Static Analysis of Concurrent Programs

MPRI 2–6: Abstract Interpretation, application to verification and static analysis

Antoine Miné

Year 2018–2019

Course 16
12 February 2019
Introduction

Concurrent programming

**Idea:**
Decompose a program into a set of (loosely) interacting processes.

**Why concurrent programs?**

- **exploit** parallelism in current computers
  (multi-processors, multi-cores, hyper-threading)

  “Free lunch is over”
  change in Moore’s law  \((\times 2\) transistors every 2 years)

- **exploit** several computers
  (distributed computing)

- **ease** of programming
  (GUI, network code, reactive programs)
Introduction

Scope

In this course:  static thread model

- implicit communication through shared memory
- explicit communication through synchronisation primitives
- fixed number of threads (no dynamic creation of threads)
- numeric programs (real-valued variables)

Goal:  static analysis

- infer numeric program invariants
- parameterized by a choice of numeric abstract domains
- discover run-time errors (e.g., division by 0)
- discover data-races (unprotected accesses by concurrent threads)
- discover deadlocks (some threads block each other indefinitely)
Outline

- From sequential to concurrent abstract interpreters
  - alternate sequential semantics
    (a denotational semantics with run-time errors)
  - interleaving concurrent semantics (not scalable)
  - interference-based semantics (non-relational)
  - weakly consistent memory models
  - synchronization: data-races, priorities, locks, and deadlocks

- Abstract rely-guarantee
  - rely-guarantee proof method
  - a complete modular concrete semantics
  - examples of relational interference abstractions

- Application: the AstréeA analyzer
Simple structured numeric language

- finite set of (toplevel) threads: prog$_1$ to prog$_n$
- finite set of numeric program variables $V \in \mathbb{V}$
- finite set of statement locations $\ell \in \mathbb{L}$
- finite set of potential error locations $\omega \in \mathbb{\Omega}$

Structured language syntax

```latex
parprog ::= \ell prog_1 \mid \ldots \mid \ell prog_n \quad \text{(parallel composition)}

\ell prog ::= \ell V \leftarrow \text{exp} \quad \text{(assignment)}
| \ell \text{if exp} \triangleright 0 \text{ then } \ell prog \text{ fi}
| \ell \text{while exp} \triangleright 0 \text{ do } \ell prog \text{ done}
| \ell prog; \ell prog \quad \text{(sequence)}

exp ::= \ell V \mid [c_1, c_2] \mid - \text{exp} \mid \text{exp} \diamond \omega \text{ exp}

\ell V \in \mathbb{R} \cup \{+\infty, -\infty\}, \diamond, \triangleright \in \{+, -, \times, /\}, \triangleright \in \{\;=, <, \ldots\}\}
```

Course 16
Static Analysis of Concurrent Programs
Antoine Miné
Sequential semantics: reminders
Reminder: transition systems

**Transition system:**  \((\Sigma, \tau, \mathcal{I})\)

- \(\Sigma\): set of program states
- \(\tau \subseteq \Sigma \times \Sigma\): transition relation (we note \((\sigma, \sigma') \in \tau\) as \(\sigma \xrightarrow{\tau} \sigma'\))
- \(\mathcal{I} \subseteq \Sigma\): initial states
Sequential semantics: reminders

Reminder: traces of a transition system

Maximal trace semantics: \( M_\infty \in \mathcal{P}(\Sigma^\infty) \)

Total executions \( \sigma_0, \ldots, \sigma_n, \ldots \)

- starting in an initial state \( \sigma_0 \in I \) and either
- ending in a blocking state in \( B \overset{\text{def}}{=} \{ \sigma \mid \forall \sigma' : \sigma \not\rightarrow \tau \sigma' \} \)
- or infinite

\[
M_\infty \overset{\text{def}}{=} \{ \sigma_0, \ldots, \sigma_n \mid \sigma_0 \in I \land \sigma_n \in B \land \forall i < n : \sigma_i \rightarrow \tau \sigma_{i+1} \} \cup \\
\{ \sigma_0, \ldots, \sigma_n \ldots \mid \sigma_0 \in I \land \forall i < \omega : \sigma_i \rightarrow \tau \sigma_{i+1} \}
\]
Finite prefix trace semantics: \( \mathcal{T}_p \in \mathcal{P}(\Sigma^*) \)

Finite prefixes of executions (semantics of testing):
\[
\mathcal{T}_p \overset{\text{def}}{=} \{ \sigma_0, \ldots, \sigma_n \mid n \geq 0, \sigma_0 \in I, \forall i < n: \sigma_i \rightarrow_{\tau} \sigma_{i+1} \}
\]

Can prove safety properties, but not liveness (termination, etc.). Abstraction of the maximal trace semantics.

\[
\mathcal{T}_p = \text{lfp } F_p \text{ where } F_p(X) = I \cup \{ \sigma_0, \ldots, \sigma_{n+1} \mid \sigma_0, \ldots, \sigma_n \in X \land \sigma_n \rightarrow_{\tau} \sigma_{n+1} \}
\]
**Sequential semantics: reminders**

**Reminder: reachable state abstraction**

**Reachable state semantics:** \( \mathcal{R} \in \mathcal{P}(\Sigma) \)

Reachable states in any execution:
\[
\mathcal{R} \overset{\text{def}}{=} \{ \sigma \mid \exists n \geq 0, \sigma_0, \ldots, \sigma_n: \sigma_0 \in \mathcal{I}, \forall i < n: \sigma_i \rightarrow \tau \sigma_{i+1} \land \sigma = \sigma_n \}
\]

Can prove (non-)reachability, but not ordering.
Abstraction of the finite trace semantics.

\[
\mathcal{R} = \text{lfp} \ F_{\mathcal{R}} \text{ where } F_{\mathcal{R}}(X) = \mathcal{I} \cup \{ \sigma \mid \exists \sigma' \in X: \sigma' \rightarrow \tau \sigma \}
\]
States of a sequential program, with errors

Simple sequential numeric programs: \( \text{parprog ::= } \ell^i \text{ prog } \ell^x \).

**Program states:** \( \Sigma \overset{\text{def}}{=} (\mathcal{L} \times \mathcal{E}) \cup \Omega \)
- a control state in \( \mathcal{L} \), and
- either a memory state: an environment in \( \mathcal{E} \overset{\text{def}}{=} \mathbb{V} \rightarrow \mathbb{R} \)
- or an error state, in \( \Omega \)

**Initial states:**
start at the first control point \( \ell^i \) with variables set to 0:
\( \mathcal{I} \overset{\text{def}}{=} \{ (\ell^i, \lambda \mathbb{V}.0) \} \)

Note that \( \mathcal{P}(\Sigma) \simeq (\mathcal{L} \rightarrow \mathcal{P}(\mathcal{E})) \times \mathcal{P}(\Omega) \):
- a state property in \( \mathcal{P}(\mathcal{E}) \) at each program point in \( \mathcal{L} \)
- and a set of errors in \( \mathcal{P}(\Omega) \)
Expression semantics with errors

**Expression semantics:** \( E[\exp] : \mathcal{E} \rightarrow (\mathcal{P}(\mathbb{R}) \times \mathcal{P}(\Omega)) \)

- \( E[\ V\ ] \rho \quad \text{def} = \langle \{ \rho(V) \}, \emptyset \rangle \)
- \( E[[c_1, c_2]] \rho \quad \text{def} = \langle \{ x \in \mathbb{R} | c_1 \leq x \leq c_2 \}, \emptyset \rangle \)
- \( E[-e] \rho \quad \text{def} = \langle \{ -v | \in V \}, O \rangle \)
- \( E[e_1 \odotomega e_2] \rho \quad \text{def} = \langle \{ v_1 \odot v_2 | v_i \in V_i, \odot \neq / \lor v_2 \neq 0 \}, O_1 \cup O_2 \cup \{ \omega \text{ if } \odot = / \land 0 \in V_2 \} \rangle \)

- defined by structural induction on the syntax
- evaluates in an environment \( \rho \) to a set of values
- also returns a set of accumulated errors
  (here, only divisions by zero)
Sequential semantics: reminders

Semantics in denotational form

Input-output function $C[ prog ]$

\[
C[ prog ] : (\mathcal{P}(E) \times \mathcal{P}(\Omega)) \rightarrow (\mathcal{P}(E) \times \mathcal{P}(\Omega))
\]

$C[ X \leftarrow e ] \langle R, O \rangle \overset{\text{def}}{=} \langle \emptyset, O \rangle \sqcup \bigsqcup_{\rho \in R} \langle \{ \rho[X \mapsto v] \mid v \in V_\rho \}, O_\rho \rangle$

$C[ e \triangleright 0 ] \langle R, O \rangle \overset{\text{def}}{=} \langle \emptyset, O \rangle \sqcup \bigsqcup_{\rho \in R} \langle \{ \rho \mid \exists v \in V_\rho : v \triangleright 0 \}, O_\rho \rangle$

where $\langle V_\rho, O_\rho \rangle \overset{\text{def}}{=} E[ e ] \rho$

$C[ \text{if } e \triangleright 0 \text{ then } s \text{ fi} ] X \overset{\text{def}}{=} (C[ s ] \circ C[ e \triangleright 0 ]) X \sqcup C[ e \triangleright 0 ] X$

$C[ \text{while } e \triangleright 0 \text{ do } s \text{ done } ] X \overset{\text{def}}{=} C[ e \triangleright 0 ] (\text{lfp } \lambda Y. X \sqcup (C[ s ] \circ C[ e \triangleright 0 ]) Y)$

$C[ s_1 ; s_2 ] \overset{\text{def}}{=} C[ s_2 ] \circ C[ s_1 ]$

- mutate memory states in $E$, accumulate errors in $\Omega$
  $\sqcup$ is the element-wise $\cup$ in $\mathcal{P}(E) \times \mathcal{P}(\Omega)$
- structured: nested loops yield nested fixpoints
- big-step: forget information on intermediate locations $\ell$
Extend a numeric abstract domain $\mathcal{E}^#$ abstracting $\mathcal{P}(\mathcal{E})$ to $\mathcal{D}^# \overset{\text{def}}{=} \mathcal{E}^# \times \mathcal{P}(\Omega)$.

$$C^#\left[\text{prog}\right] : \mathcal{D}^# \rightarrow \mathcal{D}^#$$

$C^#\left[ X \leftarrow e \right] \langle R^#, O \rangle$ and $C^#\left[ e \triangleright 0 \right] \langle R^#, O \rangle$ are given

$$C^#\left[ \text{if } e \triangleright 0 \text{ then } s \text{ fi} \right] X^# \overset{\text{def}}{=} (C^#[s] \circ C^#\left[ e \triangleright 0 \right]) X^# \sqcup C^#\left[ e \triangleright 0 \right] X^#$$

$$C^#\left[ \text{while } e \triangleright 0 \text{ do } s \text{ done} \right] X^# \overset{\text{def}}{=} C^#\left[ e \triangleright 0 \right] (\lim \lambda Y^#. Y^# \triangleright (X^# \sqcup (C^#[s] \circ C^#\left[ e \triangleright 0 \right]) Y^#))$$

$$C^#\left[ s_1 ; s_2 \right] \overset{\text{def}}{=} C^#\left[ s_2 \right] \circ C^#\left[ s_1 \right]$$

- the abstract interpreter mimics an actual interpreter
Sequential semantics: reminders

Equational vs. denotational form

**Equational:**

\[
\begin{align*}
X_1 &= \top \\
X_2 &= F_2(X_1) \\
X_3 &= F_3(X_1) \\
X_4 &= F_4(X_3, X_4)
\end{align*}
\]

---

**Denotational:**

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

\[
\begin{align*}
C[\text{while } c \text{ do } b \text{ done}] X & \overset{\text{def}}{=} C[\neg c] (\text{lfp } \lambda Y. X \cup C[b] (C[c] Y)) \\
C[\text{if } c \text{ then } t \text{ fi}] X & \overset{\text{def}}{=} C[t] (C[c] X) \cup C[\neg c] X
\end{align*}
\]

- linear memory in program length
- flexible solving strategy
- flexible context sensitivity
- easy to adapt to concurrency, using a product of CFG

---

- linear memory in program depth
- fixed iteration strategy
- fixed context sensitivity
  (follows the program structure)
- no inductive definition of the product
  \(\rightarrow\) thread-modular analysis
Concurrent semantics
**Concurrent semantics**

**Multi-thread execution model**

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell_1$ while random do</td>
<td>$\ell_4$ while random do</td>
</tr>
<tr>
<td>$\ell_2$ if $x &lt; y$ then</td>
<td>$\ell_5$ if $y &lt; 100$ then</td>
</tr>
<tr>
<td>$\ell_3$ $x \leftarrow x + 1$</td>
<td>$\ell_6$ $y \leftarrow y + [1,3]$</td>
</tr>
</tbody>
</table>

**Execution model:**

- finite number of threads
- the memory is shared $(x, y)$
- each thread has its own program counter
- execution interleaves steps from threads $t_1$ and $t_2$
  (assignments and tests are assumed to be atomic)

$\implies$ we have the global invariant $0 \leq x \leq y \leq 102$
Labelled transition system: \((\Sigma, \mathcal{A}, \tau, \mathcal{I})\)

- \(\Sigma\): set of program states
- \(\mathcal{A}\): set of actions
- \(\tau \subseteq \Sigma \times \mathcal{A} \times \Sigma\): transition relation, we note \((\sigma, a, \sigma') \in \tau\) as \((\sigma \xrightarrow{a}_\tau \sigma')\)
- \(\mathcal{I} \subseteq \Sigma\): initial states

Labelled traces: sequences of states interspersed with actions
denoted as \(\sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \cdots \sigma_n \xrightarrow{a_n} \sigma_{n+1}\)
From concurrent programs to labelled transition systems

Notations:
- concurrent program: $\text{parprog ::= } \ell_1 \text{prog}_1 \ell_x || \cdots || \ell_n \text{prog}_n \ell_x$
- threads identifiers: $\mathbb{T} \overset{\text{def}}{=} \{ 1, \ldots, n \}$

Program states: $\Sigma \overset{\text{def}}{=} \left( (\mathbb{T} \rightarrow \mathcal{L}) \times \mathcal{E} \right) \cup \Omega$
- a control state $L(t) \in \mathcal{L}$ for each thread $t \in \mathbb{T}$ and
- a single shared memory state $\rho \in \mathcal{E}$
- or an error state $\omega \in \Omega$

Initial states:
threads start at their first control point $\ell^i_t$, variables are set to 0: $\mathcal{I} \overset{\text{def}}{=} \{ (\lambda t. \ell^i_t, \lambda V.0) \}$

Actions: thread identifiers: $\mathcal{A} \overset{\text{def}}{=} \mathbb{T}$
Transition relation: \( \tau \subseteq \Sigma \times A \times \Sigma \)

\[
\begin{align*}
(L, \rho) \xrightarrow{t}_\tau (L', \rho') & \iff (L(t), \rho) \xrightarrow{\tau[\text{prog}_t]} (L'(t), \rho') \land \\
& \quad \forall u \neq t: L(u) = L'(u)
\end{align*}
\]

\[
(L, \rho) \xrightarrow{t}_\tau \omega \iff (L(t), \rho) \xrightarrow{\tau[\text{prog}_t]} \omega
\]

- based on the transition relation of individual threads
  seen as sequential processes \( \text{prog}_t \):

  \( \tau[\text{prog}] \subseteq (L \times E) \times ((L \times E) \cup \Omega) \)

  - choose a thread \( t \) to run
  - update \( \rho \) and \( L(t) \)
  - leave \( L(u) \) intact for \( u \neq t \)

  (See course 3 for the full definition of \( \tau[\text{prog}] \).)

- each \( \sigma \rightarrow \sigma' \) in \( \tau[\text{prog}_t] \) leads to many transitions in \( \tau \)!
Interleaved trace semantics

Maximal and finite prefix trace semantics are as before:

**Blocking states:** \( \mathcal{B} \overset{\text{def}}{=} \{ \sigma | \forall \sigma': \forall t: \sigma \xrightarrow{t} \sigma' \} \)

**Maximal traces:** \( \mathcal{M}_\infty \) (finite or infinite)

\[
\mathcal{M}_\infty \overset{\text{def}}{=} \{ \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n | n \geq 0 \land \sigma_0 \in \mathcal{I} \land \sigma_n \in \mathcal{B} \land \forall i < n: \sigma_i \xrightarrow{t_i} \sigma_{i+1} \} \cup \\
\{ \sigma_0 \xrightarrow{t_0} \sigma_1 \cdots | n \geq 0 \land \sigma_0 \in \mathcal{I} \land \forall i < \omega: \sigma_i \xrightarrow{t_i} \sigma_{i+1} \}
\]

**Finite prefix traces:** \( \mathcal{T}_p \)

\[
\mathcal{T}_p \overset{\text{def}}{=} \{ \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n | n \geq 0 \land \sigma_0 \in \mathcal{I} \land \forall i < n: \sigma_i \xrightarrow{t_i} \sigma_{i+1} \}
\]

\[\mathcal{T}_p = \text{lfp} F_p \text{ where} \]
\[F_p(X) = \mathcal{I} \cup \{ \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_n} \sigma_{n+1} | n \geq 0 \land \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in X \land \sigma_n \xrightarrow{t_n} \sigma_{n+1} \} \]
Fairness conditions: avoid threads being denied to run

Given $\text{enabled}(\sigma, t) \overset{\text{def}}{=} \exists \sigma' \in \Sigma: \sigma \overset{t}{\rightarrow} \sigma'$,
an infinite trace $\sigma_0 \overset{t_0}{\rightarrow} \cdots \overset{t_n}{\rightarrow} \cdots$ is:

- weakly fair if $\forall t \in T$: 
  
  $(\exists i: \forall j \geq i: \text{enabled}(\sigma_j, t)) \implies (\forall i: \exists j \geq i: a_j = t)$

  (no thread can be continuously enabled without running)

- strongly fair if $\forall t \in T$: 
  
  $(\forall i: \exists j \geq i: \text{enabled}(\sigma_j, t)) \implies (\forall i: \exists j \geq i: a_j = t)$

  (no thread can be infinitely often enabled without running)

Proofs under fairness conditions: given:

- the maximal traces $\mathcal{M}_\infty$ of a program
- a property $X$ to prove (as a set of traces)
- the set $F$ of all (weakly or strongly) fair and of finite traces

$\implies$ prove $\mathcal{M}_\infty \cap F \subseteq X$ instead of $\mathcal{M}_\infty \subseteq X$
Fairness (cont.)

Example: while $x \geq 0$ do $x \leftarrow x + 1$ done || $x \leftarrow -1$

- may not terminate without fairness
- always terminates under weak and strong fairness

Finite prefix trace abstraction

$M_\infty \cap F \subseteq X$ is abstracted into testing $\alpha_\preceq (M_\infty \cap F) \subseteq \alpha_\preceq (X)$

for all fairness conditions $F$, $\alpha_\preceq (M_\infty \cap F) = \alpha_\preceq (M_\infty) = T_p$

$\implies$ fairness-dependent properties cannot be proved with finite prefixes only

In the following, we ignore fairness conditions.

(see [Cous85])
Equational state semantics

State abstraction $\mathcal{R}$: as before

- $\mathcal{R} \overset{\text{def}}{=} \{ \sigma \mid \exists n \geq 0, \sigma_0 \xrightarrow{t_0} \cdots \sigma_n : \sigma_0 \in I \ \forall i < n : \sigma_i \xrightarrow{t_i} \sigma_{i+1} \land \sigma = \sigma_n \}$
- $\mathcal{R} = \alpha_p(T_p)$ where $\alpha_p(X) \overset{\text{def}}{=} \{ \sigma \mid \exists n \geq 0, \sigma_0 \xrightarrow{t_0} \cdots \sigma_n \in X : \sigma = \sigma_n \}$
- $\mathcal{R} = \text{lfp } F_{\mathcal{R}}$ where $F_{\mathcal{R}}(X) = I \cup \{ \sigma \mid \exists \sigma' \in X, t \in \mathcal{T} : \sigma' \xrightarrow{t} \sigma \}$

Equational form: (without handling errors in $\Omega$)

- for each $L \in \mathcal{T} \rightarrow \mathcal{L}$, a variable $\mathcal{X}_L$ with value in $\mathcal{E}$
- equations are derived from thread equations $eq(prog_t)$ as:

  $\mathcal{X}_{L_1} = \bigcup_{t \in \mathcal{T}} \{ F(\mathcal{X}_{L_2}, \ldots, \mathcal{X}_{L_N}) \mid
  \exists (\mathcal{X}_{\ell_1} = F(\mathcal{X}_{\ell_2}, \ldots, \mathcal{X}_{\ell_N})) \in eq(prog_t):
  \forall i \leq N : L_i(t) = \ell_i, \forall u \neq t : L_i(u) = L_1(u) \}$

  Join with $\cup$ equations from $eq(prog_t)$ updating a single thread $t \in \mathcal{T}$.

  (See course 3 for the full definition of $eq(prog)$.)
Product of control-flow graphs:

- control state = tuple of program points
  \[\Rightarrow\] combinatorial explosion of abstract states

- transfer functions are duplicated
Equational state semantics (example)

Example: inferring $0 \leq x \leq y \leq 102$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell_1$ while random do $\ell_2$ if $x &lt; y$ then $\ell_3 x \leftarrow x + 1$</td>
<td>$\ell_4$ while random do $\ell_5$ if $y &lt; 100$ then $\ell_6 y \leftarrow y + [1,3]$</td>
</tr>
</tbody>
</table>

Equation system:

\[
\begin{align*}
\mathcal{X}_{1,4} &= \mathcal{I} \\
\mathcal{X}_{2,4} &= \mathcal{X}_{1,4} \cup C[ x \geq y ] \mathcal{X}_{2,4} \cup C[ x \leftarrow x + 1 ] \mathcal{X}_{3,4} \\
\mathcal{X}_{3,4} &= C[ x < y ] \mathcal{X}_{2,4} \\
\mathcal{X}_{1,5} &= \mathcal{X}_{1,4} \cup C[ y \geq 100 ] \mathcal{X}_{1,5} \cup C[ y \leftarrow y + [1,3] ] \mathcal{X}_{1,6} \\
\mathcal{X}_{2,5} &= \mathcal{X}_{1,5} \cup C[ x \geq y ] \mathcal{X}_{2,5} \cup C[ x \leftarrow x + 1 ] \mathcal{X}_{3,5} \cup C[ y \geq 100 ] \mathcal{X}_{2,5} \cup C[ y \leftarrow y + [1,3] ] \mathcal{X}_{2,6} \\
\mathcal{X}_{3,5} &= C[ x < y ] \mathcal{X}_{2,5} \cup \mathcal{X}_{3,4} \cup C[ y \geq 100 ] \mathcal{X}_{3,5} \cup C[ y \leftarrow y + [1,3] ] \mathcal{X}_{3,6} \\
\mathcal{X}_{1,6} &= C[ y \geq 100 ] \mathcal{X}_{1,5} \\
\mathcal{X}_{2,6} &= \mathcal{X}_{1,6} \cup C[ x \geq y ] \mathcal{X}_{2,6} \cup C[ x \leftarrow x + 1 ] \mathcal{X}_{3,6} \cup C[ y < 100 ] \mathcal{X}_{2,5} \\
\mathcal{X}_{3,6} &= C[ x < y ] \mathcal{X}_{2,6} \cup C[ y < 100 ] \mathcal{X}_{3,5}
\end{align*}
\]
**Concurrent semantics**

**Equational state semantics (example)**

**Example: inferring** $0 \leq x \leq y \leq 102$

<table>
<thead>
<tr>
<th></th>
<th>$t_1$</th>
<th></th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell_1$</td>
<td>while random do</td>
<td>$\ell_4$</td>
<td>while random do</td>
</tr>
<tr>
<td>$\ell_2$</td>
<td>if $x &lt; y$ then</td>
<td>$\ell_5$</td>
<td>if $y &lt; 100$ then</td>
</tr>
<tr>
<td>$\ell_3$</td>
<td>$x \leftarrow x + 1$</td>
<td>$\ell_6$</td>
<td>$y \leftarrow y + [1,3]$</td>
</tr>
</tbody>
</table>

**Pros:**
- easy to construct
- easy to further abstract in an abstract domain $E^\#$

**Cons:**
- explosion of the number of variables and equations
- explosion of the size of equations $\implies$ efficiency issues
- the equation system does not reflect the program structure (not defined by induction on the concurrent program)
Wish-list

We would like to:

- keep information attached to **syntactic** program locations
  (control points in $\mathcal{L}$, not control point tuples in $\mathbb{T} \to \mathcal{L}$)

- be able to **abstract away control information**
  (precision/cost trade-off control)

- avoid **duplicating** thread instructions

- have a computation structure based on the **program syntax**
  (denotational style)

**Ideally:** thread-modular denotational-style semantics

(analyze each thread independently by induction on its syntax)
Simple interference semantics
Thread-modular analysis with simple interferences

Principle:
- analyze each thread in isolation
Thread-modular analysis with simple interferences

Principle:

- analyze each thread in isolation
- gather the values written into each variable by each thread
  \[\Rightarrow\] so-called interferences
  suitably abstracted in an abstract domain, such as intervals
Thread-modular analysis with simple interferences

Principle:
- analyze each thread in isolation
- gather the values written into each variable by each thread
  \(\Rightarrow\) 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

Principle:

- analyze each thread in isolation
- gather the values written into each variable by each thread
  $\implies$ 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
  $\implies$ one more level of fixpoint iteration
Simple interference semantics

Example

$t_1$

$\ell_1$ while random do
$\ell_2$ if $x < y$ then
$\ell_3$ $x \leftarrow x + 1$

$t_2$

$\ell_4$ while random do
$\ell_5$ if $y < 100$ then
$\ell_6$ $y \leftarrow y + [1,3]$
Example

Analysis of \( t_1 \) in isolation

(1): \( x = y = 0 \) \( \mathcal{X}_1 = I \)
(2): \( x = y = 0 \) \( \mathcal{X}_2 = \mathcal{X}_1 \cup \mathcal{C}[x \leftarrow x + 1] \mathcal{X}_3 \cup \mathcal{C}[x \geq y] \mathcal{X}_2 \)
(3): \( \perp \) \( \mathcal{X}_3 = \mathcal{C}[x < y] \mathcal{X}_2 \)
Simple interference semantics

Example

\[ t_1 \]
\[ \ell_1 \text{ while random do } \]
\[ \ell_2 \text{ if } x < y \text{ then } \]
\[ \ell_3 \quad x \leftarrow x + 1 \]

\[ t_2 \]
\[ \ell_4 \text{ while random do } \]
\[ \ell_5 \text{ if } y < 100 \text{ then } \]
\[ \ell_6 \quad y \leftarrow y + [1,3] \]

Analysis of \( t_2 \) in isolation

(4): \( x = y = 0 \)
\( \chi_4 = \emptyset \)

(5): \( x = 0, \ y \in [0,102] \)
\( \chi_5 = \chi_4 \cup \mathbb{C}[y \leftarrow y + [1,3]] \chi_6 \cup \mathbb{C}[y \geq 100] \chi_5 \)

(6): \( x = 0, \ y \in [0,99] \)
\( \chi_6 = \mathbb{C}[y < 100] \chi_5 \)

output interferences: \( y \leftarrow [1,102] \)
Example

Re-analysis of $t_1$ with interferences from $t_2$

input interferences: $y \leftarrow [1, 102]$

(1): $x = y = 0$

$x_1 = \emptyset$

(2): $x \in [0, 102], y = 0$

$x_2 = x_{1a} \cup C[x \leftarrow x + 1] \cup x \geq (y | [1, 102]) \cup x_2$

(3): $x \in [0, 102], y = 0$

$x_3 = C[x < (y | [1, 102])] \cup x_2$

output interferences: $x \leftarrow [1, 102]$

subsequent re-analyses are identical (fixpoint reached)
**Example**

\[
\ell_1 \quad \text{while random do} \\
\ell_2 \quad \text{if } x < y \text{ then} \\
\ell_3 \quad x \leftarrow x + 1 \\
\ell_4 \quad \text{while random do} \\
\ell_5 \quad \text{if } y < 100 \text{ then} \\
\ell_6 \quad y \leftarrow y + [1, 3]
\]

**Derived abstract analysis:**

- similar to a **sequential** program analysis, but iterated  
  (can be parameterized by arbitrary abstract domains)

- **efficient** (few reanalyses are required in practice)

- interferences are **non-relational** and **flow-insensitive**  
  (limit inherited from the concrete semantics)

**Limitation:**

We get \( x, y \in [0, 102] \); we don't get that \( x \leq y \)  

simplistic view of thread interferences (volatile variables)  

based on an **incomplete** concrete semantics!
Interferences in \( \mathcal{I} \) \( \overset{\text{def}}{=} T \times V \times R \)

\( \langle t, X, v \rangle \) means: \( t \) can store the value \( v \) into the variable \( X \)

We define the analysis of a thread \( t \) with respect to a set of interferences \( I \subseteq \mathcal{I} \).

**Expressions with interference**: for thread \( t \)

\( E_t[\text{exp}] : (\mathcal{E} \times \mathcal{P}(\mathcal{I})) \rightarrow (\mathcal{P}(R) \times \mathcal{P}(\Omega)) \)

- Apply interferences to read variables:
  \( E_t[\text{X}] \langle \rho, I \rangle \overset{\text{def}}{=} \langle \{ \rho(X) \} \cup \{ v | \exists u \neq t: \langle u, X, v \rangle \in I \}, \emptyset \rangle \)

- Pass recursively \( I \) down to sub-expressions:
  \( E_t[\text{e}] \langle \rho, I \rangle \overset{\text{def}}{=} \langle \text{let } \langle V, O \rangle = E_t[\text{e}] \langle \rho, I \rangle \text{ in } \langle \{ -v \mid v \in V \}, O \rangle \rangle \)

...
Denotational semantics with interferences (cont.)

**Statements with interference:** for thread $t$

\[
C_t[\text{prog}] : (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega) \times \mathcal{P}(\mathcal{I})) \rightarrow (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega) \times \mathcal{P}(\mathcal{I}))
\]

- pass interferences to expressions
- collect new interferences due to assignments
- accumulate interferences from inner statements

\[
C_t[ X \leftarrow e ] \langle R, O, I \rangle \overset{\text{def}}{=} \\
\langle \emptyset, O, I \rangle \sqcup \bigsqcup_{\rho \in R} \langle \{ \rho[X \mapsto v] \mid v \in V_\rho \}, O_\rho, \{ \langle t, X, v \rangle \mid v \in V_\rho \} \rangle
\]

\[
C_t[ s_1; s_2 ] \overset{\text{def}}{=} C_t[ s_2 ] \circ C_t[ s_1 ]
\]

\[
\vdots
\]

noting \( \langle V_\rho, O_\rho \rangle \overset{\text{def}}{=} E_t[ e ] \langle \rho, I \rangle \)

\(\sqcup\) is now the element-wise \(\cup\) in \(\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega) \times \mathcal{P}(\mathcal{I})\)
Program semantics: \( P[\text{parprog}] \subseteq \Omega \)

Given \( \text{parprog} ::= \text{prog}_1 \parallel \cdots \parallel \text{prog}_n \), we compute:

\[
P[\text{parprog}] \overset{\text{def}}{=} \left( \text{lfp } \lambda \langle O, I \rangle. \bigsqcup_{t \in T} [C_t[\text{prog}_t]]\langle \mathcal{E}_0, \emptyset, I \rangle\right)_{\Omega, \emptyset}
\]

- each thread analysis starts in an initial environment set \( \mathcal{E}_0 \overset{\text{def}}{=} \{ \lambda V.0 \} \)
- \( [X]_{\Omega, \emptyset} \) projects \( X \in \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega) \times \mathcal{P}(\mathbb{I}) \) on \( \mathcal{P}(\Omega) \times \mathcal{P}(\mathbb{I}) \) and interferences and errors from all threads are joined (the output environments are ignored)
- \( P[\text{parprog}] \) only outputs the set of possible run-time errors
**Abstract interferences** \( \#^\# \)

\[ \mathcal{P}(\emptyset) \overset{\text{def}}{=} \mathcal{P}(T \times V \times \mathbb{R}) \text{ is abstracted as } \emptyset^\# \overset{\text{def}}{=} (T \times V) \to R^\# \]

where \( R^\# \) abstracts \( \mathcal{P}(\mathbb{R}) \) (e.g. intervals)

**Abstract semantics with interferences** \( C_t^\#[s] \)

derived from \( C^\#[s] \) in a generic way:

**Example:** \( C_t^\#[X \leftarrow e] \langle R^\#, \Omega, I^\# \rangle \)

- for each \( Y \) in \( e \), get its interference \( Y_{R}^\# = \bigcup_{R}^\# \{ I^\#(u, Y) \mid u \neq t \} \)
- if \( Y_{R}^\# \neq \bot_{R}^\# \), replace \( Y \) in \( e \) with \( \text{get}(Y, R^\#) \cup_{R}^\# Y_{R}^\# \)
  (where \( \text{get}(Y, R^\#) \) extracts the abstract values in \( R^\# \) of a variable \( Y \) from \( R^\# \in \mathcal{E}^\# \))
- compute \( \langle R'^\#, O' \rangle = C^\#[e] \langle R^\#, O \rangle \)
- enrich \( I^\#(t, X) \) with \( \text{get}(X, R'^\#) \)
Simple interference semantics

Static analysis with interferences

**Abstract analysis**

\[
\text{P}^\#[[\text{parprog}]] \overset{\text{def}}{=} \lim \lambda \langle O, I^\# \rangle \cdot \langle O, I^\# \rangle \nabla \bigcup_{t \in T} \left[ \text{C}^\#[[\text{prog}_t]] \langle \mathcal{E}_0^\#, \emptyset, I^\# \rangle \right] \Omega, \emptyset^\# \right] \Omega
\]

- **effective** analysis by **structural induction**
- termination ensured by a **widening**
- parameterized by a choice of abstract domains \( \mathcal{R}^\#, \mathcal{E}^\# \)

- **interferences** are **flow-insensitive** and **non-relational** in \( \mathcal{R}^\# \)
- **thread analysis** remains **flow-sensitive** and **relational** in \( \mathcal{E}^\# \)

(reminder: \( [X]_{\Omega}, [Y]_{\Omega, \emptyset^\#} \) keep only \( X \)'s component in \( \Omega \), \( Y \)'s components in \( \Omega \) and \( \emptyset^\# \))
Control paths

**atomic** ::= $X ← exp | exp \otimes 0$

### Control paths

\[ \pi : prog \rightarrow \mathcal{P}(atomic^*) \]

\[ \pi(X ← e) \overset{\text{def}}{=} \{ X ← e \} \]

\[ \pi(\text{if } e \otimes 0 \text{ then } s \text{ fi}) \overset{\text{def}}{=} (\{ e \otimes 0 \} \cdot \pi(s)) \cup \{ e \not\otimes 0 \} \]

\[ \pi(\text{while } e \otimes 0 \text{ do } s \text{ done}) \overset{\text{def}}{=} (\bigcup_{i \geq 0}(\{ e \otimes 0 \} \cdot \pi(s))^i) \cdot \{ e \not\otimes 0 \} \]

\[ \pi(s_1; s_2) \overset{\text{def}}{=} \pi(s_1) \cdot \pi(s_2) \]

\[ \pi(prog) \] is a (generally infinite) set of finite control paths
Path-based concrete semantics of sequential programs

Join-over-all-path semantics

\[ \Pi[P]: (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)) \rightarrow (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)) \quad P \subseteq \text{atomic}^* \]

\[ \Pi[P]\langle R, O \rangle \overset{\text{def}}{=} \bigsqcup_{s_1\ldots s_n \in P} (C[s_n] \circ \cdots \circ C[s_1])\langle R, O \rangle \]

Semantic equivalence

\[ C[\text{prog}] = \Pi[\pi(\text{prog})] \]

(not true in the abstract)

Advantages:
- easily extended to concurrent programs (path interleavings)
- able to model program transformations (weak memory models)

Course 16 Static Analysis of Concurrent Programs Antoine Miné p. 38 / 87
**Concurrent control paths**

\[ \pi_* \overset{\text{def}}{=} \{ \text{interleavings of } \pi(\text{prog}_t), \ t \in T \} \]

\[ = \{ p \in \text{atomic}^* \mid \forall t \in T, \ proj_t(p) \in \pi(\text{prog}_t) \} \]

**Interleaving program semantics**

\[ P_*[\text{parprog}] \overset{\text{def}}{=} \prod[\pi_*]\langle E_0, \emptyset \rangle_\Omega \]

\((proj_t(p) \text{ keeps only the atomic statement in } p \text{ coming from thread } t)\)

\((\simeq \text{sequentially consistent executions [Lamport 79]})\)

**Issues:**

- too many paths to consider exhaustively
- no induction structure to iterate on
  \(\implies\) abstract as a denotational semantics
- unrealistic assumptions on granularity and memory consistency
Soundness of the interference semantics

Soundness theorem

\[ P_\ast[\text{parprog}] \subseteq P[\text{parprog}] \]

Proof sketch:

- define \( \Pi_t[P][X] \overset{\text{def}}{=} \bigsqcup \{ C_t[s_1;\ldots;s_n] X | s_1 \cdot \ldots \cdot s_n \in P \} \), then \( \Pi_t[\pi(s)] = C_t[s] \);

- given the interference fixpoint \( I \subseteq \) from \( P[\text{parprog}] \), prove by recurrence on the length of \( p \in \pi_\ast \) that:
  - \( \forall t \in T, \forall \rho \in [\Pi[P][\langle E_0, \emptyset \rangle]], \exists \rho' \in [\Pi_t[\text{proj}_t(p)][\langle E_0, \emptyset, I \rangle]] \) such that \( \forall X \in V, \rho(X) = \rho'(X) \) or \( \langle u, X, \rho(X) \rangle \in I \) for some \( u \neq t \).
  - \( [\Pi[P][\langle E_0, \emptyset \rangle]] \subseteq \bigcup_{t \in T} [\Pi_t[\text{proj}_t(p)][\langle E_0, \emptyset, I \rangle]] \)

Note: sound but not complete
Weakly consistent memories
Issues with weak consistency

(simplified Dekker mutual exclusion algorithm)

\( S_1 \) and \( S_2 \) cannot execute simultaneously.
Issues with weak consistency

(simplified Dekker mutual exclusion algorithm)

$S_1$ and $S_2$ can execute simultaneously.
Not a sequentially consistent behavior!

Caused by:

- write FIFOs, caches, distributed memory
- hardware or compiler optimizations, transformations
- ...

behavior accepted by Java [Mans05]
**Total Store Ordering:** model for intel x86

- each thread writes to a FIFO queue
- queues are flushed non-deterministically to the shared memory
- a thread reads back from its queue if possible and from shared memory otherwise
Out of thin air principle

(example from causality test case #4 for Java by Pugh et al.)

We should not have $R_1 = 42$. 
Weakly consistent memories

Out of thin air principle

Original program

| R1 ← X; | R ← Y; |
| Y ← R1 | X ← R2 |

“Optimized” program

| Y ← 42; |
| R1 ← X; |
| Y ← R1 | X ← R2 |

(example from causality test case #4 for Java by Pugh et al.)

We should not have $R_1 = 42$.

Possible if we allow speculative writes!

$\rightarrow$ we disallow this kind of program transformations.

(also forbidden in Java)
Atomicity and granularity

We assumed that assignments are atomic…
Atomicity and granularity

We assumed that assignments are atomic... but that may not be the case.

The second program admits more behaviors e.g.: $X = 1$ at the end of the program

[Reyn04]
Path-based definition of weak consistency

Acceptable control path transformations: \( p \rightsquigarrow q \)

only reduce interferences and errors

- **Reordering:** \( X_1 \leftarrow e_1 \cdot X_2 \leftarrow e_2 \rightsquigarrow X_2 \leftarrow e_2 \cdot X_1 \leftarrow e_1 \)
  
  (if \( X_1 \notin \text{var}(e_2) \), \( X_2 \notin \text{var}(e_1) \), and \( e_1 \) does not stop the program)

- **Propagation:** \( X \leftarrow e \cdot s \rightsquigarrow X \leftarrow e \cdot s[e/X] \)
  
  (if \( X \notin \text{var}(e) \), \( \text{var}(e) \) are thread-local, and \( e \) is deterministic)

- **Factorization:** \( s_1 \cdot \ldots \cdot s_n \rightsquigarrow X \leftarrow e \cdot s_1[X/e] \cdot \ldots \cdot s_n[X/e] \)
  
  (if \( X \) is fresh, \( \forall i, \text{var}(e) \cap \text{lval}(s_i) = \emptyset \), and \( e \) has no error)

- **Decomposition:** \( X \leftarrow e_1 + e_2 \rightsquigarrow T \leftarrow e_1 \cdot X \leftarrow T + e_2 \)
  
  (change of granularity)

  
  \[ \ldots \]

but **NOT:**

- “out-of-thin-air” writes: \( X \leftarrow e \rightsquigarrow X \leftarrow 42 \cdot X \leftarrow e \)
Weakly consistent memories

Soundness of the interference semantics

Interleaving semantics of transformed programs $P'[\text{parprog}]$

- $\pi'(s) \overset{\text{def}}{=} \{ p \mid \exists p' \in \pi(s) : p' \rightsquigarrow^* p \}$
- $\pi_*$ $\overset{\text{def}}{=} \{ \text{interleavings of } \pi'(\text{prog}_t), t \in T \}$
- $P'[\text{parprog}] \overset{\text{def}}{=} [\prod_{\pi_*} \langle E_0, \emptyset \rangle]_{\Omega}$

Soundness theorem

$P_*'[\text{parprog}] \subseteq P[\text{parprog}]$

$\implies$ the interference semantics is sound wrt. weakly consistent memories and changes of granularity
Locks
Synchronization primitives

\[\text{prog} ::= \text{lock}(m) \mid \text{unlock}(m)\]

\(m \in M:\) finite set of non-recursive mutexes

Scheduling

mutexes ensure mutual exclusion

at each time, each mutex can be locked by a single thread
Mutual exclusion

We use a refinement of the simple interference semantics by partitioning wrt. an abstract local view of the scheduler $\mathbb{C}$

- $\mathcal{E} \rightsquigarrow \mathcal{E} \times \mathbb{C}$,
  $\mathcal{E}^\# \rightsquigarrow \mathbb{C} \rightarrow \mathcal{E}^\#

- $I \overset{\text{def}}{=} T \times V \times R \rightsquigarrow I \overset{\text{def}}{=} T \times \mathbb{C} \times V \times R$,
  $I^\# \overset{\text{def}}{=} (T \times V) \rightarrow \mathcal{R}^\# \rightsquigarrow I^\# \overset{\text{def}}{=} (T \times \mathbb{C} \times V) \rightarrow \mathcal{R}^\#

$\mathbb{C} \overset{\text{def}}{=} \mathbb{C}_{\text{race}} \cup \mathbb{C}_{\text{sync}}$ separates data-race writes $\mathbb{C}_{\text{race}}$

and well-synchronized writes $\mathbb{C}_{\text{sync}}$
Data-race effects \( C_{race} \cong \mathcal{P}(M) \)

Across read / write not protected by a mutex.

Partition wrt. mutexes \( M \subseteq \mathcal{M} \) held by the current thread \( t \).

\[
\begin{align*}
C_t[ X \leftarrow e ] \langle \rho, M, I \rangle & \text{ adds } \{ \langle t, M, X, v \rangle \mid v \in E_t[ X ] \langle \rho, M, I \rangle \} \text{ to } I \\
E_t[ X ] \langle \rho, M, I \rangle &= \{ \rho(X) \} \cup \{ v \mid \langle t', M', X, v \rangle \in I, t \neq t', M \cap M' = \emptyset \}
\end{align*}
\]

Bonus: we get a data-race analysis for free!
Mutual exclusion

Well-synchronized effects \( \mathcal{C}_{sync} \simeq M \times \mathcal{P}(M) \)

- last write before unlock affects first read after lock
- partition interferences wrt. a protecting mutex \( m \) (and \( M \))
- \( \mathcal{C}_t[unlock(m)] \langle \rho, M, I \rangle \) stores \( \rho(X) \) into \( I \)
- \( \mathcal{C}_t[lock(m)] \langle \rho, M, I \rangle \) imports values form \( I \) into \( \rho \)
- imprecision: non-relational, largely flow-insensitive

\[ \implies \mathcal{C} \simeq \mathcal{P}(M) \times (\{\text{data} \setminus \text{race}\} \cup M) \]
### Example analysis

#### abstract consumer/producer

<table>
<thead>
<tr>
<th>$N$ consumers</th>
<th>$N$ producers</th>
</tr>
</thead>
<tbody>
<tr>
<td>while random do</td>
<td></td>
</tr>
<tr>
<td>lock(m); $\ell_1$</td>
<td></td>
</tr>
<tr>
<td>if $X &gt; 0$ then $\ell_2$ $X \leftarrow X - 1$ fi;</td>
<td></td>
</tr>
<tr>
<td>unlock(m); $\ell_3$ $Y \leftarrow X$</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>while random do</td>
</tr>
<tr>
<td>lock(m);</td>
<td></td>
</tr>
<tr>
<td>$X \leftarrow X + 1$;</td>
<td></td>
</tr>
<tr>
<td>if $X &gt; 100$ then $X \leftarrow 100$ fi;</td>
<td></td>
</tr>
<tr>
<td>unlock(m)</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td></td>
</tr>
</tbody>
</table>

Assuming we have several ($N$) producers and consumers:

- **no data-race interference** *(proof of the absence of data-race)*
- **well-synchronized interferences:**
  - *consumer*: $x \leftarrow [0, 99]$
  - *producer*: $x \leftarrow [1, 100]$

$\implies$ we get that $x \in [0, 100]$

*(without locks, if $N > 1$, our concrete semantics cannot bound $x$!)*
Locks and priorities

priority-based critical sections

<table>
<thead>
<tr>
<th>high thread</th>
<th>low thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>L ← isLocked(m);</td>
<td>lock(m);</td>
</tr>
<tr>
<td>if L = 0 then</td>
<td>Z ← Y;</td>
</tr>
<tr>
<td>Y ← Y+1;</td>
<td>Y ← 0;</td>
</tr>
<tr>
<td>yield()</td>
<td>unlock(m)</td>
</tr>
</tbody>
</table>

Real-time scheduling

- only the highest priority unblocked thread can run
- lock and yield may block
- yielding threads wake up non-deterministically
  preemption lower-priority threads
- explicit synchronisation enforces memory consistency
  prevents data races
Locks and priorities

priority-based critical sections

<table>
<thead>
<tr>
<th>high thread</th>
<th>low thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \leftarrow \text{isLocked}(m)$;</td>
<td>$\text{lock}(m)$;</td>
</tr>
<tr>
<td>if $L = 0$ then</td>
<td>$Z \leftarrow Y$;</td>
</tr>
<tr>
<td>$Y \leftarrow Y+1$;</td>
<td>$Y \leftarrow 0$;</td>
</tr>
<tr>
<td>$\text{yield}()$</td>
<td>$\text{unlock}(m)$</td>
</tr>
</tbody>
</table>

Partition interferences and environments wrt. scheduling state

- partition wrt. mutexes tested with $\text{isLocked}$
- $X \leftarrow \text{isLocked}(m)$ creates two partitions (in C)
  - $P_0$ where $X = 0$ and $m$ is free
  - $P_1$ where $X = 1$ and $m$ is locked
- $P_0$ handled as if $m$ where locked
- blocking primitives merge $P_0$ and $P_1$ ($\text{lock, yield}$)
**Priority-based scheduling**

**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
  - again, handled similarly to locks, with a richer $\mathcal{C}$
Deadlock checking

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock$(a)$</td>
<td>lock$(a)$</td>
</tr>
<tr>
<td>lock$(c)$</td>
<td>lock$(b)$</td>
</tr>
<tr>
<td>unlock$(c)$</td>
<td>unlock$(a)$</td>
</tr>
<tr>
<td>lock$(b)$</td>
<td>lock$(a)$</td>
</tr>
<tr>
<td>unlock$(b)$</td>
<td>unlock$(a)$</td>
</tr>
<tr>
<td>unlock$(a)$</td>
<td>unlock$(b)$</td>
</tr>
</tbody>
</table>

During the analysis, gather:

- all reachable **mutex configurations**: $R \subseteq \mathbb{T} \times \mathcal{P}(\mathbb{M})$
- **lock instructions** from these configurations $R \times \mathbb{M}$
Deadlock checking

During the analysis, gather:

- all reachable mutex configurations: $R \subseteq T \times P(M)$
- lock instructions from these configurations $R \times M$

Then, 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.
Rely-guarantee proof method
Rely-guarantee proof method

Reminder: Floyd–Hoare logic

Logic to prove properties about sequential programs [Hoar69].

**Hoare triples:** \( \{P\} \text{prog}\{Q\} \)

- annotate programs with logic assertions \( \{P\} \text{prog}\{Q\} \)
  (if \( P \) holds before prog, then \( Q \) holds after prog)
- check that \( \{P\}\text{prog}\{Q\} \) is derivable with the following rules
  (the assertions are program invariants)

\[
\begin{align*}
\{P[e/X]\} & X \leftarrow e \{P\} \\
\{P\} s_1 \{Q\} & \quad \{Q\} s_2 \{R\} \\
\{P\} s_1 ; s_2 \{R\} & \\
\{P \land e \not\to 0\} s \{Q\} & P \land e \not\to 0 \Rightarrow Q \\
\{P\} \text{if } e \not\to 0 \text{ then } s \text{ fi } \{Q\} & \\
\{P \land e \not\to 0\} s \{P\} & \{P\} \text{ while } e \not\to 0 \text{ do } s \text{ done } \{P \land e \not\to 0\} \\
\{P'\} s \{Q'\} & P \Rightarrow P' \quad Q' \Rightarrow Q \\
\{P\} s \{Q\} & 
\end{align*}
\]

Link with abstract interpretation:
- the equations reachability semantics \( (X_\ell)_{\ell \in \mathcal{L}} \) provides the most precise Hoare triples in fixpoint constructive form
Owicki–Gries proof method

Extension of Floyd–Hoare to concurrent programs [Owic76].

**Principle:** add a new rule, for $||$

\[
\begin{align*}
\{ P_1 \} s_1 \{ Q_1 \} & \quad \{ P_2 \} s_2 \{ Q_2 \} \\
\{ P_1 \land P_2 \} s_1 \land s_2 \{ Q_1 \land Q_2 \}
\end{align*}
\]
Owicki–Gries proof method

Extension of Floyd–Hoare to concurrent programs [Owic76].

\textbf{Principle:} add a new rule, for \mid

\[
\begin{align*}
\{P_1\} s_1 \{Q_1\} & \quad \{P_2\} s_2 \{Q_2\} \\
\{P_1 \land P_2\} s_1 \mid \mid s_2 \{Q_1 \land Q_2\}
\end{align*}
\]

This rule is not always sound!

\text{e.g., we have} \quad \{X = 0, Y = 0\} X \leftarrow 1 \{X = 1, Y = 0\}

\text{and} \quad \{X = 0, Y = 0\} \text{if } X = 0 \text{ then } Y \leftarrow 1 \text{ fi} \{X = 0, Y = 1\}

\text{but not} \quad \{X = 0, Y = 0\} X \leftarrow 1 \mid \mid \text{if } X = 0 \text{ then } Y \leftarrow 1 \text{ fi} \{false\}

\implies \quad \text{we need a side-condition to the rule:} \quad \{P_1\} s_1 \{Q_1\} \text{ and } \{P_2\} s_2 \{Q_2\} \text{ must not interfere}
Owicki–Gries proof method (cont.)

**interference freedom**

given proofs $\Delta_1$ and $\Delta_2$ of $\{P_1\} s_1 \{Q_1\}$ and $\{P_2\} s_2 \{Q_2\}$

$\Delta_1$ does not interfere with $\Delta_2$ if:
- for any $\Phi$ appearing before a statement in $\Delta_1$
- for any $\{P'_2\} s'_2 \{Q'_2\}$ appearing in $\Delta_2$
- $\{\Phi \land P'_2\} s'_2 \{\Phi\}$ holds
- and moreover $\{Q_1 \land P'_2\} s'_2 \{Q_1\}$

i.e.: the assertions used to prove $\{P_1\} s_1 \{Q_1\}$ are stable by $s_2$

e.g.,

\[
\begin{align*}
\{X = 0, Y \in [0, 1]\} & \quad X \leftarrow 1 \{X = 1, Y \in [0, 1]\} \\
\{X \in [0, 1], Y = 0\} & \quad \text{if } X = 0 \text{ then } Y \leftarrow 1 \quad |\{X \in [0, 1], Y \in [0, 1]\} \\
\Rightarrow & \quad \{X = 0, Y = 0\} \quad X \leftarrow 1 \quad |\{X = 1, Y \in [0, 1]\}
\end{align*}
\]

**Summary:**

- **pros:** the invariants are local to the threads
- **cons:** the proof is not compositional

(proving one thread requires delving into the proof of other threads)

$\Rightarrow$ not satisfactory
Jones’ rely-guarantee proof method

Idea: explicit interferences with (more) annotations \[\text{[Jone81]}\].

Rely-guarantee “quintuples”: \(R, G \vdash \{P\} \text{prog} \{Q\}\)

- if \(P\) is true before \text{prog} is executed
- and the effect of other threads is included in \(R\) (rely)
- then \(Q\) is true after \text{prog}
- and the effect of \text{prog} is included in \(G\) (guarantee)

where:

- \(P\) and \(Q\) are assertions on states \((\text{in } \mathcal{P}(\Sigma))\)
- \(R\) and \(G\) are assertions on transitions \((\text{in } \mathcal{P}(\Sigma \times \mathcal{A} \times \Sigma))\)

The parallel composition rule becomes:

\[
\begin{align*}
R \lor G_2, G_1 \vdash \{P_1\} s_1 \{Q_1\} & & R \lor G_1, G_2 \vdash \{P_2\} s_2 \{Q_2\} \\
\hline
R, G_1 \lor G_2 \vdash \{P_1 \land P_2\} s_1 \mid\mid s_2 \{Q_1 \land Q_2\}
\end{align*}
\]
**Rely-guarantee example**

### checking \( t_1 \)

\[
\ell_1 \quad \text{while random do}
\]

\[
\ell_2 \quad \text{if } x < y \text{ then}
\]

\[
\ell_3 \quad x \leftarrow x+1
\]

fi

\[
\ell_4 \quad \text{done}
\]

**at** \( \ell_1 \) : \( x = y = 0 \)

**at** \( \ell_2 \) : \( x, y \in [0, 102], \ x \leq y \)

**at** \( \ell_3 \) : \( x \in [0, 101], \ y \in [1, 102], \ x < y \)

### checking \( t_2 \)

\[
\ell_4 \quad \text{while random do}
\]

\[
\ell_5 \quad \text{if } y < 100 \text{ then}
\]

\[
\ell_6 \quad y \leftarrow y + [1,3]
\]

fi

\[
\ell_7 \quad \text{done}
\]

**at** \( \ell_4 \) : \( x = y = 0 \)

**at** \( \ell_5 \) : \( x, y \in [0, 102], \ x \leq y \)

**at** \( \ell_6 \) : \( x \in [0, 99], \ y \in [0, 99], \ x \leq y \)
Rely-guarantee example

In this example:

- guarantee exactly what is relied on \((R_1 = G_1 \text{ and } R_2 = G_2)\)
- rely and guarantee are global assertions

**Benefits of rely-guarantee:**

- invariants are still local to threads
- checking a thread does not require looking at other threads, only at an abstraction of their semantics
We must rely on and guarantee that each thread increments \( x \) exactly once!

**Solution:**

Auxiliary variables do not change the semantics but store extra information:

- Past values of variables (history of the computation)
- Program counter of other threads (pc)

**Example:**

<table>
<thead>
<tr>
<th>( t_1 )</th>
<th>( t_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell_1 ) ( x \leftarrow x + 1 ) ( \ell_2 )</td>
<td>( \ell_3 ) ( x \leftarrow x + 1 ) ( \ell_4 )</td>
</tr>
</tbody>
</table>

**Goal:** prove \( \{ x = 0 \} t_1 \parallel t_2 \{ x = 2 \} \).
Auxiliary variables

Goal: prove \( \{ x = 0 \} t_1 \parallel t_2 \{ x = 2 \} \).
we must rely on and guarantee that
each thread increments \( x \) exactly once!

Solution: auxiliary variables
do not change the semantics but store extra information:
- past values of variables (history of the computation)
- program counter of other threads \((pc_t)\)

Example: for \( t_1 \):
\[
\{(pc_2 = \ell 3 \land x = 0) \lor (pc_2 = \ell 4 \land x = 1)\}
\]
\[
x \leftarrow x + 1
\]
\[
\{(pc_2 = \ell 3 \land x = 1) \lor (pc_2 = \ell 4 \land x = 2)\}
\]
Rely-guarantee as abstract interpretation
Main idea: separate execution steps

- from the current thread \textit{a}
  - found by analysis by induction on the syntax of \textit{a}
- from other threads \textit{b}
  - given as parameter in the analysis of \textit{a}
  - inferred during the analysis of \textit{b}
Reachable states projected on thread $t$: $\mathcal{R}(t)$

- attached to thread control point in $\mathcal{L}$, not control state in $\mathbb{T} \rightarrow \mathcal{L}$
- remember other thread’s control point as “auxiliary variables”
  (required for completeness)

$\mathcal{R}(t) \overset{\text{def}}{=} \pi_t(\mathcal{R}) \subseteq \mathcal{L} \times (\mathbb{V} \cup \{ pc_{t'} \mid t \neq t' \in \mathbb{T} \}) \rightarrow \mathbb{R}$

where $\pi_t(R) \overset{\text{def}}{=} \{ \langle L(t), \rho \mid \forall t' \neq t: pc_{t'} \mapsto L(t') \rangle \mid \langle L, \rho \rangle \in R \}$
Interferences generated by $t$: $A(t)$ (\(\simeq\) guarantees on transitions)

Extract the transitions with action $t$ observed in $\mathcal{T}_p$

(subset of the transition system, containing only transitions actually used in reachability)

$A(t) \overset{\text{def}}{=} \alpha^\parallel(\mathcal{T}_p)(t)$

where $\alpha^\parallel(X)(t) \overset{\text{def}}{=} \{ \langle \sigma_i, \sigma_{i+1} \rangle \mid \exists \sigma_0 \overset{a_1}{\longrightarrow} \sigma_1 \cdots \overset{a_n}{\longrightarrow} \sigma_n \in X : a_{i+1} = t \}$
Thread-modular concrete semantics

We express $RI(t)$ and $A(t)$ directly from the transition system, without computing $Tp$

**States:** $RI$

Interleave:
- transitions from the current thread $t$
- transitions from interferences $A$ by other threads

$RI(t) = \text{lfp } R_t(A)$, where

$$R_t(Y)(X) \overset{\text{def}}{=} \pi_t(I) \cup \{ \pi_t(\sigma') | \exists \pi_t(\sigma) \in X: \sigma \xrightarrow{t} \tau \sigma' \} \cup \{ \pi_t(\sigma') | \exists \pi_t(\sigma) \in X: \exists t' \neq t: \langle \sigma, \sigma' \rangle \in Y(t') \}$$

$\implies$ similar to reachability for a sequential program, up to $A$
Thread-modular concrete semantics

We express $\mathcal{R}l(t)$ and $A(t)$ directly from the transition system, without computing $\mathcal{T}_p$

**Interferences:** $A$

Collect transitions from a thread $t$ and reachable states $\mathcal{R}$:

$A(t) = B(\mathcal{R}l)(t)$, where

$$B(Z)(t) \overset{\text{def}}{=} \{ \langle \sigma, \sigma' \rangle \mid \pi_t(\sigma) \in Z(t) \land \sigma \xrightarrow{\tau}^t \sigma' \}$$
We express $R_l(t)$ and $A(t)$ directly from the transition system, without computing $T_p$

Recursive definition:

- $R_l(t) = \text{lfp } R_t(A)$
- $A(t) = B(R_l)(t)$

$\implies$ express the most precise solution as nested fixpoints:

$$R_l = \text{lfp } \lambda Z. \lambda t. \text{lfp } R_t(B(Z))$$

**Completeness:** $\forall t: R_l(t) \simeq R \quad (\pi_t \text{ is bijective thanks to auxiliary variables})$
Fixpoint form

Constructive fixpoint form:
Use Kleene’s iteration to construct fixpoints:

1. $\mathcal{R}l = \text{lfp } H = \bigcup_{n \in \mathbb{N}} H^n(\lambda t.\emptyset)$
   in the pointwise powerset lattice $\prod_{t \in \mathcal{T}} \{t\} \rightarrow \mathcal{P}(\Sigma_t)$

2. $H(Z)(t) = \text{lfp } R_t(B(Z)) = \bigcup_{n \in \mathbb{N}} (R_t(B(Z)))^n(\emptyset)$
   in the powerset lattice $\mathcal{P}(\Sigma_t)$
   (similar to the sequential semantics of thread $t$ in isolation)

$\Rightarrow$ nested iterations
Abstract rely-guarantee

**Suggested algorithm:** nested iterations with acceleration

once abstract domains for states and interferences are chosen

- start from $\mathcal{R}^\#_0 \overset{\text{def}}{=} A^\#_0 \overset{\text{def}}{=} \lambda t. \bot^\#$
- while $A^\#_n$ is not stable
  - compute $\forall t \in T : \mathcal{R}^\#_{n+1}(t) \overset{\text{def}}{=} \text{lfp } R^\#_t(A^\#_n)$
    by iteration with widening $\triangledown$
    ($\simeq$ separate analysis of each thread)
  - compute $A^\#_{n+1} \overset{\text{def}}{=} A^\#_n \triangledown B^\#(\mathcal{R}^\#_{n+1})$
- when $A^\#_n = A^\#_{n+1}$, return $\mathcal{R}^\#_n$

$\implies$ thread-modular analysis
parameterized by abstract domains
able to easily reuse existing sequential analyses
Thread-modular abstractions
Flow-insensitive abstraction:

- reduce as much control information as possible
- but keep flow-sensitivity on each thread’s control location

Local state abstraction: remove auxiliary variables

\[ \alpha^{nf}_{R}(X) \overset{\text{def}}{=} \{ (\ell, \rho|_{\mathcal{V}}) \mid (\ell, \rho) \in X \} \cup (X \cap \Omega) \]

Interference abstraction: remove all control state

\[ \alpha^{nf}_{A}(Y) \overset{\text{def}}{=} \{ (\rho, \rho') \mid \exists L, L' \in \mathbb{T} \rightarrow \mathcal{L}: ((L, \rho), (L', \rho')) \in Y \} \]
Flow-insensitive abstraction (cont.)

Flow-insensitive fixpoint semantics: (omitting errors $\Omega$)

We apply $\alpha_{\mathcal{R}}^{nf}$ and $\alpha_{\mathcal{A}}^{nf}$ to the nested fixpoint semantics.

\[ \mathcal{R}^{nf} \overset{\text{def}}{=} \text{lfp} \lambda Z. \lambda t. \text{lfp} \mathcal{R}^{nf} t (B^{nf} (Z)) \]

where

\[ B^{nf} (Z)(t) \overset{\text{def}}{=} \{ (\rho, \rho') \mid \exists \ell, \ell' \in \mathcal{L}: (\ell, \rho) \in Z(t) \land (\ell, \rho) \rightarrow t (\ell', \rho') \} \]

(extract interferences from reachable states)

\[ \mathcal{R}^{nf} t (Y)(X) \overset{\text{def}}{=} \mathcal{R}^{loc} t (X) \cup A^{nf} t (Y)(X) \]

(interleave steps)

\[ \mathcal{R}^{loc} t (X) \overset{\text{def}}{=} \{(\ell^i_t, \lambda V.0)\} \cup \{(\ell', \rho') \mid \exists (\ell, \rho) \in X: (\ell, \rho) \rightarrow t (\ell', \rho') \} \]

(thread step)

\[ A^{nf} t (Y)(X) \overset{\text{def}}{=} \{(\ell, \rho') \mid \exists \rho, u \neq t: (\ell, \rho) \in X \land (\rho, \rho') \in Y(u) \} \]

(interference step)

Cost/precision trade-off:

- less variables
  \[ \implies \text{subsequent numeric abstractions are more efficient} \]

- insufficient to analyze $x \leftarrow x + 1 \parallel x \leftarrow x + 1$
Thread-modular abstractions

Retrieving the simple interference-based analysis

**Cartesian abstraction:** on interferences

- forget the relations between variables
- forget the relations between values before and after transitions (input-output relationship)
- only remember which variables are modified, and their value:

\[ \alpha_{nr}^A(Y) \stackrel{\text{def}}{=} \lambda V. \{ x \in V \mid \exists (\rho, \rho') \in Y : \rho(V) \neq x \land \rho'(V) = x \} \]

- to apply interferences, we get, in the nested fixpoint form:

\[ A_{nr}^t(Y)(X) \stackrel{\text{def}}{=} \{ (\ell, \rho[V \mapsto v]) \mid (\ell, \rho) \in X, V \in V, \exists u \neq t : v \in Y(u)(V) \} \]

- no modification on the state
  (the analysis of each thread can still be relational)

\[ \implies \text{we get back our simple interference analysis!} \]

Finally, use a numeric abstract domain \( \alpha : \mathcal{P}(\mathbb{V} \to \mathbb{R}) \to \mathcal{D}^\# \)
(for interferences, \( \mathbb{V} \to \mathcal{P}(\mathbb{R}) \) is abstracted as \( \mathbb{V} \to \mathcal{D}^\# \))
Thread-modular abstractions

From traces to thread-modular analyses

abstract states

*abstract interferences*

local states

**abstract interferences**

interferences

interleaved execution trace prefixes

**static analyzer**

**rely-guarantee** (without aux. variables)

**rely-guarantee** (with aux. variables)

**test**
Thread-modular abstractions

Weakly relational interferences

Clock thread

while Clock < \(10^6\) do
  Clock ← Clock + 1;
  ...
done

Accumulator thread

while random do
  Prec ← Clock;
  ...
  delta ← Clock - Prec;
  if random then x ← x + delta endif;
  ...
done

- clock is a global, increasing clock
- x accumulates periods of time
- no overflow on Clock - Prec, nor x ← x + delta

To prove this, we need **relational abstractions** of interferences
(keep input-output relationships)
**Abstraction:**

map variables to \( \uparrow \) monotonic or \( \top \) don’t know

\[ \alpha^\text{mono}_A(Y) \overset{\text{def}}{=} \lambda V. \text{if } \forall \langle \rho, \rho' \rangle \in Y: \rho(V) \leq \rho'(V) \text{ then } \uparrow \text{ else } \top \]

- keep some input-output relationships
- forgets all relations between variables
- flow-insensitive

**Inference and use**

- **gather:**
  \[ A^\text{mono}(t)(V) = \uparrow \iff \text{all assignments to } V \text{ in } t \text{ have the form } V \leftarrow V + e, \text{ with } e \geq 0 \]

- **use:** combined with non-relational interferences
  - if \( \forall t: A^\text{mono}(t)(V) = \uparrow \)
  - then any test with non-relational interference \( C[V | [a, b]] \)
  - can be strengthened into \( C[X \leq V] \)
Thread-modular abstractions

Relational invariant interferences

**Abstraction:** keep relations maintained by interferences

- remove control state in interferences
- keep mutex state $M$
- forget input-output relationships
- keep relationships between variables

\[
\alpha_{inv}^A(Y) \overset{\text{def}}{=} \{ \langle M, \rho \rangle \mid \exists \rho' : \langle \langle M, \rho \rangle, \langle M, \rho' \rangle \rangle \in Y \lor \langle \langle M, \rho' \rangle, \langle M, \rho \rangle \rangle \in Y \}
\]

\[
\langle M, \rho \rangle \in \alpha_{inv}^A(Y) \implies \langle M, \rho \rangle \in \alpha_{inv}^A(Y) \text{ after any sequence of interferences from } Y
\]

**Lock invariant:**

\[
\{ \rho \mid \exists t \in T, M : \langle M, \rho \rangle \in \alpha_{inv}(\cdot(t)), m \notin M \}
\]

- property maintained outside code protected by $m$
- possibly broken while $m$ is locked
- restored before unlocking $m$
Relational lock invariants

Improved interferences: mixing simple interferences and lock invariants
Relational lock invariants

Improved interferences: mixing simple interferences and lock invariants

- apply non-relational data-race interferences
- unless threads hold a common lock (mutual exclusion)
Relational lock invariants

Improved interferences: mixing simple interferences and lock invariants

- apply non-relational data-race interferences
  unless threads hold a common lock (mutual exclusion)

- apply non-relational well-synchronized interferences at lock points
  then intersect with the lock invariant

- gather lock invariants for lock / unlock pairs
**Relational lock invariants**

**Improved interferences:** mixing simple interferences and lock invariants

- apply non-relational data-race interferences **unless** threads hold a common lock (mutual exclusion)
- apply non-relational well-synchronized interferences at lock points **then** intersect with the lock invariant
- gather lock invariants for lock / unlock pairs
Thread-modular abstractions

Weakly relational interference example

<table>
<thead>
<tr>
<th>analyzing ( t_1 )</th>
<th>( t_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>while random do</td>
<td>( x ) unchanged</td>
</tr>
<tr>
<td>lock(m);</td>
<td>( y ) incremented</td>
</tr>
<tr>
<td>if ( x &lt; y ) then</td>
<td>0 ( \leq ) ( y ) ( \leq ) 102</td>
</tr>
<tr>
<td>( x \leftarrow x + 1; )</td>
<td></td>
</tr>
<tr>
<td>unlock(m)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>analyzing ( t_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y ) unchanged</td>
</tr>
<tr>
<td>0 ( \leq ) ( x ), ( x \leq ) ( y )</td>
</tr>
<tr>
<td>while random do</td>
</tr>
<tr>
<td>lock(m);</td>
</tr>
<tr>
<td>if ( y &lt; 100 ) then</td>
</tr>
<tr>
<td>( y \leftarrow y + [1,3]; )</td>
</tr>
<tr>
<td>unlock(m)</td>
</tr>
</tbody>
</table>

Using all three interference abstractions:
- non-relational interferences (0 \( \leq \) \( y \) \( \leq \) 102, 0 \( \leq \) \( x \))
- lock invariants, with the octagon domain (\( x \leq y \))
- monotonic interferences (\( y \) monotonic)

we can prove automatically that \( x \leq y \) holds
Application: The AstréeA analyzer
The Astrée 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
The AstréeA analyzer

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 $\implies$ minimal code modifications
- additionally: 4 KB ARINC 653 OS model

Target application:
- ARINC 653 embedded avionic application
- 15 threads, 1.6 Mlines
- embedded reactive code + network code + string formatting
- many variables, arrays, loops
- shallow call graph, no dynamic allocation
From simple interferences to relational interferences

<table>
<thead>
<tr>
<th>monotonicity domain</th>
<th>relational lock invariants</th>
<th>analysis time</th>
<th>memory</th>
<th>iterations</th>
<th>alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>25h 26mn</td>
<td>22 GB</td>
<td>6</td>
<td>4616</td>
</tr>
<tr>
<td>✓</td>
<td>×</td>
<td>30h 30mn</td>
<td>24 GB</td>
<td>7</td>
<td>1100</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>110h 38mn</td>
<td>90 GB</td>
<td>7</td>
<td>1009</td>
</tr>
</tbody>
</table>
Summary
We presented static analysis methods that are:

- inspired from thread-modular proof methods
- abstractions of complete concrete semantics (for safety properties)
- sound for all interleavings
- sound for weakly consistent memory semantics (when using non-relational, flow-insensitive interference abstraction)
- aware of scheduling, priorities and synchronization
- parameterized by (possibly relational) abstract domains (independent domains for state abstraction and interference abstraction)


