Thread-Modular Analysis of Concurrent Programs

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

Antoine Miné

Year 2022–2023

Course 16
20 February 2023
Concurrent programming

Principle: 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, $\times 2$ 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)
Introduction

Scope

In this course: static thread model
- implicit communications through shared memory
- explicit communications through synchronisation primitives
- fixed number of threads
- numeric programs

Goal: static analysis
- infer numeric program invariants
- parameterized by a choice of numeric abstract domains
- discover run-time errors
- discover data-races
- discover deadlocks
- application to analyzing embedded C programs
Outline

- Simple concurrent language
- Non-modular concurrent semantics
- Simple interference thread-modular concurrent semantics
- Weakly consistent memories
- Locks and synchronization
- Abstract rely-guarantee thread-modular concurrent semantics
- Relational interference abstractions
- 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

\[
\begin{align*}
\text{prog} &::= \ell \text{stmt}_1 \ell \parallel \ldots \parallel \ell \text{stmt}_n \ell \\
\ell \text{stmt} &::= \ell V \leftarrow \text{exp} \ell \\
&\quad \mid \ell \text{if} \exp \succeq 0 \text{then} \ell \text{stmt} \ell \text{fi} \ell \\
&\quad \mid \ell \text{while} \ell \exp \succeq 0 \text{do} \ell \text{stmt} \ell \text{done} \ell \\
&\quad \mid \ell \text{stmt}; \ell \text{stmt} \ell \\
\text{exp} &::= \ell V \mid [c_1, c_2] \mid - \text{exp} \mid \text{exp} \odot \text{exp} \\
c_1, c_2 &\in \mathbb{R} \cup \{+\infty, -\infty\}, \odot \in \{+, -, \times, /, \omega\}, \succeq \in \{=, <, \ldots\}
\end{align*}
\]
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 $0 \leq x \leq y \leq 102$
Language and semantics

Trace-based semantic model

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: 

\[ \text{prog} ::= \ell_1^i \text{stmt}_1 \ell_1^x \ | \ \cdots \ | \ \ell_n^i \text{stmt}_n \ell_n^x \]

Threads are numbered: 

\[ \mathbb{T} \overset{\text{def}}{=} \{1, \ldots, n\} \]

**Program states:** 

\[ \Sigma \overset{\text{def}}{=} (\mathbb{T} \rightarrow \mathcal{L}) \times \mathcal{E} \]

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

**Initial states:**

Threads start at their first control point \( \ell_t^i \), variables are set to 0:

\[ \mathcal{I} \overset{\text{def}}{=} \{\langle \lambda t.\ell_t^i, \lambda V.0 \rangle\} \]

**Actions:** Actions are thread identifiers:

\[ \mathcal{A} \overset{\text{def}}{=} \mathbb{T} \]
From concurrent programs to labelled transition systems

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

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

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

Maximal and finite prefix trace semantics are as before:

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

**Maximal traces:** \( \mathcal{M}_\infty \) (finite or infinite)
\[
\mathcal{M}_\infty \overset{\text{def}}{=} \{ \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \mid n \geq 0 \land \sigma_0 \in \mathcal{I} \land \sigma_n \in \mathcal{B} \land \forall i < n : \sigma_i \xrightarrow{t_i} \tau \sigma_{i+1} \} \cup \\
\{ \sigma_0 \xrightarrow{t_0} \sigma_1 \cdots \mid n \geq 0 \land \sigma_0 \in \mathcal{I} \land \forall i < \omega : \sigma_i \xrightarrow{t_i} \tau \sigma_{i+1} \}
\]

**Finite prefix traces:** \( \mathcal{T}_p \)
\[
\mathcal{T}_p \overset{\text{def}}{=} \{ \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \mid n \geq 0 \land \sigma_0 \in \mathcal{I} \land \forall i < n : \sigma_i \xrightarrow{t_i} \tau \sigma_{i+1} \}
\]
\[
\mathcal{T}_p = \text{lfp } F_p \text{ where } F_p(\mathcal{X}) = \mathcal{I} \cup \{ \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_n} \sigma_{n+1} \mid n \geq 0 \land \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in \mathcal{X} \land \sigma_n \xrightarrow{t_n} \tau \sigma_{n+1} \}
\]
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 \( \mathcal{M}_\infty \) 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 \( || \) \( x \leftarrow -2 \)
- **may not terminate without fairness**
- **always** terminates under **weak and strong fairness**

Finite prefix trace abstraction

\( \mathcal{M}_\infty \cap F \subseteq X \) is abstracted into testing \( \alpha_{\leq}(\mathcal{M}_\infty \cap F) \subseteq \alpha_{\leq}(X) \)

for all fairness conditions \( F \), \( \alpha_{\leq}(\mathcal{M}_\infty \cap F) = \alpha_{\leq}(\mathcal{M}_\infty) = \mathcal{T}_p \)

recall that \( \alpha_{\leq}(\mathcal{T}) \overset{\text{def}}{=} \{ t \in \Sigma^* \mid \exists u \in \mathcal{T} : t \preceq u \} \) is the finite prefix abstraction
and \( \mathcal{T} = \alpha_{\leq}(\mathcal{M}_\infty) \)

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

---

In the rest of the course, we ignore fairness conditions
**Reminder : Reachable state semantics:** \( R \in \mathcal{P}(\Sigma) \)

Reachable states in any execution:

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

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

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

Abstraction of the finite trace semantics.

\[
R = \alpha_p(T_p) \text{ where } \alpha_p(X) \overset{\text{def}}{=} \{ \sigma | \exists n \geq 0, \sigma_0 \xrightarrow{t_0} \cdots \sigma_n \in X: \sigma = \sigma_n \}
\]
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: \( \mathcal{P}(\mathcal{L} \times \mathcal{E}) \simeq \mathcal{L} \rightarrow \mathcal{P}(\mathcal{E}) \)

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

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

\( \Rightarrow \) set of recursive equations on \( \mathcal{X}_\ell \)

**Example:**

\[
\begin{align*}
\ell_1^1 & : i \leftarrow 2; \\
\ell_2^2 & : n \leftarrow [-\infty, +\infty]; \\
\ell_3^3 & : \text{while } \ell_4^4 : i < n \text{ do} \\
\ell_5^5 & : \text{if } [0, 1] = 0 \text{ then} \\
\ell_6^6 & : i \leftarrow i + 1 \\
\end{align*}
\]

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

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

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

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

- 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:**

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

- 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 \( \implies \) thread-modular analysis
Non-modular concurrent semantics
Equational concurrent state semantics

Equational form:

- for each $L \in T \rightarrow L$, a variable $X_L$ with value in $E$
- equations are derived from thread equations $eq(stmt_t)$ as:

$$X_{L_1} = \bigcup_{t \in T} \{ F(X_{L_2}, \ldots, X_{L_N}) \mid$$

$$\exists (X_{\ell_1} = F(X_{\ell_2}, \ldots, X_{\ell_N})) \in eq(stmt_t):$$

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

Join with $\cup$ equations from $eq(stmt_t)$ updating a single thread $t \in T$.

(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
  \[\implies\text{combinatorial explosion of abstract states}\]
- 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></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>

### Equation system:

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

<table>
<thead>
<tr>
<th>Example: inferring $0 \leq x \leq y \leq 102$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
</tr>
<tr>
<td>$\ell_1$ while random do</td>
</tr>
<tr>
<td>$\ell_2$ if $x &lt; y$ then</td>
</tr>
<tr>
<td>$\ell_3$ $x \leftarrow x + 1$</td>
</tr>
<tr>
<td>$t_2$</td>
</tr>
<tr>
<td>$\ell_4$ while random do</td>
</tr>
<tr>
<td>$\ell_5$ if $y &lt; 100$ then</td>
</tr>
<tr>
<td>$\ell_6$ $y \leftarrow y + [1,3]$</td>
</tr>
</tbody>
</table>

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

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

- 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
  \[\Rightarrow\] 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
  - 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
  - one more level of fixpoint iteration
Example

\begin{align*}
\textit{t}_1 & \quad \ell_1 & \text{while random do} \\
& \quad \ell_2 & \text{if } x < y \text{ then} \\
& \quad \ell_3 & x \leftarrow x + 1 \\
\textit{t}_2 & \quad \ell_4 & \text{while random do} \\
& \quad \ell_5 & \text{if } y < 100 \text{ then} \\
& \quad \ell_6 & y \leftarrow y + [1,3]
\end{align*}
Example

\[
\begin{align*}
\text{Analysis of } t_1 \text{ in isolation} & \\
(1): x = y = 0 & \quad \mathcal{X}_1 = I \\
(2): x = y = 0 & \quad \mathcal{X}_2 = \mathcal{X}_1 \cup C[x \leftarrow x + 1] \mathcal{X}_3 \cup C[x \geq y] \mathcal{X}_2 \\
(3): \bot & \quad \mathcal{X}_3 = C[x < y] \mathcal{X}_2
\end{align*}
\]
**Simple interference semantics**

**Intuition**

**Example**

### t1

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

### t2

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

---

**Analysis of t2 in isolation**

1. \( x = y = 0 \) \( X_4 = \emptyset \)
2. \( x = 0, y \in [0, 102] \) \( X_5 = X_4 \cup C[y \leftarrow y + [1, 3]] \cup C[y \geq 100] X_5 \)
3. \( x = 0, y \in [0, 99] \) \( X_6 = C[y < 100] X_5 \)

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

\[ \begin{align*}
    &t_1 \\
    &\ell_1 \text{ while random do} \\
    &\quad \ell_2 \text{ if } x < y \text{ then} \\
    &\quad \quad \ell_3 x \leftarrow x + 1 \\
    &t_2 \\
    &\ell_4 \text{ while random do} \\
    &\quad \ell_5 \text{ if } y < 100 \text{ then} \\
    &\quad \quad \ell_6 y \leftarrow y + [1, 3]
\end{align*} \]

Re-analysis of \( t_1 \) with interferences from \( t_2 \)

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

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

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

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

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

#### Derived abstract analysis:
- similar to a **sequential** program analysis, but iterated
  can be parameterized by arbitrary abstract domains
- **efficient** few reanalisys 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 $\mathcal{I} \overset{\text{def}}{=} T \times V \times \mathbb{R}$

$\langle t, X, v \rangle$ means that $t$ 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** : $E_t[\text{exp}] : \mathcal{E} \times \mathcal{P}(\mathcal{I}) \rightarrow \mathcal{P}(\mathbb{R}) \times \mathcal{P}(\Omega)$ for thread $t$

- add interference $I \in \mathcal{I}$, 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:
  
  $$E_t[X] \langle \rho, I \rangle \overset{\text{def}}{=} \langle \{ \rho(X) \} \cup \{ v | \exists u \neq t : \langle u, X, v \rangle \in I \} \rangle, \emptyset \rangle$$

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

- etc.
Denotational semantics with interferences (cont.)

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

$$C_t[\text{stmt}] : \mathcal{P}(E) \times \mathcal{P}(%0x09)\Omega) \times \mathcal{P}(I) \rightarrow \mathcal{P}(E) \times \mathcal{P}(\Omega) \times \mathcal{P}(I)$$

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

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

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

... 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: $P[prog] \subseteq \Omega$

Given $prog ::= stmt_1 \parallel \cdots \parallel stmt_n$, we compute:

$$P[prog] \overset{\text{def}}{=} \bigcup_{t \in T} \lambda \langle O, I \rangle \cdot [C_t[stmt_t] \langle E_0, \emptyset, I \rangle]_{\Omega, I}$$

- Each thread analysis starts in an initial environment set $E_0 \overset{\text{def}}{=} \{ \lambda V.0 \}$
- $[X]_{\Omega, I}$ projects $X \in P(E) \times P(\Omega) \times P(\emptyset)$ on $P(\Omega) \times P(\emptyset)$ and interferences and errors from all threads are joined
- The output environments from a thread analysis are not easily exploitable
- $P[prog]$ only outputs the set of possible run-time errors

We will need to prove the soundness of $P[prog]$ with respect to the interleaving semantics...
Interference abstraction

**Abstract interferences $\#$**

$\mathcal{P}(\emptyset) \overset{\text{def}}{=} \mathcal{P}(\top \times \emptyset \times \mathbb{R})$ is abstracted as $\# \overset{\text{def}}{=} (\top \times \emptyset) \rightarrow \mathcal{R}\#$

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

**Abstract semantics with interferences $C^\#_t[e]$**

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

Abstract analysis

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

- **effective** analysis by **structural induction**
- \( P^\#[\text{prog}] \) is sound with respect to \( P[\text{prog}] \)
- 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: \([X]_{\Omega}, [Y]_{\Omega, I^\#}\) keep only \(X\)'s component in \(\Omega\), \(Y\)'s components in \(\Omega\) and \(I^\#\)
Path-based soundness proof
Control paths of a sequential program

**atomic** ::= \( X \leftarrow \text{exp} \mid \text{exp} \triangleright 0 \)

**Control paths**

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

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

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

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

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

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

e.g. \( \pi(i \leftarrow 0; \text{while } i < 10 \text{ do } i \leftarrow i + 1 \text{ done}; x \leftarrow i) = i \leftarrow 0 \cdot (i < 10 \cdot i \leftarrow i + 1)^* \cdot x \leftarrow i \)
Join-over-all-path semantics

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

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

Semantic equivalence

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

no longer true in the abstract
Path-based concrete semantics of concurrent programs

**Concurrent control paths**

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

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

**Interleaving program semantics**

\[ P_\ast[\text{prog}] \overset{\text{def}}{=} [\prod[\pi_\ast] \langle E_0, \emptyset \rangle]_\Omega \]

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

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

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

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

**Notes:**

- sound but not complete
- can be extended to soundness proof under weakly consistent memories
Weakly consistent memories
Issues with weak consistency

program written

\[
\begin{align*}
F_1 &\leftarrow 1; \\
\text{if } F_2 &= 0 \text{ then} \\
S_1 &
\end{align*}
\]

\[
\begin{align*}
F_2 &\leftarrow 1; \\
\text{if } F_1 &= 0 \text{ then} \\
S_2 &
\end{align*}
\]

(simplified Dekker mutual exclusion algorithm)

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

<table>
<thead>
<tr>
<th>program written</th>
<th>program executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_1 \leftarrow 1$; if $F_2 = 0$ then $S_1$ fi</td>
<td>if $F_2 = 0$ then $F_1 \leftarrow 1$; if $F_1 = 0$ then $S_1$ fi</td>
</tr>
<tr>
<td>$F_2 \leftarrow 1$; $S_2$ fi</td>
<td>$F_2 \leftarrow 1$; $S_2$ fi</td>
</tr>
</tbody>
</table>

(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

We should not have $R_1 = 42$. 

(Original program)

<table>
<thead>
<tr>
<th>Original program</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1 \leftarrow X$; $R \leftarrow Y$; $Y \leftarrow R_1$; $X \leftarrow R_2$</td>
</tr>
</tbody>
</table>

(example from causality test case #4 for Java by Pugh et al.)
Out of thin air principle

We should not have $R_1 = 42$.

Possible if we allow speculative writes!

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

(also forbidden in Java)
Atomicity and granularity

original program

\[
X \leftarrow X + 1 \quad | \quad X \leftarrow X + 1
\]

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: \( p \leadsto q \)
only reduce interferences and errors

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

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

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

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

- \ldots

but NOT:

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

Soundness of the interference semantics

Interleaving semantics of transformed programs $P'_*[\text{prog}]$

- $\pi'(s) \overset{\text{def}}{=} \{ p \mid \exists p' \in \pi(s) : p' \leadsto^* p \}$
- $\pi'_* \overset{\text{def}}{=} \{ \text{interleavings of } \pi'(\text{stmt}_t), t \in T \}$
- $P'_*[\text{prog}] \overset{\text{def}}{=} [\Pi[\pi'_*]\langle \mathcal{E}_0, \emptyset \rangle]_\Omega$

Soundness theorem

$P'_*[\text{prog}] \subseteq P[\text{prog}]$

$\implies$ the interference semantics is sound wrt. weakly consistent memories and changes of granularity
Locks and synchronization
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 \( \mathbb{C} \)

\[
\begin{align*}
\mathcal{E} & \rightsquigarrow \mathcal{E} \times \mathbb{C}, \quad \mathcal{E}' \rightsquigarrow \mathbb{C} \rightarrow \mathcal{E}' \\
\mathcal{I} & \overset{\text{def}}{=} \mathbb{T} \times \mathbb{V} \times \mathbb{R} \rightsquigarrow \mathcal{I} \overset{\text{def}}{=} \mathbb{T} \times \mathbb{C} \times \mathbb{V} \times \mathbb{R}, \\
\mathcal{I}' & \overset{\text{def}}{=} (\mathbb{T} \times \mathbb{V}) \rightarrow \mathcal{R}' \rightsquigarrow \mathcal{I}' \overset{\text{def}}{=} (\mathbb{T} \times \mathbb{C} \times \mathbb{V}) \rightarrow \mathcal{R}' \\
\mathbb{C} & \overset{\text{def}}{=} \mathbb{C}_{\text{race}} \cup \mathbb{C}_{\text{sync}} \text{ separates} \\
\text{data-race writes } & \mathbb{C}_{\text{race}} \\
\text{well-synchronized writes } & \mathbb{C}_{\text{sync}}
\end{align*}
\]
Mutual exclusion

Data-race effects \( \mathcal{C}_{\text{race}} \simeq \mathcal{P}(M) \)

Across read / write not protected by a mutex.
Partition wrt. mutexes \( M \subseteq \mathbb{M} \) held by the current thread \( t \).

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

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

Well-synchronized effects

\[ \mathcal{C}_{sync} \simeq \mathbb{M} \times \mathcal{P}(\mathbb{M}) \]

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

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

**abstract consumer/producer**

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

- **no data-race interference**

- **well-synchronized interferences:**
  - **consumer**: $x \leftarrow [0, 99]$
  - **producer**: $x \leftarrow [1, 100]$

  $\Rightarrow$ we can prove that $y \in [0, 100]$

  without locks, we cannot get $y \leq 100$

Can be generalized to several consumers and producers.
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: $R \subseteq \mathbb{T} \times \mathcal{P}(\mathbb{M})$
- lock instructions from these configurations $R \times \mathbb{M}$

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.

generalization to larger cycles, with more threads involved in a deadlock, is easy
Priority-based scheduling

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

Analysis: refined transfer of interference based on priority
- partition interferences wrt. thread and priority
  - support for manual priority change, and for priority ceiling protocol
- higher priority processes inject state from yield into every point
- lower priority processes inject data-race interferences into yield
Beyond non-relational interferences
Inspiration from program logics
Reminder: Floyd–Hoare logic

Logic to prove properties about sequential programs [Hoar69].

**Hoare triples:** \{P\} stmt \{Q\}

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

\[
\begin{align*}
\{P[e/X]\} X & \leftarrow e \{P\} \\
\{P\} s_1 \{Q\} \quad \{Q\} s_2 \{R\} & \quad \{P\} \text{ if } e \triangleright 0 \text{ then } s \text{ fi} \{Q\} \\
\{P\} s_1 ; s_2 \{R\} & \quad \{P \wedge e \triangleright 0\} s \{P\} \\
\{P \wedge e \triangleright 0\} & \quad \{P\} \text{ while } e \triangleright 0 \text{ do } s \text{ done } \{P \wedge e \triangleright 0\} \\
\{P'\} s \{Q'\} \quad P \Rightarrow P' \quad Q' \Rightarrow Q & \quad \{P\} s \{Q\}
\end{align*}
\]

Link with abstract interpretation:
- the equations reachability semantics \(\langle X_\ell \rangle_{\ell \in \mathcal{L}}\) 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 before stmt is executed
- and the effect of other threads is included in $R$ (rely)
- then $Q$ is true after stmt
- and the effect of 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:

$$
\frac{R \lor G_2, G_1 \vdash \{P_1\} s_1 \{Q_1\} \quad R \lor G_1, G_2 \vdash \{P_2\} s_2 \{Q_2\}}{R, G_1 \lor G_2 \vdash \{P_1 \land P_2\} s_1 || s_2 \{Q_1 \land Q_2\}}
$$
## Rely-guarantee example

<table>
<thead>
<tr>
<th>Checking $t_1$</th>
<th>Checking $t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell^1$ while random do $\ell^2$ if $x &lt; y$ then $\ell^3$ $x \leftarrow x+1$ fi done</td>
<td></td>
</tr>
<tr>
<td>$\ell^4$ while random do $\ell^5$ if $y &lt; 100$ then $\ell^6$ $y \leftarrow y + [1,3]$ fi done</td>
<td></td>
</tr>
</tbody>
</table>

- $\ell^1$: $x = y = 0$
- $\ell^2$: $x, y \in [0, 102], x \leq y$
- $\ell^3$: $x \in [0, 101], y \in [1, 102], x < y$
- $\ell^4$: $x = y = 0$
- $\ell^5$: $x, y \in [0, 102], x \leq y$
- $\ell^6$: $x \in [0, 99], y \in [0, 99], x \leq y$
Rely-guarantee example

### checking $t_1$

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

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

### checking $t_2$

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

- $\ell_4$: $x = y = 0$
- at $\ell_5$: $x, y \in [0, 102]$, $x \leq y$
- at $\ell_6$: $x \in [0, 99]$, $y \in [0, 99]$, $x \leq y$

In this example:

- guarantee exactly what is relied on ($R_1 = G_1$ 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
Main idea: **separate** execution steps

- from the **current thread** \( a \)
  - found by analysis by induction on the syntax of \( a \)

- from **other threads** \( b \)
  - given as parameter in the analysis of \( a \)
  - inferred during the analysis of \( b \)

\[ \Rightarrow \text{express the semantics from the point of view of a single thread} \]
Trace decomposition

Reachable states projected on thread $t$: $\mathcal{R}l(t)$

- attached to thread control point in $L$, not control state in $T \to L$
- remember other thread's control point as "auxiliary variables"
  (required for completeness)

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

where $\pi_t(R) \overset{\text{def}}{=} \{ \langle L(t), \rho [\forall t' \neq t: pc_{t'} \mapsto L(t')] \rangle | \langle L, \rho \rangle \in R \}$
Trace decomposition

**Interferences generated by** \( t \): \( A(t) \)  
(\( \simeq \) guarantees on transitions)

Extract the transitions with action \( t \) observed in \( T_p \)  
(subset of the transition system, containing only transitions actually used in reachability)

\[
A(t) \overset{\text{def}}{=} \alpha^I(T_p)(t)
\]

where \( \alpha^I(X)(t) \overset{\text{def}}{=} \{ \langle \sigma_i, \sigma_{i+1} \rangle \mid \exists \sigma_0 \xrightarrow{a_0} \sigma_1 \cdots \xrightarrow{a_{n-1}} \sigma_n \in X : a_i = t \} \)
Thread-modular concrete semantics

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

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

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

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

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 $A$
We express $R_{l}(t)$ and $A(t)$ directly from the transition system, without computing $T_{p}$

**Interferences:** $A$

Collect transitions from a thread $t$ and reachable states $R$:

$$A(t) = B(R_{l})(t), \text{ where}$$

$$B(Z)(t) \overset{\text{def}}{=} \{ \langle \sigma, \sigma' \rangle \mid \pi_{t}(\sigma) \in Z(t) \land \sigma \xrightarrow{\tau}^{t} \sigma' \}$$
Thread-modular concrete semantics

We express $\mathcal{RI}(t)$ and $A(t)$ directly from the transition system, without computing $T_p$.

Recursive definition:
- $\mathcal{RI}(t) = \text{lfp } R_t(A)$
- $A(t) = B(\mathcal{RI})(t)$

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

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

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

any property provable with the interleaving semantics can be proven with the thread-modular semantics!
Fixpoint form

**Constructive fixpoint form:**

Use Kleene’s iteration to construct fixpoints:

- \( Rl = \text{lfp} \ H = \bigcup_{n \in \mathbb{N}} H^n(\lambda t.\emptyset) \)
  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)

\[ \implies \text{nested iterations} \]
Abstract rely-guarantee

**Suggested algorithm:** nested iterations with acceleration

once abstract domains for states and interferences are chosen

- start from $\mathcal{R}l_0^\# \overset{\text{def}}{=} A_0^\# \overset{\text{def}}{=} \lambda t. \bot^\#$
- while $A_n^\#$ is not stable
  - compute $\forall t \in \mathbb{T}: \mathcal{R}l_{n+1}^\#(t) \overset{\text{def}}{=} \text{lfp } R^\#_t(A_n^\#)$ by iteration with widening $\nabla$
    $(\simeq$ separate analysis of each thread$)$
  - compute $A_{n+1}^\# \overset{\text{def}}{=} A_n^\# \nabla B^\#(\mathcal{R}l_{n+1}^\#)$
  - when $A_n^\# = A_{n+1}^\#$, return $\mathcal{R}l_n^\#$

$\implies$ thread-modular analysis parameterized by abstract domains (only source of approximation)
able to easily reuse existing sequential analyses
Retrieving 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

\[ \alpha_R^{nf}(X) \overset{\text{def}}{=} \{ \langle \ell, \rho \rangle \mid \langle \ell, \rho \rangle \in X \} \cup X \]

Interference abstraction: remove all control state

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

**Flow-insensitive fixpoint semantics:**

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

$$\mathcal{R}^{nf} \overset{\text{def}}{=} \text{lfp } \lambda Z. \lambda t. \text{lfp } R^{nf}_t(B^{nf}(Z)),$$

where

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

- $R^{loc}_t(X) \overset{\text{def}}{=} \{ \langle \ell^i_t, \lambda V.0 \rangle \} \cup \{ \langle \ell', \rho' \rangle \mid \exists \langle \ell, \rho \rangle \in X: \langle \ell, \rho \rangle \rightarrow_t \langle \ell', \rho' \rangle \}$ (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:

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

- 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 \( \mathbb{V} \rightarrow \mathcal{D}^\# \)
A note on unbounded thread creation

**Extension:** relax the finiteness constraint on $T$

- there is still a **finite syntactic set** of threads $T_s$
- some threads $T_\infty \subseteq T_s$ can have several instances
  (possibly an unbounded number)

**Flow-insensitive analysis:**

- local state and interference domains have finite dimensions
  $(E_t$ and $(L \times E) \times (L \times E)$, as opposed to $E$ and $E \times E$)
- all instances of a thread $t \in T_s$ are isomorphic
  $\implies$ iterate the analysis on the finite set $T_s$ (instead of $T$)
- we must handle **self-interferences** for threads in $T_\infty$:

$$A^nf_t(Y)(X) \overset{\text{def}}{=} \{ (\ell, \rho') | \exists \rho, u: (u \neq t \lor t \in T_\infty) \land (\ell, \rho) \in X \land (\rho, \rho') \in Y(u) \}$$
From traces to thread-modular analyses

abstract states
\((\mathbb{T} \times \mathcal{L}) \rightarrow \mathcal{E}^\#\)  
abstract interferences
\(\mathbb{T} \rightarrow \mathcal{E}^\#\)

local states
\((\mathbb{T} \times \mathcal{L}) \rightarrow \mathcal{P}(\mathcal{E})\)
non-relational interferences
\(\mathbb{T} \rightarrow \mathcal{P}(\mathcal{E})\)

flow-insensitive interferences
\((\mathbb{T} \times \mathcal{L}) \rightarrow \mathcal{P}(\mathcal{E} \times \mathcal{E})\)

interferences
\(\mathbb{A} : \mathbb{T} \rightarrow \mathcal{P}(\Sigma \times \Sigma)\)

interleaved execution trace prefixes
\(\mathcal{T}_p \in \mathcal{P}(\Sigma^*)\)

static analyzer  
static analyzer

rely-guarantee  
rely-guarantee

concrete executions
Relational thread-modular abstractions
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 $p_{c_b} \in \forall_a$ 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 $\mathcal{P}((\forall_a \rightarrow \mathbb{Z}) \times (\forall_a \rightarrow \mathbb{Z})) \sim \mathcal{P}((\forall_a \cup \forall'_a) \rightarrow \mathbb{Z})$

- $X \in \forall_a$ 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.: $\{ (x, x + 1) \mid x \in [0, 10] \}$ is represented as $x' = x + 1 \land x \in [0, 10]$

$\Rightarrow$ 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)

<table>
<thead>
<tr>
<th>( t_1 )</th>
<th>( t_2 )</th>
</tr>
</thead>
</table>
| \[ \text{while } z < 10000 \]  
  \[ \quad z \leftarrow z + 1 \]  
  \[ \quad \text{if } y < c \text{ then } y \leftarrow y + 1 \]  
  \[ \text{done} \] | \[ \text{while } z < 10000 \]  
  \[ \quad z \leftarrow z + 1 \]  
  \[ \quad \text{if } x < y \text{ then } x \leftarrow x + 1 \]  
  \[ \text{done} \] |

\[
\begin{array}{c}
\text{time (s)} \\
\hline
0.08 & 2.27 & 6.00 \\
\end{array}
\]

![Graph showing scalability in the number of threads](image)
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

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

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

**Lock invariant:**

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

- 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
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
**Monotonicity abstraction**

**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 \( C[ X \leq (V | [a, b])] \) 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
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 \( \Rightarrow \) 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</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


