Precise Thread-Modular Abstract Interpretation of Concurrent Programs using Relational Interference Abstractions

Raphaël Monat [1] Antoine Miné [2]

[1] ENS de Lyon, Lyon, France

[2] LIP6, UPMC, Paris, France

- Concurrent architectures are more and more widespread.
- Concurrent programs are much more difficult to design and analyze than sequential ones. Testing is inefficient.

$\Rightarrow$ good setting to apply static analysis by abstract interpretation.

Concurrent program analysis

Lots of possible executions to analyze: combinatorial explosion.
Thread-modular approaches analyze threads separately, but lose precision in the process.

Goal

Can we perform a thread-modular analysis as precise as a non-modular one?

Contributions

- Thread-modular analysis.
- Highly parametrizable can be flow-sensitive and relational.
- Sweet spot: relational, partially flow-sensitive.
- Simple implementation scaling with a few variables but a lot of threads.
- Able to prove mutual exclusions, while still avoiding explosion in the number of control states.

Limitations

- Sequential Consistency.
- Fixed number of threads.
- Parametrization (partitioning).
- All variables are linked in the relational domain: no scalability in the number of variables.

Bakery mutual exclusion algorithm, two threads:

```
var c0:bool, c1:bool, n0:int, n1:int;
initial not(c0) and not(c1) and n0 == 0 and n1 == 0;
```

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

Non-modular analysis: iteration on the CFG product having n^2 nodes.
Thread-modular analysis: each thread is analyzed separately (multiple iterations).

In A. Miné, "Relational thread-modular static value analysis by abstract interpretation" (VMCAI'14):

- thread-modular, flow-insensitive and non-relational analysis
- refined using partially relational abstractions (such as the monotonicity abstract domain).

Here:

- Goal: be as precise as a non-modular approach, and see if performance is better.
- Concrete semantics is the same, only the abstractions used differ.
- Focus on short, complex programs where previous work is not precise.
- Fully-relational, partially flow-sensitive analysis.

Definition

Program state domain:

$$\mathcal{S} = \quad (\mathcal{T} \to \mathcal{L}) \quad \times \quad (\mathcal{V} \to \mathbb{Z})$$

Definition

Program state domain:

$$\mathcal{S} = \underbrace{(\mathcal{T} \to \mathcal{L})}_{\text{control state } \mathcal{C}} \times \underbrace{(\mathcal{V} \to \mathbb{Z})}_{\text{memory state } \mathcal{M}}$$

Definition

Program state domain:

$$\mathcal{S} = \underbrace{(\mathcal{T} \to \mathcal{L})}_{\text{control state } \mathcal{C}} \times \underbrace{(\mathcal{V} \to \mathbb{Z})}_{\text{memory state } \mathcal{M}}$$

Interference

An interference is a modification of a program state by a thread.

Definition

Program state domain:

$$\mathcal{S} = \underbrace{(\mathcal{T} \to \mathcal{L})}_{\text{control state } \mathcal{C}} \times \underbrace{(\mathcal{V} \to \mathbb{Z})}_{\text{memory state } \mathcal{M}}$$

Interference

An interference is a modification of a program state by a thread.

An interference is a relation between program states. Interference domain:

$$\mathcal{I} = \mathcal{S} \times \mathcal{S}$$

$((c_1, m_1), (c_2, m_2)) \in \mathcal{I}$: $c_1 \leadsto c_2$, $m_1 \leadsto m_2$.

Idea

1. Set $I = \emptyset$.
2. Analyze each thread in isolation.
3. Collect the interference created by each thread, store them in I.

Idea

1. Set $I = \emptyset$.
2. Analyze each thread in isolation.
3. Collect the interference created by each thread, store them in I.
4. Analyze each thread separately, using I.
5. Collect the interference, update I.
6. Go back to step 4 until the set of interference is stable.

Idea

1. Set $I = \emptyset$.
2. Analyze each thread in isolation.
3. Collect the interference created by each thread, store them in I.
4. Analyze each thread separately, using I.
5. Collect the interference, update I.
6. Go back to step 4 until the set of interference is stable.

Remark

This analysis is sound only when every interference has been uncovered.

First iteration:

```
var c0:bool, c1:bool, n0:int, n1:int;
initial not(c0) and not(c1) and n0 == 0 and n1 == 0;
```

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

First iteration:

```
var c0:bool, c1:bool, n0:int, n1:int;
initial not(c0) and not(c1) and n0 == 0 and n1 == 0;
```

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

$$c0 : false \rightsquigarrow true$$

First iteration:

```
var c0:bool, c1:bool, n0:int, n1:int;
initial not(c0) and not(c1) and n0 == 0 and n1 == 0;
```

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

$$c0 : false \rightsquigarrow true$$
$$n0 : 0 \rightsquigarrow 1 + n1$$

First iteration:

```
var c0:bool, c1:bool, n0:int, n1:int;
initial not(c0) and not(c1) and n0 == 0 and n1 == 0;
```

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

$$c0 : false \rightsquigarrow true$$
$$n0 : 0 \rightsquigarrow 1 + n1$$
$$c0 : true \rightsquigarrow false$$

First iteration:

```
var c0:bool, c1:bool, n0:int, n1:int;
initial not(c0) and not(c1) and n0 == 0 and n1 == 0;
```

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

$c0 : false \rightsquigarrow true$

$n0 : 0 \rightsquigarrow 1 + n1$

$c0 : true \rightsquigarrow false$

$n0 : 1 + n1 \rightsquigarrow 0$

First iteration:

```
var c0:bool, c1:bool, n0:int, n1:int;
initial not(c0) and not(c1) and n0 == 0 and n1 == 0;
```

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

$$c0 : false \rightsquigarrow true$$
$$n0 : 0 \rightsquigarrow 1 + n1$$
$$c0 : true \rightsquigarrow false$$
$$n0 : 1 + n1 \rightsquigarrow 0$$

$$c1 : false \rightsquigarrow true$$

First iteration:

```
var c0:bool, c1:bool, n0:int, n1:int;
initial not(c0) and not(c1) and n0 == 0 and n1 == 0;
```

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

$c0 : false \rightsquigarrow true$

$n0 : 0 \rightsquigarrow 1 + n1$

$c0 : true \rightsquigarrow false$

$n0 : 1 + n1 \rightsquigarrow 0$

$c1 : false \rightsquigarrow true$

$n1 : 0 \rightsquigarrow 1 + n0$

First iteration:

```
var c0:bool, c1:bool, n0:int, n1:int;
initial not(c0) and not(c1) and n0 == 0 and n1 == 0;
```

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

$$c0 : false \rightsquigarrow true$$
$$n0 : 0 \rightsquigarrow 1 + n1$$
$$c0 : true \rightsquigarrow false$$
$$n0 : 1 + n1 \rightsquigarrow 0$$

$$c1 : false \rightsquigarrow true$$
$$n1 : 0 \rightsquigarrow 1 + n0$$
$$c1 : true \rightsquigarrow false$$

First iteration:

```
var c0:bool, c1:bool, n0:int, n1:int;
initial not(c0) and not(c1) and n0 == 0 and n1 == 0;
```

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

$c0 : false \rightsquigarrow true$

$n0 : 0 \rightsquigarrow 1 + n1$

$c0 : true \rightsquigarrow false$

$n0 : 1 + n1 \rightsquigarrow 0$

$c1 : false \rightsquigarrow true$

$n1 : 0 \rightsquigarrow 1 + n0$

$c1 : true \rightsquigarrow false$

$n1 : 1 + n0 \rightsquigarrow 0$

Second iteration:

```
var c0:bool, c1:bool, n0:int, n1:int;
initial not(c0) and not(c1) and n0 == 0 and n1 == 0;
```

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

$c0 : false \leadsto true$

$n0 : 0 \leadsto 1 + n1$

$c0 : true \leadsto false$

$n0 : 1 + n1 \leadsto 0$

Now, `c0` and `(n0 > 0 and (n0 <= n1))` can be true.

Encoding

An interference is a relation in $\mathcal{P}(\mathcal{S} \times \mathcal{S})$: we can duplicate the dimensions using primed variables:
If thread t executes ${}^1 x = f(x)^2$, encode it as
$x' = f(x) \wedge \mathsf{pc}_t = 1 \wedge \mathsf{pc}'_t = 2$.

Remarks

- We used polyhedra and octagons to represent interference.
- $x' > x$ expresses that x can only increase through a thread interference.
- Control is also encoded using a variable: relations between control and data can be expressed.

State of a thread t: forget about control location of the other threads.

$$\alpha_{\mathcal{M}}(t) : \begin{cases} \mathcal{P}(\mathcal{C} \times \mathcal{M}) & \longrightarrow & \mathcal{L} \to \mathcal{P}(\mathcal{M}) \\ X & \longmapsto & \lambda L.\{e \mid (c, e) \in X \ \wedge c(t) = L\} \end{cases}$$

"Intra-thread flow sensitivity".

Interference: abstract control locations using $\alpha_{\mathcal{L}} : \mathcal{L} \to \mathcal{L}^\#$, i.e. the abstract control domain is $\mathcal{C}^\# = \mathcal{T} \to \mathcal{L}^\#$. Then, partitioning is performed, depending on the value of $\mathcal{L}^\#$.

Choice of $\mathcal{L}^\#$ strongly impacts performance:

- $\mathcal{L}^\# = \mathcal{L}$: back to the CFG product.
- $|\mathcal{L}^\#| = 1$: flow-insensitive approach.

In the mutual exclusions algorithms tested (Bakery, Peterson), partitioning before the critical section was sufficient.

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

In the mutual exclusions algorithms tested (Bakery, Peterson), partitioning before the critical section was sufficient.

```
while (true) do
  c0 = true
  n0 = 1 + max(n0, n1)
  c0 = false

  wait(not (c1))
  wait(not (n1 > 0 and (n1 < n0)))
  /* Critical Section */

  n0 = 0
done
```

```
while (true) do
  c1 = true
  n1 = 1 + max(n0, n1)
  c1 = false

  wait(not (c0))
  wait(not (n0 > 0 and (n0 <= n1)))
  /* Critical Section */

  n1 = 0
done
```

Batman, a BAsic Thread-Modular ANalyzer

- Written in Ocaml.
- Using either Apron[1] or BDDApron[2] libraries.
- Toy language similar to ConcurInterproc.

Performance comparison with ConcurInterproc[3], a non-thread-modular analyzer using the same libraries, similar input toy language, able to perform modular procedure analysis.

[1] Jeannet and Miné, "Apron: A library of numerical abstract domains for static analysis".

[2] Jeannet, BddApron, `http://tinyurl.com/bddapron`.

[3] Jeannet, "Relational interprocedural verification of concurrent programs".

```
while (true) do
  wait(f == 1)
  x = 1
  f = [1, 3]
done
```

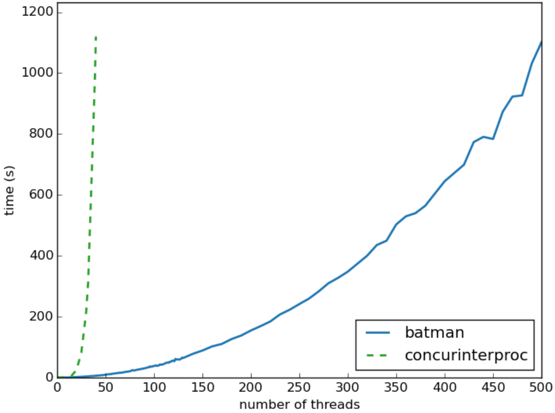
```
while (true) do
  wait(f == 2)
  x = 2
  f = [1, 3]
done
```

```
while (true) do
  wait(f == 3)
  x = 3
  f = [1, 3]
done
```

```
while (true) do
  wait(f == 1)
  x = 1
  f = [1, 3]
done
```

```
while (true) do
  wait(f == 2)
  x = 2
  f = [1, 3]
done
```

```
while (true) do
  wait(f == 3)
  x = 3
  f = [1, 3]
done
```



Algorithm	Number of threads	Time, polyhedron	Time, octagons
Peterson	2	0.67s	0.72s
Bakery	3	6.5s	27s
Bakery	4	49s	6m 33s
Bakery	5	5m 10s	49m 45s
Bakery	6	–	151m 8s
Bakery	7	–	12h

Timeout: 24h.

Model Checking:

- Thread-Modular approaches[4].
- Partial order reduction techniques[5].
- Bounded model checking[6].

Abstract Interpretation:

- Thread-modular, flow-sensitive approach using constraints[7].
- Duet tool[8], focuses on analyzing parameterized concurrent programs.

---

[4] Flanagan and Shaz Qadeer, “Thread-modular model checking”.

[5] Godefroid, “Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem”.

[6] S. Qadeer and Rehof, “Context-bounded model checking of concurrent software”.

[7] Kusano and Wang, “Flow-sensitive Composition of Thread-modular Abstract Interpretation”.

[8] Farzan and Kincaid., “Duet: Static analysis for unbounded parallelism”.

- Can control trade-off between analysis cost and precision.
- Relatively precise, avoid control-state explosion of non-modular approaches: we can use the minimal number of abstract control points to be sufficiently precise, and still be efficient.
- Mutual exclusion inferred!

Future work: analyze real-world programs!

- Add support of local variables.
- Use of packing techniques.
- POSIX Threads.
- Weakly consistent memories.