Thread-Modular Abstractions for the Static Analysis of Concurrent Programs

Antoine Miné

LIP6
University Pierre and Marie Curie
Paris, France

Workshops on Numerical and Symbolic Abstract Domains &
Static Analysis of Concurrent Software
11 September 2016
Edinburgh, Scotland

Goal

Concurrent programs are hard to program, and hard to verify, due to

- combinatorial exposition of execution paths
- errors lurking in hard-to-find corner cases

No good sound static analysis tool for concurrent programs?

Goal: sound static analysis of concurrent programs

- for C programs (pointers, structs, floats, etc.)
- check for run-time errors, data-races, and deadlocks
- using flexible, composable abstractions
- with the potential for high scalability and high precision
- specialized, application-specific analyzer

$\Longrightarrow$ AstréeA: Astrée for concurrent embedded software.

Don't miss: talks by David Delmas (Airbus) and Stephan Wilhelm (AbsInt) this afternoon!

Concurrent execution model

- fixed set of threads
- executions as interleavings of atomic actions from threads
  Note: data-race analysis will report atomicity-related issues
- real-time, priority-based scheduling
  fully preemptive scheduling is the norm
  priorities may prevent some interactions, up to the analysis to discover
- shared-memory (all global variables shared)
- low-level synchronization with locks
  higher-level primitives through stub libraries
  (POSIX, ARINC 653, OSEK/AUTOSAR)
- memory consistency model:
  from sequential consistency + data-race freedom checking
  to partial store ordering (hardware memory model)
  to allowed program transformations (compiler memory model)
  depending on the abstract domain

Outline

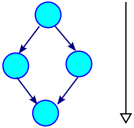
- Simple interference-based analysis
  - principle of thread-modular analysis
  - application to the AstréeA analyzer

- Towards more precise interference abstractions
  - rely-guarantee in abstract interpretation form
  - relational & flow-sensitive interference abstractions
  - applications

- Conclusion

Analysis with simple interferences

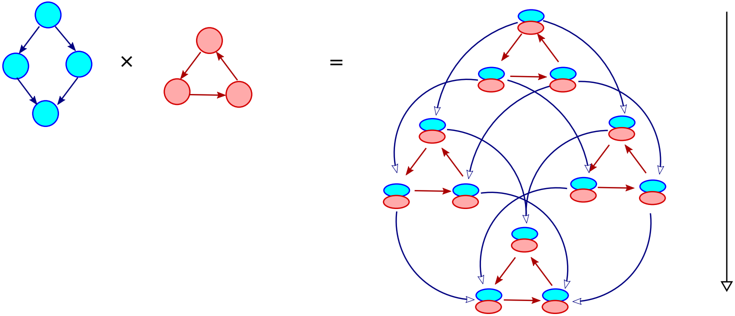
Thread-modular vs. non-thread-modular analysis



Sequential analysis:

- one abstract state per program point
- one transfer function per instruction
- various iteration schemes with widening

Thread-modular vs. non-thread-modular analysis



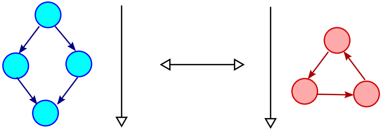
Natural extension to multi-thread: CFG product

- control state = tuple of program points
  $\Longrightarrow$ combinatorial explosion of abstract states
- transfer functions are duplicated

Not practical for high scalability...

Beyond partial-order reduction: we need abstraction *a priori*

Thread-modular vs. non-thread-modular analysis

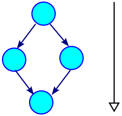


Thread-modular analysis:

- analyze each thread separately
- also analyze their interaction

CFG-based vs. syntax-based

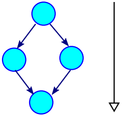
CFG-based:



$$\begin{cases} X_1 = \top \\ X_2 = F_2(X_1) \\ X_3 = F_3(X_1) \\ X_4 = F_4(X_3, X_4) \end{cases}$$

CFG-based vs. syntax-based

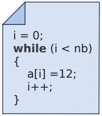
CFG-based:



$$\left\{ \begin{array}{l} X_1 = \top \\ X_2 = F_2(X_1) \\ X_3 = F_3(X_1) \\ X_4 = F_4(X_3, X_4) \end{array} \right.$$

- linear memory in program length
- flexible solving strategy
  flexible context sensitivity
- easy to adapt to concurrency, both in thread-modular and CFG product way

Syntax-based:



```
i = 0;
while (i < nb)
{
   a[i] =12;
   i++;
}
```

$$[\![\textbf{while } c \textbf{ do } b]\!] X \overset{\text{def}}{=} [\![\neg c]\!] (\text{lfp } \lambda Y. X \cup [\![b]\!]([\![c]\!] Y))$$

$$[\![\textbf{if } c \textbf{ then } t]\!] X \overset{\text{def}}{=} [\![t]\!]([\![c]\!] X) \cup [\![\neg c]\!] X$$

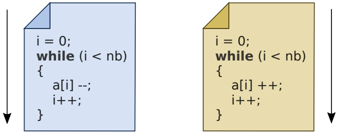
...

- linear memory in program depth
- fixed iteration strategy
  fixed context sensitivity
  (follows the program structure)
- no practical induction definition of product
  $\Longrightarrow$ thread-modular analysis

for scalability on large programs, memory is a limiting factor

$\Rightarrow$ **we use an interpreter by induction on the syntax**

Thread-modular analysis with simple interferences



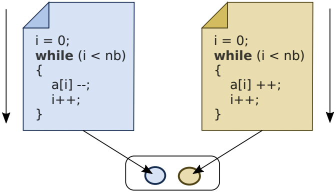
```
i = 0;
while (i < nb)
{
   a[i] --;
   i++;
}
```

```
i = 0;
while (i < nb)
{
   a[i] ++;
   i++;
}
```

Principle:

- analyze each thread in isolation

Thread-modular analysis with simple interferences



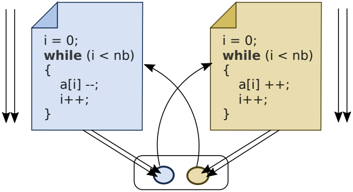
```
i = 0;
while (i < nb)
{
   a[i] --;
   i++;
}
```

```
i = 0;
while (i < nb)
{
   a[i] ++;
   i++;
}
```

Principle:

- analyze each thread in isolation
- gather the values written into each variable by each thread
  $\Longrightarrow$ so-called interferences
  suitably abstracted in an abstract domain, such as intervals

Thread-modular analysis with simple interferences



```
i = 0;
while (i < nb)
{
    a[i] --;
    i++;
}
```

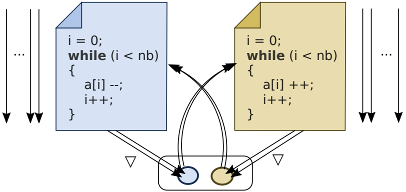
```
i = 0;
while (i < nb)
{
    a[i] ++;
    i++;
}
```

Principle:

- analyze each thread in isolation
- gather the values written into each variable by each thread
  $\Longrightarrow$ so-called interferences

  suitably abstracted in an abstract domain, such as intervals
- reanalyze threads, injecting these values at each read

Thread-modular analysis with simple interferences



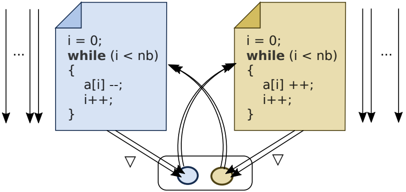
```
i = 0;
while (i < nb)
{
   a[i] --;
   i++;
}
```

```
i = 0;
while (i < nb)
{
   a[i] ++;
   i++;
}
```

Principle:

- analyze each thread in isolation
- gather the values written into each variable by each thread
  $\Longrightarrow$ so-called interferences
  suitably abstracted in an abstract domain, such as intervals
- reanalyze threads, injecting these values at each read
- iterate until stabilization while widening interferences

Thread-modular analysis with simple interferences



Principle:

- analyze each thread in isolation
- gather the values written into each variable by each thread
  $\Longrightarrow$ so-called interferences
  suitably abstracted in an abstract domain, such as intervals
- reanalyze threads, injecting these values at each read
- iterate until stabilization while widening interferences

Benefits:

- very similar to a sequential analysis (high reusability)
- efficient!

Simple abstract interferences: Example

Simple interference analysis: with interval abstraction

t_1

```
while random do
    if $x < y$ then
        $x \leftarrow x + 1$
done
```

t_2

```
while random do
    if $y < 100$ then
        $y \leftarrow y + [1, 3]$
done
```

Simple abstract interferences: Example

Simple interference analysis: with interval abstraction

```
t_1
while random do
    if $x < y$ then
        $x \leftarrow x + 1$
done
```

```
t_2
while random do
    if $y < 100$ then
        $y \leftarrow y + [1, 3]$
done
```

iteration	t_1	t_2
1	$\emptyset$	$\emptyset$

Analysis as separate sequential processes, without interferences

$\Longrightarrow$ t_2 writes $[1, 102]$ into y

Simple abstract interferences: Example

Simple interference analysis: with interval abstraction

t_1

```
while random do
    if $x < y$ then
        $x \leftarrow x + 1$
done
```

t_2

```
while random do
    if $y < 100$ then
        $y \leftarrow y + [1, 3]$
done
```

iteration	t_1	t_2
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$y \mapsto [1, 102]$

add the information that t_2 writes $[1, 102]$ into y, for t_1 to use

$\Longrightarrow$ t_1 now writes into x

Simple abstract interferences: Example

Simple interference analysis: with interval abstraction

```
t1                              | t2
                                |
while random do                 | while random do
    if $x < y$ then             |     if $y < 100$ then
        $x \leftarrow x + 1$    |         $y \leftarrow y + [1, 3]$
done                            | done
```

iteration	t_1	t_2	
1	$\emptyset$	$\emptyset$	
2	$\emptyset$	$y \mapsto [1, 102]$	
3	$x \mapsto [1, 102]$	$y \mapsto [1, 102]$	(stable)

t_1 writes into x, but this is not visible by t_2;
we reach a stable point

$\Longrightarrow$ program invariant: $x, y \in [0, 102]$

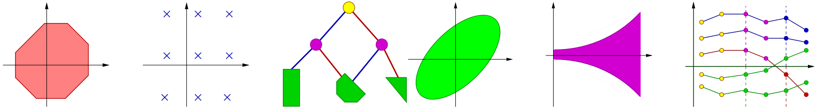
The Astrée(A) analyzer

Astrée:

- started as an academic project by : P. Cousot, R. Cousot, J. Feret, A. Miné, X. Rival, B. Blanchet, D. Monniaux, L. Mauborgne
- checks for absence of run-time error in embedded synchronous C code
- applied to Airbus software with zero alarm (A340 in 2003, A380 in 2004)
- industrialized by AbsInt since 2009

Design by refinement:

- incompleteness: any static analyzer fails on infinitely many programs
- completeness: any program can be analyzed by some static analyzer
- in practice:
  - from target programs and properties of interest
  - start with a simple and fast analyzer (interval)
  - while there are false alarms, add new / tweak abstract domains



Integrating simple interferences into Astrée

From Astrée to AstréeA:

- follow-up project: Astrée for concurrent embedded C code (2012–2016)
- interferences abstracted using stock non-relation domains
- memory domain instrumented to gather / inject interferences
- added an extra iterator
- added loop invariant caching (large speedup wrt. naive iteration on the syntax)

⟹ minimal code modifications

- additionally: 4 KB ARINC 653 OS model

First results:

- ARINC 653 embedded avionic application
- 15 threads, 1.6 Mlines
- embedded reactive code + network code + string formatting
- 4616 alarms in 25h, 22GB, with 6 iterations

Limitations of simple interferences

```
t_1
while random do
    if $x < y$ then
        $x \leftarrow x + 1$
done
```

```
t_2
while random do
    if $y < 100$ then
        $y \leftarrow y + [1, 3]$
done
```

- the analysis finds $x, y \in [0, 102]$
- but, in fact, $0 \leq x \leq y \leq 102$

Cause:

we started from a concrete semantics which is incomplete for reachability

Simple interferences perform a flow-insensitive, non-relational abstraction even before applying any abstract domain!

Towards more powerful interference abstractions

Rely-guarantee reasoning

checking t_1

	t_1	t_2
$(1a)$	**while** random **do**	x unchanged
$(2a)$	**if** $x < y$ **then**	y incremented
$(3a)$	$x \leftarrow x + 1$	$0 \leq y \leq 102$

$(1a) : x = y = 0$
$(2a) : x, y \in [0, 102],\ x \leq y$
$(3a) : x \in [0, 101],\ y \in [1, 102],\ x < y$

checking t_2

t_1		t_2
y unchanged	$(1b)$	**while** random **do**
$0 \leq x \leq y$	$(2b)$	**if** $y < 100$ **then**
	$(3b)$	$y \leftarrow y + [1, 3]$

$(1b) : x = y = 0$
$(2b) : x, y \in [0, 102],\ x \leq y$
$(3b) : x, y \in [0, 99],\ x \leq y$

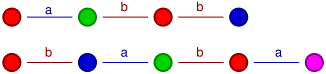
Rely–guarantee: proof method introduced by Jones in 1981

- generalized Hoare logics (by structural induction ⇒ thread-modular)
- requires thread-local invariant assertions and guarantees on transitions generated by other threads
- checks each thread against an abstraction of the other threads
- allows proving that $x \leq y$ holds!

We look for a static analysis, not a proof method

$\Longrightarrow$ infer automatically invariants and guarantees (i.e., interferences)

Complete concrete interference semantics

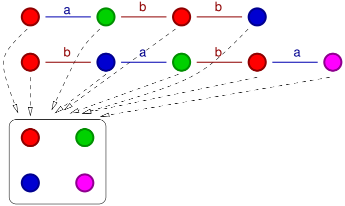


Whole-program concrete semantics:

- transition system: $\sigma \xrightarrow{a} \sigma'$ (action of thread a on state σ)
- executions = partial finite trace semantics

$$\mathcal{F} \overset{\text{def}}{=} \text{lfp}\, \lambda X.\, I \cup \{\, \sigma_0 \xrightarrow{a_1} \cdots \sigma_i \xrightarrow{a_{i+1}} \sigma_{i+1} \mid \sigma_0 \xrightarrow{a_1} \cdots \sigma_i \in X \wedge \sigma_i \xrightarrow{a_{i+1}}_{\tau} \sigma_{i+1} \,\}$$

Complete concrete interference semantics



Whole-program concrete semantics:

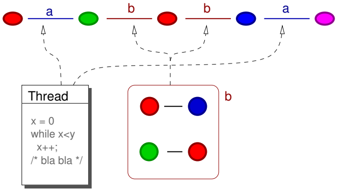
- transition system: $\sigma \xrightarrow{a} \sigma'$ (action of thread a on state σ)
- executions = partial finite trace semantics

$$\mathcal{F} \overset{\text{def}}{=} \text{lfp}\, \lambda X.\, I \cup \{\, \sigma_0 \xrightarrow{a_1} \cdots \sigma_i \xrightarrow{a_{i+1}} \sigma_{i+1} \mid \sigma_0 \xrightarrow{a_1} \cdots \sigma_i \in X \land \sigma_i \xrightarrow{a_{i+1}}_{\tau} \sigma_{i+1} \,\}$$

- collecting semantics = reachable states

$$\mathcal{R} \overset{\text{def}}{=} \alpha^{reach}(\mathcal{F}) \overset{\text{def}}{=} \{\, \sigma \mid \exists \sigma_0 \xrightarrow{a_1} \cdots \sigma_n \in F : \exists i \leq n : \sigma = \sigma_i \,\}$$

Complete concrete interference semantics

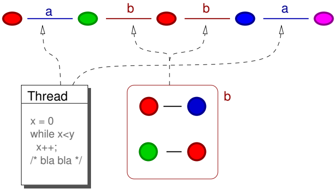


Thread-modular concrete semantics:

Given a thread *a*, a program execution interleaves

- steps from the thread
- steps from other threads

Complete concrete interference semantics



Thread-modular concrete semantics:

Given a thread *a*, a program execution interleaves

- steps from the thread
- steps from other threads

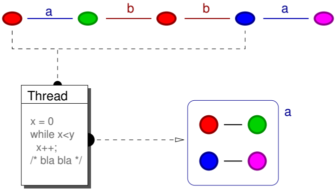
$\mathcal{R}(a) = \text{lfp } R_a(\mathcal{I})$, where

$$R_a(\mathcal{I})(X) \overset{\text{def}}{=} I \cup \{\, \sigma' \mid \exists \sigma \in X : \sigma \xrightarrow{a} \sigma' \,\} \cup \{\, \sigma' \mid \exists \sigma \in X : \exists a' \neq a : \langle \sigma, \sigma' \rangle \in \mathcal{I}(a') \,\}$$

$\Longrightarrow$ similar to reachability for a sequential program

- up to $\mathcal{I}$
- using enriched control state (auxiliary variables)

Complete concrete interference semantics

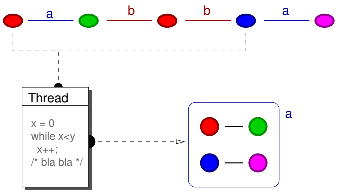


Thread-modular concrete semantics:

To get $\mathcal{I}$, we collect transitions from $\mathcal{R}$:

$$\mathcal{I}(a) = B(\mathcal{R}(a)), \text{ where } B(\mathcal{R}(a)) \stackrel{\text{def}}{=} \{\, \langle \sigma, \sigma' \rangle \mid \sigma \in \mathcal{R}(a) \land \sigma \xrightarrow{a} \sigma' \,\}$$

Complete concrete interference semantics



Thread-modular concrete semantics:

To sum up, we have a mutually recursive definition:

$$\begin{cases} \mathcal{R}(a) = \text{lfp } R_a(\mathcal{I}) \\ \mathcal{I}(a) = B(\mathcal{R}(a)) \end{cases}$$

$\Longrightarrow$ we express the most precise solution as nested fixpoints:

$$\mathcal{R} = \text{lfp } \lambda Z.\, \lambda a.\, \text{lfp } R_a(B(Z))$$

We retrieve the idea of iterating analyses with interference

Simple interferences as an abstraction

Completeness:

our concrete semantics computes exactly the reachability semantics
$\Longrightarrow$ any program can be analyzed by some thread-modular analyzer

Retrieving simple interferences using abstractions

From concrete interference $\mathcal{I} \in \mathfrak{I}$ where $\begin{cases} \mathfrak{I} \overset{\text{def}}{=} \mathcal{T} \to \mathcal{P}(\Sigma \times \Sigma) \\ \Sigma \overset{\text{def}}{=} (\mathcal{T} \to \mathcal{L}) \times (\mathcal{V} \to \mathit{Val}) \end{cases}$

to **simple interferences** in $\mathcal{T} \times \mathcal{V} \to \mathcal{P}(\mathit{Val})$

- remove control information $\mathcal{T} \to \mathcal{L}$
- remove relationality, only keep value for variables that changed

$$\alpha(I)(t, V) \overset{\text{def}}{=} \{\, \rho'(V) \mid ((c, \rho), (c', \rho')) \in I(t) \wedge \rho(V) \neq \rho(V') \,\}$$

Application:

develop new abstractions and combine them with simple interferences
to improve AstréeA by specialization

Fully relational interferences

From concrete interference $\mathcal{I} \in \mathfrak{I}$ in $\mathfrak{I} \overset{\text{def}}{=} \mathcal{T} \to \mathcal{P}(\Sigma \times \Sigma)$

to fully relational interferences in $(\mathcal{T} \times \mathcal{L} \times \mathcal{L}) \to \mathcal{P}(\Sigma')$

where $\Sigma' \overset{\text{def}}{=} (\mathcal{V}_{\mathcal{L}} \cup \mathcal{V} \cup \mathcal{V}') \to \mathbb{R}$

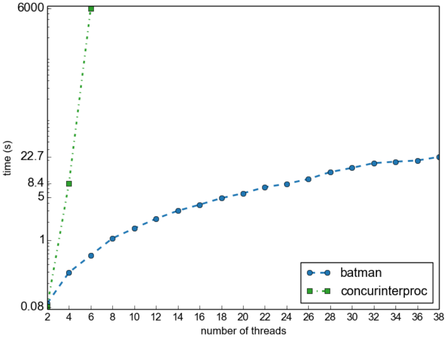
- model relations as memory states with input / output variables $\mathcal{V}$, $\mathcal{V}'$
  e.g.: $\{\, (x, x+1) \mid x \in [0, 10] \,\}$ is represented as $x' = x + 1 \wedge x \in [0, 10]$
- remember control state of other threads in numeric variables $\mathcal{V}_{\mathcal{L}}$

$\Longrightarrow$ model interferences in a relational numeric domain

Benefits and drawbacks:

- very precise:
  the only source of imprecision comes from the numeric domain
  partitioning possible, especially wrt. control information
- expressive: represents variable relations and intput/output relations
- costly: must apply a (possibly large) relation at each program step

Experiments with fully relational interferences [R. Monat]



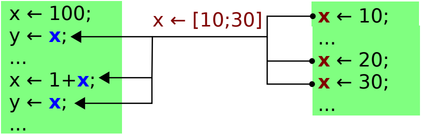
```
t_1
while z < 10000
  z ← z + 1
  if y < c then y ← y + 1
done
```

```
t_2
while z < 10000
  z ← z + 1
  if x < y then x ← x + 1
done
```

Experiments by R. Monat (not part of AstréeA)

Scalability in the number of threads (assuming fixed number of variables)

Lock-partitioning of simple interferences



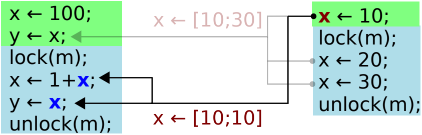
Without lock:

- all writes into x on the right affect all reads from x on the left
- interferences taken into account through expression injection

$$\begin{array}{lll} y \leftarrow x & \text{becomes} & y \leftarrow x \cup [10;30] \\ x \leftarrow 1 + x & \text{becomes} & x \leftarrow 1 + (x \cup [10;30]) \end{array}$$

  and then use a regular transfer function
- we detect the presence of data-races

Lock-partitioning of simple interferences



x ← 100;
y ← x;
lock(m);
x ← 1+**x**;
y ← **x**;
unlock(m);

x ← [10;30]

x ← [10;10]

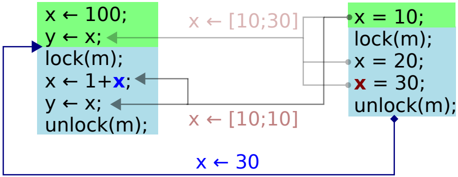
x ← 10;
lock(m);
x ← 20;
x ← 30;
unlock(m);

With locks:

- partition interferences wrt. locks held
- the first $y \leftarrow x$ is still $y \leftarrow x \cup [10;30]$
  the second $y \leftarrow x$ is now $y \leftarrow x \cup [10;10]$

  these interferences are caused by data-races

Lock-partitioning of simple interferences



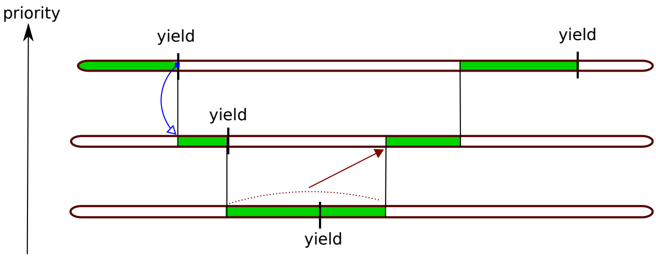
<u>With locks:</u>

- partition interferences wrt. locks held
- the first $y \leftarrow x$ is still $y \leftarrow x \cup [10;30]$
  the second $y \leftarrow x$ is now $y \leftarrow x \cup [10;10]$

  these interferences are caused by data-races
- the last write to x before unlock
  influences all reads from x between lock and update of x
  $\Longrightarrow$ we transfer the values of x from unlock to lock instruction

  these are well-synchronized interferences

Priority-based scheduling



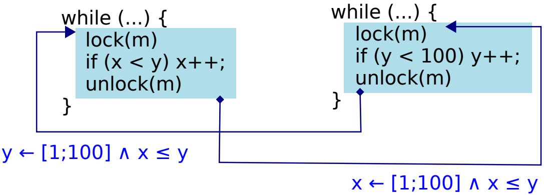
Real-time scheduling:

- priorities are strict (but possibly dynamic)
- a process can only be preempted by a process of strictly higher priority
- a process can block for an indeterminate amount of time (yield, lock)

Analysis: refined transfer of interference based on priority

- partition interferences wrt. thread and priority
  support for manual priority change, and for priority ceiling protocol
- higher priority processes inject state from yield into every point
- lower priority processes inject data-race interferences into yield

Relational lock invariants



```
while (...) {
  lock(m)
  if (x < y) x++;
  unlock(m)
}
```

```
while (...) {
  lock(m)
  if (y < 100) y++;
  unlock(m)
}
```

y ← [1;100] ∧ x ≤ y

x ← [1;100] ∧ x ≤ y

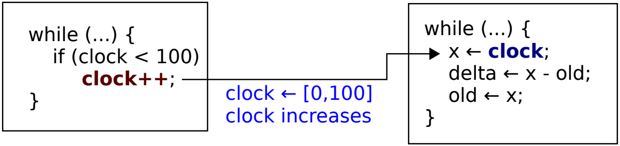
Idea: use (costly) relational interferences only at lock instructions

Rationale: locks often protect important, complex invariants

- data-race interference unchanged (here, ∅, as there is no data-race)
- well-synchronized interferences now carry:
  - a set of written values
  - a state property left invariant by the block
    (intersection of state at lock and at unlock point)

we don't keep input/output relation

Monotonicity interference



```
while (...) {
   if (clock < 100)
      clock++;
}
```

clock ← [0,100]
clock increases

```
while (...) {
   x ← clock;
   delta ← x - old;
   old ← x;
}
```

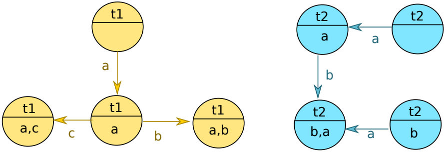
Idea: specialized domain to keep simple input/output relations

- clock is only increased (i.e., monotonic)
  - easy to infer (check all assignments)
  - easy to represent (one flow-insensitive flag per variable)
  - easy to exploit: new value of clock - old value of clock ≥ 0

very common pattern in control-command software

Deadlock checking

t_1	t_2
lock(a)	**lock**(a)
lock(c)	**lock**(b)
unlock(c)	**unlock**(a)
lock(b)	**lock**(a)
unlock(b)	**unlock**(a)
unlock(a)	**unlock**(b)

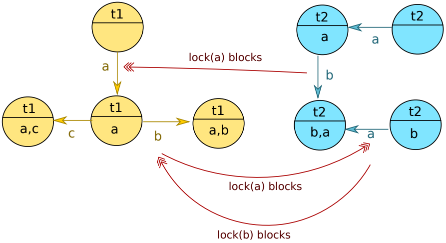


During the analysis, gather:

- all reachable mutex configurations: $R \subseteq \mathcal{T} \times \mathcal{P}(mutexes)$
- lock instructions from these configurations $R \times mutex$

Deadlock checking

```
t1             | t2
---------------+--------------
lock(a)        | lock(a)
   lock(c)     | lock(b)
   unlock(c)   | unlock(a)
   lock(b)     |    lock(a)
   unlock(b)   |    unlock(a)
unlock(a)      | unlock(b)
```



During the analysis, gather:

- all reachable mutex configurations: $R \subseteq \mathcal{T} \times \mathcal{P}(mutexes)$
- lock instructions from these configurations $R \times mutex$

After the analysis, construct a blocking graph between lock instructions

- $((t, m), \ell)$ blocks $((t', m'), \ell')$ if

  $t \neq t'$ and $m \cap m' = \emptyset$ (configurations not in mutual exclusion)

  $\ell \in m'$ (blocking lock)

A deadlock is a cycle in the blocking graph.

generalization to larger cycles, with more threads involved in a deadlock, is easy

Application to AstréeA

monotonicity domain	relational lock invariants	analysis time	memory	iterations	alarms
×	×	25h 26mn	22 GB	6	4616
✓	×	30h 30mn	24 GB	7	1100
✓	✓	110h 38mn	90 GB	7	1009

We only integrated into AstréeA a part of the proposed abstractions

Still scalability concerns with relational lock invariants (packing needed)

Reminder: embedded ARINC 653 C application with 15 threads, 1.6 Mlines

Weak memory consistency

Multi-core CPU and optimizing compilers enforce weak memory consistency

$\Longrightarrow$ an analysis sound only for sequential consistency
may not be sound for the actual memory model!

Soundness argument: on a per abstraction basis

- simple interferences: sound for reordering of independent R/W
  (includes PSO, TSO, traditional compiler optimization)
- monotonicity abstraction: sound for TSO & PSO
- relational lock invariants: sound for DRF guarantee
  if no data-race!
  (includes C, C++, Java)
- full relational interferences: sound for SC only

Conclusion

Conclusion

We proposed a static analysis framework for concurrent programs:

- sound for all interleavings
  and in some cases weakly consistent memories
- thread-modular
  scalable, able to reuse existing analyzers
- parameterized by abstract domains
  able to reuse existing domains
- constructed by abstraction of a complete method
  enable refinement to arbitrary precision
- presented several abstraction instances (relational, flow-sensitive)
- presented encouraging experimental results

Future work:

- specialization of state and interference domains for AstréeA
- bridge the gap between full relational and non-relational inteferences
- bridge the gap between arbitrary preemption and sequentializable
  flow-sensitive or even history-sensitive interference abstraction, e.g.: initialization