store to *l* (CO)
- For each load, a store having written the read value (RF)
- We combine all permutations of CO/RF using backtracking

Filter out forbidden executions

- Apply model rules to deduce more ordering relations
- Incoherent execution: an action must happen before itself (which means that some relations exhibit a cycle)

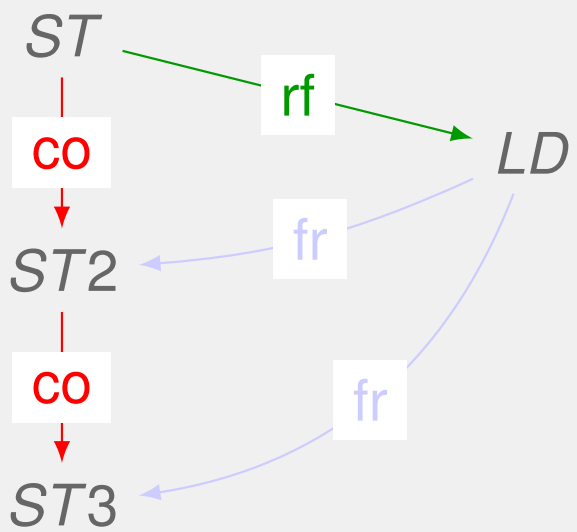
Express and derive relations with CHR

Ordering relations vs. CHR constraints

Relation	CHR constraint
$x := 0 \xrightarrow{CO} x := 1$	$\texttt{co}(\texttt{x} := 0, \texttt{x} := 1)$
$x := 2 \xrightarrow{RF} r := x$	$\texttt{rf}(\texttt{x} := 2, \texttt{r} := \texttt{x}(2))$

CHR rules to derive new relations :

- $\texttt{rf}(\texttt{ST}, \texttt{LD}), \texttt{co}(\texttt{ST}, \texttt{ST2}) \Rightarrow \texttt{fr}(\texttt{LD}, \texttt{ST2})$.
- $\texttt{fr}(\texttt{LD}, \texttt{ST2}), \texttt{co}(\texttt{ST2}, \texttt{ST3}) \Rightarrow \texttt{fr}(\texttt{LD}, \texttt{ST3})$.



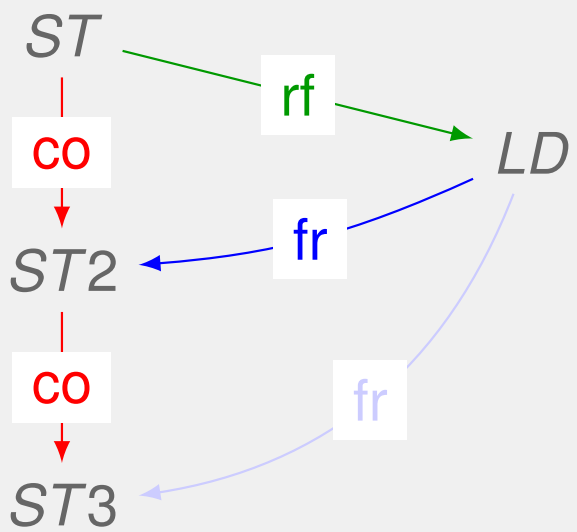
Express and derive relations with CHR

Ordering relations vs. CHR constraints

Relation	CHR constraint
$x := 0 \xrightarrow{CO} x := 1$	$\texttt{co}(\texttt{x} := 0, \texttt{x} := 1)$
$x := 2 \xrightarrow{RF} r := x$	$\texttt{rf}(\texttt{x} := 2, \texttt{r} := \texttt{x}(2))$

CHR rules to derive new relations :

- $\texttt{rf}(\texttt{ST}, \texttt{LD}), \texttt{co}(\texttt{ST}, \texttt{ST2}) \Rightarrow \texttt{fr}(\texttt{LD}, \texttt{ST2})$.
- $\texttt{fr}(\texttt{LD}, \texttt{ST2}), \texttt{co}(\texttt{ST2}, \texttt{ST3}) \Rightarrow \texttt{fr}(\texttt{LD}, \texttt{ST3})$.



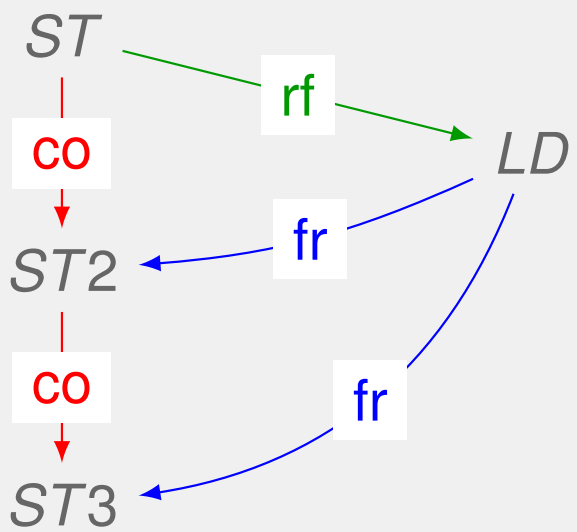
Express and derive relations with CHR

Ordering relations vs. CHR constraints

Relation	CHR constraint
$x := 0 \xrightarrow{CO} x := 1$	$\texttt{co}(\texttt{x} := 0, \texttt{x} := 1)$
$x := 2 \xrightarrow{RF} r := x$	$\texttt{rf}(\texttt{x} := 2, \texttt{r} := \texttt{x}(2))$

CHR rules to derive new relations :

- $\texttt{rf}(\texttt{ST}, \texttt{LD}), \texttt{co}(\texttt{ST}, \texttt{ST2}) \Rightarrow \texttt{fr}(\texttt{LD}, \texttt{ST2})$.
- $\texttt{fr}(\texttt{LD}, \texttt{ST2}), \texttt{co}(\texttt{ST2}, \texttt{ST3}) \Rightarrow \texttt{fr}(\texttt{LD}, \texttt{ST3})$.

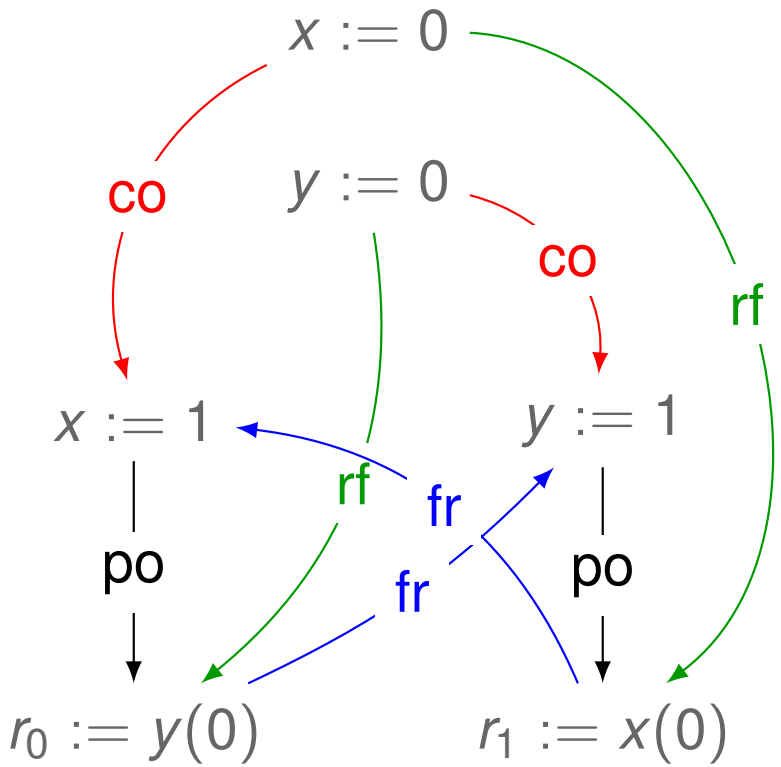


Definition of a model : SC

Definition of a model

- Define a new relation as the union of maintained relations

```
co(I,J) ==> sc(I,J).
rf(I,J) ==> sc(I,J).
fr(I,J) ==> sc(I,J).
po(I,J) ==> sc(I,J).
```

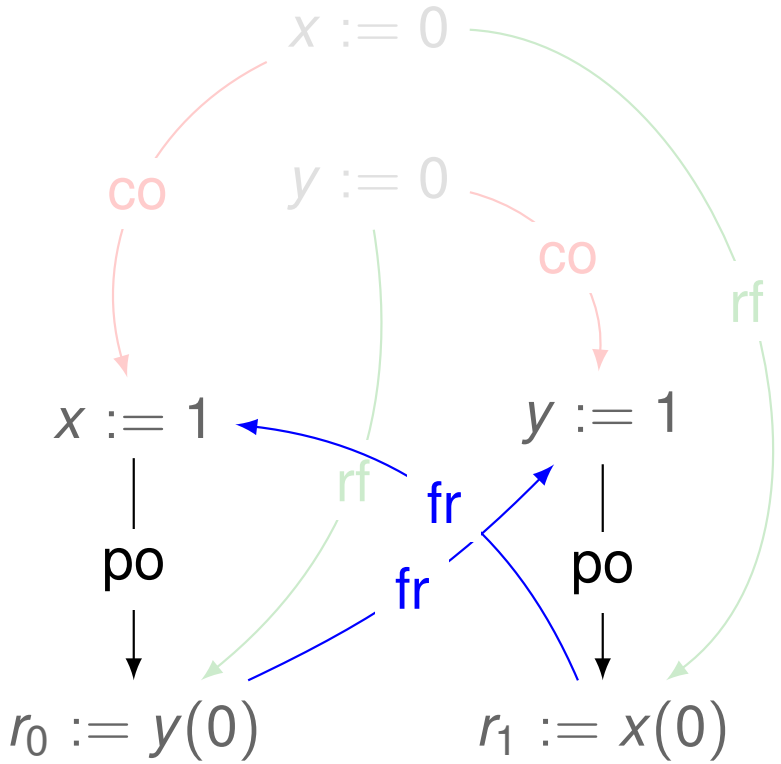


Definition of a model : SC

Definition of a model

- Define a new relation as the union of maintained relations

```
co(I,J) ==> sc(I,J).
rf(I,J) ==> sc(I,J).
fr(I,J) ==> sc(I,J).
po(I,J) ==> sc(I,J).
```



Validation

Available Models

- SC, TSO, PSO and ARM (Almost finished)
- Easy to extend new memory models

Program samples from Herd

Our solver has been tested on 18 examples of litmus programs

- Message passing,
- Basic uniproc relations,
- ...

We observe the same results found by Herd.

Performance tests

Multiple message passing

Example	Model	#exec	Herd	CHR Solver
MP3	Generic	147 436	**1.2s**	3.3s
	PSO	2 258	**3.8s**	6.4s
	TSO	800	4.1s	**3.2s**
	SC	678	5.5s	**3.3s**
MP4	Generic	255 000 000	**1405s**	$>$ 1h
	PSO	516 030	$>$ 1h	**2796s**
	TSO	96 498	$>$ 1h	**752s**
	SC	81 882	$>$ 1h	**747s**

Early pruning makes the solver efficient for “long” programs.
Our solver was not so hard to implement.

Table of Contents

Contribution

A method for the verification of concurrent code

- Under sequentially consistent memory model
- Based on code and specification transformation

Our method is

- Successfully applied on a part of a microkernel [FMICS 2015]
- Now automated in a Frama-C plugin [SCAM 2016]
- Proved sound on paper, to be proved using Coq.

A CHR based solver for weak memory behaviors [CSTVA 2016]

- It can identify executions allowed by a memory model
- It has interesting performances on “big” programs

Future Work

Make the plugin compatible with other plugins

- Currently only WP is supported
- Determine critical features

Make further experiments

- Integrate function call in CONC2SEQ
- Analyze more case studies

Add new features to our CHR solver

- Complete ARM
- Arithmetic and branching instructions

Combine CONC2SEQ and our CHR solver

- Validate sequentialy consistent behavior using the solver

Publications

Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue
A case study on formal verification of the Anaxagoros hypervisor paging system with Frama-C
In *International Workshop on Formal Methods for Industrial Critical Systems* (FMICS 2015), LNCS, pages 15–30, Oslo, Norway, June 2015. Springer

Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue
A CHR-Based Solver for Weak Memory Behaviors
In *Proceedings of the 7th Workshop on Constraint Solvers in Testing, Verification (CSTVA 2016), and Analysis co-located with The International Symposium on Software Testing and Analysis (ISSTA 2016), Saarbrücken, Germany, July 17th, 2016.*, pages 15–22, 2016

Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue
Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs
In *16th IEEE International Working Conference on Source Code Analysis and Manipulation* (SCAM 2016), 2-3 October 2016, Raleigh, NC USA, 2016

Publications

Thank you for your attention !

Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue
A case study on formal verification of the Anaxagoros hypervisor paging system with Frama-C
In *International Workshop on Formal Methods for Industrial Critical Systems* (FMICS 2015), LNCS, pages 15–30, Oslo, Norway, June 2015. Springer

Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue
A CHR-Based Solver for Weak Memory Behaviors
In *Proceedings of the 7th Workshop on Constraint Solvers in Testing, Verification (CSTVA 2016), and Analysis co-located with The International Symposium on Software Testing and Analysis (ISSTA 2016), Saarbrücken, Germany, July 17th, 2016.*, pages 15–22, 2016

Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue
Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs
In *16th IEEE International Working Conference on Source Code Analysis and Manipulation* (SCAM 2016), 2-3 October 2016, Raleigh, NC USA, 2016

Thank you for your attention

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