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)

Scope

In this course: static thread model

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

Goal: static analysis

- infer numeric program invariants
- parameterized by a choice of numeric abstract domains
- discover run-time errors (e.g., divisions by 0)
- discover data-races (unprotected accesses by concurrent threads)
- discover deadlocks (some threads block each other indefinitely)
- application to analyzing embedded C programs

Outline

- Simple concurrent language
- Non-modular concurrent semantics
- Simple interference thread-modular concurrent semantics
- 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: $\texttt{stmt}_1$ to $\texttt{stmt}_n$
- 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

$$\texttt{prog} ::= {}^\ell\texttt{stmt}_1{}^\ell \,||\, \ldots \,||\, {}^\ell\texttt{stmt}_n{}^\ell \quad \textit{(parallel composition)}$$

$$\begin{array}{lcll} {}^\ell\texttt{stmt}^\ell & ::= & {}^\ell V \leftarrow \texttt{exp}^\ell & \textit{(assignment)} \\ & | & {}^\ell\texttt{if exp} \bowtie 0 \texttt{ then } {}^\ell\texttt{stmt}^\ell \texttt{ fi}^\ell & \textit{(conditional)} \\ & | & {}^\ell\texttt{while } {}^\ell\texttt{exp} \bowtie 0 \texttt{ do } {}^\ell\texttt{stmt}^\ell \texttt{ done}^\ell & \textit{(loop)} \\ & | & {}^\ell\texttt{stmt}; {}^\ell\texttt{stmt}^\ell & \textit{(sequence)} \end{array}$$

$$\texttt{exp} ::= V \mid [c_1, c_2] \mid -\texttt{exp} \mid \texttt{exp} \diamond \texttt{exp}$$

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

Multi-thread execution model

t_1	t_2
ℓ1 `while random do` ℓ2 `if x < y then` ℓ3 `x ← x + 1`	ℓ4 `while random do` ℓ5 `if y < 100 then` ℓ6 `y ← 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 $0 \leq x \leq y \leq 102$

Semantic model: labelled transition systems

simple extension of 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 $\sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \cdots \sigma_n \xrightarrow{a_n} \sigma_{n+1}$

τ is omitted on $\rightarrow$ for traces for simplicity

From concurrent programs to labelled transition systems

- given: $\texttt{prog} ::= {}^{\ell^i_1}\texttt{stmt}_1{}^{\ell^x_1} \,||\, \cdots \,||\, {}^{\ell^i_n}\texttt{stmt}_n{}^{\ell^x_n}$
- threads are numbered: $\mathbb{T} \stackrel{\text{def}}{=} \{1, \ldots, n\}$

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

- a control state $L(t) \in \mathcal{L}$ for each thread $t \in \mathbb{T}$ and
- a single shared memory state $\rho \in \mathcal{E} \stackrel{\text{def}}{=} \mathbb{V} \to \mathbb{Z}$

Initial states:

threads start at their first control point ℓ^i_t, variables are set to 0:
$\mathcal{I} \stackrel{\text{def}}{=} \{ \langle \lambda t.\ell^i_t, \lambda V.0 \rangle \}$

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

From concurrent programs to labelled transition systems

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

$$\langle\, L,\, \rho \,\rangle \xrightarrow{t}_{\tau} \langle\, L',\, \rho' \,\rangle \stackrel{\text{def}}{\iff} \begin{array}{l} \langle\, L(t),\, \rho \,\rangle \rightarrow_{\tau[\mathtt{stmt}_t]} \langle\, L'(t),\, \rho' \,\rangle \wedge \\ \forall u \neq t\colon L(u) = L'(u) \end{array}$$

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

  see course 2 for the full definition of $\tau[\mathtt{stmt}]$

- each transition $\sigma \rightarrow_{\tau[\mathtt{stmt}_t]} \sigma'$ leads to many transitions $\rightarrow_{\tau}$!

Interleaved trace semantics

Maximal and finite prefix trace semantics are as before:

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

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

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

Finite prefix traces: $\mathcal{T}_p$

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

$\mathcal{T}_p = \text{lfp}\, F_p$ where $F_p(X) = \mathcal{I} \cup \{\, \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_n} \sigma_{n+1} \mid n \geq 0 \wedge \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in X \wedge \sigma_n \xrightarrow{t_n}_\tau \sigma_{n+1} \,\}$

Fairness

Fairness conditions: avoid threads being denied to run forever

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

an infinite trace $\sigma_0 \xrightarrow{t_0} \cdots \sigma_n \xrightarrow{t_n} \cdots$ 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 $\mathcal{M}_\infty$ 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 $\alpha_{*\preceq}(\mathcal{M}_\infty \cap F) \subseteq \alpha_{*\preceq}(X)$

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

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

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

In the rest of the course, we ignore fairness conditions

Reachability semantics for concurrent programs

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

Reachable states in any execution:

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

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

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

Abstraction of the finite trace semantics.

$\mathcal{R} = \alpha_p(\mathcal{T}_p)$ where $\alpha_p(X) \stackrel{\text{def}}{=} \{\, \sigma \mid \exists n \geq 0, \sigma_0 \xrightarrow{t_0} \cdots \sigma_n \in X: \sigma = \sigma_n \,\}$

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: $\mathcal{P}(\mathcal{L} \times \mathcal{E}) \simeq \mathcal{L} \to \mathcal{P}(\mathcal{E})$

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

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

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

Example:

ℓ^1 $i \leftarrow 2$;
ℓ^2 $n \leftarrow [-\infty, +\infty]$;
ℓ^3 while ℓ^4 $i < n$ do
  ℓ^5 if $[0,1] = 0$ then
    ℓ^6 $i \leftarrow i + 1$
  fi
ℓ^7 done
ℓ^8

$$
\begin{aligned}
\mathcal{X}_1 &= \mathcal{I} \\
\mathcal{X}_2 &= \mathsf{C}[\![\, i \leftarrow 2 \,]\!]\, \mathcal{X}_1 \\
\mathcal{X}_3 &= \mathsf{C}[\![\, n \leftarrow [-\infty, +\infty] \,]\!]\, \mathcal{X}_2 \\
\mathcal{X}_4 &= \mathcal{X}_3 \cup \mathcal{X}_7 \\
\mathcal{X}_5 &= \mathsf{C}[\![\, i < n \,]\!]\, \mathcal{X}_4 \\
\mathcal{X}_6 &= \mathcal{X}_5 \\
\mathcal{X}_7 &= \mathcal{X}_5 \cup \mathsf{C}[\![\, i \leftarrow i + 1 \,]\!]\, \mathcal{X}_6 \\
\mathcal{X}_8 &= \mathsf{C}[\![\, i \geq n \,]\!]\, \mathcal{X}_4
\end{aligned}
$$

Denotational state semantics

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

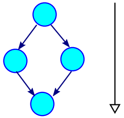
$C[\![\,\texttt{stmt}\,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

$$
\begin{array}{lcl}
C[\![\, X \leftarrow e \,]\!]\, R & \stackrel{\text{def}}{=} & \{\, \rho[X \mapsto v] \mid \rho \in R, v \in E[\![\, e \,]\!]\, \rho \,\} \\
C[\![\, e \bowtie 0 \,]\!]\, R & \stackrel{\text{def}}{=} & \{\, \rho \in R \mid \exists v \in E[\![\, e \,]\!]\, \rho \colon v \bowtie 0 \,\} \\
C[\![\, \texttt{if } e \bowtie 0 \texttt{ then } s \texttt{ fi} \,]\!]\, R & \stackrel{\text{def}}{=} & (C[\![\, s \,]\!] \circ C[\![\, e \bowtie 0 \,]\!])R \sqcup C[\![\, e \not\bowtie 0 \,]\!]\, R \\
C[\![\, s_1; s_2 \,]\!] & \stackrel{\text{def}}{=} & C[\![\, s_2 \,]\!] \circ C[\![\, s_1 \,]\!] \\
C[\![\, \texttt{while } e \bowtie 0 \texttt{ do } s \texttt{ done} \,]\!]\, R & \stackrel{\text{def}}{=} & C[\![\, e \not\bowtie 0 \,]\!]\, (\mathrm{lfp}\, \lambda Y. R \sqcup (C[\![\, s \,]\!] \circ C[\![\, e \bowtie 0 \,]\!])Y)
\end{array}
$$

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

Equational vs. denotational form

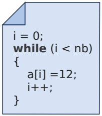
Equational:



$$\begin{cases} \mathcal{X}_1 = \top \\ \mathcal{X}_2 = F_2(\mathcal{X}_1) \\ \mathcal{X}_3 = F_3(\mathcal{X}_1) \\ \mathcal{X}_4 = F_4(\mathcal{X}_3, \mathcal{X}_4) \end{cases}$$

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

Denotational:



$$\mathsf{C}[\![\, \texttt{while}\ c\ \texttt{do}\ b\ \texttt{done} \,]\!]\, X \stackrel{\text{def}}{=}$$
$$\quad \mathsf{C}[\![\, \neg c \,]\!]\, (\mathrm{lfp}\, \lambda Y.X \cup \mathsf{C}[\![\, b \,]\!]\, (\mathsf{C}[\![\, c \,]\!]\, Y))$$
$$\mathsf{C}[\![\, \texttt{if}\ c\ \texttt{then}\ t\ \texttt{fi} \,]\!]\, X \stackrel{\text{def}}{=}$$
$$\quad \mathsf{C}[\![\, t \,]\!]\, (\mathsf{C}[\![\, c \,]\!]\, X) \cup \mathsf{C}[\![\, \neg c \,]\!]\, X$$

...

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

Non-modular concurrent semantics

Equational concurrent state semantics

Equational form:

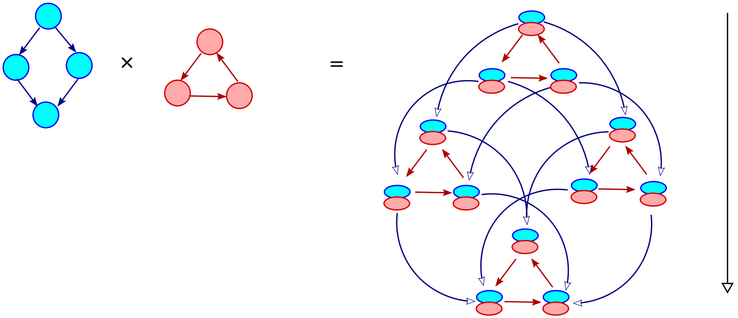
- for each $L \in \mathbb{T} \to \mathcal{L}$, a variable $\mathcal{X}_L$ with value in $\mathcal{E}$
- equations are derived from thread equations $eq(\texttt{stmt}_t)$ as:

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

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

(see course 2 for the full definition of $eq(\texttt{stmt})$)

Equational state semantics (illustration)



Product of control-flow graphs:

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

Equational state semantics (example)

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

t_1	t_2
$\ell 1$ `while random do`	$\ell 4$ `while random do`
$\ell 2$ `if x < y then`	$\ell 5$ `if y < 100 then`
$\ell 3$ `x ← x + 1`	$\ell 6$ `y ← y + [1,3]`

Equation system:

$$
\begin{aligned}
\mathcal{X}_{1,4} &= \mathcal{I} \\
\mathcal{X}_{2,4} &= \mathcal{X}_{1,4} \cup \mathsf{C}[\![\, x \ge y \,]\!]\, \mathcal{X}_{2,4} \cup \mathsf{C}[\![\, x \leftarrow x+1 \,]\!]\, \mathcal{X}_{3,4} \\
\mathcal{X}_{3,4} &= \mathsf{C}[\![\, x < y \,]\!]\, \mathcal{X}_{2,4} \\
\mathcal{X}_{1,5} &= \mathcal{X}_{1,4} \cup \mathsf{C}[\![\, y \ge 100 \,]\!]\, \mathcal{X}_{1,5} \cup \mathsf{C}[\![\, y \leftarrow y+[1,3] \,]\!]\, \mathcal{X}_{1,6} \\
\mathcal{X}_{2,5} &= \mathcal{X}_{1,5} \cup \mathsf{C}[\![\, x \ge y \,]\!]\, \mathcal{X}_{2,5} \cup \mathsf{C}[\![\, x \leftarrow x+1 \,]\!]\, \mathcal{X}_{3,5} \cup \\
&\quad\ \mathcal{X}_{2,4} \cup \mathsf{C}[\![\, y \ge 100 \,]\!]\, \mathcal{X}_{2,5} \cup \mathsf{C}[\![\, y \leftarrow y+[1,3] \,]\!]\, \mathcal{X}_{2,6} \\
\mathcal{X}_{3,5} &= \mathsf{C}[\![\, x < y \,]\!]\, \mathcal{X}_{2,5} \cup \mathcal{X}_{3,4} \cup \mathsf{C}[\![\, y \ge 100 \,]\!]\, \mathcal{X}_{3,5} \cup \mathsf{C}[\![\, y \leftarrow y+[1,3] \,]\!]\, \mathcal{X}_{3,6} \\
\mathcal{X}_{1,6} &= \mathsf{C}[\![\, y < 100 \,]\!]\, \mathcal{X}_{1,5} \\
\mathcal{X}_{2,6} &= \mathcal{X}_{1,6} \cup \mathsf{C}[\![\, x \ge y \,]\!]\, \mathcal{X}_{2,6} \cup \mathsf{C}[\![\, x \leftarrow x+1 \,]\!]\, \mathcal{X}_{3,6} \cup \mathsf{C}[\![\, y < 100 \,]\!]\, \mathcal{X}_{2,5} \\
\mathcal{X}_{3,6} &= \mathsf{C}[\![\, x < y \,]\!]\, \mathcal{X}_{2,6} \cup \mathsf{C}[\![\, y < 100 \,]\!]\, \mathcal{X}_{3,5}
\end{aligned}
$$

Equational state semantics (example)

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

t_1	t_2
$\ell 1$ `while random do`	$\ell 4$ `while random do`
$\ell 2$ `if x < y then`	$\ell 5$ `if y < 100 then`
$\ell 3$ `x ← x + 1`	$\ell 6$ `y ← y + [1,3]`

Pros:

- easy to construct
- easy to further abstract in an abstract domain $\mathcal{E}^\sharp$

Cons:

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

Wish-list

We would like to:

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

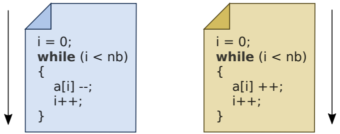
Ideally: thread-modular denotational-style semantics

analyze each thread independently by induction on its syntax

but remain sound with respect to all interleavings !

Simple interference semantics

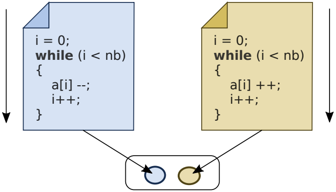
Thread-modular analysis with simple interferences



Principle:

- analyze each thread in isolation

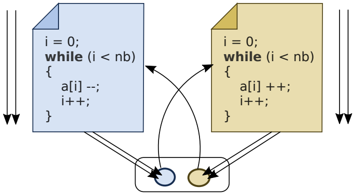
Thread-modular analysis with simple interferences



Principle:

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

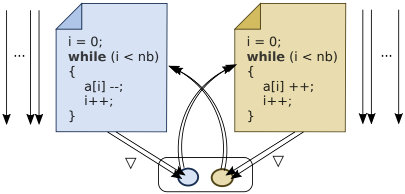
Thread-modular analysis with simple interferences



Principle:

- analyze each thread in isolation
- gather the values written into each variable by each thread
  $\Longrightarrow$ 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
  $\Longrightarrow$ 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
  $\Longrightarrow$ 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): $x = y = 0$ $\mathcal{X}_2 = \mathcal{X}_1 \cup \mathsf{C}[\![\, x \leftarrow x + 1 \,]\!]\, \mathcal{X}_3 \cup \mathsf{C}[\![\, x \geq y \,]\!]\, \mathcal{X}_2$

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

Example

t_1

```
ℓ1 while random do
  ℓ2 if x < y then
    ℓ3 x ← x + 1
```

t_2

```
ℓ4 while random do
  ℓ5 if y < 100 then
    ℓ6 y ← y + [1,3]
```

Analysis of t_2 in isolation

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

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

Example

t_1

```
ℓ1 while random do
  ℓ2 if x < y then
    ℓ3 x ← x + 1
```

t_2

```
ℓ4 while random do
  ℓ5 if y < 100 then
    ℓ6 y ← y + [1,3]
```

Re-analysis of t_1 with interferences from t_2

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

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

(2): $x \in [0, 102],\ y = 0$ $\quad \mathcal{X}_2 = \mathcal{X}_{1a} \cup \mathsf{C}[\![\, x \leftarrow x + 1 \,]\!]\, \mathcal{X}_3 \cup \mathsf{C}[\![\, x \geq (y \mid [1, 102]) \,]\!]\, \mathcal{X}_2$

(3): $x \in [0, 102],\ y = 0$ $\quad \mathcal{X}_3 = \mathsf{C}[\![\, x < (y \mid [1, 102]) \,]\!]\, \mathcal{X}_2$

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

subsequent re-analyses are identical (fixpoint reached)

Example

t_1

```
ℓ1 while random do
  ℓ2 if x < y then
    ℓ3 x ← x + 1
```

t_2

```
ℓ4 while random do
  ℓ5 if y < 100 then
    ℓ6 y ← 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 (we'll fix that later)

Formalizing the simple interference 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 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 $I \subseteq \mathbb{I}$.

Expressions : $\mathrm{E_t}[\![\, \mathrm{exp} \,]\!] : \mathcal{E} \times \mathcal{P}(\mathbb{I}) \to \mathcal{P}(\mathbb{R}) \times \mathcal{P}(\Omega)$ for thread t

- add interference $I \in \mathbb{I}$, 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:
  $\mathrm{E_t}[\![\, X \,]\!]\,\langle\, \rho,\, I \,\rangle \stackrel{\text{def}}{=} \langle\, \{\, \rho(X) \,\} \cup \{\, v \mid \exists u \neq t \colon \langle\, u,\, X,\, v \,\rangle \in I \,\},\, \emptyset \,\rangle$
- Pass recursively I down to sub-expressions:
  $\mathrm{E_t}[\![\, -e \,]\!]\,\langle\, \rho,\, I \,\rangle \stackrel{\text{def}}{=} \text{let } \langle\, V,\, O \,\rangle = \mathrm{E_t}[\![\, e \,]\!]\,\langle\, \rho,\, I \,\rangle \text{ in } \langle\, \{\, -v \mid v \in V \,\},\, O \,\rangle$
- etc.

Denotational semantics with interferences (cont.)

Statements with interference: for thread t

$$\mathsf{C_t}[\![\, \mathtt{stmt} \,]\!] : \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega) \times \mathcal{P}(\mathbb{I}) \to \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega) \times \mathcal{P}(\mathbb{I})$$

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

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

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

...

noting $\langle\, V_\rho,\ O_\rho \,\rangle \stackrel{\text{def}}{=} \mathsf{E_t}[\![\, e \,]\!] \langle\, \rho,\ I \,\rangle$
$\sqcup$ 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: $P[\![\,\texttt{prog}\,]\!] \subseteq \Omega$

Given $\texttt{prog} ::= \texttt{stmt}_1 \mid\mid \cdots \mid\mid \texttt{stmt}_n$, we compute:

$$P[\![\,\texttt{prog}\,]\!] \stackrel{\text{def}}{=} \left[\text{lfp}\, \lambda\langle\, O,\, I\,\rangle. \bigsqcup_{t\in\mathbb{T}} \, [C_t[\![\,\texttt{stmt}_t\,]\!]\,\langle\, \mathcal{E}_0,\, \emptyset,\, I\,\rangle]_{\Omega,\mathbb{I}}\right]_\Omega$$

- each thread analysis starts in an initial environment set $\mathcal{E}_0 \stackrel{\text{def}}{=} \{\, \lambda V.0\,\}$
- $[X]_{\Omega,\mathbb{I}}$ projects $X \in \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega) \times \mathcal{P}(\mathbb{I})$ on $\mathcal{P}(\Omega) \times \mathcal{P}(\mathbb{I})$ and interferences and errors from all threads are joined

  the output environments from a thread analysis are not easily exploitable
- $P[\![\,\texttt{prog}\,]\!]$ only outputs the set of possible run-time errors

We will need to prove the soundness of $P[\![\,\texttt{prog}\,]\!]$ with respect to the interleaving semantics...

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

Abstract semantics with interferences $\mathsf{C}_t^\sharp[\![s]\!]$

derived from $\mathsf{C}^\sharp[\![s]\!]$ in a generic way:

Example: $\mathsf{C}_t^\sharp[\![X \leftarrow e]\!] \langle R^\sharp, \Omega, I^\sharp \rangle$

- for each Y in e, get its interference $Y^\sharp_{\mathcal{R}} = \bigsqcup^\sharp_{\mathcal{R}} \{ I^\sharp\langle u, Y \rangle \mid u \neq t \}$
- if $Y^\sharp_{\mathcal{R}} \neq \bot^\sharp_{\mathcal{R}}$, replace Y in e with $get\langle Y, R^\sharp \rangle \sqcup^\sharp_{\mathcal{R}} Y^\sharp_{\mathcal{R}}$
  $get(Y, R^\sharp)$ extracts the abstract values variable Y from $R^\sharp \in \mathcal{E}^\sharp$
- compute $\langle R^{\sharp\prime}, O' \rangle = \mathsf{C}^\sharp[\![e]\!] \langle R^\sharp, O \rangle$
- enrich $I^\sharp\langle t, X \rangle$ with $get(X, R^{\sharp\prime})$

Static analysis with interferences

Abstract analysis

$$\mathsf{P}^\sharp[\![\,\texttt{prog}\,]\!] \stackrel{\text{def}}{=} \left[\lim \lambda\langle\, O,\, I^\sharp\,\rangle.\langle\, O,\, I^\sharp\,\rangle \mathbin{\triangledown} \bigsqcup\nolimits^\sharp_{t\in\mathbb{T}} \left[\, \mathsf{C}^\sharp_t[\![\,\texttt{stmt}_t\,]\!]\langle\, \mathcal{E}^\sharp_0,\, \emptyset,\, I^\sharp\,\rangle\,\right]_{\Omega,\mathbb{I}^\sharp}\right]_{\Omega}$$

- effective analysis by structural induction
- $\mathsf{P}^\sharp[\![\,\texttt{prog}\,]\!]$ is sound with respect to $\mathsf{P}[\![\,\texttt{prog}\,]\!]$
- termination ensured by a widening
- parameterized by a choice of abstract domains $\mathcal{R}^\sharp$, $\mathcal{E}^\sharp$

- interferences are flow-insensitive and non-relational in $\mathcal{R}^\sharp$
- thread analysis remains flow-sensitive and relational in $\mathcal{E}^\sharp$

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

Path-based soundness proof

Control paths of a sequential program

$atomic ::= X \leftarrow \texttt{exp} \mid \texttt{exp} \bowtie 0$

Control paths

$\underline{\pi} : \texttt{stmt} \rightarrow \mathcal{P}(atomic^*)$

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

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

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

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

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

e.g. $\pi(i \leftarrow 0;\ \texttt{while } i < 10 \texttt{ do } i \leftarrow i + 1 \texttt{ done};\ x \leftarrow i) = i \leftarrow 0 \cdot (i < 10 \cdot i \leftarrow i + 1)^* \cdot x \leftarrow i$

Path-based concrete semantics of sequential programs

Join-over-all-path semantics

$\Pi[\![P]\!] : (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)) \to (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)) \quad P \subseteq atomic^*$

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

Semantic equivalence

$$\mathsf{C}[\![\texttt{stmt}]\!] = \Pi[\![\pi(\texttt{stmt})]\!]$$

no longer true in the abstract

Path-based concrete semantics of concurrent programs

Concurrent control paths

$$\pi_* \stackrel{\text{def}}{=} \{\, \text{interleavings of } \pi(\texttt{stmt}_t),\ t \in \mathbb{T} \,\}$$
$$= \{\, p \in \mathit{atomic}^* \mid \forall t \in \mathbb{T},\ \mathit{proj}_t(p) \in \pi(\texttt{stmt}_t) \,\}$$

Interleaving program semantics

$$\mathsf{P}_*[\![\, \texttt{prog} \,]\!] \stackrel{\text{def}}{=} [\, \Pi[\![\, \pi_* \,]\!] \langle\, \mathcal{E}_0, \emptyset \,\rangle \,]_\Omega$$

($\mathit{proj}_t(p)$ 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
  $\Longrightarrow$ abstract as a denotational semantics

Soundness of the interference semantics

Soundness theorem

$$\mathrm{P}_*[\![\,\texttt{prog}\,]\!] \subseteq \mathrm{P}[\![\,\texttt{prog}\,]\!]$$

Proof sketch:

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

Notes:

- sound but not complete
- can be extended to soundness proof under weakly consistent memories

Weakly consistent memories

Issues with weak consistency

program written

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

(simplified Dekker mutual exclusion algorithm)

S_1 and S_2 cannot execute simultaneously.

Issues with weak consistency

program written			program executed	

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

(simplified Dekker mutual exclusion algorithm)

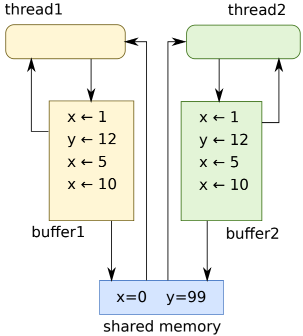
S_1 and S_2 can execute simultaneously.
Not a sequentially consistent behavior!

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

original program	
R1 ← X; Y ← R1	R ← Y; X ← R2

⟶

"optimized" program	
Y ← 42; R1 ← X; Y ← R1	R2 ← Y; X ← R2

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

We should not have $R_1 = 42$.

Possible if we allow speculative writes!
$\Longrightarrow$ 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

original program

$X \leftarrow X + 1 \;|\; X \leftarrow X + 1$

$\longrightarrow$

executed program

$r_1 \leftarrow X + 1$	$r_2 \leftarrow X + 1$
$X \leftarrow r_1$	$X \leftarrow r_2$

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 $X_1 \notin var(e_2)$, $X_2 \notin var(e_1)$, and e_1 does not stop the program)
- Propagation: $X \leftarrow e \cdot s \rightsquigarrow X \leftarrow e \cdot s[e/X]$
  (if $X \notin var(e)$, $var(e)$ are thread-local, and e is deterministic)
- Factorization: $s_1 \cdot \ldots \cdot s_n \rightsquigarrow X \leftarrow e \cdot s_1[X/e] \cdot \ldots \cdot s_n[X/e]$
  (if X is fresh, $\forall i$, $var(e) \cap lval(s_i) = \emptyset$, 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 $\mathsf{P}'_*[\![\,\texttt{prog}\,]\!]$

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

Soundness theorem

$\mathsf{P}'_*[\![\,\texttt{prog}\,]\!] \subseteq \mathsf{P}[\![\,\texttt{prog}\,]\!]$

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

Locks and synchronization

Scheduling

Synchronization primitives

$$\begin{array}{lcl} \texttt{stmt} & ::= & \texttt{lock}(m) \\ & | & \texttt{unlock}(m) \end{array}$$

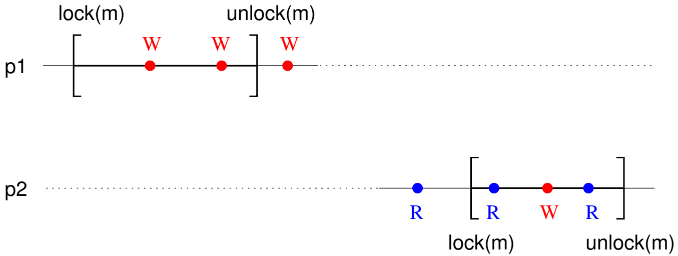
$m \in \mathbb{M}$: finite set of non-recursive mutexes

Scheduling

mutexes ensure mutual exclusion

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

Mutual exclusion



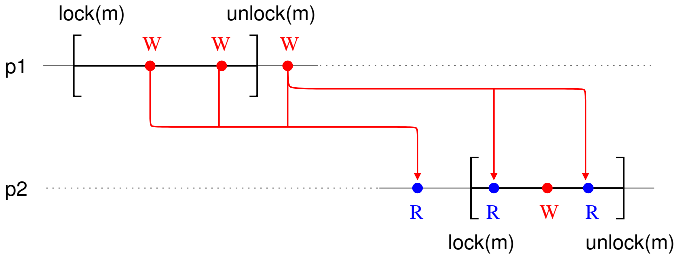
We use a refinement of the simple interference semantics by partitioning wrt. an abstract local view of the scheduler $\mathbb{C}$

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

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

- data-race writes $\mathbb{C}_{race}$
- well-synchronized writes $\mathbb{C}_{sync}$

Mutual exclusion



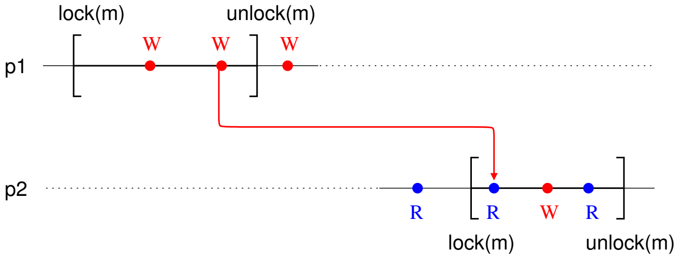
Data-race effects $\quad \mathbb{C}_{race} \simeq \mathcal{P}(\mathbb{M})$

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

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

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)
- $C_t[\![\,\texttt{unlock}(m)\,]\!]\langle\,\rho,\, M,\, I\,\rangle$ stores $\rho(X)$ into I
- $C_t[\![\,\texttt{lock}(m)\,]\!]\langle\,\rho,\, M,\, I\,\rangle$ imports values form I into ρ
- imprecision: non-relational, largely flow-insensitive

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

Example analysis

abstract consumer/producer

consumer	producer
`while random do`	`while random do`
`lock(m);` $^{\ell 1}$	`lock(m);`
`if X>0 then` $^{\ell 2}$`X←X-1 fi;`	`X←X+1;`
`unlock(m);`	`if X>100 then X←100 fi;`
$^{\ell 3}$`Y←X`	`unlock(m)`
`done`	`done`

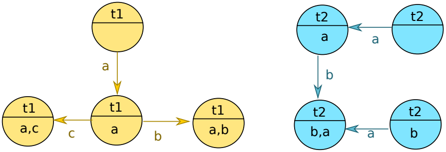
- no data-race interference (proof of absence of data-race)
- well-synchronized interferences:
  consumer: $x \leftarrow [0, 99]$
  producer: $x \leftarrow [1, 100]$
- $\implies$ we can prove that $y \in [0, 100]$

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

Can be generalized to several consumers and producers.

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)

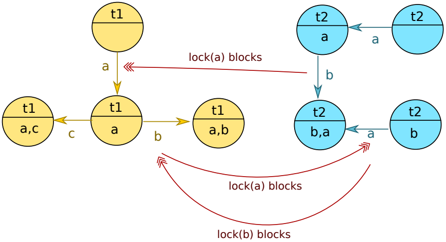


During the analysis, gather:

- 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 $R \times \mathbb{M}$

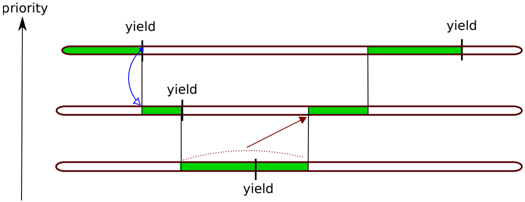
Then, construct a blocking graph between lock instructions

- $((t, m), \ell)$ 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.

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: $\{P\}\,\texttt{stmt}\,\{Q\}$

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

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

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

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

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

Jones' rely-guarantee proof method

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

Rely-guarantee "quintuples": $R, G \vdash \{P\}\,\texttt{stmt}\,\{Q\}$

- 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 $\mathcal{P}(\Sigma \times \mathcal{A} \times \Sigma)$)

The parallel composition rule is:

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

Rely-guarantee example

checking t_1

```
ℓ1 while random do
  ℓ2 if x < y then
    ℓ3 x ← 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

```
ℓ4 while random do
  ℓ5 if y < 100 then
    ℓ6 y ← y + [1,3]
  fi
done
```

at $\ell 4: \; x = y = 0$
at $\ell 5: \; x, y \in [0, 102], \; x \leq y$
at $\ell 6: \; x \in [0, 99], \; y \in [0, 99], \; x \leq y$

Rely-guarantee example

checking t_1

```
ℓ1 while random do
  ℓ2 if x < y then
    ℓ3 x ← x+1
  fi
done
```

x unchanged
y incremented
$0 \leq y \leq 102$

$\ell 1: \ x = y = 0$
$\ell 2: \ x, y \in [0, 102], \ x \leq y$
$\ell 3: \ x \in [0, 101], \ y \in [1, 102], \ x < y$

checking t_2

y unchanged
$0 \leq x \leq y$

```
ℓ4 while random do
  ℓ5 if y < 100 then
    ℓ6 y ← y + [1,3]
  fi
done
```

at $\ell 4: \ x = y = 0$
at $\ell 5: \ x, y \in [0, 102], \ x \leq y$
at $\ell 6: \ x \in [0, 99], \ y \in [0, 99], \ x \leq y$

In this example:

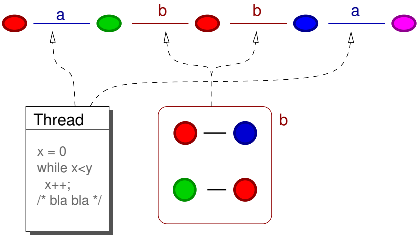
- guarantee exactly what is relied on ($R_1 = G_1$ and $R_2 = G_2$)
- rely and guarantee are global assertions

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

Modularity: main idea

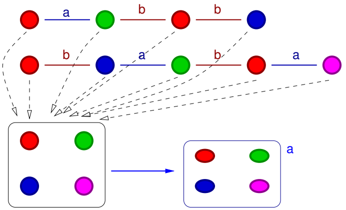


Main idea: separate execution steps

- 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*

$\Longrightarrow$ express the semantics from the point of view of a single thread

Trace decomposition



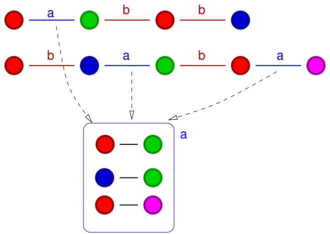
Reachable states projected on thread t: $\mathcal{RI}(t)$

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

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

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)

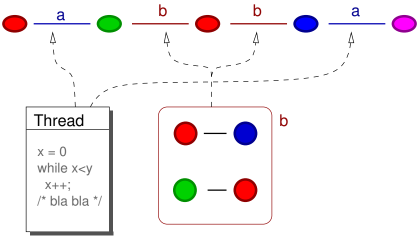
Extract the transitions with action t observed in $\mathcal{T}_p$

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

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

where $\alpha^{\mathbb{I}}(X)(t) \stackrel{\text{def}}{=} \{ \langle \sigma_i, \sigma_{i+1} \rangle \mid \exists \sigma_0 \stackrel{a_0}{\to} \sigma_1 \cdots \stackrel{a_{n-1}}{\to} \sigma_n \in X \colon a_i = t \}$

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:

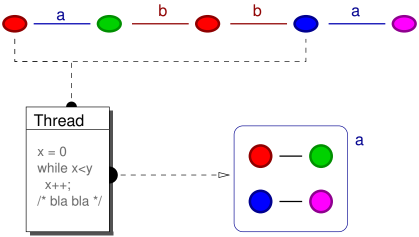
- transitions from the current thread t
- transitions from interferences A by other threads

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

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

$\implies$ similar to reachability for a sequential program, up to A

Thread-modular concrete semantics



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

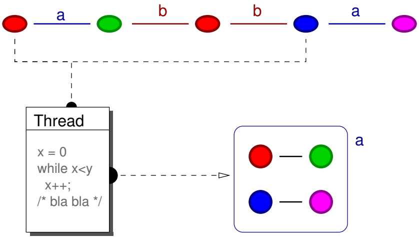
Interferences: A

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

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

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

Thread-modular concrete semantics



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

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: $\forall t: \mathcal{RI}(t) \simeq \mathcal{R}$ (π_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:

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

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

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

  in the powerset lattice $\mathcal{P}(\Sigma_t)$

  (similar to the sequential semantics of thread t in isolation)

$\Longrightarrow$ nested iterations

Abstract rely-guarantee

Suggested algorithm: nested iterations with acceleration

once abstract domains for states and interferences are chosen

- start from $\mathcal{RI}_0^\sharp \stackrel{\text{def}}{=} A_0^\sharp \stackrel{\text{def}}{=} \lambda t.\bot^\sharp$
- while $A_n^\sharp$ is not stable
  - compute $\forall t \in \mathbb{T}: \mathcal{RI}_{n+1}^\sharp(t) \stackrel{\text{def}}{=} \text{lfp } R_t^\sharp(A_n^\sharp)$
    by iteration with widening $\triangledown$

    ($\simeq$ separate analysis of each thread)
  - compute $A_{n+1}^\sharp \stackrel{\text{def}}{=} A_n^\sharp \triangledown B^\sharp(\mathcal{RI}_{n+1}^\sharp)$
- when $A_n^\sharp = A_{n+1}^\sharp$, return $\mathcal{RI}_n^\sharp$

$\Longrightarrow$ 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

Flow-insensitive abstraction:

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

Local state abstraction: remove auxiliary variables

$$\alpha^{nf}_{\mathcal{R}}(X) \stackrel{\text{def}}{=} \{\, \langle\, \ell,\, \rho_{|\mathbb{V}} \,\rangle \,|\, \langle\, \ell,\, \rho \,\rangle \in X \,\} \cup X$$

Interference abstraction: remove all control state

$$\alpha^{nf}_{A}(Y) \stackrel{\text{def}}{=} \{\, \langle\, \rho,\, \rho' \,\rangle \,|\, \exists L, L' \in \mathbb{T} \to \mathcal{L} \colon \langle\, \langle\, L,\, \rho \,\rangle,\, \langle\, L',\, \rho' \,\rangle \,\rangle \in Y \,\}$$

Flow-insensitive abstraction (cont.)

Flow-insensitive fixpoint semantics:

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

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

- $B^{nf}(Z)(t) \stackrel{\text{def}}{=} \{\, \langle \rho, \rho' \rangle \mid \exists \ell, \ell' \in \mathcal{L}\colon \langle \ell, \rho \rangle \in Z(t) \land \langle \ell, \rho \rangle \to_t \langle \ell', \rho' \rangle \,\}$
  (extract interferences from reachable states)
- $R^{nf}_t(Y)(X) \stackrel{\text{def}}{=} R^{loc}_t(X) \cup A^{nf}_t(Y)(X)$
  (interleave steps)
- $R^{loc}_t(X) \stackrel{\text{def}}{=} \{\langle \ell^i_t, \lambda V.0 \rangle\} \cup \{\, \langle \ell', \rho' \rangle \mid \exists \langle \ell, \rho \rangle \in X\colon \langle \ell, \rho \rangle \to_t \langle \ell', \rho' \rangle \,\}$
  (thread step)
- $A^{nf}_t(Y)(X) \stackrel{\text{def}}{=} \{\, \langle \ell, \rho' \rangle \mid \exists \rho,\, u \neq t\colon \langle \ell, \rho \rangle \in X \land \langle \rho, \rho' \rangle \in Y(u) \,\}$
  (interference step)

Cost/precision trade-off:

- less variables
  $\Longrightarrow$ subsequent numeric abstractions are more efficient
- insufficient to analyze $x \leftarrow x + 1 \,||\, x \leftarrow x + 1$

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:

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

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

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

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

$\Longrightarrow$ we get back our simple interference analysis!

Finally, use a numeric abstract domain $\alpha : \mathcal{P}(\mathbb{V} \to \mathbb{R}) \to \mathcal{D}^\sharp$
for interferences, $\mathbb{V} \to \mathcal{P}(\mathbb{R})$ is abstracted as $\mathbb{V} \to \mathcal{D}^\sharp$

A note on unbounded thread creation

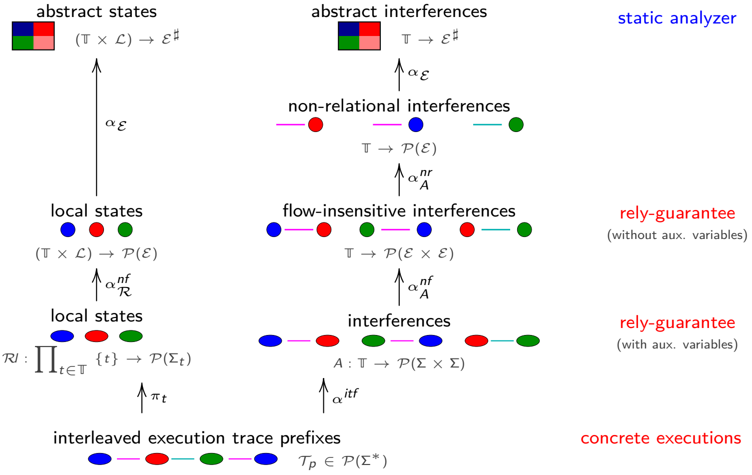
Extension: relax the finiteness constraint on $\mathbb{T}$

- there is still a finite syntactic set of threads $\mathbb{T}_s$
- some threads $\mathbb{T}_\infty \subseteq \mathbb{T}_s$ can have several instances
  (possibly an unbounded number)

Flow-insensitive analysis:

- local state and interference domains have finite dimensions
  ($\mathcal{E}_t$ and $(\mathcal{L} \times \mathcal{E}) \times (\mathcal{L} \times \mathcal{E})$, as opposed to $\mathcal{E}$ and $\mathcal{E} \times \mathcal{E}$)
- all instances of a thread $t \in \mathbb{T}_s$ are isomorphic
  $\Longrightarrow$ iterate the analysis on the finite set $\mathbb{T}_s$ (instead of $\mathbb{T}$)
- we must handle self-interferences for threads in $\mathbb{T}_\infty$:
  $A_t^{nf}(Y)(X) \stackrel{\text{def}}{=}$
  $\{\, (\ell, \rho') \,|\, \exists \rho, u \colon (u \neq t \vee t \in \mathbb{T}_\infty) \wedge (\ell, \rho) \in X \wedge (\rho, \rho') \in Y(u) \,\}$

From traces to thread-modular analyses



Relational thread-modular abstractions

Fully relational interferences with numeric domains

Reachability : $\mathcal{RI}(t) : \mathcal{L} \to \mathcal{P}(\mathbb{V}_a \to \mathbb{Z})$

approximated as usual with one numeric abstract element per label

auxiliary variables $pc_b \in \mathbb{V}_a$ 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 $\mathcal{P}((\mathbb{V}_a \to \mathbb{Z}) \times (\mathbb{V}_a \to \mathbb{Z})) \simeq \mathcal{P}((\mathbb{V}_a \cup \mathbb{V}'_a) \to \mathbb{Z})$

- $X \in \mathbb{V}_a$ value of variable X or auxiliary variable in the pre-state
- $X' \in \mathbb{V}'_a$ value of variable X or auxiliary variable in the post-state

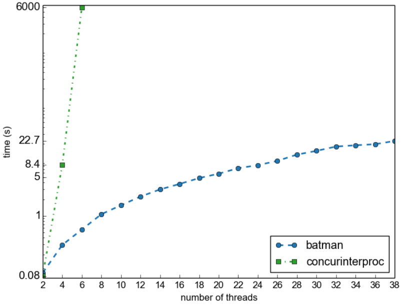
e.g.: $\{ (x, x+1) \mid x \in [0, 10] \}$ is represented as $x' = x + 1 \wedge x \in [0, 10]$

$\Longrightarrow$ 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



t_1	t_2

while $z < 10000$
  $z \leftarrow z + 1$
  if $y < c$ then $y \leftarrow y + 1$
done
	while $z < 10000$
	  $z \leftarrow z + 1$
	  if $x < y$ then $x \leftarrow x + 1$
	done

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

Partially relational 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

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

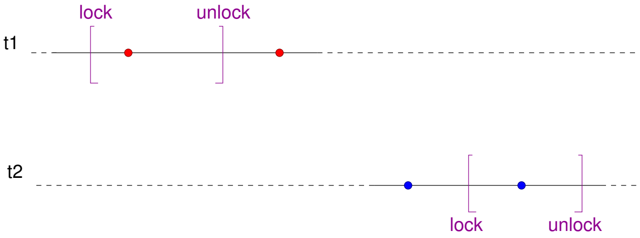
$\langle M, \rho \rangle \in \alpha_A^{\text{inv}}(Y) \Longrightarrow \langle M, \rho \rangle \in \alpha_A^{\text{inv}}(Y)$ after any sequence of interferences from Y

Lock invariant:

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

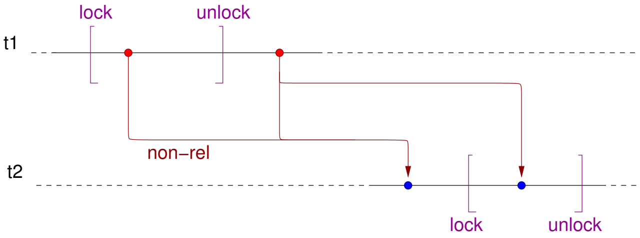
- property maintained outside code protected by m
- possibly broken while m is locked
- restored before unlocking m

Relational lock invariants



Improved interferences: mixing simple interferences and lock invariants

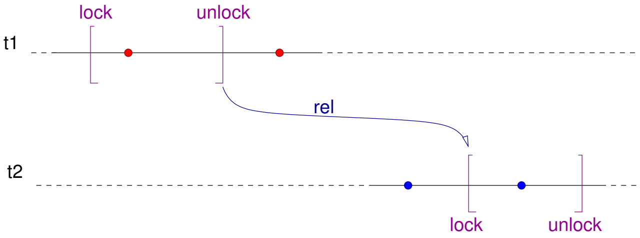
Relational lock invariants



Improved interferences: mixing simple interferences and lock invariants

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

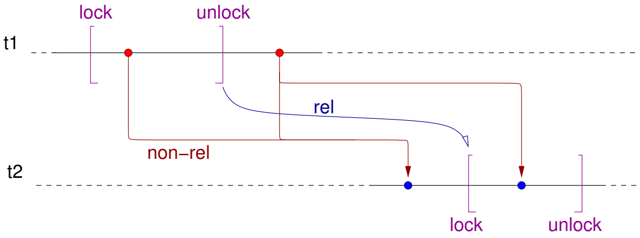
Relational lock invariants



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 $\nearrow$ monotonic or $\top$ don't know

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

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

Inference and use

- **gather:**
  $A^{\text{mono}}(t)(V) = \nearrow \iff$
  all assignments to V in t have the form $V \leftarrow V + e$, with $e \geq 0$
- **use:** combined with non-relational interferences
  if $\forall t: A^{\text{mono}}(t)(V) = \nearrow$
  then any test with non-relational interference $\mathsf{C}[\![X \leq (V \mid [a, b])]\!]$ can be strengthened into $\mathsf{C}[\![X \leq V]\!]$

Weakly relational interference example

analyzing t_1

t_1	t_2

`while random do`
`lock(m);`
`if x < y then`
`x ← x + 1;`
`unlock(m)`
	x unchanged
	y incremented
	$0 \leq y \leq 102$

analyzing t_2

t_1	t_2

y unchanged
$0 \leq x$, $x \leq y$
	`while random do`
	`lock(m);`
	`if y < 100 then`
	`y ← y + [1,3];`
	`unlock(m)`

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

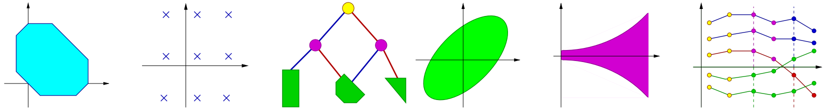
The Astrée analyzer

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 $\Longrightarrow$ 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

Conclusion

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

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.