Thread-Modular Static Analysis of Concurrent Programs

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

Antoine Miné

Year 2021–2022

Course 6
25 October
Concurrent programming

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

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

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

- exploit several computers (distributed computing)
- ease of programming (GUI, network code, reactive programs)

But concurrent programs are hard to program and hard to verify:

- combinatorial exposition of execution paths (interleavings)
- errors lurking in hard-to-find corner cases (race conditions)
- unintuitive execution models (weak memory consistency)
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., divisions by 0)
- discover data-races (unprotected accesses by concurrent threads)
- discover deadlocks (some threads block each other indefinitely)
- application to analyzing embedded C programs
Outline

- Simple concurrent language
- Non-modular concurrent semantics
- Simple interference thread-modular concurrent semantics
- Abstract rely-guarantee thread-modular concurrent semantics
- Application: the AstréeA analyzer
Language and semantics
Structured numeric language

- finite set of (toplevel) threads: $\text{stmt}_1$ to $\text{stmt}_n$
- finite set of numeric program variables $V \in \mathbb{V}$
- finite set of statement locations $\ell \in \mathcal{L}$
- locations with possible run-time errors $\omega \in \Omega$ (divisions by zero)

Structured language syntax

$$
\text{prog} ::= \ell \text{stmt}_1 \ell \parallel \ldots \parallel \ell \text{stmt}_n \ell \quad \text{(parallel composition)}
$$

$$
\ell \text{stmt} \ell ::= \ell V \leftarrow \text{exp} \ell \quad \text{(assignment)}
| \ell \text{if exp} \notdiv 0 \text{then } \ell \text{stmt} \ell \text{ fi} \ell \quad \text{(conditional)}
| \ell \text{while } \text{exp} \notdiv 0 \text{ do } \ell \text{stmt} \ell \text{ done} \ell \quad \text{(loop)}
| \ell \text{stmt} \ell ; \ell \text{stmt} \ell \quad \text{(sequence)}
$$

$$
\text{exp} ::= V | [c_1, c_2] | - \text{exp} | \text{exp} \diamond \text{exp}
$$

$$
c_1, c_2 \in \mathbb{R} \cup \{+\infty, -\infty\}, \diamond \in \{+,-,\times,\div\omega\}, \notdiv \in \{=,<,\ldots\}
$$
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 $0 \leq x \leq y \leq 102$
Semantic model: labelled transition systems

simple extension of transition systems

**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 \(\sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \cdots \sigma_n \xrightarrow{a_n} \sigma_{n+1}\)

\(\tau\) is omitted on \(\xrightarrow{\cdot}\) for traces for simplicity
From concurrent programs to labelled transition systems

- given: \( \text{prog} ::= {\ell_1}^{i} \text{stmt}_1{\ell_1}^x \parallel \cdots \parallel {\ell_n}^{i} \text{stmt}_n{\ell_n}^x \)
- threads are numbered: \( T \overset{\text{def}}{=} \{ 1, \ldots, n \} \)

Program states: \( \Sigma \overset{\text{def}}{=} (T \rightarrow \mathcal{L}) \times \mathcal{E} \)
- a control state \( L(t) \in \mathcal{L} \) for each thread \( t \in T \) and
- a single shared memory state \( \rho \in \mathcal{E} \overset{\text{def}}{=} \forall \rightarrow \mathbb{Z} \)

Initial states:

threads start at their first control point \( \ell_t^i \), variables are set to 0:
\( I \overset{\text{def}}{=} \{ \langle \lambda t.\ell_t^i, \lambda V.0 \rangle \} \)

Actions: actions are thread identifiers: \( A \overset{\text{def}}{=} T \)
From concurrent programs to labelled transition systems

**Transition relation:** \( \tau \subseteq \Sigma \times A \times \Sigma \)

\[
\langle L, \rho \rangle \xrightarrow{t} \tau \langle L', \rho' \rangle \iff \langle L(t), \rho \rangle \rightarrow_{\tau[\text{stmt}_t]} \langle L'(t), \rho' \rangle \land \\
\forall u \neq t: L(u) = L'(u)
\]

- based on the transition relation of individual threads seen as sequential processes \( \text{stmt}_t \): \( \tau[\text{stmt}_t] \subseteq (\mathcal{L} \times \mathcal{E}) \times (\mathcal{L} \times \mathcal{E}) \)
  - choose a thread \( t \) to run
  - update \( \rho \) and \( L(t) \)
  - leave \( L(u) \) intact for \( u \neq t \)

  see course 2 for the full definition of \( \tau[\text{stmt}] \)

- each transition \( \sigma \rightarrow_{\tau[\text{stmt}_t]} \sigma' \) leads to many transitions \( \rightarrow_{\tau} \)!
Interleaved trace semantics

Maximal and finite prefix trace semantics are as before:

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

**Maximal traces:** \( M_\infty \) (finite or infinite)

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

**Finite prefix traces:** \( T_p \)

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

\[T_p = \text{lfp } F_p\text{ where } \]

\[
F_p(X) = I \cup \{ \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_n} \sigma_{n+1} \mid n \geq 0 \land \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in X \land \sigma_n \xrightarrow{t_n} \tau \sigma_{n+1} \}
\]
Fairness conditions: avoid threads being denied to run forever

Given \( \text{enabled}(\sigma, t) \overset{\text{def}}{\iff} \exists \sigma' \in \Sigma : \sigma \xrightarrow{t} \tau \sigma' \)

an infinite trace \( \sigma_0 \xrightarrow{t_0} \cdots \sigma_n \xrightarrow{t_n} \cdots \) 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 \( M_\infty \) 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 \( M_\infty \cap F \subseteq X \) instead of \( M_\infty \subseteq X \)
Fairness (cont.)

Example: \( \text{while } x \geq 0 \text{ do } x \leftarrow x + 1 \text{ done } \parallel x \leftarrow -2 \)
- 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 \( \alpha_\star \leq (M_\infty \cap F) \subseteq \alpha_\star \leq (X) \)

for all fairness conditions \( F \), \( \alpha_\star \leq (M_\infty \cap F) = \alpha_\star \leq (M_\infty) = T_p \)

recall that \( \alpha_\star \leq (T) \overset{\text{def}}{=} \{ t \in \Sigma^* | \exists u \in T : t \leq u \} \) is the finite prefix abstraction
and \( T = \alpha_\star \leq (M_\infty) \)

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

In the following, we ignore fairness conditions
Reachability semantics for concurrent programs

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

Reachable states in any execution:

\[
\mathcal{R} \overset{\text{def}}{=} \left\{ \sigma \mid \exists n \geq 0, \sigma_0, \ldots, \sigma_n: \sigma_0 \in \mathcal{I}, \forall i < n: \exists t \in \mathcal{T}: \sigma_i \xrightarrow{t} \tau \sigma_{i+1} \land \sigma = \sigma_n \right\}
\]

\[
\mathcal{R} = \text{lfp } F_{\mathcal{R}} \text{ where } F_{\mathcal{R}}(X) = \mathcal{I} \cup \{ \sigma \mid \exists \sigma' \in X, t \in \mathcal{T}: \sigma' \xrightarrow{t} \tau \sigma \}
\]

Can prove (non-)reachability, but not ordering, termination, liveness and cannot exploit fairness.

Abstraction of the finite trace semantics.

\[
\mathcal{R} = \alpha_p(\mathcal{T}_p) \text{ where } \alpha_p(X) \overset{\text{def}}{=} \left\{ \sigma \mid \exists n \geq 0, \sigma_0 \xrightarrow{t_0} \cdots \sigma_n \in X: \sigma = \sigma_n \right\}
\]
Reminders: sequential semantics
Equational state semantics of sequential program

- see lfp $f$ as the least solution of an equation $x = f(x)$
- partition states by control: $\mathcal{P}(L \times E) \simeq L \to \mathcal{P}(E)$

$\mathcal{X}_\ell \in \mathcal{P}(E)$: invariant at $\ell \in L$

$\forall \ell \in L: \mathcal{X}_\ell \overset{\text{def}}{=} \{ m \in E | \langle \ell, m \rangle \in R \}$

$\implies$ set of recursive equations on $\mathcal{X}_\ell$

**Example:**

\[
\begin{align*}
\ell^1 & \quad i \leftarrow 2; \\
\ell^2 & \quad n \leftarrow [-\infty, +\infty]; \\
\ell^3 & \quad \text{while } \ell^4 \ i < n \text{ do} \\
\ell^5 & \quad \text{if } [0, 1] = 0 \text{ then} \\
\ell^6 & \quad i \leftarrow i + 1 \\
\ell^7 & \quad \text{fi} \\
\ell^8 & \quad \text{done}
\end{align*}
\]

\[
\begin{align*}
\mathcal{X}_1 & = \mathcal{I} \\
\mathcal{X}_2 & = C[ i \leftarrow 2 ] \mathcal{X}_1 \\
\mathcal{X}_3 & = C[ n \leftarrow [-\infty, +\infty] ] \mathcal{X}_2 \\
\mathcal{X}_4 & = \mathcal{X}_3 \cup \mathcal{X}_7 \\
\mathcal{X}_5 & = C[ i < n ] \mathcal{X}_4 \\
\mathcal{X}_6 & = \mathcal{X}_5 \\
\mathcal{X}_7 & = \mathcal{X}_5 \cup C[ i \leftarrow i + 1 ] \mathcal{X}_6 \\
\mathcal{X}_8 & = C[ i \geq n ] \mathcal{X}_4
\end{align*}
\]
Denotational state semantics

Alternate view as an input-output state function $C[\text{stmt}]$

$$C[\text{stmt}] : \mathcal{P}(E) \rightarrow \mathcal{P}(E)$$

- $C[\text{X} \leftarrow e] R \quad \overset{\text{def}}{=} \{ \rho[\text{X} \mapsto v] | \rho \in R, v \in E[\text{e}] \rho \}$
- $C[\text{e} \triangleright 0] R \quad \overset{\text{def}}{=} \{ \rho \in R | \exists v \in E[\text{e}] \rho : v \triangleright 0 \}$
- $C[\text{if} \ e \triangleright 0 \ \text{then} \ s \ \text{fi}] R \quad \overset{\text{def}}{=} (C[s] \circ C[\text{e} \triangleright 0]) R \cup C[\text{e} \triangleright 0] R$
- $C[s_1; s_2] \quad \overset{\text{def}}{=} C[s_2] \circ C[s_1]$
- $C[\text{while} \ e \triangleright 0 \ \text{do} \ s \ \text{done}] R \quad \overset{\text{def}}{=} C[\text{e} \triangleright 0] (\text{lfp} \lambda Y. R \cup (C[s] \circ C[\text{e} \triangleright 0]) Y)$

- mutate memory states in $E$
- structured: nested loops yield nested fixpoints
- big-step: forget information on intermediate locations $\ell$
- mimics an actual interpreter
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:**

\[
\begin{align*}
i &= 0; \\
while (i < nb) \\
\{ \\
    a[i] &= 12; \\
i &= i + 1;
\}
\end{align*}
\]

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

- 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
  \[ \implies \text{thread-modular analysis} \]
Non-modular concurrent semantics
Equational concurrent state semantics

**Equational form:**

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

$$\mathcal{X}_L = \bigcup_{t \in \mathbb{T}} \{ F(\mathcal{X}_{L_2}, \ldots, \mathcal{X}_{L_N}) | \exists (\mathcal{X}_{\ell_1} = F(\mathcal{X}_{\ell_2}, \ldots, \mathcal{X}_{\ell_N})) \in eq(stmt_t):$$

$$\forall i \leq N: L_i(t) = \ell_i, \forall u \neq t: L_i(u) = L_1(u) \}$$

Join with $\bigcup$ equations from $eq(stmt_t)$ updating a single thread $t \in \mathbb{T}$.

(see course 2 for the full definition of $eq(stmt)$)
Equational state semantics (illustration)

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

Equation system:

\[
\begin{align*}
X_{1,4} &= I \\
X_{2,4} &= X_{1,4} \cup C[x \geq y] X_{2,4} \cup C[x \leftarrow x + 1] X_{3,4} \\
X_{3,4} &= C[x < y] X_{2,4} \\
X_{1,5} &= X_{1,4} \cup C[y \geq 100] X_{1,5} \cup C[y \leftarrow y + [1,3]] X_{1,6} \\
X_{2,5} &= X_{1,5} \cup C[x \geq y] X_{2,5} \cup C[x \leftarrow x + 1] X_{3,5} \cup \\
& \quad (X_{2,4} \cup C[y \geq 100]) X_{2,5} \cup C[y \leftarrow y + [1,3]] X_{2,6} \\
X_{3,5} &= C[x < y] X_{2,5} \cup X_{3,4} \cup C[y \geq 100] X_{3,5} \cup C[y \leftarrow y + [1,3]] X_{3,6} \\
X_{1,6} &= C[y < 100] X_{1,5} \\
X_{2,6} &= X_{1,6} \cup C[x \geq y] X_{2,6} \cup C[x \leftarrow x + 1] X_{3,6} \cup C[y < 100] X_{2,5} \\
X_{3,6} &= C[x < y] X_{2,6} \cup C[y < 100] X_{3,5}
\end{align*}
\]
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</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>

**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
- \(\rightarrow\) 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 $\mathbb{T} \rightarrow \mathcal{L}$)

- 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
but **remain sound** with respect to all interleavings!
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
Principle:

- analyze each thread in isolation
- gather the values written into each variable by each thread
  \[\implies\text{so-called interferences}\]
  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
  \[\Rightarrow\] 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
  \[\Rightarrow\] one more level of fixpoint iteration
Example

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

$t_1$

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

Analysis of $t_1$ in isolation

(1): $x = y = 0 \quad \mathcal{X}_1 = \perp$

(2): $x = y = 0 \quad \mathcal{X}_2 = \mathcal{X}_1 \cup C[x \leftarrow x + 1] \mathcal{X}_3 \cup C[x \geq y] \mathcal{X}_2$

(3): $\perp \quad \mathcal{X}_3 = C[x < y] \mathcal{X}_2$
Example

Analysis of $t_2$ in isolation

(4): $x = y = 0$ \hspace{1cm} $\mathcal{X}_4 = I$

(5): $x = 0, \ y \in [0, 102]$ \hspace{1cm} $\mathcal{X}_5 = \mathcal{X}_4 \cup C[y \leftarrow y + [1, 3]] \cup C[y \geq 100] \cup \mathcal{X}_5$

(6): $x = 0, \ y \in [0, 99]$ \hspace{1cm} $\mathcal{X}_6 = C[y < 100] \cup \mathcal{X}_5$

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

<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>

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

**Input interferences:** $y \leftarrow [1, 102]$

1. $x = y = 0$ \quad $\mathcal{X}_1 = I$
2. $x \in [0, 102], y = 0$ \quad $\mathcal{X}_2 = \mathcal{X}_{1a} \cup C[x \leftarrow x + 1] \mathcal{X}_3 \cup C[x \geq (y | [1, 102])] \mathcal{X}_2$
3. $x \in [0, 102], y = 0$ \quad $\mathcal{X}_3 = C[x < (y | [1, 102])] \mathcal{X}_2$

**Output interferences:** $x \leftarrow [1, 102]$

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

```plaintext
\[\begin{align*}
\ell_1 & \text{ while random do} \\
\ell_2 & \text{ if } x < y \text{ then} \\
\ell_3 & x \leftarrow x + 1
\end{align*}\]

\[\begin{align*}
\ell_4 & \text{ while random do} \\
\ell_5 & \text{ if } y < 100 \text{ then} \\
\ell_6 & y \leftarrow y + [1, 3]
\end{align*}\]
```

**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 (we’ll fix that later)
Formalizing the simple interference semantics
Denotational semantics with interferences

Interferences in \( I \) \( \triangleq T \times V \times R \)
\( \langle t, X, n \rangle \) means that \( t \) can store the value \( n \) into the variable \( X \)

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

**Expressions** : \( E_t[\exp] : \mathcal{E} \times \mathcal{P}(I) \rightarrow \mathcal{P}(R) \times \mathcal{P}(\Omega) \) for thread \( t \)

- add interference \( I \in I \), as input
- add error information \( \omega \in \Omega \) as output
  locations of / operators that can cause a division by 0

Example:

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

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

- etc.
Denotational semantics with interferences (cont.)

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

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

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

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

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

... notating $\langle V_{\rho}, O_{\rho} \rangle \overset{\text{def}}{=} E_t[e] \langle \rho, I \rangle$

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

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

\[
P[\text{prog}] \overset{\text{def}}{=} \text{lfp } \lambda \langle O, I \rangle. \bigcup_{t \in T} [C_t[\text{stmt}_t] \langle E_0, \emptyset, I \rangle]_{\Omega, \emptyset}
\]

- each thread analysis starts in an initial environment set \( E_0 \overset{\text{def}}{=} \{ \lambda V.0 \} \)

- \([X]_{\Omega, \emptyset}\) projects \( X \in \mathcal{P}(E) \times \mathcal{P}(\Omega) \times \mathcal{P}(\emptyset) \) on \( \mathcal{P}(\Omega) \times \mathcal{P}(\emptyset) \)
  and interferences and errors from all threads are joined
  
  the output environments from a thread analysis are not easily exploitable

- \( P[\text{prog}] \) only outputs the set of possible run-time errors

We will need to prove the soundness of \( P[\text{prog}] \)
with respect to the interleaving semantics...
Interference abstraction

**Abstract interferences \( \# \)**

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

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

**Abstract semantics with interferences \( \mathcal{C}_t^\#[s] \)**

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

**Example:**  \( \mathcal{C}_t^\#[X \leftarrow e]\langle R^\#, \Omega, I^\# \rangle \)

- for each \( Y \) in \( e \), get its interference \( Y^\#_\mathcal{R} = \bigsqcup_{\mathcal{R}} \{ I^\# \langle u, Y \rangle \mid u \neq t \} \)
- if \( Y^\#_\mathcal{R} \neq \bot^\#_\mathcal{R} \), replace \( Y \) in \( e \) with \( \text{get}(Y, R^\#) \sqcup Y^\#_\mathcal{R} \)
  - \( \text{get}(Y, R^\#) \) extracts the abstract values variable \( Y \) from \( R^\# \in \mathcal{E}^\# \)
- compute \( \langle R'^\#, O' \rangle = \mathcal{C}_t^\#[e]\langle R^\#, O \rangle \)
- enrich \( I^\# \langle t, X \rangle \) with \( \text{get}(X, R'^\#) \)
Static analysis with interferences

Abstract analysis

\[ P^\#[ prog ] \overset{\text{def}}{=} \lim_\lambda \langle O, I^\# \rangle \triangledown \bigcup_{t \in T} [ C^\#_t[ stmt_t ] \langle E^\#_0, \emptyset, I^\# \rangle ]_{\Omega, I^\#} \Omega \]

- effective analysis by structural induction
- \( P^\#[ prog ] \) is sound with respect to \( P[ prog ] \)
- termination ensured by a widening
- parameterized by a choice of abstract domains \( R^\#, E^\# \)

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

reminder: \( [X]_{\Omega}, [Y]_{\Omega, I^\#} \) keep only \( X \)'s component in \( \Omega \), \( Y \)'s components in \( \Omega \) and \( I^\# \)
Path-based soundness proof
Control paths of a sequential program

\[ \text{atomic} ::= X \leftarrow \text{exp} \mid \text{exp} \nless \text{0} \]

Control paths

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

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

\[ \pi(\text{if } e \nless 0 \text{ then } s \text{ fi}) \overset{\text{def}}{=} (\{ e \nless 0 \} \cdot \pi(s)) \cup \{ e \nless 0 \} \]

\[ \pi(\text{while } e \nless 0 \text{ do } s \text{ done}) \overset{\text{def}}{=} \left( \bigcup_{i \geq 0} (\{ e \nless 0 \} \cdot \pi(s))^i \right) \cdot \{ e \nless 0 \} \]

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

\( \pi(\text{stmt}) \) is a (generally infinite) set of finite control paths

\[ \text{e.g. } \pi(i \leftarrow 0; \text{while } i < 10 \text{ do } i \leftarrow i + 1 \text{ done}; \ x \leftarrow i) = i \leftarrow 0 \cdot (i < 10 \cdot i \leftarrow i + 1)^* \cdot x \leftarrow i \]
Path-based concrete semantics of sequential programs

Join-over-all-path semantics

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

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

Semantic equivalence

\[ C[\text{stmt}] = \prod[\pi(\text{stmt})] \]

no longer true in the abstract
Simple interference semantics

Path-based soundness proof

Path-based concrete semantics of concurrent programs

### Concurrent control paths

\[
\pi_* \overset{\text{def}}{=} \{ \text{interleavings of } \pi(\text{stmt}_t), \ t \in \mathbb{T} \} \\
= \{ p \in \text{atomic}^* \mid \forall t \in \mathbb{T}, \ \text{proj}_t(p) \in \pi(\text{stmt}_t) \}
\]

### Interleaving program semantics

\[
P_*[\text{prog}] \overset{\text{def}}{=} [\Pi[\pi_*] \langle \mathcal{E}_0, \emptyset \rangle]_\Omega
\]

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

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

### Issues:

- too many paths to consider exhaustively
- no induction structure to iterate on

\implies \text{abstract as a denotational semantics}
Soundness of the interference semantics

Soundness theorem

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

Proof sketch:

- define \( \prod_t P \times X \overset{\text{def}}{=} \bigsqcup \{ C_t[s_1; \ldots; s_n] X \mid s_1 \cdot \ldots \cdot s_n \in P \} \), then \( \prod_t \pi(s) = C_t[s] \);
- given the interference fixpoint \( I \subseteq I \) from \( P[\text{prog}] \), prove by recurrence on the length of \( p \in \pi_\ast \) that:
  - \( \forall \rho \in \prod[\ E_0, \emptyset] \ E \), \( \forall t \in T \),
    \( \exists \rho' \in \prod[\ \text{proj}_t(p)]\langle E_0, \emptyset, I \rangle \ E \) such that
    \( \forall X \in V, \rho(X) = \rho'(X) \) or \( \langle u, X, \rho(X) \rangle \in I \) for some \( u \neq t \).
  - \( \prod[\ E_0, \emptyset] \omega \subseteq \bigcup_{t \in T} \prod[\ \text{proj}_t(p)]\langle E_0, \emptyset, I \rangle \omega \)

Notes:

- sound but not complete
- can be extended to soundness proof under weakly consistent memories
Locks and synchronization
Scheduling

Synchronization primitives

\[
\text{stmt} ::= \text{lock}(m) \\
\quad | \quad \text{unlock}(m)
\]

\[m \in \mathbb{M} : \text{finite set of non-recursive mutexes}\]

Scheduling

mutexes ensure mutual exclusion

at each time, each mutex can be locked by a single thread
We use a refinement of the simple interference semantics by partitioning wrt. an abstract local view of the scheduler $\mathcal{C}$

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

$\mathcal{C} \overset{\text{def}}{=} \mathcal{C}_{race} \cup \mathcal{C}_{sync}$ separates

- data-race writes $\mathcal{C}_{race}$
- well-synchronized writes $\mathcal{C}_{sync}$
Simple interference semantics
Locks and synchronization

Mutual exclusion

\[
\begin{array}{c}
\text{lock}(m) & \text{unlock}(m) \\
p_1 & \text{lock}(m) & \text{unlock}(m) \\
& W & W & W \\
& R & R & W & R
\end{array}
\]

\textbf{Data-race effects} \quad \mathcal{C}_{race} \simeq \mathcal{P}(\mathbb{M})

Across read / write not protected by a mutex.

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

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

\textbf{Bonus:} we get a data-race analysis for free!
Mutual exclusion

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

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

\[ \mathbb{C} \cong \mathcal{P}(\mathbb{M}) \times (\{\text{data} \rightarrow \text{race}\} \cup \mathbb{M}) \]
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: $R \subseteq T \times \mathcal{P}(M)$
- lock instructions from these configurations $R \times M$
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: $R \subseteq T \times \mathcal{P}(M)$
- 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.

A generalization to larger cycles, with more threads involved in a deadlock, is easy.
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
Beyond non-relational interferences
Inspiration from program logics
Reminder: Floyd–Hoare logic

Logic to prove properties about **sequential** programs [Hoar69].

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

- annotate programs with **logic assertions** \( \{ P \} \text{stmt} \{ Q \} \)
  (if \( P \) holds before \( \text{stmt} \), then \( Q \) holds after \( \text{stmt} \))

- check that \( \{ P \} \text{stmt} \{ Q \} \) is derivable with the following rules
  (the assertions are program invariants)

\[
\begin{align*}
\{ P[e/X] \} & X \leftarrow e \{ P \} \\
\{ P \} & \text{stmt} \{ Q \} \quad \{ Q \} \text{stmt} \{ R \} \\
\{ P \} & \text{stmt}_1; \text{stmt}_2 \{ R \} \\
\{ P \} & \text{stmt} \{ Q \} \\
\end{align*}
\]

\[
\begin{align*}
\{ P \land e \not\| 0 \} & \text{stmt} \{ Q \} \\
\{ P \} & \text{if } e \not\| 0 \text{ then stmt } \text{fi} \{ Q \} \\
\{ P \land e \not\| 0 \} & \text{stmt} \{ P \} \\
\{ P \} & \text{while } e \not\| 0 \text{ do stmt done } \{ P \land e \not\| 0 \} \\
\{ P' \} & \text{stmt} \{ Q' \} \\
\end{align*}
\]

Link with abstract interpretation:

- the equations reachability semantics \( (X_\ell)_{\ell \in \mathcal{L}} \) provides the **most precise Hoare triples in fixpoint constructive form**
Jones’ rely-guarantee proof method

Idea: explicit interferences with (more) annotations [Jone81].

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

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

where:

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

The parallel composition rule is:

\[
R \lor G_2, G_1 \vdash \{ P_1 \} s_1 \{ Q_1 \} \quad R \lor G_1, G_2 \vdash \{ P_2 \} s_2 \{ Q_2 \}
\]

\[
R, G_1 \lor G_2 \vdash \{ P_1 \land P_2 \} s_1 \parallel s_2 \{ Q_1 \land Q_2 \}
\]
## Rely-guarantee example

### Checking $t_1$

<table>
<thead>
<tr>
<th>$\ell_1$</th>
<th>while random do</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell_2$</td>
<td>if $x &lt; y$ then</td>
</tr>
<tr>
<td>$\ell_3$</td>
<td>$x \leftarrow x+1$</td>
</tr>
<tr>
<td>fi</td>
<td>done</td>
</tr>
</tbody>
</table>

- $\ell_1$: $x = y = 0$
- $\ell_2$: $x, y \in [0, 102], x \leq y$
- $\ell_3$: $x \in [0, 101], y \in [1, 102], x < y$

### Checking $t_2$

<table>
<thead>
<tr>
<th>$\ell_4$</th>
<th>while random do</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell_5$</td>
<td>if $y &lt; 100$ then</td>
</tr>
<tr>
<td>$\ell_6$</td>
<td>$y \leftarrow y + [1,3]$</td>
</tr>
<tr>
<td>fi</td>
<td>done</td>
</tr>
</tbody>
</table>

- $\ell_4$: $x = y = 0$
- $\ell_5$: $x, y \in [0, 102], x \leq y$
- $\ell_6$: $x \in [0, 99], y \in [0, 99], x \leq y$
Beyond non-relational interferences

Inspiration from program logics

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:**
- more precise: can prove \(x \leq y\)
- invariants are still local to threads
- checking a thread does not require looking at other threads, only at an abstraction of their semantics
Rely-guarantee as abstract interpretation
Modularity: main idea

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}

\implies express the semantics from the point of view of a single thread
Reachable states projected on thread $t$: $\mathcal{R}(t)$

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

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

where $\pi_t(R) \overset{\text{def}}{=} \{⟨L(t), ρ[∀t' \neq t: pc_{t'} \mapsto L(t')]⟩ \mid ⟨L, ρ⟩ \in R\}$
Trace decomposition

Interferences generated by $t$: $A(t)$ (≈ 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)

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

where $\alpha^l(X)(t) \overset{\text{def}}{=} \{ \langle \sigma_i, \sigma_{i+1} \rangle | \exists \sigma_0 \overset{a_0}{\rightarrow} \cdots \overset{a_{n-1}}{\rightarrow} \sigma_n \in X : a_i = t \}$
We express $\mathcal{R}(t)$ and $A(t)$ directly from the transition system, without computing $\mathcal{T}_p$

**States:** $\mathcal{R}$

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

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

$$R_t(Y)(X) \overset{\text{def}}{=} \pi_t(I) \cup \{ \pi_t(\sigma') | \exists \pi_t(\sigma) \in X: \sigma \xrightarrow{\tau} \sigma' \} \cup$$

$$\{ \pi_t(\sigma') | \exists \pi_t(\sigma) \in X: \exists t' \neq t: \langle \sigma, \sigma' \rangle \in Y(t') \}$$

$\implies$ similar to reachability for a sequential program, up to $A$
We express $\mathcal{R}(t)$ and $A(t)$ directly from the transition system, without computing $T_p$

**Interferences:** $A$

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

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

where

$$B(\mathcal{Z})(t) \overset{\text{def}}{=} \{ \langle \sigma, \sigma' \rangle \mid \pi_t(\sigma) \in \mathcal{Z}(t) \land \sigma \overset{t}{\rightarrow} \sigma' \}$$
Thread-modular concrete semantics

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

Recursive definition:

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

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

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

Completeness: $\forall t: \mathcal{R}(t) \simeq \mathcal{R}$  ($\pi_t$ is bijective thanks to auxiliary variables)

any property provable with the interleaving semantics can be proven with the thread-modular semantics!
Constructive fixpoint form:

Use Kleene’s iteration to construct fixpoints:

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

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

$\implies$ nested iterations
Abstract rely-guarantee

**Suggested algorithm:** nested iterations with acceleration

Once abstract domains for states and interferences are chosen

- Start from $Rl^0 ≜ A^0 ≜ \lambda t. \bot$
- While $A^t_n$ is not stable
  - Compute $\forall t \in T: Rl^{t+1}_n(t) ≜ \text{lfp } R^t_t(A^t_n)$ by iteration with widening $\nabla$
    - ($\simeq$ separate analysis of each thread)
  - Compute $A^{t+1}_n ≜ A^t_n \nabla B^t_n(Rl^{t+1}_n)$
  - When $A^t_n = A^{t+1}_n$, return $Rl^n$  

$\Longrightarrow$ thread-modular analysis parameterized by abstract domains (only source of approximation)

 Able to easily reuse existing sequential analyses
Beyond non-relational interferences

Retrieving thread-modular abstractions
Flow-insensitive abstraction

**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

\[
\alpha_{\mathcal{R}}^{nf}(X) \overset{\text{def}}{=} \{ \langle \ell, \rho|_v \rangle \mid \langle \ell, \rho \rangle \in X \} \cup X
\]

**Interference abstraction:** remove all control state

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

Flow-insensitive fixpoint semantics:

We apply $\alpha^{nf}_R$ and $\alpha^{nf}_A$ to the nested fixpoint semantics.

$R^{nf}$ defined as $\text{lfp } \lambda Z. \lambda t. \text{lfp } R^{nf}_t(B^{nf}(Z))$, where

- $B^{nf}(Z)(t) \overset{\text{def}}{=} \{ \langle \rho, \rho' \rangle \mid \exists \ell, \ell' \in L: \langle \ell, \rho \rangle \in Z(t) \land \langle \ell, \rho \rangle \rightarrow_t \langle \ell', \rho' \rangle \}$
  (extract interferences from reachable states)

- $R^{nf}_t(Y)(X) \overset{\text{def}}{=} R^{loc}_t(X) \cup A^{nf}_t(Y)(X)$
  (interleave steps)

- $R^{loc}_t(X) \overset{\text{def}}{=} \{ \langle \ell^t, \lambda V.0 \rangle \} \cup \{ \langle \ell', \rho' \rangle \mid \exists \langle \ell, \rho \rangle \in X: \langle \ell, \rho \rangle \rightarrow_t \langle \ell', \rho' \rangle \}$
  (thread step)

- $A^{nf}_t(Y)(X) \overset{\text{def}}{=} \{ \langle \ell, \rho' \rangle \mid \exists \rho, u \neq t: \langle \ell, \rho \rangle \in X \land \langle \rho, \rho' \rangle \in Y(u) \}$
  (interference step)

Cost/precision trade-off:

- less variables
  $\implies$ subsequent numeric abstractions are more efficient

- insufficient to analyze $x \leftarrow x + 1 \parallel x \leftarrow x + 1$
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:
  \[ \alpha_{A}^{nr}(Y) \overset{\text{def}}{=} \lambda V. \{ x \in V \mid \exists \langle \rho, \rho' \rangle \in Y: \rho(V) \neq x \land \rho'(V) = x \} \]
- to apply interferences, we get, in the nested fixpoint form:
  \[ A_{t}^{nr}(Y)(X) \overset{\text{def}}{=} \{ \langle \ell, \rho[V \mapsto v] \rangle \mid \langle \ell, \rho \rangle \in X, V \in V, \exists u \neq t: v \in Y(u)(V) \} \]
- 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 \( \alpha : \mathcal{P}(\mathbb{V} \rightarrow \mathbb{R}) \rightarrow \mathcal{D}^{\#} \)
for interferences, \( \mathbb{V} \rightarrow \mathcal{P}(\mathbb{R}) \) is abstracted as \( \mathbb{V} \rightarrow \mathcal{D}^{\#} \)
Beyond non-relational interferences

Retrieving thread-modular abstractions

From traces to thread-modular analyses

**Abstract states**

\[(T \times L) \rightarrow \mathcal{E}^\#\]

**Abstract interferences**

\[T \rightarrow \mathcal{E}^\#\]

**Static analyzer**

**Non-relational interferences**

\[T \rightarrow \mathcal{P}(\mathcal{E})\]

**Rely-guarantee**

(without aux. variables)

**Flow-insensitive interferences**

\[T \rightarrow \mathcal{P}(\mathcal{E} \times \mathcal{E})\]

**Rely-guarantee**

(with aux. variables)

**Interferences**

\[A : T \rightarrow \mathcal{P}(\Sigma \times \Sigma)\]

**Concrete executions**

\[T_p \in \mathcal{P}(\Sigma^*)\]
Relational thread-modular abstractions
Fully relational interferences with numeric domains

Reachability: \( \mathcal{R}(t) : \mathcal{L} \rightarrow \mathcal{P}(\forall a \rightarrow \mathbb{Z}) \)
approximated as usual with one numeric abstract element per label

auxiliary variables \( pc_b \in \forall a \) are kept (program labels as numbers)

Interferences: \( A(t) \in \mathcal{P}(\Sigma \times \Sigma) \)
a numeric relation can be expressed in a classic numeric domain
as \( \mathcal{P}(\forall a \rightarrow \mathbb{Z}) \times (\forall a \rightarrow \mathbb{Z}) \) \( \simeq \) \( \mathcal{P}(\forall a \cup \forall_a' \rightarrow \mathbb{Z}) \)

- \( X \in \forall a \) value of variable \( X \) or auxiliary variable in the pre-state
- \( X' \in \forall_a' \) value of variable \( X \) or auxiliary variable in the post-state

e.g.: \{ (\( x, x + 1 \) | \( x \in [0, 10] \) \} is represented as \( x' = x + 1 \wedge x \in [0, 10] \)

\[ \implies \text{use one global abstract element per thread} \]

Benefits and drawbacks:

- simple: reuse stock numeric abstractions and thread iterators
- precise: the only source of imprecision is the numeric domain
- costly: must apply a (possibly large) relation at each program step
Experiments with fully relational interferences

Experiments by R. Monat
Scalability in the number of threads (assuming fixed number of variables)

\[ t_1 \]
\[
\text{while } z < 10000 \\
\quad z \leftarrow z + 1 \\
\quad \text{if } y < c \text{ then } y \leftarrow y + 1 \\
\text{done}
\]

\[ t_2 \]
\[
\text{while } z < 10000 \\
\quad z \leftarrow z + 1 \\
\quad \text{if } x < y \text{ then } x \leftarrow x + 1 \\
\text{done}
\]
Beyond non-relational interferences
Relational thread-modular abstractions

Partially relational interferences

**Abstraction:** keep relations maintained by interferences

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

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

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

**Lock invariant:**

$$\{ \rho | \exists t \in \mathcal{T}, M: \langle M, \rho \rangle \in \alpha_{inv}^{A}(\emptyset(t)), m \notin M \}$$

- 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
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
Monotonicity abstraction

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

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

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

**Inference and use**

**gather:**
\(A_\text{mono}(t)(V) = \uparrow \iff\) all assignments to \(V\) in \(t\) 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 \(C[X \leq (V | [a, b])]\) can be strengthened into \(C[X \leq V]\)
Weakly relational interference example

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 \((y \text{ monotonic})\)

we can prove automatically that \(x \leq y\) holds

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

<table>
<thead>
<tr>
<th>analyzing (t_2)</th>
<th>(t_1)</th>
<th>(t_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y) unchanged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0 \leq x, x \leq y)</td>
<td></td>
<td>(0 \leq y \leq 102)</td>
</tr>
<tr>
<td>(y \leftarrow y + [1,3];)</td>
<td></td>
<td>unlocked(m)</td>
</tr>
</tbody>
</table>
Application: The AstréeA 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*
- *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>
Conclusion
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
- aware of scheduling, priorities and synchronization
- **parameterized** by (possibly relational) abstract domains (independent domains for state abstraction and interference abstraction)
Bibliography


