

Static Analysis of Concurrent Programs

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

Antoine Miné

Year 2018–2019

Course 16

12 February 2019

Concurrent programming

Idea:

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

Why concurrent programs?

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

“Free lunch is over”
change in Moore's law (×2 transistors every 2 years)
- exploit several computers
(distributed computing)
- ease of programming
(GUI, network code, reactive programs)

Scope

In this course: **static thread model**

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

Goal: **static analysis**

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

Outline

- From sequential to concurrent abstract interpreters
 - alternate sequential semantics
(a denotational semantics with run-time errors)
 - interleaving concurrent semantics (not scalable)
 - interference-based semantics (non-relational)
 - weakly consistent memory models
 - synchronization: data-races, priorities, locks, and deadlocks
- Abstract rely-guarantee
 - rely-guarantee proof method
 - a complete modular concrete semantics
 - examples of relational interference abstractions
- Application : the AstréeA analyzer

Simple structured numeric language

- finite set of (toplevel) **threads**: prog_1 to prog_n
- finite set of numeric program variables $V \in \mathbb{V}$
- finite set of statement locations $\ell \in \mathcal{L}$
- finite set of potential error locations $\omega \in \Omega$

Structured language syntax

$\text{parprog} ::= \ell \text{prog}_1^\ell \parallel \dots \parallel \ell \text{prog}_n^\ell$ *(parallel composition)*

$\ell \text{prog}^\ell ::= \ell V \leftarrow \text{exp}^\ell$ *(assignment)*

| $\ell \text{if } \text{exp} \bowtie 0 \text{ then } \ell \text{prog}^\ell \text{ fi}$ *(conditional)*

| $\ell \text{while } \ell \text{exp} \bowtie 0 \text{ do } \ell \text{prog}^\ell \text{ done}^\ell$ *(loop)*

| $\ell \text{prog}; \ell \text{prog}^\ell$ *(sequence)*

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

$c_1, c_2 \in \mathbb{R} \cup \{+\infty, -\infty\}, \diamond \in \{+, -, \times, /\}, \bowtie \in \{=, <, \dots\}$

Sequential semantics: reminders

Reminder: transition systems

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

- Σ : set of program states
- $\tau \subseteq \Sigma \times \Sigma$: transition relation (we note $(\sigma, \sigma') \in \tau$ as $\sigma \rightarrow_{\tau} \sigma'$)
- $\mathcal{I} \subseteq \Sigma$: initial states

Reminder: traces of a transition system

Maximal trace semantics: $\mathcal{M}_\infty \in \mathcal{P}(\Sigma^\infty)$

Total executions $\sigma_0, \dots, \sigma_n, \dots$

- starting in an initial state $\sigma_0 \in \mathcal{I}$ and either
- **ending** in a blocking state in $\mathcal{B} \stackrel{\text{def}}{=} \{ \sigma \mid \forall \sigma': \sigma \not\rightarrow_\tau \sigma' \}$
- or **infinite**

$$\mathcal{M}_\infty \stackrel{\text{def}}{=} \{ \sigma_0, \dots, \sigma_n \mid \sigma_0 \in \mathcal{I} \wedge \sigma_n \in \mathcal{B} \wedge \forall i < n: \sigma_i \rightarrow_\tau \sigma_{i+1} \} \cup \{ \sigma_0, \dots, \sigma_n \dots \mid \sigma_0 \in \mathcal{I} \wedge \forall i < \omega: \sigma_i \rightarrow_\tau \sigma_{i+1} \}$$

Reminder: prefix trace abstraction

Finite prefix trace semantics: $\mathcal{T}_p \in \mathcal{P}(\Sigma^*)$

Finite prefixes of executions (semantics of testing):

$$\mathcal{T}_p \stackrel{\text{def}}{=} \{ \sigma_0, \dots, \sigma_n \mid n \geq 0, \sigma_0 \in \mathcal{I}, \forall i < n: \sigma_i \rightarrow_{\tau} \sigma_{i+1} \}$$

Can prove **safety** properties, but not **liveness** (termination, etc.).

Abstraction of the maximal trace semantics.

$$\mathcal{T}_p = \text{lfp } F_p \text{ where } F_p(X) = \mathcal{I} \cup \{ \sigma_0, \dots, \sigma_{n+1} \mid \sigma_0, \dots, \sigma_n \in X \wedge \sigma_n \rightarrow_{\tau} \sigma_{n+1} \}$$

Reminder: reachable state abstraction

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

Reachable states in any execution:

$$\mathcal{R} \stackrel{\text{def}}{=} \{ \sigma \mid \exists n \geq 0, \sigma_0, \dots, \sigma_n: \sigma_0 \in \mathcal{I}, \forall i < n: \sigma_i \rightarrow_{\tau} \sigma_{i+1} \wedge \sigma = \sigma_n \}$$

Can prove (non-)reachability, but not ordering.

Abstraction of the finite trace semantics.

$$\mathcal{R} = \text{lfp } F_{\mathcal{R}} \text{ where } F_{\mathcal{R}}(X) = \mathcal{I} \cup \{ \sigma \mid \exists \sigma' \in X: \sigma' \rightarrow_{\tau} \sigma \}$$

States of a sequential program, with errors

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

Program states: $\Sigma \stackrel{\text{def}}{=} (\mathcal{L} \times \mathcal{E}) \cup \Omega$

- a **control** state in \mathcal{L} , and
- either a **memory** state: an environment in $\mathcal{E} \stackrel{\text{def}}{=} \mathbb{V} \rightarrow \mathbb{R}$
- or an **error state**, in Ω

Initial states:

start at the first control point ℓ^i with variables set to 0:

$$\mathcal{I} \stackrel{\text{def}}{=} \{ (\ell^i, \lambda V.0) \}$$

Note that $\mathcal{P}(\Sigma) \simeq (\mathcal{L} \rightarrow \mathcal{P}(\mathcal{E})) \times \mathcal{P}(\Omega)$:

- a state property in $\mathcal{P}(\mathcal{E})$ at each program point in \mathcal{L}
- and a set of errors in $\mathcal{P}(\Omega)$

Expression semantics with errors

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

$$E[V] \rho \stackrel{\text{def}}{=} \langle \{ \rho(V) \}, \emptyset \rangle$$

$$E[[c_1, c_2]] \rho \stackrel{\text{def}}{=} \langle \{ x \in \mathbb{R} \mid c_1 \leq x \leq c_2 \}, \emptyset \rangle$$

$$E[-e] \rho \stackrel{\text{def}}{=} \text{let } \langle V, O \rangle = E[e] \rho \text{ in} \\ \langle \{ -v \mid v \in V \}, O \rangle$$

$$E[e_1 \diamond_{\omega} e_2] \rho \stackrel{\text{def}}{=} \text{let } \langle V_1, O_1 \rangle = E[e_1] \rho \text{ in} \\ \text{let } \langle V_2, O_2 \rangle = E[e_2] \rho \text{ in} \\ \langle \{ v_1 \diamond v_2 \mid v_i \in V_i, \diamond \neq / \vee v_2 \neq 0 \}, \\ O_1 \cup O_2 \cup \{ \omega \text{ if } \diamond = / \wedge 0 \in V_2 \} \rangle$$

- defined by structural induction on the syntax
- evaluates in an environment ρ to a **set of values**
- also returns a set of **accumulated errors**
(here, only divisions by zero)

Semantics in denotational form

Input-output function $C[\text{prog}]$

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

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

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

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

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

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

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

- mutate memory states in \mathcal{E} , accumulate errors in Ω
(\sqcup is the element-wise \cup in $\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)$)
- structured: nested loops yield nested fixpoints
- big-step: forget information on intermediate locations ℓ

Abstract semantics in denotational form

Extend a numeric abstract domain $\mathcal{E}^\#$ abstracting $\mathcal{P}(\mathcal{E})$ to $\mathcal{D}^\# \stackrel{\text{def}}{=} \mathcal{E}^\# \times \mathcal{P}(\Omega)$.

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

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

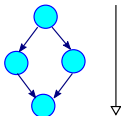
$$C^\#[\text{if } e \bowtie 0 \text{ then } s \text{ fi}] X^\# \stackrel{\text{def}}{=} (C^\#[s] \circ C^\#[e \bowtie 0]) X^\# \sqcup^\# C^\#[e \nabla 0] X^\#$$

$$C^\#[\text{while } e \bowtie 0 \text{ do } s \text{ done}] X^\# \stackrel{\text{def}}{=} C^\#[e \nabla 0] (\text{lim} \lambda Y^\#. Y^\# \nabla (X^\# \sqcup (C^\#[s] \circ C^\#[e \bowtie 0]) Y^\#))$$

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

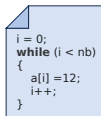
- the abstract interpreter mimics an actual interpreter

Equational vs. denotational form

Equational:

$$\left\{ \begin{array}{l} \mathcal{X}_1 = \top \\ \mathcal{X}_2 = F_2(\mathcal{X}_1) \\ \mathcal{X}_3 = F_3(\mathcal{X}_1) \\ \mathcal{X}_4 = F_4(\mathcal{X}_3, \mathcal{X}_4) \end{array} \right.$$

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

Denotational:

$$\begin{aligned} C[\text{while } c \text{ do } b \text{ done}] X &\stackrel{\text{def}}{=} \\ &C[\neg c] (\text{Ifp } \lambda Y. X \cup C[b] (C[c] Y)) \\ C[\text{if } c \text{ then } t \text{ fi}] X &\stackrel{\text{def}}{=} \\ &C[t] (C[c] X) \cup C[\neg c] X \\ &\dots \end{aligned}$$

- 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

Concurrent semantics

Multi-thread execution model

t_1	t_2
ℓ_1 while random do ℓ_2 if $x < y$ then ℓ_3 $x \leftarrow x + 1$	ℓ_4 while random do ℓ_5 if $y < 100$ then ℓ_6 $y \leftarrow y + [1,3]$

Execution model:

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

\implies we have the global invariant $0 \leq x \leq y \leq 102$

Labelled transition systems

Labelled transition system: $(\Sigma, \mathcal{A}, \tau, \mathcal{I})$

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

Labelled traces: sequences of states interspersed with actions

denoted as $\sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \cdots \sigma_n \xrightarrow{a_n} \sigma_{n+1}$

From concurrent programs to labelled transition systems

Notations:

- concurrent program:

$$\text{parprog} ::= \ell_1^i \text{prog}_1^{\ell_1^x} \parallel \dots \parallel \ell_n^i \text{prog}_n^{\ell_n^x}$$

- threads identifiers: $\mathbb{T} \stackrel{\text{def}}{=} \{1, \dots, n\}$

Program states: $\Sigma \stackrel{\text{def}}{=} ((\mathbb{T} \rightarrow \mathcal{L}) \times \mathcal{E}) \cup \Omega$

- a **control** state $L(t) \in \mathcal{L}$ for each thread $t \in \mathbb{T}$ and
- a single **shared memory** state $\rho \in \mathcal{E}$
- or an error state $\omega \in \Omega$

Initial states:

threads start at their first control point ℓ_t^i , variables are set to 0:

$$\mathcal{I} \stackrel{\text{def}}{=} \{(\lambda t. \ell_t^i, \lambda V. 0)\}$$

Actions: thread identifiers: $\mathcal{A} \stackrel{\text{def}}{=} \mathbb{T}$

From concurrent programs to labelled transition systems

Transition relation: $\tau \subseteq \Sigma \times \mathcal{A} \times \Sigma$

$$(L, \rho) \xrightarrow{t}_{\tau} (L', \rho') \iff (L(t), \rho) \rightarrow_{\tau[\text{prog}_t]} (L'(t), \rho') \wedge \forall u \neq t: L(u) = L'(u)$$

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

- based on the transition relation of individual threads seen as sequential processes prog_t :

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

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

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

- each $\sigma \rightarrow \sigma'$ in $\tau[\text{prog}_t]$ leads to **many transitions** in τ !

Interleaved trace semantics

Maximal and finite prefix trace semantics are as before:

Blocking states: $\mathcal{B} \stackrel{\text{def}}{=} \{ \sigma \mid \forall \sigma' : \forall t : \sigma \not\stackrel{t}{\rightarrow}_{\tau} \sigma' \}$

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

$$\mathcal{M}_{\infty} \stackrel{\text{def}}{=} \{ \sigma_0 \xrightarrow{t_0} \dots \xrightarrow{t_{n-1}} \sigma_n \mid n \geq 0 \wedge \sigma_0 \in \mathcal{I} \wedge \sigma_n \in \mathcal{B} \wedge \forall i < n : \sigma_i \xrightarrow{t_i}_{\tau} \sigma_{i+1} \} \cup \\ \{ \sigma_0 \xrightarrow{t_0} \sigma_1 \dots \mid n \geq 0 \wedge \sigma_0 \in \mathcal{I} \wedge \forall i < \omega : \sigma_i \xrightarrow{t_i}_{\tau} \sigma_{i+1} \}$$

Finite prefix traces: \mathcal{T}_p

$$\mathcal{T}_p \stackrel{\text{def}}{=} \{ \sigma_0 \xrightarrow{t_0} \dots \xrightarrow{t_{n-1}} \sigma_n \mid n \geq 0 \wedge \sigma_0 \in \mathcal{I} \wedge \forall i < n : \sigma_i \xrightarrow{t_i}_{\tau} \sigma_{i+1} \}$$

$\mathcal{T}_p = \text{lfp } F_p$ where

$$F_p(X) = \mathcal{I} \cup \{ \sigma_0 \xrightarrow{t_0} \dots \xrightarrow{t_n} \sigma_{n+1} \mid n \geq 0 \wedge \sigma_0 \xrightarrow{t_0} \dots \xrightarrow{t_{n-1}} \sigma_n \in X \wedge \sigma_n \xrightarrow{t_n}_{\tau} \sigma_{n+1} \}$$

Fairness

Fairness conditions: avoid threads being denied to run

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

an infinite trace $\sigma_0 \xrightarrow{t_0} \dots \sigma_n \xrightarrow{t_n} \dots$ is:

- **weakly fair** if $\forall t \in \mathbb{T}$:
 $(\exists i: \forall j \geq i: enabled(\sigma_j, t)) \implies (\forall i: \exists j \geq i: a_j = t)$
 (no thread can be continuously enabled without running)
- **strongly fair** if $\forall t \in \mathbb{T}$:
 $(\forall i: \exists j \geq i: enabled(\sigma_j, t)) \implies (\forall i: \exists j \geq i: a_j = t)$
 (no thread can be infinitely often enabled without running)

Proofs under fairness conditions given:

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

Fairness (cont.)

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

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

Finite prefix trace abstraction

$\mathcal{M}_\infty \cap F \subseteq X$ is abstracted into testing $\alpha_{*\underline{\gamma}}(\mathcal{M}_\infty \cap F) \subseteq \alpha_{*\underline{\gamma}}(X)$

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

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

In the following, we ignore fairness conditions.

(see [Cous85])

Equational state semantics

State abstraction \mathcal{R} : as before

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

Equational form: (without handling errors in Ω)

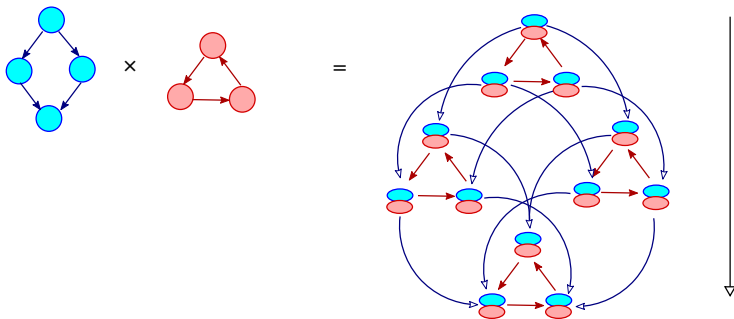
- for each $L \in \mathbb{T} \rightarrow \mathcal{L}$, a variable \mathcal{X}_L with value in \mathcal{E}
- equations are derived from thread equations $eq(\text{prog}_t)$ as:

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

Join with \cup equations from $eq(\text{prog}_t)$ updating a single thread $t \in \mathbb{T}$.

(See course 3 for the full definition of $eq(\text{prog})$.)

Equational state semantics (illustration)



Product of control-flow graphs:

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

Equational state semantics (example)

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

t_1	t_2
ℓ^1 while random do ℓ^2 if $x < y$ then ℓ^3 $x \leftarrow x + 1$	ℓ^4 while random do ℓ^5 if $y < 100$ then ℓ^6 $y \leftarrow y + [1,3]$

Equation system:

$$\mathcal{X}_{1,4} = \mathcal{I}$$

$$\mathcal{X}_{2,4} = \mathcal{X}_{1,4} \cup C[x \geq y] \mathcal{X}_{2,4} \cup C[x \leftarrow x + 1] \mathcal{X}_{3,4}$$

$$\mathcal{X}_{3,4} = C[x < y] \mathcal{X}_{2,4}$$

$$\mathcal{X}_{1,5} = \mathcal{X}_{1,4} \cup C[y \geq 100] \mathcal{X}_{1,5} \cup C[y \leftarrow y + [1, 3]] \mathcal{X}_{1,6}$$

$$\mathcal{X}_{2,5} = \mathcal{X}_{1,5} \cup C[x \geq y] \mathcal{X}_{2,5} \cup C[x \leftarrow x + 1] \mathcal{X}_{3,5} \cup \\ \mathcal{X}_{2,4} \cup C[y \geq 100] \mathcal{X}_{2,5} \cup C[y \leftarrow y + [1, 3]] \mathcal{X}_{2,6}$$

$$\mathcal{X}_{3,5} = C[x < y] \mathcal{X}_{2,5} \cup \mathcal{X}_{3,4} \cup C[y \geq 100] \mathcal{X}_{3,5} \cup C[y \leftarrow y + [1, 3]] \mathcal{X}_{3,6}$$

$$\mathcal{X}_{1,6} = C[y < 100] \mathcal{X}_{1,5}$$

$$\mathcal{X}_{2,6} = \mathcal{X}_{1,6} \cup C[x \geq y] \mathcal{X}_{2,6} \cup C[x \leftarrow x + 1] \mathcal{X}_{3,6} \cup C[y < 100] \mathcal{X}_{2,5}$$

$$\mathcal{X}_{3,6} = C[x < y] \mathcal{X}_{2,6} \cup C[y < 100] \mathcal{X}_{3,5}$$

Equational state semantics (example)

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

t_1	t_2
ℓ_1 while random do ℓ_2 if $x < y$ then ℓ_3 $x \leftarrow x + 1$	ℓ_4 while random do ℓ_5 if $y < 100$ then ℓ_6 $y \leftarrow y + [1,3]$

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:

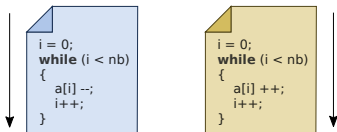
- keep information attached to **syntactic** program locations
(control points in \mathcal{L} , not control point tuples in $\mathbb{T} \rightarrow \mathcal{L}$)
- be able to **abstract away control information**
(precision/cost trade-off control)
- avoid **duplicating** thread instructions
- have a computation structure based on the **program syntax**
(denotational style)

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

(analyze each thread independently by induction on its syntax)

Simple interference semantics

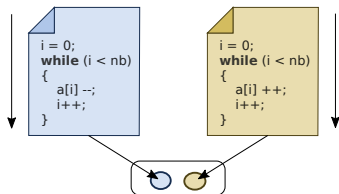
Thread-modular analysis with simple interferences



Principle:

- analyze each thread in **isolation**

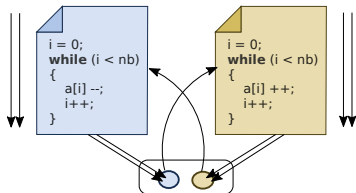
Thread-modular analysis with simple interferences



Principle:

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

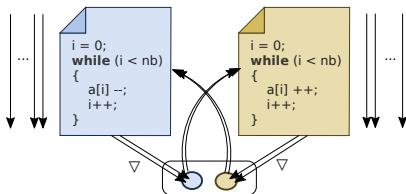
Thread-modular analysis with simple interferences



Principle:

- analyze each thread in **isolation**
- **gather** the **values** written into each variable by each thread
 ⇒ 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

 t_1

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

 t_2

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

Example

 t_1

```

 $\ell^1$  while random do
   $\ell^2$  if  $x < y$  then
     $\ell^3$   $x \leftarrow x + 1$ 
  
```

 t_2

```

 $\ell^4$  while random do
   $\ell^5$  if  $y < 100$  then
     $\ell^6$   $y \leftarrow y + [1, 3]$ 
  
```

Analysis of t_1 in isolation

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

Example

 t_1

```

 $\ell^1$  while random do
   $\ell^2$  if  $x < y$  then
     $\ell^3$   $x \leftarrow x + 1$ 
  
```

 t_2

```

 $\ell^4$  while random do
   $\ell^5$  if  $y < 100$  then
     $\ell^6$   $y \leftarrow y + [1, 3]$ 
  
```

Analysis of t_2 in isolation

(4): $x = y = 0$ $\mathcal{X}_4 = I$ (5): $x = 0, y \in [0, 102]$ $\mathcal{X}_5 = \mathcal{X}_4 \cup C[y \leftarrow y + [1, 3]] \mathcal{X}_6 \cup C[y \geq 100] \mathcal{X}_5$ (6): $x = 0, y \in [0, 99]$ $\mathcal{X}_6 = C[y < 100] \mathcal{X}_5$ output interferences: $y \leftarrow [1, 102]$

Example

 t_1

```

 $\ell^1$  while random do
   $\ell^2$  if  $x < y$  then
     $\ell^3$   $x \leftarrow x + 1$ 
  
```

 t_2

```

 $\ell^4$  while random do
   $\ell^5$  if  $y < 100$  then
     $\ell^6$   $y \leftarrow y + [1, 3]$ 
  
```

Re-analysis of t_1 with interferences from t_2

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

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

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

subsequent re-analyses are identical (fixpoint reached)

Example

 t_1

```

 $\ell^1$  while random do
   $\ell^2$  if  $x < y$  then
     $\ell^3$   $x \leftarrow x + 1$ 
  
```

 t_2

```

 $\ell^4$  while random do
   $\ell^5$  if  $y < 100$  then
     $\ell^6$   $y \leftarrow y + [1, 3]$ 
  
```

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!

Denotational semantics with interferences

Interferences in $\mathbb{I} \stackrel{\text{def}}{=} \mathbb{T} \times \mathbb{V} \times \mathbb{R}$

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

We define the analysis of a thread t
with respect to a set of interferences $I \subseteq \mathbb{I}$.

Expressions with interference: for thread t

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

- Apply interferences to read variables:

$$E_t[X] \langle \rho, I \rangle \stackrel{\text{def}}{=} \langle \{\rho(X)\} \cup \{v \mid \exists u \neq t: \langle u, X, v \rangle \in I\}, \emptyset \rangle$$

- Pass recursively I down to sub-expressions:

$$E_t[-e] \langle \rho, I \rangle \stackrel{\text{def}}{=} \text{let } \langle V, O \rangle = E_t[e] \langle \rho, I \rangle \text{ in } \langle \{-v \mid v \in V\}, O \rangle$$

...

Denotational semantics with interferences (cont.)

Statements with interference: for thread t

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

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

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

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

...

noting $\langle V_\rho, O_\rho \rangle \stackrel{\text{def}}{=} E_t[e] \langle \rho, I \rangle$
 \sqcup is now the element-wise \cup in $\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega) \times \mathcal{P}(\mathbb{I})$

Denotational semantics with interferences (cont.)

Program semantics: $P[\text{parprog}] \subseteq \Omega$

Given $\text{parprog} ::= \text{prog}_1 \parallel \dots \parallel \text{prog}_n$, we compute:

$$P[\text{parprog}] \stackrel{\text{def}}{=} \left[\text{Ifp } \lambda \langle O, I \rangle. \sqcup_{t \in \mathbb{T}} [C_t[\text{prog}_t] \langle \mathcal{E}_0, \emptyset, I \rangle]_{\Omega, \parallel} \right]_{\Omega}$$

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

Interference abstraction

Abstract interferences \mathbb{I}^\sharp

$\mathcal{P}(\mathbb{I}) \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{T} \times \mathbb{V} \times \mathbb{R})$ is abstracted as $\mathbb{I}^\sharp \stackrel{\text{def}}{=} (\mathbb{T} \times \mathbb{V}) \rightarrow \mathcal{R}^\sharp$
 where \mathcal{R}^\sharp abstracts $\mathcal{P}(\mathbb{R})$ (e.g. intervals)

Abstract semantics with interferences $C_t^\sharp \llbracket s \rrbracket$

derived from $C^\sharp \llbracket s \rrbracket$ in a generic way:

Example: $C_t^\sharp \llbracket X \leftarrow e \rrbracket \langle R^\sharp, \Omega, I^\sharp \rangle$

- for each Y in e , get its interference $Y_{\mathcal{R}}^\sharp = \bigsqcup_{\mathcal{R}}^\sharp \{ I^\sharp \langle u, Y \rangle \mid u \neq t \}$
- if $Y_{\mathcal{R}}^\sharp \neq \perp_{\mathcal{R}}^\sharp$, replace Y in e with $\text{get} \langle Y, R^\sharp \rangle \sqcup_{\mathcal{R}}^\sharp Y_{\mathcal{R}}^\sharp$
 (where $\text{get} \langle Y, R^\sharp \rangle$ extracts the abstract values in \mathcal{R}^\sharp of a variable Y from $R^\sharp \in \mathcal{E}^\sharp$)
- compute $\langle R^{\sharp'}, O' \rangle = C^\sharp \llbracket e \rrbracket \langle R^\sharp, O \rangle$
- enrich $I^\sharp \langle t, X \rangle$ with $\text{get} \langle X, R^{\sharp'} \rangle$

Static analysis with interferences

Abstract analysis

$$P^\# \llbracket \text{parprog} \rrbracket \stackrel{\text{def}}{=} \left[\text{lim } \lambda \langle O, I^\# \rangle. \langle O, I^\# \rangle \nabla \sqcup_{t \in \mathbb{T}} \left[C_t^\# \llbracket \text{prog}_t \rrbracket \langle \mathcal{E}_0^\#, \emptyset, I^\# \rangle \right]_{\Omega, \mathbb{I}^\#} \right]_{\Omega}$$

- **effective** analysis by **structural induction**
- termination ensured by a **widening**
- parameterized by a choice of abstract domains $\mathcal{R}^\#, \mathcal{E}^\#$
- **interferences** are **flow-insensitive** and **non-relational** in $\mathcal{R}^\#$
- **thread analysis** remains **flow-sensitive** and **relational** in $\mathcal{E}^\#$

(reminder: $[X]_{\Omega}$, $[Y]_{\Omega, \mathbb{I}^\#}$ keep only X 's component in Ω , Y 's components in Ω and $\mathbb{I}^\#$)

Path-based semantics

Control paths

$atomic ::= X \leftarrow exp \mid exp \bowtie 0$

Control paths

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

$\pi(X \leftarrow e) \stackrel{\text{def}}{=} \{X \leftarrow e\}$

$\pi(\text{if } e \bowtie 0 \text{ then } s \text{ fi}) \stackrel{\text{def}}{=} (\{e \bowtie 0\} \cdot \pi(s)) \cup \{e \not\bowtie 0\}$

$\pi(\text{while } e \bowtie 0 \text{ do } s \text{ done}) \stackrel{\text{def}}{=} \left(\bigcup_{i \geq 0} (\{e \bowtie 0\} \cdot \pi(s))^i \right) \cdot \{e \not\bowtie 0\}$

$\pi(s_1; s_2) \stackrel{\text{def}}{=} \pi(s_1) \cdot \pi(s_2)$

$\pi(prog)$ is a (generally infinite) set of finite control paths

Path-based concrete semantics of sequential programs

Join-over-all-path semantics

$$\sqcap \llbracket P \rrbracket : (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)) \rightarrow (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)) \quad P \subseteq \text{atomic}^*$$

$$\sqcap \llbracket P \rrbracket \langle R, O \rangle \stackrel{\text{def}}{=} \bigsqcup_{s_1 \dots s_n \in P} (C \llbracket s_n \rrbracket \circ \dots \circ C \llbracket s_1 \rrbracket) \langle R, O \rangle$$

Semantic equivalence

$$C \llbracket \text{prog} \rrbracket = \sqcap \llbracket \pi(\text{prog}) \rrbracket$$

(not true in the abstract)

Advantages:

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

Path-based concrete semantics of concurrent programs

Concurrent control paths

$$\begin{aligned}\pi_* &\stackrel{\text{def}}{=} \{ \text{interleavings of } \pi(\text{prog}_t), t \in \mathbb{T} \} \\ &= \{ p \in \text{atomic}^* \mid \forall t \in \mathbb{T}, \text{proj}_t(p) \in \pi(\text{prog}_t) \}\end{aligned}$$

Interleaving program semantics

$$P_* \llbracket \text{parprog} \rrbracket \stackrel{\text{def}}{=} [\sqcap \llbracket \pi_* \rrbracket \langle \mathcal{E}_0, \emptyset \rangle]_{\Omega}$$

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

(\simeq sequentially consistent executions [Lamport 79])

Issues:

- too many paths to consider exhaustively
- no induction structure to iterate on
 \implies **abstract** as a **denotational** semantics
- unrealistic assumptions on granularity and memory consistency

Soundness of the interference semantics

Soundness theorem

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

Proof sketch:

- define $\sqcap_t[P]X \stackrel{\text{def}}{=} \bigsqcup \{C_t[s_1; \dots; s_n]X \mid s_1 \dots s_n \in P\}$,
then $\sqcap_t[\pi(s)] = C_t[s]$;
- given the interference fixpoint $I \subseteq \mathbb{I}$ from $P[\text{parprog}]$,
prove by recurrence on the length of $p \in \pi_*$ that:
 - $\forall t \in \mathbb{T}, \forall \rho \in [\sqcap_t[p]\langle \mathcal{E}_0, \emptyset \rangle]_{\mathcal{E}}$,
 $\exists \rho' \in [\sqcap_t[\text{proj}_t(p)]\langle \mathcal{E}_0, \emptyset, I \rangle]_{\mathcal{E}}$ such that
 $\forall X \in \mathbb{V}, \rho(X) = \rho'(X)$ or $\langle u, X, \rho(X) \rangle \in I$ for some $u \neq t$.
 - $[\sqcap_t[p]\langle \mathcal{E}_0, \emptyset \rangle]_{\Omega} \subseteq \bigcup_{t \in \mathbb{T}} [\sqcap_t[\text{proj}_t(p)]\langle \mathcal{E}_0, \emptyset, I \rangle]_{\Omega}$

Note: sound but not complete

Weakly consistent memories

Issues with weak consistency

program written

```
 $F_1 \leftarrow 1;$       |  $F_2 \leftarrow 1;$   
if  $F_2 = 0$  then    | if  $F_1 = 0$  then  
   $S_1$               |    $S_2$   
fi                  | fi
```

(simplified Dekker mutual exclusion algorithm)

S_1 and S_2 **cannot** execute simultaneously.

Issues with weak consistency

program written

$F_1 \leftarrow 1;$	$F_2 \leftarrow 1;$
if $F_2 = 0$ then	if $F_1 = 0$ then
S_1	S_2
fi	fi



program executed

if $F_2 = 0$ then	if $F_1 = 0$ then
$F_1 \leftarrow 1;$	$F_2 \leftarrow 1;$
S_1	S_2
fi	fi

(simplified Dekker mutual exclusion algorithm)

S_1 and S_2 can execute simultaneously.

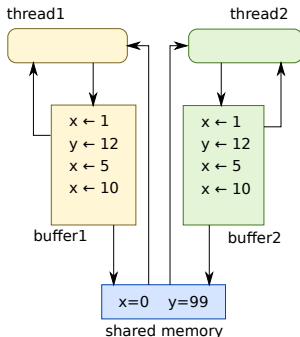
Not a sequentially consistent behavior!

Caused by:

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

behavior accepted by Java [Mans05]

Hardware memory model example: TSO



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

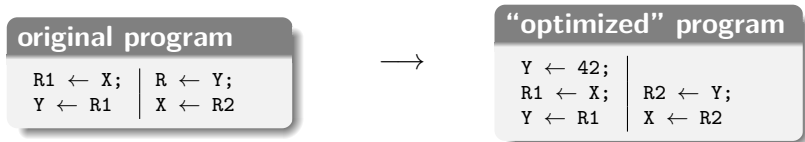
original program

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

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

We should not have $R_1 = 42$.

Out of thin air principle



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

We should not have $R_1 = 42$.

Possible if we allow speculative writes!

⇒ we **disallow** this kind of program transformations.

(also forbidden in Java)

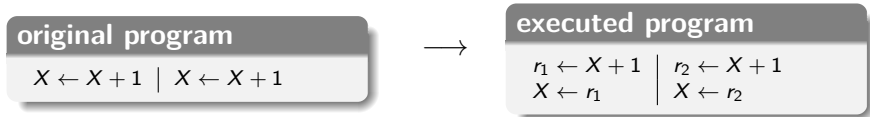
Atomicity and granularity

original program

$X \leftarrow X + 1 \mid X \leftarrow X + 1$

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: $p \rightsquigarrow q$

only reduce interferences and errors

- **Reordering:** $X_1 \leftarrow e_1 \cdot X_2 \leftarrow e_2 \rightsquigarrow X_2 \leftarrow e_2 \cdot X_1 \leftarrow e_1$
(if $X_1 \notin \text{var}(e_2)$, $X_2 \notin \text{var}(e_1)$, and e_1 does not stop the program)
- **Propagation:** $X \leftarrow e \cdot s \rightsquigarrow X \leftarrow e \cdot s[e/X]$
(if $X \notin \text{var}(e)$, $\text{var}(e)$ are thread-local, and e is deterministic)
- **Factorization:** $s_1 \cdot \dots \cdot s_n \rightsquigarrow X \leftarrow e \cdot s_1[X/e] \cdot \dots \cdot s_n[X/e]$
(if X is fresh, $\forall i, \text{var}(e) \cap \text{lval}(s_i) = \emptyset$, and e has no error)
- **Decomposition:** $X \leftarrow e_1 + e_2 \rightsquigarrow T \leftarrow e_1 \cdot X \leftarrow T + e_2$
(change of granularity)
- ...

but **NOT:**

- "out-of-thin-air" writes: $X \leftarrow e \rightsquigarrow X \leftarrow 42 \cdot X \leftarrow e$

Soundness of the interference semantics

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

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

Soundness theorem

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

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

Locks

Scheduling

Synchronization primitives

```
prog ::= lock(m)
      | unlock(m)
```

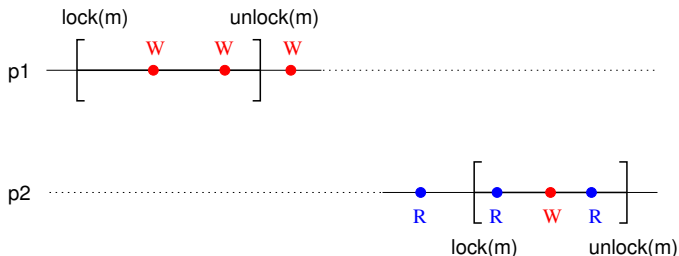
$m \in \mathbb{M}$: finite set of non-recursive mutexes

Scheduling

mutexes ensure **mutual exclusion**

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

Mutual exclusion



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

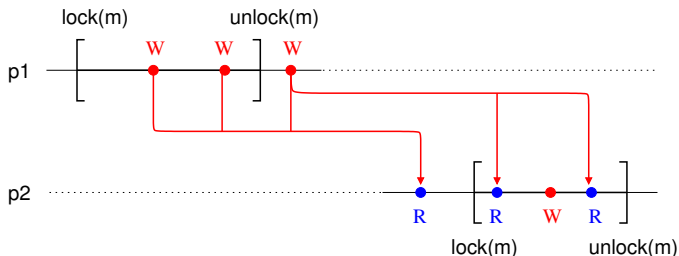
- $\mathcal{E} \rightsquigarrow \mathcal{E} \times \mathbb{C}$, $\mathcal{E}^\# \rightsquigarrow \mathbb{C} \rightarrow \mathcal{E}^\#$
- $\mathbb{I} \stackrel{\text{def}}{=} \mathbb{T} \times \mathbb{V} \times \mathbb{R} \rightsquigarrow \mathbb{I} \stackrel{\text{def}}{=} \mathbb{T} \times \mathbb{C}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

$\mathbb{C} \stackrel{\text{def}}{=} \mathbb{C}_{\text{race}} \cup \mathbb{C}_{\text{sync}}$ separates

data-race writes \mathbb{C}_{race}

and well-synchronized writes \mathbb{C}_{sync}

Mutual exclusion



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

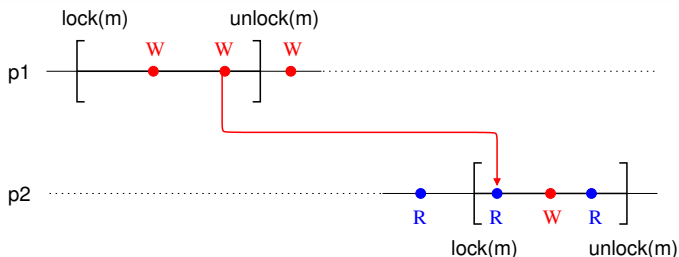
Across read / write not protected by a mutex.

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

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

Mutual exclusion



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

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

$$\implies \mathbb{C} \simeq \mathcal{P}(\mathbb{M}) \times (\{\text{data} - \text{race}\} \cup \mathbb{M})$$

Example analysis

abstract consumer/producer

N consumers	N producers
<pre> while random do lock(m); ℓ^1 if $X > 0$ then $\ell^2 X \leftarrow X - 1$ fi; unlock(m); $\ell^3 Y \leftarrow X$ done </pre>	<pre> while random do lock(m); $X \leftarrow X + 1$; if $X > 100$ then $X \leftarrow 100$ fi; unlock(m) done </pre>

Assuming we have several (N) producers and consumers:

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

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

Locks and priorities

priority-based critical sections

high thread	low thread
$L \leftarrow \text{isLocked}(m);$ if $L = 0$ then $Y \leftarrow Y+1;$ yield()	$\text{lock}(m);$ $Z \leftarrow Y;$ $Y \leftarrow 0;$ $\text{unlock}(m)$

Real-time scheduling

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

Locks and priorities

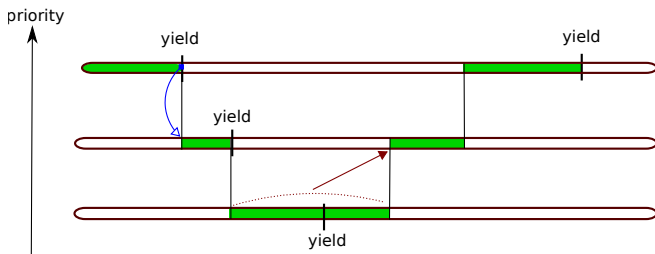
priority-based critical sections

high thread	low thread
$L \leftarrow \text{isLocked}(m);$ if $L = 0$ then $Y \leftarrow Y+1;$ yield()	$\text{lock}(m);$ $Z \leftarrow Y;$ $Y \leftarrow 0;$ $\text{unlock}(m)$

Partition interferences and **environments** wrt. scheduling state

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

Priority-based scheduling



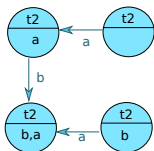
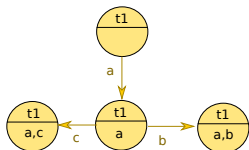
Analysis: refined transfer of interference based on priority

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

⇒ again, handled similarly to locks, with a richer \mathbb{C}

Deadlock checking

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

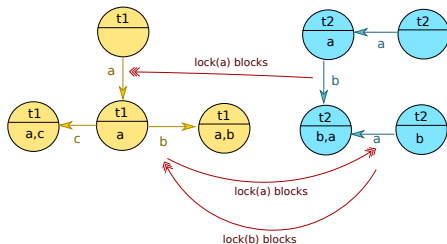


During the analysis, gather:

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

Deadlock checking

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



During the analysis, gather:

- all reachable **mutex configurations**: $R \subseteq \mathbb{T} \times \mathcal{P}(\mathbb{M})$
- **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.

Rely-guarantee proof method

Reminder: Floyd–Hoare logic

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

Hoare triples: $\{P\} \text{prog} \{Q\}$

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

$$\frac{}{\{P[e/X]\} X \leftarrow e \{P\}} \quad \frac{\{P \wedge e \bowtie 0\} s \{Q\} \quad P \wedge e \not\bowtie 0 \Rightarrow Q}{\{P\} \text{if } e \bowtie 0 \text{ then } s \text{ fi } \{Q\}}$$

$$\frac{\{P\} s_1 \{Q\} \quad \{Q\} s_2 \{R\}}{\{P\} s_1; s_2 \{R\}} \quad \frac{\{P \wedge e \bowtie 0\} s \{P\}}{\{P\} \text{while } e \bowtie 0 \text{ do } s \text{ done } \{P \wedge e \not\bowtie 0\}}$$

$$\frac{\{P'\} s \{Q'\} \quad P \Rightarrow P' \quad Q' \Rightarrow Q}{\{P\} s \{Q\}}$$

Link with abstract interpretation:

- the equations reachability semantics $(\mathcal{X}_\ell)_{\ell \in \mathcal{L}}$ provides the **most precise Hoare triples in fixpoint constructive form**

Owicki–Gries proof method

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

Principle: add a new rule, for \parallel

$$\frac{\{P_1\} s_1 \{Q_1\} \quad \{P_2\} s_2 \{Q_2\}}{\{P_1 \wedge P_2\} s_1 \parallel s_2 \{Q_1 \wedge Q_2\}}$$

Owicki–Gries proof method

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

Principle: add a new rule, for \parallel

$$\frac{\{P_1\} s_1 \{Q_1\} \quad \{P_2\} s_2 \{Q_2\}}{\{P_1 \wedge P_2\} s_1 \parallel s_2 \{Q_1 \wedge Q_2\}}$$

This rule is **not always sound!**

e.g., we have $\{X = 0, Y = 0\} X \leftarrow 1 \{X = 1, Y = 0\}$
 and $\{X = 0, Y = 0\} \text{if } X = 0 \text{ then } Y \leftarrow 1 \text{ fi } \{X = 0, Y = 1\}$
 but not $\{X = 0, Y = 0\} X \leftarrow 1 \parallel \text{if } X = 0 \text{ then } Y \leftarrow 1 \text{ fi } \{false\}$

\implies we need a side-condition to the rule:

$\{P_1\} s_1 \{Q_1\}$ and $\{P_2\} s_2 \{Q_2\}$ **must not interfere**

Owicki–Gries proof method (cont.)

interference freedom

given proofs Δ_1 and Δ_2 of $\{P_1\} s_1 \{Q_1\}$ and $\{P_2\} s_2 \{Q_2\}$

Δ_1 does not interfere with Δ_2 if:

for any Φ appearing before a statement in Δ_1

for any $\{P'_2\} s'_2 \{Q'_2\}$ appearing in Δ_2

$\{\Phi \wedge P'_2\} s'_2 \{\Phi\}$ holds

and moreover $\{Q_1 \wedge P'_2\} s'_2 \{Q_1\}$

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

e.g., $\{X = 0, Y \in [0, 1]\} X \leftarrow 1 \{X = 1, Y \in [0, 1]\}$
 $\{X \in [0, 1], Y = 0\}$ if $X = 0$ then $Y \leftarrow 1$ fi $\{X \in [0, 1], Y \in [0, 1]\}$

$\implies \{X = 0, Y = 0\} X \leftarrow 1 \parallel$ if $X = 0$ then $Y \leftarrow 1$ fi $\{X = 1, Y \in [0, 1]\}$

Summary:

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

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

\implies not satisfactory

Jones' rely-guarantee proof method

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

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

- if P is true before prog is executed
- **and the effect of other threads is included in R** (rely)
- then Q is true after prog
- **and the effect of prog 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 $\mathcal{P}(\Sigma \times \mathcal{A} \times \Sigma)$)

The parallel composition rule becomes:

$$\frac{R \vee G_2, G_1 \vdash \{P_1\} s_1 \{Q_1\} \quad R \vee G_1, G_2 \vdash \{P_2\} s_2 \{Q_2\}}{R, G_1 \vee G_2 \vdash \{P_1 \wedge P_2\} s_1 \parallel s_2 \{Q_1 \wedge Q_2\}}$$

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

```

at ℓ_1 : $x = y = 0$ at ℓ_2 : $x, y \in [0, 102]$, $x \leq y$ at ℓ_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

```

at ℓ_4 : $x = y = 0$ at ℓ_5 : $x, y \in [0, 102]$, $x \leq y$ at ℓ_6 : $x \in [0, 99]$, $y \in [0, 99]$, $x \leq y$

Rely-guarantee example

checking t_1

ℓ^1 while random do ℓ^2 if $x < y$ then ℓ^3 $x \leftarrow x+1$ fi done	x unchanged y incremented $0 \leq y \leq 102$
---	---

ℓ^1 : $x = y = 0$

ℓ^2 : $x, y \in [0, 102], x \leq y$

ℓ^3 : $x \in [0, 101], y \in [1, 102], x < y$

checking t_2

y unchanged $0 \leq x \leq y$	ℓ^4 while random do ℓ^5 if $y < 100$ then ℓ^6 $y \leftarrow y + [1,3]$ fi done
------------------------------------	---

at ℓ^4 : $x = y = 0$

at ℓ^5 : $x, y \in [0, 102], x \leq y$

at ℓ^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:

- invariants are still local to threads
- checking a thread does not require looking at other threads, only at an **abstraction of their semantics**

Auxiliary variables

Example

t_1	t_2
$\ell_1 \ x \leftarrow x + 1 \ \ell_2$	$\ell_3 \ x \leftarrow x + 1 \ \ell_4$

Goal: prove $\{x = 0\} t_1 \parallel t_2 \{x = 2\}$.

Auxiliary variables

Example

t_1	t_2
$\ell_1 \ x \leftarrow x + 1 \ \ell_2$	$\ell_3 \ x \leftarrow x + 1 \ \ell_4$

Goal: prove $\{x = 0\} t_1 \parallel t_2 \{x = 2\}$.

we must rely on and guarantee that
 each thread increments x exactly once!

Solution: auxiliary variables

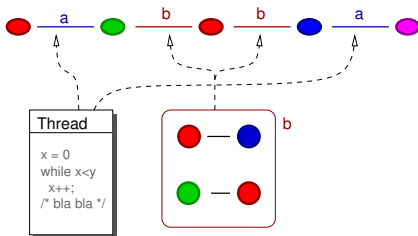
do not change the semantics but store extra information:

- past values of variables (history of the computation)
- program counter of other threads (pc_t)

Example: for t_1 : $\{(pc_2 = \ell_3 \wedge x = 0) \vee (pc_2 = \ell_4 \wedge x = 1)\}$
 $x \leftarrow x + 1$
 $\{(pc_2 = \ell_3 \wedge x = 1) \vee (pc_2 = \ell_4 \wedge x = 2)\}$

Rely-guarantee as abstract interpretation

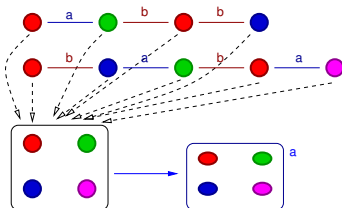
Modularity: main idea



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*

Trace decomposition



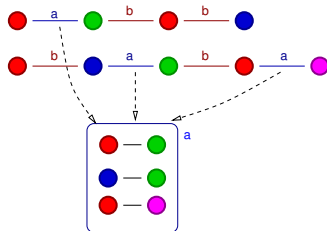
Reachable states projected on thread t : $\mathcal{RI}(t)$

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

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

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

Trace decomposition



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

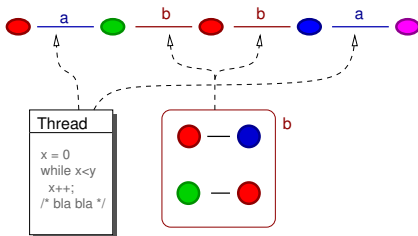
Extract the transitions with action t **observed** in \mathcal{T}_p

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

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

$$\text{where } \alpha^{\parallel}(X)(t) \stackrel{\text{def}}{=} \{ \langle \sigma_i, \sigma_{i+1} \rangle \mid \exists \sigma_0 \xrightarrow{a_1} \sigma_1 \cdots \xrightarrow{a_n} \sigma_n \in X : a_{i+1} = t \}$$

Thread-modular concrete semantics



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

States: $\mathcal{R}I$

Interleave:

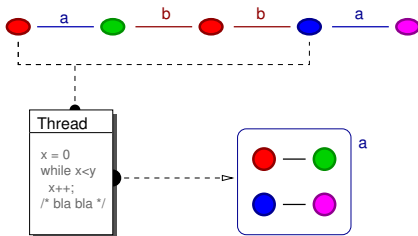
- transitions from the current thread t
- transitions from interferences A by other threads

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

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

\Rightarrow similar to reachability for a sequential program, up to A

Thread-modular concrete semantics



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

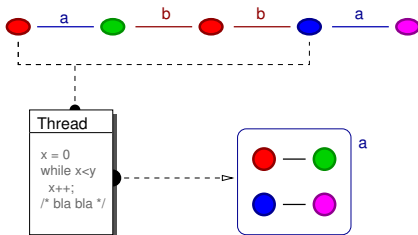
Interferences: A

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

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

$$B(\mathcal{Z})(t) \stackrel{\text{def}}{=} \{ \langle \sigma, \sigma' \rangle \mid \pi_t(\sigma) \in \mathcal{Z}(t) \wedge \sigma \xrightarrow{t} \sigma' \}$$

Thread-modular concrete semantics



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

Recursive definition:

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

\Rightarrow express the most precise solution as nested fixpoints:

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

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

Fixpoint form

Constructive fixpoint form:

Use Kleene's iteration to construct fixpoints:

- $\mathcal{R}I = \text{lfp } H = \bigsqcup_{n \in \mathbb{N}} H^n(\lambda t. \emptyset)$

in the pointwise powerset lattice $\prod_{t \in \mathbb{T}} \{t\} \rightarrow \mathcal{P}(\Sigma_t)$

- $H(Z)(t) = \text{lfp } R_t(B(Z)) = \bigcup_{n \in \mathbb{N}} (R_t(B(Z)))^n(\emptyset)$

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 $\mathcal{R}I_0^\# \stackrel{\text{def}}{=} A_0^\# \stackrel{\text{def}}{=} \lambda t. \perp^\#$
- while $A_n^\#$ is not stable
 - compute $\forall t \in \mathbb{T}: \mathcal{R}I_{n+1}^\#(t) \stackrel{\text{def}}{=} \text{lfp } R_t^\#(A_n^\#)$
by iteration with widening ∇
(\simeq separate analysis of each thread)
 - compute $A_{n+1}^\# \stackrel{\text{def}}{=} A_n^\# \nabla B^\#(\mathcal{R}I_{n+1}^\#)$
- when $A_n^\# = A_{n+1}^\#$, return $\mathcal{R}I_n^\#$

\implies thread-modular analysis
parameterized by abstract domains
able to easily reuse existing sequential analyses

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

$$\alpha_{\mathcal{R}}^{nf}(X) \stackrel{\text{def}}{=} \{ (\ell, \rho|_{\mathcal{V}}) \mid (\ell, \rho) \in X \} \cup (X \cap \Omega)$$

Interference abstraction: remove **all** control state

$$\alpha_A^{nf}(Y) \stackrel{\text{def}}{=} \{ (\rho, \rho') \mid \exists L, L' \in \mathbb{T} \rightarrow \mathcal{L}: ((L, \rho), (L', \rho')) \in Y \}$$

Flow-insensitive abstraction (cont.)

Flow-insensitive fixpoint semantics: (omitting errors Ω)

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

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

- $B^{nf}(Z)(t) \stackrel{\text{def}}{=} \{(\rho, \rho') \mid \exists \ell, \ell' \in \mathcal{L}: (\ell, \rho) \in Z(t) \wedge (\ell, \rho) \rightarrow_t (\ell', \rho')\}$
(extract interferences from reachable states)
- $R_t^{nf}(Y)(X) \stackrel{\text{def}}{=} R_t^{loc}(X) \cup A_t^{nf}(Y)(X)$ (interleave steps)
- $R_t^{loc}(X) \stackrel{\text{def}}{=} \{(\ell'_i, \lambda V. 0)\} \cup \{(\ell', \rho') \mid \exists (\ell, \rho) \in X: (\ell, \rho) \rightarrow_t (\ell', \rho')\}$ (thread step)
- $A_t^{nf}(Y)(X) \stackrel{\text{def}}{=} \{(\ell, \rho') \mid \exists \rho, u \neq t: (\ell, \rho) \in X \wedge (\rho, \rho') \in Y(u)\}$ (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:

$$\alpha_A^{nr}(Y) \stackrel{\text{def}}{=} \lambda V. \{x \in \mathbb{V} \mid \exists (\rho, \rho') \in Y: \rho(V) \neq x \wedge \rho'(V) = x\}$$

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

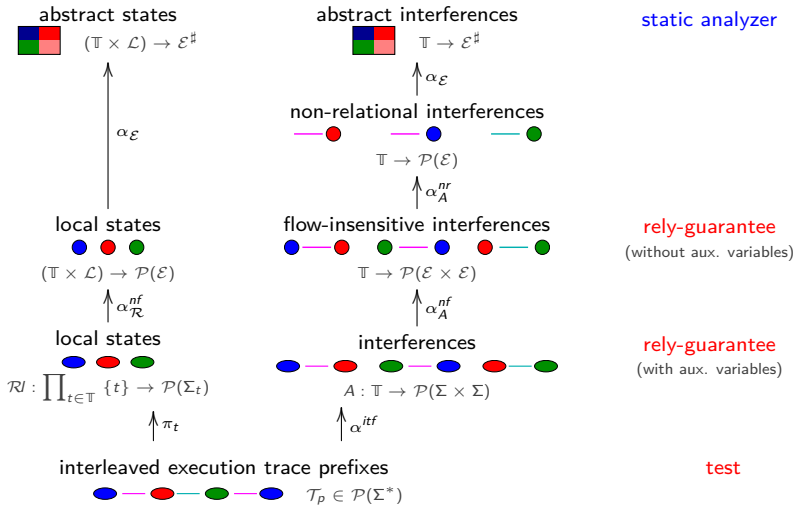
$$A_t^{nr}(Y)(X) \stackrel{\text{def}}{=} \{(\ell, \rho[V \mapsto v]) \mid (\ell, \rho) \in X, V \in \mathbb{V}, \exists u \neq t: v \in Y(u)(V)\}$$

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

\implies 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 $\mathbb{V} \rightarrow \mathcal{D}^\#$)

From traces to thread-modular analyses



Weakly relational interferences

Clock thread

```
while Clock < 106 do
  Clock ← Clock + 1;
  ...
done
```

Accumulator thread

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

- clock is a global, increasing clock
- x accumulates periods of time
- **no overflow** on $\text{Clock} - \text{Prec}$, nor $x \leftarrow x + \text{delta}$

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

Monotonicity abstraction

Abstraction:

map variables to \uparrow monotonic or \top don't know

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

- 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 $V \leftarrow V + e$, with $e \geq 0$

- **use:** combined with non-relational interferences

$$\text{if } \forall t: A^{\text{mono}}(t)(V) = \uparrow$$

then any test with non-relational interference $C \llbracket X \leq (V \mid [a, b]) \rrbracket$
can be strengthened into $C \llbracket X \leq V \rrbracket$

Relational invariant interferences

Abstraction: keep relations maintained by interferences

- remove control state in interferences (α_A^{nf})
- keep mutex state M (set of mutexes held)
- forget input-output relationships
- keep relationships between variables

$$\alpha_A^{inv}(Y) \stackrel{\text{def}}{=} \{ \langle M, \rho \rangle \mid \exists \rho' : \langle \langle M, \rho \rangle, \langle M, \rho' \rangle \rangle \in Y \vee \langle \langle M, \rho' \rangle, \langle M, \rho \rangle \rangle \in Y \}$$

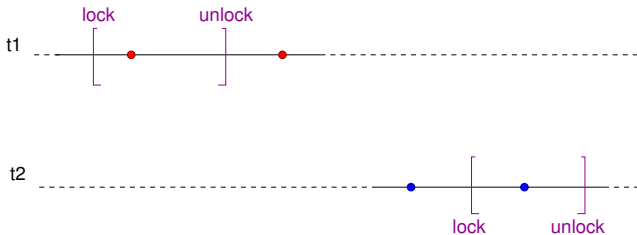
$\langle M, \rho \rangle \in \alpha_A^{inv}(Y) \implies \langle M, \rho \rangle \in \alpha_A^{inv}(Y)$ after any sequence of interferences from Y

Lock invariant:

$$\{ \rho \mid \exists t \in \mathcal{T}, M : \langle M, \rho \rangle \in \alpha_A^{inv}(\llbracket t \rrbracket), m \notin M \}$$

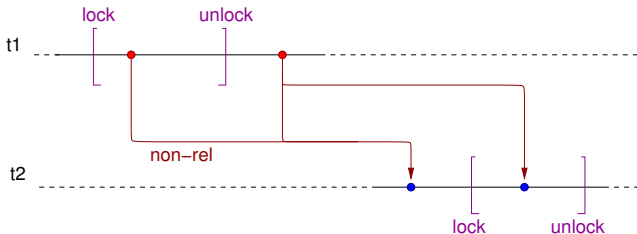
- property maintained outside code protected by m
- possibly broken while m is locked
- restored before unlocking m

Relational lock invariants



Improved interferences: mixing simple interferences and lock invariants

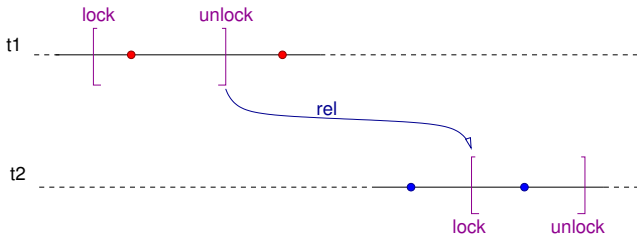
Relational lock invariants



Improved interferences: mixing simple interferences and lock invariants

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

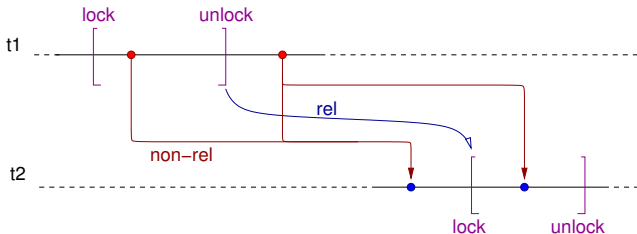
Relational lock invariants



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

Weakly relational interference example

analyzing t_1

t_1	t_2
<pre>while random do lock(m); if x < y then x ← x + 1; unlock(m)</pre>	<pre>x unchanged y incremented 0 ≤ y ≤ 102</pre>

analyzing t_2

t_1	t_2
<pre>y unchanged 0 ≤ x, x ≤ y</pre>	<pre>while random do lock(m); if y < 100 then y ← y + [1,3]; unlock(m)</pre>

Using all three interference abstractions:


- non-relational interferences ($0 \leq y \leq 102, 0 \leq x$)
- lock invariants, with the octagon domain ($x \leq y$)
- monotonic interferences (y monotonic)

we can prove automatically that $x \leq y$ holds

Application: The AstréeA analyzer

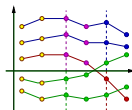
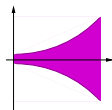
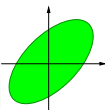
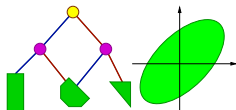
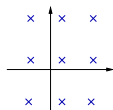
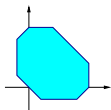
The Astrée analyzer

Astrée:

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

Design by refinement:

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



The AstréeA analyzer

From Astrée to AstréeA:

- follow-up project: **Astrée for concurrent embedded C code** (2012–2016)
- interferences abstracted using stock non-relation domains
- memory domain instrumented to gather / inject interferences
- added an extra iterator \implies minimal code modifications
- additionally: 4 KB ARINC 653 OS model

Target application:

- ARINC 653 embedded avionic application
- **15 threads, 1.6 Mlines**
- embedded reactive code + network code + string formatting
- many variables, arrays, loops
- shallow call graph, no dynamic allocation



From simple interferences to relational interferences

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

Summary

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**
- **sound** for **weakly consistent memory** semantics
(when using non-relational, flow-insensitive interference abstraction)
- aware of **scheduling**, **priorities** and **synchronization**
- **parameterized** by (possibly relational) **abstract domains**
(independent domains for state abstraction and interference abstraction)

Bibliography

Bibliography

[Bour93] **F. Bourdoncle**. *Efficient chaotic iteration strategies with widenings*. In Proc. FMPA'93, LNCS vol. 735, pp. 128–141, Springer, 1993.

[Carr09] **J.-L. Carré & C. Hymans**. *From single-thread to multithreaded: An efficient static analysis algorithm*. In arXiv:0910.5833v1, EADS, 2009.

[Cous84] **P. Cousot & R. Cousot**. *Invariance proof methods and analysis techniques for parallel programs*. In Automatic Program Construction Techniques, chap. 12, pp. 243–271, Macmillan, 1984.

[Cous85] **R. Cousot**. *Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles*. In Thèse d'Etat es sc. math., INP Lorraine, Nancy, 1985.

[Hoar69] **C. A. R. Hoare**. *An axiomatic basis for computer programming*. In Com. ACM, 12(10):576–580, 1969.

Bibliography (cont.)

[Jone81] **C. B. Jones**. *Development methods for computer programs including a notion of interference*. In PhD thesis, Oxford University, 1981.

[Lamp77] **L. Lamport**. *Proving the correctness of multiprocess programs*. In IEEE Trans. on Software Engineering, 3(2):125–143, 1977.

[Lamp78] **L. Lamport**. *Time, clocks, and the ordering of events in a distributed system*. In Comm. ACM, 21(7):558–565, 1978.

[Mans05] **J. Manson, B. Pugh & S. V. Adve**. *The Java memory model*. In Proc. POPL'05, pp. 378–391, ACM, 2005.

[Miné12] **A. Miné**. *Static analysis of run-time errors in embedded real-time parallel C programs*. In LMCS 8(1:26), 63 p., arXiv, 2012.

[Owic76] **S. Owicki & D. Gries**. *An axiomatic proof technique for parallel programs I*. In Acta Informatica, 6(4):319–340, 1976.

Bibliography (cont.)

[Reyn04] **J. C. Reynolds**. *Toward a grainless semantics for shared-variable concurrency*. In Proc. FSTTCS'04, LNCS vol. 3328, pp. 35–48, Springer, 2004.

[Sara07] **V. A. Saraswat, R. Jagadeesan, M. M. Michael & C. von Praun**. *A theory of memory models*. In Proc. PPOPP'07, pp. 161–172, ACM, 2007.