Static analysis by abstract interpretation of concurrent programs

Antoine Miné

Habilitation
École normale supérieure
Paris, France

28 May 2013

Ariane 5 example (1996)

Cause: software error

- arithmetic overflow in unprotected data conversion from 64-bit float to 16-bit integer
- uncaught software exception $\Longrightarrow$ self-destruct sequence

Raised awareness about the importance of program verification:
even simple errors can have dramatic consequences
and are difficult to find *a priori*...

Ariane 5 example (1996)

...despite progress in:

- safer programming languages (Ada)
- rigorous development processes (embedded critical software)
- extensive testing (but not exhaustive)

Formal methods can help
(provide rigorous, mathematical insurance)

Reasoning about programs

Example

$i \leftarrow 2$
$n \leftarrow$ **input** $[-100, 100]$
while $i \leq n$ **do**

if random() then
$i \leftarrow i + 2$

Program proof: deductive method on a logic of programs

- pioneered by [Floyd 1967], [Hoare 1969], [Turing 1949]

Reasoning about programs

Example

$\{i{=}0, n{=}0\}$

$i \leftarrow 2$ $\{i{=}2, n{=}0\}$

$n \leftarrow$ **input** $[-100, 100]$ $\{i{=}2, -100{\leq}n{\leq}100]\}$

while $\{i{\geq}2, i{\leq}\max(2,n{+}2), -100{\leq}n{\leq}100]\}$ $i \leq n$ **do**

  $\{i{\geq}2, i{\leq}n, 2{\leq}n{\leq}100\}$

  if random() then

    $i \leftarrow i + 2$

$\{n{<}i{\leq}\max(2,n{+}2), -100{\leq}n{\leq}100\}$

Program proof: deductive method on a logic of programs

- pioneered by [Floyd 1967], [Hoare 1969], [Turing 1949]
- rely on the programmer to insert properties
- prove that they are (inductive) invariant
  (possibly with computer assistance)

Reasoning about programs

Example

$\{i{=}0, n{=}0\}$

$i \leftarrow 2$ $\{i{=}2, n{=}0\}$

$n \leftarrow$ **input** $[-100, 100]$ $\{i{=}2, -100{\le}n{\le}100]\}$

while $\{i{\ge}2, i{\le}\max(2,n{+}2), -100{\le}n{\le}100]\}$ $i \le n$ **do**

  $\{i{\ge}2, i{\le}n, 2{\le}n{\le}100\}$

  if random() then

    $i \leftarrow i + 2$

$\{n{<}i{\le}\max(2,n{+}2), -100{\le}n{\le}100\}$

how can we infer invariants?
(especially loop invariants)

generally undecidable
$\Longrightarrow$ use approximations

Semantic-based static analysis

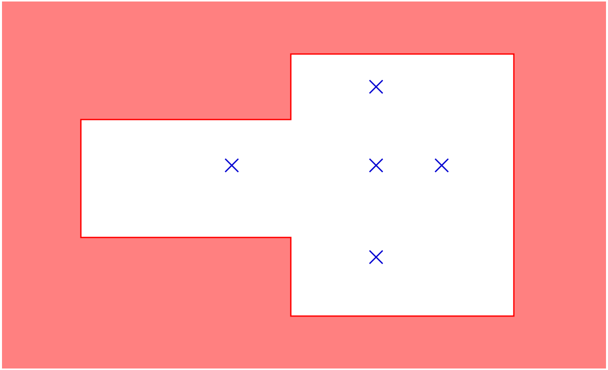
Static analysis:

- analyses directly the source code (not a reduced model)
- automatic and always terminating
- sound (full control and data coverage)
- incomplete (properties missed, false alarms)
- traditionally used in low precision settings (e.g., optimization)
  now precise enough for validation (few false alarms)
- parametrized and adaptable to different classes of programs

Abstract interpretation: unifying theory of program semantics

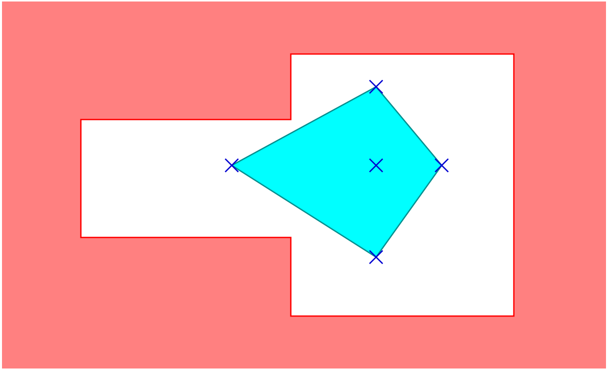
- introduced in [Cousot Cousot 1976]
- theoretical tools to design and compare static analyzes

Correctness proof and false alarms



The program is correct $(\text{blue} \cap \text{red} = \emptyset)$

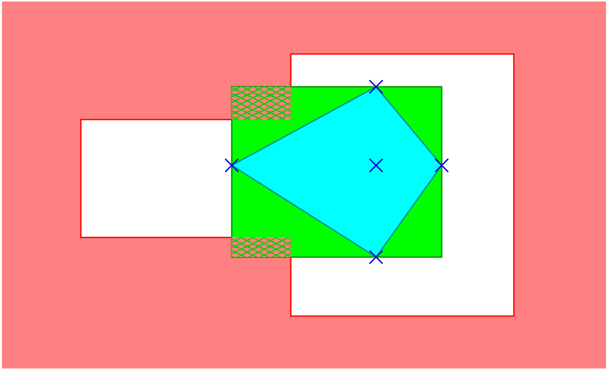
Correctness proof and false alarms



The program is correct $(\text{blue} \cap \text{red} = \emptyset)$

A polyhedral abstraction can prove the correctness $(\text{cyan} \cap \text{red} = \emptyset)$

Correctness proof and false alarms



The program is correct $(\text{blue} \cap \text{red} = \emptyset)$

A polyhedral abstraction can prove the correctness $(\text{cyan} \cap \text{red} = \emptyset)$

An interval abstraction cannot $(\text{green} \cap \text{red} \neq \emptyset$, false alarm)

Concurrent programming

Idea:

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

Why concurrent programs?

- can 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)

- can exploit several computers
  (distributed computing)

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

  $\Longrightarrow$ found in embedded critical applications (event-driven)

Concurrent programs verification

Concurrent programs are hard to design and hard to verify:

- programs are highly non-deterministic
  (many possible scheduling, execution interleavings)
  $\Longrightarrow$ testing is costly and ineffective, with low coverage
- errors appear in corner cases
- new kinds of errors (data-races, deadlocks)
- weakly consistent memory
  (no more total order of memory operations,
  causing unexpected behaviors)

Outline

- **Abstract interpretation primer**
  - static analysis of sequential programs
  - numeric abstract domains
- **Analysis of concurrent programs**
  - rely/guarantee reasoning, in abstract interpretation form
  - thread-modular interference-based analysis
  - advanced topics on interferences
    - soundness in weak memory consistency models
    - mutual exclusion and priorities
    - relational interferences
- **Implementation and experimentation**
  - Astrée: industrial static analyzer for sequential programs
  - AstréeA: prototype analyzer for concurrent programs
- **Conclusion**

Introduction to abstract interpretation

Principles of abstract interpretation

Key design steps:

1. Define a concrete semantics of the language
   - precise mathematical definition of programs
   - assumed correct (often w.r.t. informal specification)
   - uncomputable or combinatorial
   - constructive form (iterations up to fixpoints)
2. Extract a subset of properties of interest
   - goal properties & intermittent properties
   - generally infinite or very large classes (intervals, polyhedra)
   - with an algebra: sound abstract operators
3. Design abstract domains
   - data-structure encoding
   - algorithms implementing the abstract operators
   - extrapolation operators (approximate fixpoints)

Transition systems

Formal model of programs (Σ, τ, I)

- Σ: set of program states
- $\tau \subseteq \Sigma \times \Sigma$: transition relation, $\sigma \to \sigma'$ (execution step)
- $I \subseteq \Sigma$: set of initial states

Transition systems

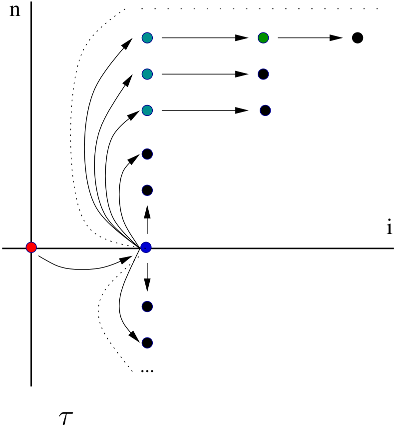
Formal model of programs (Σ, τ, I)

- Σ: set of program states
- $\tau \subseteq \Sigma \times \Sigma$: transition relation, $\sigma \to \sigma'$ (execution step)
- $I \subseteq \Sigma$: set of initial states

Example

[1] $i \leftarrow 2$
[2] $n \leftarrow$ **input** $[-100, 100]$
[3] **while** [4] $i \leq n$ **do**
  if random() then
    $i \leftarrow i + 2$
5

$\Sigma = \{\, 1, 2, 3, 4, 5 \,\} \times \mathbb{Z}^2$
$I = \{\, (1, 0, 0) \,\}$



Trace semantics

Partial execution traces $\mathbb{T}$

- set of execution traces, in $\mathcal{P}(\Sigma^*)$
- $\mathbb{T} \stackrel{\text{def}}{=} \text{lfp } F$ where
  $F(T) \stackrel{\text{def}}{=} I \cup \{ \langle \sigma_0, \ldots, \sigma_{n+1} \rangle \mid \langle \sigma_0, \ldots, \sigma_n \rangle \in T \land \sigma_n \to \sigma_{n+1} \}$

Expressiveness:

computing $\mathbb{T}$ is equivalent to exhaustive test
$\Longrightarrow$ can answer question about program safety

Cost:

$\mathbb{T}$ is often very large or unbounded
$\Longrightarrow$ well-defined mathematically but not computable

State semantics

State semantics $\mathbb{S}$:

- set of reachable states, in $\mathcal{P}(\Sigma)$
- $\mathbb{S} \stackrel{\text{def}}{=} \mathrm{lfp}\, G$ where $G(S) \stackrel{\text{def}}{=} I \cup \{\, \sigma \mid \exists \sigma' \in S \colon \sigma' \to \sigma \,\}$

Abstraction of the trace semantics:

- $\mathbb{S} = \alpha_{state}(\mathbb{T})$ where
  $\alpha_{state}(T) \stackrel{\text{def}}{=} \{\, \sigma_i \mid \exists \langle \sigma_0, \ldots, \sigma_n \rangle \in T \colon i \in [0, n] \,\}$

Expressiveness:

- forget the ordering of states in traces:
  $\alpha_{state}(\{$● — ● — ● — ●$\}) = \{$● ● ●$\}$
- still sufficient to prove safety properties
  (the program never reaches an error state)

Instantiation on a simple language

Language syntax

stat	$::=$	$X \leftarrow expr$	(*assignment*)
	$\mid$	**if** $expr \bowtie 0$ **then** $stat$	(*conditional*)
	$\mid$	**while** $expr \bowtie 0$ **do** $stat$	(*loop*)
	$\mid$	$stat; stat$	(*sequence*)
expr	$::=$	$X \mid [c_1, c_2] \mid expr \diamond_\ell expr \mid \cdots$	
$X \in \mathcal{V}$		finite set of variables	

$c_1, c_2 \in \mathbb{R}$, $\diamond \in \{+, -, \times, /\}$, $\bowtie \in \{=, >, \geq, <, \leq\}$

Idealized language:

- fixed, finite set of numeric variables (with value in $\mathbb{R}$)
- no function
- sequential (no concurrency)

Semantic of expressions and commands

States: $\Sigma \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{E}$

- control state $\ell \in \mathcal{L}$ (syntactic location)
- memory state $\sigma \in \mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{R}$ (maps variables to values)

Expression semantics: $\mathsf{E}[\![\, expr \,]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{R})$

$$\begin{array}{lcl}
\mathsf{E}[\![\, [c_1, c_2] \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{\, v \in \mathbb{R} \mid c_1 \leq v \leq c_2 \,\} \\
\mathsf{E}[\![\, X \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{\, \rho(X) \,\} \\
\mathsf{E}[\![\, -e_1 \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{\, -v \mid v \in \mathsf{E}[\![\, e_1 \,]\!] \,\} \\
\mathsf{E}[\![\, e_1 \diamond e_2 \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{\, v_1 \diamond v_2 \mid v_i \in \mathsf{E}[\![\, e_i \,]\!]\, \rho,\ \diamond \neq / \vee v_2 \neq 0 \,\}
\end{array}$$

Command semantics: $\mathsf{C}[\![\, stat \,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

$$\begin{array}{lcl}
\mathsf{C}[\![\, V \leftarrow e \,]\!]\, R & \stackrel{\text{def}}{=} & \{\, \rho[V \mapsto v] \mid \rho \in R,\ v \in \mathsf{e}[\![\, \rho \,]\!] \,\} \\
\mathsf{C}[\![\, e \bowtie 0 \,]\!]\, R & \stackrel{\text{def}}{=} & \{\, \rho \mid \rho \in R,\ \exists v \in \mathsf{e}[\![\, \rho \,]\!] : v \bowtie 0 \,\}
\end{array}$$

State semantic as equation systems

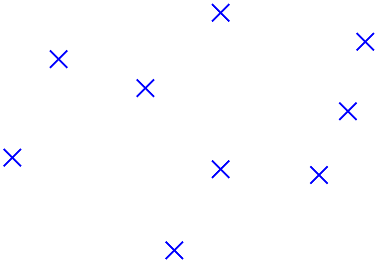
```
1 i ← 2
2 n ← input [−100, 100]
3 while 4 i ≤ n do
      5 if random() then
            i ← i + 2 6
7
```

$$\begin{aligned}
\mathcal{X}_1 &= \{ (0,0) \} \\
\mathcal{X}_2 &= C[\![i \leftarrow 2]\!] \mathcal{X}_1 \\
\mathcal{X}_3 &= C[\![n \leftarrow [-100, 100]]\!] \mathcal{X}_2 \\
\mathcal{X}_4 &= \mathcal{X}_3 \cup \mathcal{X}_6 \\
\mathcal{X}_5 &= C[\![i \leq n]\!] \mathcal{X}_4 \\
\mathcal{X}_6 &= \mathcal{X}_5 \cup C[\![i \leftarrow i + 2]\!] \mathcal{X}_5 \\
\mathcal{X}_7 &= C[\![i > n]\!] \mathcal{X}_4
\end{aligned}$$

where:

- $\forall \ell \in \mathcal{L}: \mathcal{X}_\ell \subseteq \mathcal{E}$ (states are partitioned by control location)
- (recursive) equation system stems from the program syntax
- program semantics is the least solution of the system
  (least fixpoint $\Longrightarrow$ most precise invariant)
- it can be solved by increasing iteration:
  $\forall \ell \in \mathcal{L}: \mathcal{X}_\ell^0 = \emptyset, \quad \forall i > 0: \mathcal{X}_\ell^{i+1} = F_\ell(\mathcal{X}_1^i, \ldots, \mathcal{X}_{|\mathcal{L}|}^i)$
  (may require transfinite iterations! $\Longrightarrow$ not computable)

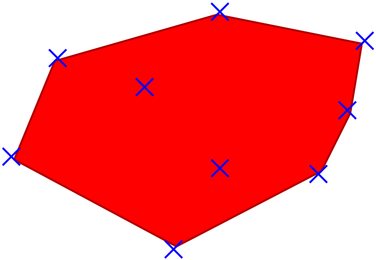
Numeric domains



concrete sets, in $\mathcal{P}(\mathcal{E})$: $\{\langle 0, 3\rangle, \langle 5.5, 0\rangle, \langle 12, 7\rangle, \ldots\}$ (not computable)

Numeric domains

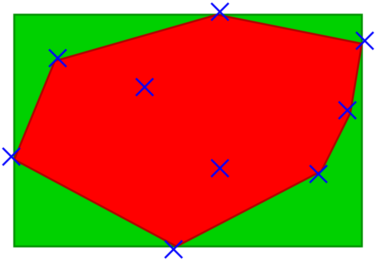
We abstract $\mathcal{P}(\mathcal{E}) \simeq \mathcal{P}(\mathbb{R}^{|\mathcal{V}|})$ further



concrete sets, in $\mathcal{P}(\mathcal{E})$: $\{\langle 0, 3 \rangle, \langle 5.5, 0 \rangle, \langle 12, 7 \rangle, \ldots\}$ (not computable)
polyhedra: $6X + 11Y \geq 33 \wedge \cdots$ (exponential cost)

Numeric domains

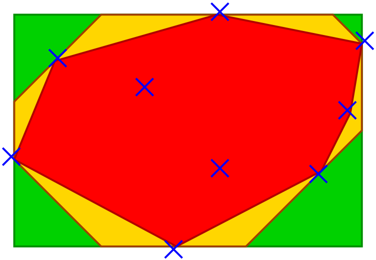
We abstract $\mathcal{P}(\mathcal{E}) \simeq \mathcal{P}(\mathbb{R}^{|\mathcal{V}|})$ further



concrete sets, in $\mathcal{P}(\mathcal{E})$:	$\{\langle 0, 3 \rangle, \langle 5.5, 0 \rangle, \langle 12, 7 \rangle, \ldots\}$	(not computable)
polyhedra:	$6X + 11Y \geq 33 \wedge \cdots$	(exponential cost)
intervals:	$X \in [0, 12] \wedge Y \in [0, 8]$	(linear cost)

Numeric domains

We abstract $\mathcal{P}(\mathcal{E}) \simeq \mathcal{P}(\mathbb{R}^{|\mathcal{V}|})$ further



concrete sets, in $\mathcal{P}(\mathcal{E})$:	$\{\langle 0, 3\rangle, \langle 5.5, 0\rangle, \langle 12, 7\rangle, \ldots\}$	(not computable)
polyhedra:	$6X + 11Y \geq 33 \wedge \cdots$	(exponential cost)
intervals:	$X \in [0, 12] \wedge Y \in [0, 8]$	(linear cost)
octagons:	$X + Y \geq 3 \wedge Y \geq 0 \wedge \cdots$	(cubic cost)

Trade-off between cost and expressiveness / precision

Static analysis

1 $i \leftarrow 2$
2 $n \leftarrow$ **input** $[-100, 100]$
3 **while** 4 $i \leq n$ **do**
  5 **if random() then**
    $i \leftarrow i + 2$ 6
7

$$
\begin{array}{lcl}
\mathcal{X}_1^{\sharp i+1} & \stackrel{\text{def}}{=} & \{ (0,0) \}^\sharp \\
\mathcal{X}_2^{\sharp i+1} & \stackrel{\text{def}}{=} & C^\sharp[\![i \leftarrow 2]\!] \, \mathcal{X}_1^{\sharp i} \\
\mathcal{X}_3^{\sharp i+1} & \stackrel{\text{def}}{=} & C^\sharp[\![n \leftarrow [-100, 100]]\!] \, \mathcal{X}_2^{\sharp i} \\
\mathcal{X}_4^{\sharp i+1} & \stackrel{\text{def}}{=} & \mathcal{X}_4^{\sharp i} \triangledown (\mathcal{X}_3^{\sharp i} \cup^\sharp \mathcal{X}_6^{\sharp i}) \\
\mathcal{X}_5^{\sharp i+1} & \stackrel{\text{def}}{=} & C^\sharp[\![i \leq n]\!] \, \mathcal{X}_4^{\sharp i} \\
\mathcal{X}_6^{\sharp i+1} & \stackrel{\text{def}}{=} & \mathcal{X}_5^{\sharp i} \cup^\sharp C^\sharp[\![i \leftarrow i + 2]\!] \, \mathcal{X}_5^{\sharp i} \\
\mathcal{X}_7^{\sharp i+1} & \stackrel{\text{def}}{=} & C^\sharp[\![i > n]\!] \, \mathcal{X}_4^{\sharp i}
\end{array}
$$

- abstract variables $\mathcal{X}_\ell^\sharp \in \mathcal{E}^\sharp$ replace concrete ones $\mathcal{X}_\ell \in \mathcal{P}(\mathcal{E})$
- abstract operators are used: $C^\sharp[\![\cdot]\!] : \mathcal{E}^\sharp \to \mathcal{E}^\sharp$, $\cup^\sharp : \mathcal{E}^\sharp \times \mathcal{E}^\sharp \to \mathcal{E}^\sharp$
- the system is solved by iterations
  $\mathcal{X}_\ell^{\sharp 0} \stackrel{\text{def}}{=} \emptyset^\sharp, \; \mathcal{X}_\ell^{\sharp i+1} \stackrel{\text{def}}{=} F_\ell^\sharp(\mathcal{X}_1^{\sharp i}, \ldots, \mathcal{X}_{|\mathcal{L}|}^{\sharp \; i})$
- widening $\triangledown$ is used to force convergence in finite time
  (e.g.: put unstable bounds to ∞)

  $\Longrightarrow$ effective, terminating, sound static analyzer

Contribution: floating-point polyhedra

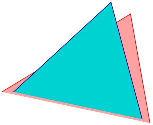
Original polyhedra use arbitrary precision rationals and double descriptions (constraints / generator) [Cousot Halbwachs 78]

Goal: use floats for improved scalability [Liqian Chen's PhD]

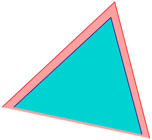
- constraints with float coefficients [Chen et al. 2008]
- constraints with float interval coefficients [Chen et al. 2009]

Algorithms: sound float versions of

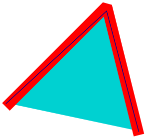
- Fourier-Motzkin elimination (approximate projection)
- guaranteed linear programming (sound enclosure)



unsound floats



sound float



sound float intervals

Contribution: domains for realistic data-types

Adapt domains from $\mathbb{R}$ to data-types found in actual programs

Machine integers: [Miné 2012]

- wrap-around semantics after overflow $(127 + 1 = -128)$
- specialized domain: modular intervals $(X \in [a, b] + c\mathbb{Z})$

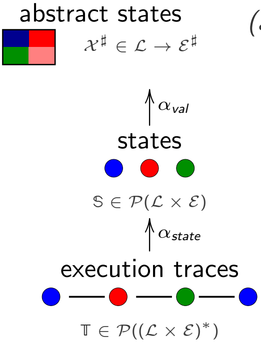
Floating-point numbers: [Miné 2004]

- handle rounding-errors (non-linear)
- abstract rounding as non-deterministic choice in intervals $(\mathit{round}(X) \rightsquigarrow X + [-\epsilon, \epsilon]X + [-\varepsilon, \varepsilon])$

Memory representation awareness: [Miné 2006]

- C union types (dynamic decomposition of the memory)
- ill-typed accesses through C pointer casts and arithmetic
- bit-level manipulation in machine integers and floats

Abstraction summary for sequential programs



(abstract invariants) — implementable, data-structures + algorithms

(invariants) — mathematical, non-computable

Static analysis of concurrent software

Concurrent language

Language extension:

- finite, fixed set of threads $stat_t$, $t \in \mathcal{T}$
- all variables $\mathcal{V}$ are shared

Execution model: non-deterministic interleaving of thread actions
(sequential consistency with atomic assignments and tests)

Labelled transition system:

- states $\Sigma \stackrel{\text{def}}{=} (\mathcal{T} \to \mathcal{L}) \times \mathcal{E}$
  (thread-local control state in $\mathcal{T} \to \mathcal{L}$, shared memory in $\mathcal{E}$)
- labelled transitions $\sigma \stackrel{t}{\to} \sigma'$, $t \in \mathcal{T}$

  $\langle L[t \mapsto \ell], \rho \rangle \stackrel{t}{\to} \langle L[t \mapsto \ell'], \rho' \rangle \iff \langle \ell, \rho \rangle \to_{stat_t} \langle \ell', \rho' \rangle$
  (derived from the transitions of individual threads)

Trace and state semantics

Labelled trace semantics:

- set of interleaved execution traces, with thread labels
- $\mathbb{T} \stackrel{\text{def}}{=} \text{lfp}\, F$ where
  $F(T) \stackrel{\text{def}}{=} I \cup \{\, \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_i} \sigma_{i+1} \mid \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{i-1}} \sigma_i \in T \wedge \sigma_i \xrightarrow{t_i} \sigma_{i+1} \,\}$

State semantics: (as before)

- $\mathbb{S} \stackrel{\text{def}}{=} \text{lfp}\, G$ where $G(S) \stackrel{\text{def}}{=} I \cup \{\, \sigma \mid \exists \sigma', t : \sigma' \xrightarrow{t} \sigma \,\}$
- $\mathbb{S} = \alpha_{state}(\mathbb{T})$ where
  $\alpha_{state}(T) \stackrel{\text{def}}{=} \{\, \sigma_i \mid \exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in \mathcal{T} : i \in [0, n] \,\}$

Idea:

forget about threads and labels
analyze as a sequential program interleaving thread statements

Equational state semantics example

Example: inferring $0 \leq x \leq y \leq 10$

t_1	t_2
while [1] true **do**	**while** [4] true **do**
[2] **if** $x < y$ **then**	[5] **if** $y < 10$ **then**
[3] $x \leftarrow x + 1$	[6] $y \leftarrow y + 1$

- attach variables $\mathcal{X}_L \in \mathcal{P}(\mathcal{E})$ to control locations $L \in \mathcal{T} \to \mathcal{L}$
- synthesize equations $\mathcal{X}_L = F_L(\mathcal{X}_{(1,\ldots,1)}, \ldots, \mathcal{X}_{(|\mathcal{L}|,\ldots,|\mathcal{L}|)})$
  from thread equations $\mathcal{X}_{\ell,t} = F_{\ell,t}(\mathcal{X}_{1,t}, \ldots, \mathcal{X}_{|\mathcal{L}|,t})$

Equational state semantics example

Example: inferring $0 \le x \le y \le 10$

t_1	t_2
while [1] true **do**	**while** [4] true **do**
[2] **if** $x < y$ **then**	[5] **if** $y < 10$ **then**
[3] $x \leftarrow x + 1$	[6] $y \leftarrow y + 1$

(Simplified) concrete equation system:

$$
\begin{aligned}
\mathcal{X}_{1,4} &= I \cup \mathrm{C}[\![\, x \leftarrow x + 1 \,]\!]\, \mathcal{X}_{3,4} \cup \mathrm{C}[\![\, x \ge y \,]\!]\, \mathcal{X}_{2,4} \\
&\qquad \cup \mathrm{C}[\![\, y \leftarrow y + 1 \,]\!]\, \mathcal{X}_{1,6} \cup \mathrm{C}[\![\, y \ge 10 \,]\!]\, \mathcal{X}_{1,5} \\
\mathcal{X}_{2,4} &= \mathcal{X}_{1,4} \cup \mathrm{C}[\![\, y \leftarrow y + 1 \,]\!]\, \mathcal{X}_{2,6} \cup \mathrm{C}[\![\, y \ge 10 \,]\!]\, \mathcal{X}_{2,5} \\
\mathcal{X}_{3,4} &= \mathrm{C}[\![\, x < y \,]\!]\, \mathcal{X}_{2,4} \cup \mathrm{C}[\![\, y \leftarrow y + 1 \,]\!]\, \mathcal{X}_{3,6} \cup \mathrm{C}[\![\, y \ge 10 \,]\!]\, \mathcal{X}_{3,5} \\
\mathcal{X}_{1,5} &= \mathrm{C}[\![\, x \leftarrow x + 1 \,]\!]\, \mathcal{X}_{3,5} \cup \mathrm{C}[\![\, x \ge y \,]\!]\, \mathcal{X}_{2,5} \cup \mathcal{X}_{1,4} \\
\mathcal{X}_{2,5} &= \mathcal{X}_{1,5} \cup \mathcal{X}_{2,4} \\
\mathcal{X}_{3,5} &= \mathrm{C}[\![\, x < y \,]\!]\, \mathcal{X}_{2,5} \cup \mathcal{X}_{3,4} \\
\mathcal{X}_{1,6} &= \mathrm{C}[\![\, x \leftarrow x + 1 \,]\!]\, \mathcal{X}_{3,6} \cup \mathrm{C}[\![\, x \ge y \,]\!]\, \mathcal{X}_{2,6} \cup \mathrm{C}[\![\, y < 10 \,]\!]\, \mathcal{X}_{1,5} \\
\mathcal{X}_{2,6} &= \mathcal{X}_{1,6} \cup \mathrm{C}[\![\, y < 10 \,]\!]\, \mathcal{X}_{2,5} \\
\mathcal{X}_{3,6} &= \mathrm{C}[\![\, x < y \,]\!]\, \mathcal{X}_{2,6} \cup \mathrm{C}[\![\, y < 10 \,]\!]\, \mathcal{X}_{3,5}
\end{aligned}
$$

Rely/guarantee proof method

Modular proof method introduced by [Jones 1981]

checking t_1

```
while ¹ true do
  ² if x < y then
    ³ x ← x + 1
```

at 1, 2 : $0 \leq x \leq y \leq 10$
at 3 : $0 \leq x < y \leq 10$

checking t_2

```
while ⁴ true do
  ⁵ if y < 10 then
    ⁶ y ← y + 1
```

at 4, 5 : $0 \leq x \leq y \leq 10$
at 6 : $0 \leq x \leq y < 10$

Annotate programs with:

- local invariants (attached to $\mathcal{L}$, not $\mathcal{T} \to \mathcal{L}$)

For each thread, prove that local invariants hold

Rely/guarantee proof method

Modular proof method introduced by [Jones 1981]

checking t_1

while [1] true **do** [2] **if** $x < y$ **then** [3] $x \leftarrow x + 1$	x unchanged y incremented $y \leq 10$

at 1, 2 : $0 \leq x \leq y \leq 10$
at 3 : $0 \leq x < y \leq 10$

checking t_2

y unchanged	**while** [4] true **do** [5] **if** $y < 10$ **then** [6] $y \leftarrow y + 1$

at 4, 5 : $0 \leq x \leq y \leq 10$
at 6 : $0 \leq x \leq y < 10$

Annotate programs with:

- local invariants (attached to $\mathcal{L}$, not $\mathcal{T} \to \mathcal{L}$)
- guarantees on transitions by other threads

For each thread, prove that local invariants and guarantees hold relying on guarantees from other threads

$\Longrightarrow$ check a thread against an abstraction of the other threads (does not require looking at other threads)

Contribution: rely/guarantee as abstract interpretation

Formalization as abstract interpretation [Miné 2012]

- constructive design (fixpoints)
- infer invariants and guarantees (instead of only checking)
- exploit existing abstractions (numeric domains)

Complementary abstractions: of the trace semantics $\mathbb{T}$

- thread-local states for $t \in \mathcal{T}$
  $\mathbb{S}_t \stackrel{\text{def}}{=} \pi_t(\alpha_{state}(\mathbb{T}))$ where
  $\pi_t\langle\, L,\, \rho \,\rangle \stackrel{\text{def}}{=} \langle\, L(t),\, \rho[\forall t' \neq t\colon pc_{t'} \mapsto L(t')] \,\rangle \in \mathcal{P}(\mathcal{L} \times \mathcal{E}_t)$
  (keep other threads' location in auxiliary variables)

- interferences generated by $t \in \mathcal{T}$
  $\mathbb{A}_t \stackrel{\text{def}}{=} \{\, \langle\, \sigma_i,\, \sigma_{i+1} \,\rangle \,|\, \exists \cdots \sigma_i \stackrel{t}{\rightarrow} \sigma_{i+1} \cdots \colon \in \mathbb{T} \,\}$
  transitions from τ actually observed in execution traces

  (relational and flow-sensitive information)

Contribution: rely/guarantee as abstract interpretation

Nested fixpoint form: for the state semantics $\mathbb{S}$

$\mathbb{S} = \mathrm{lfp}\ G$ where

$$G_t(S) \stackrel{\mathrm{def}}{=} \mathrm{lfp}\ H_t(\lambda t'.\{\,\langle \sigma, \sigma' \rangle \mid \sigma \in S_{t'},\ \sigma \stackrel{t'}{\to} \sigma'\,\})$$

$$H_t(A)(S) \stackrel{\mathrm{def}}{=} \pi_t(I \cup \{\,\sigma' \mid \exists \pi_t(\sigma) \in S\colon \sigma \stackrel{t}{\to} \sigma' \vee \exists t' \neq t\colon (\sigma, \sigma') \in A_{t'}\,\})$$

- $H_t(A)$: execute one step, in thread t or interferences A
- $G_t(S) \simeq \mathrm{lfp}\, H_t$: analyze thread t completely with fixed interferences (spawned from S)
- $\mathrm{lfp}\ G$: re-analyze all threads until interferences stabilize
- can be computed by (transfinite) iterations

Thread-modular, constructive, complete computation of safety properties

Further abstractions

State abstractions:

- forget auxiliary variables

  $\alpha_{aux}(X) \stackrel{\text{def}}{=} \{ \langle \ell, \rho_{|\mathcal{E}} \rangle \mid \langle \ell, \rho \rangle \in X \} \in \mathcal{P}(\mathcal{L} \times \mathcal{E})$

  (allows uniform analyses of threads with unbounded instances)

Interference abstractions:

- flow-insensitive abstraction:

  $\alpha_{flow}(X) \stackrel{\text{def}}{=} \{ \langle \rho, \rho' \rangle \mid \exists L, L' : \langle \langle L, \rho \rangle, \langle L', \rho' \rangle \rangle \in X \}$

  (infer global interferences)

- input-insensitive abstraction:

  $\alpha_{out}(X) \stackrel{\text{def}}{=} \{ \rho' \mid \exists \rho : \langle \rho, \rho' \rangle \in X \} \in \mathcal{P}(\mathcal{E})$

- non-relational abstraction:

  $\alpha_{val}(X) \stackrel{\text{def}}{=} \lambda V \in \mathcal{V}. \{ \rho(V) \mid \rho \in X \} \in \mathcal{V} \to \mathcal{P}(\mathbb{R})$

Further abstractions in numeric abstract domains

Application: simple interference analysis

Proposed initially and implemented in AstréeA in [Miné 2010]
reformulated as abstract rely-guarantee in [Miné 2012]

Interference abstraction in $\mathcal{I} \stackrel{\text{def}}{=} \mathcal{T} \times \mathcal{V} \times \mathbb{R}$

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

Modified semantic of expressions and commands:

$$\mathsf{E}_t[\![\, X \,]\!]\,\langle\, \rho,\, I \,\rangle \stackrel{\text{def}}{=} \{\, \rho(X) \,\} \cup \{\, v \mid \exists t' \neq t \colon \langle\, t',\, X,\, v \,\rangle \in I \,\}$$

$$\mathsf{C}_t[\![\, X \leftarrow e \,]\!]\,\langle\, R,\, I \,\rangle \stackrel{\text{def}}{=}$$
$$\langle\, \{\, \rho[X \mapsto v] \mid \rho \in R,\, v \in V_\rho \,\},\, I \cup \{\, \langle\, t,\, X,\, v \,\rangle \mid \rho \in R,\, v \in V_\rho \,\} \,\rangle$$
$$\text{where } V_\rho \stackrel{\text{def}}{=} \mathsf{E}_t[\![\, e \,]\!]\,\langle\, \rho,\, I \,\rangle$$

- analyze each thread as a sequential program with interferences $I \subseteq \mathcal{I}$
- a thread analysis infers new interferences
- iterate (with widening $\triangledown$) until stabilization

Simple interference analysis: example

Example

t_1	t_2
while [1] true **do**	**while** [4] true **do**
[2] **if** $x < y$ **then**	[5] **if** $y < 10$ **then**
[3] $x \leftarrow x + 1$	[6] $y \leftarrow y + 1$

Interference semantics:

iteration 1
$I = \emptyset$
at 2 : $x = 0,\ y = 0$
at 5 : $x = 0,\ y \in [0, 10]$
new $I = \{\langle t_2, y, 1\rangle, \ldots, \langle t_2, y, 10\rangle\}$

Simple interference analysis: example

Example

t_1	t_2
while [1] true **do**	**while** [4] true **do**
[2] **if** $x < y$ **then**	[5] **if** $y < 10$ **then**
[3] $x \leftarrow x + 1$	[6] $y \leftarrow y + 1$

Interference semantics:

iteration 2
$I = \{ \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \}$
at 2 : $x \in [0, 10]$, $y = 0$
at 5 : $x = 0$, $y \in [0, 10]$
new $I = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \}$

Simple interference analysis: example

Example

t_1	t_2
while [1] true **do**	**while** [4] true **do**
[2] **if** $x < y$ **then**	[5] **if** $y < 10$ **then**
[3] $x \leftarrow x + 1$	[6] $y \leftarrow y + 1$

Interference semantics:

iteration 3

$I = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \}$

at 2 : $x \in [0, 10]$, $y = 0$

at 5 : $x = 0$, $y \in [0, 10]$

new $I = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \}$

Simple interference analysis: example

Example

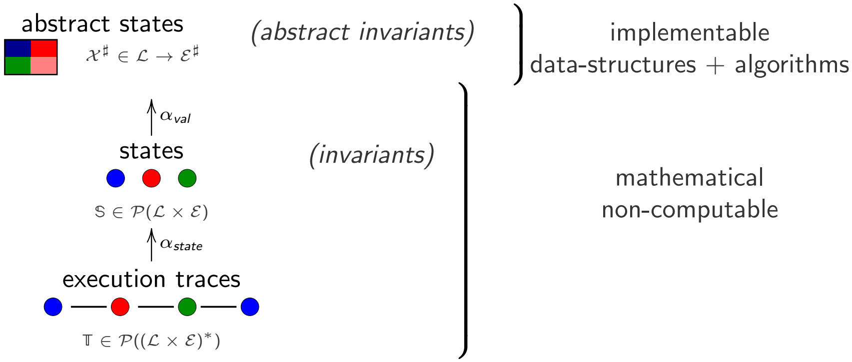
t_1	t_2
while ^{1} true **do**	**while** ^{4} true **do**
^{2} **if** $x < y$ **then**	^{5} **if** $y < 10$ **then**
^{3} $x \leftarrow x + 1$	^{6} $y \leftarrow y + 1$

Interference semantics:

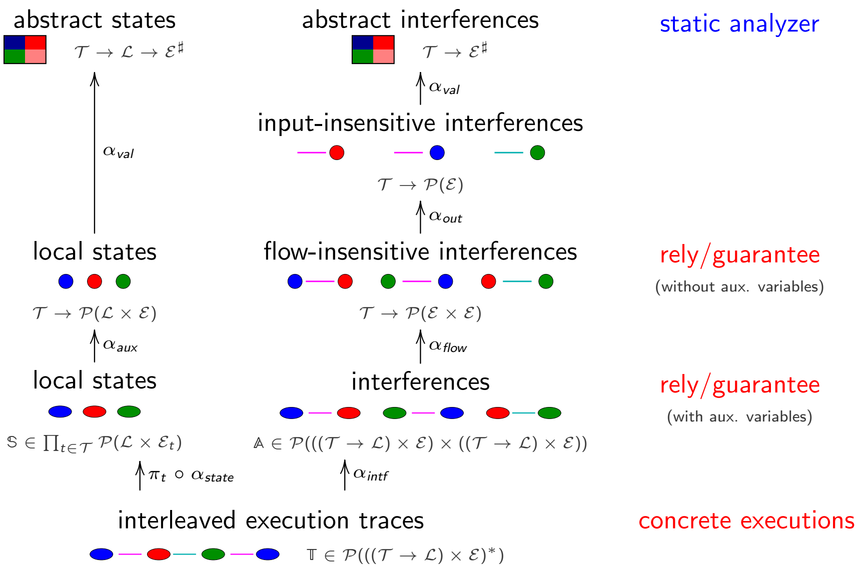
iteration 3
$I = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \}$
at 2 : $x \in [0, 10]$, $y = 0$
at 5 : $x = 0$, $y \in [0, 10]$
new $I = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \}$

Note: we cannot infer $x \leq y$ at 2, only $x, y \in [0, 10]$

Abstraction summary for sequential programs



Abstraction summary for concurrent programs



Weak memory consistency

program written

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

(simplified Dekker mutual exclusion algorithm)

S_1 and S_2 cannot execute simultaneously

Weak memory consistency

program written

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

$\longrightarrow$

program executed

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

(simplified Dekker mutual exclusion algorithm)

S_1 and S_2 can execute simultaneously

(non sequentially consistent behavior)

Causes:

- weak hardware memory model (write FIFOs, caches)
- thread-unaware compiler optimizations (reordering)
- now part of standards (Java, C, C++)

Weak memory consistency

program written

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

$\longrightarrow$

program executed

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

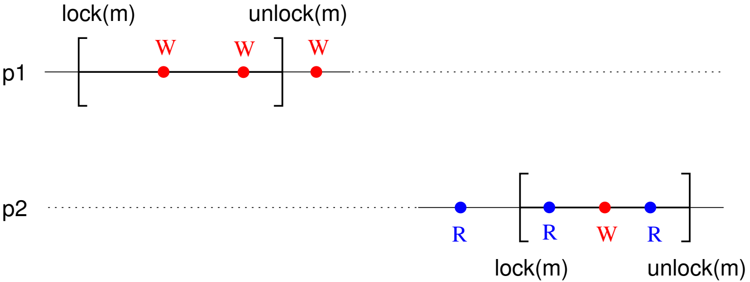
(simplified Dekker mutual exclusion algorithm)

Soundness theorem: [Miné 2011] [Alglave et al. 2011]

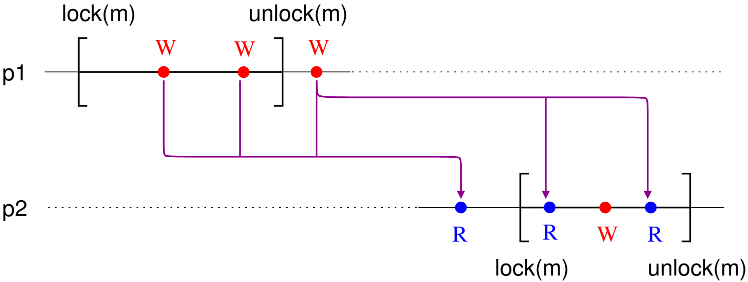
For flow-insensitive interference abstractions
the analysis is invariant by a wide range of thread transformations

- inserting FIFO buffers
- reordering of "independent" statements
- common sub-expression elimination
- change of granularity

Handling mutual exclusion



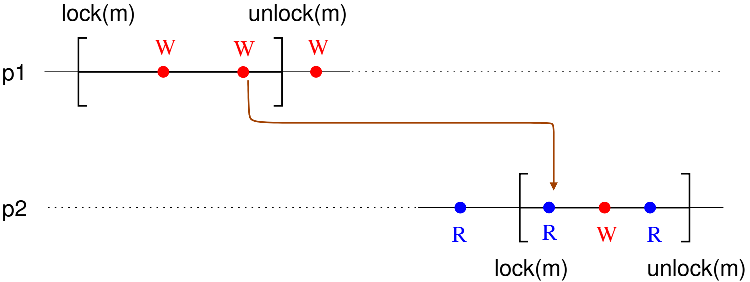
Handling mutual exclusion



No interference unless:

- write / read not protected by a common mutex (data-races), or

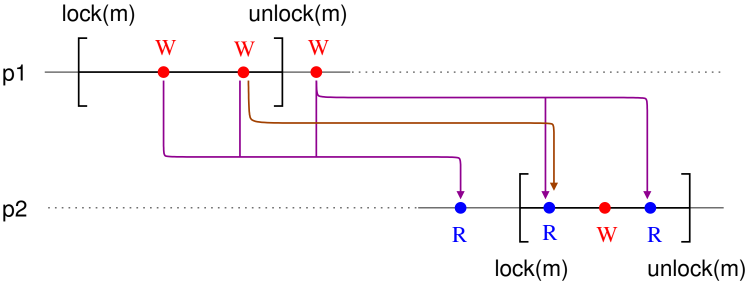
Handling mutual exclusion



No interference unless:

- write / read not protected by a common mutex (data-races), or
- last write before unlocking affects first read after lock

Handling mutual exclusion



No interference unless:

- write / read not protected by a common mutex (data-races), or
- last write before unlocking affects first read after lock

Solution:

- partition interferences wrt. mutexes
  $\mathcal{T} \times \mathcal{V} \times \mathbb{R} \rightsquigarrow \mathcal{T} \times \mathcal{P}(\textit{mutexes}) \times \mathcal{V} \times \mathbb{R}$
- extract / apply interferences at critical section boundaries

Priority-based scheduling

priority-based critical sections

high thread	low thread
$L \leftarrow$ **islocked**(m);	**lock**(m);
if $L = 0$ **then**	$Z \leftarrow Y$;
$\quad Y \leftarrow Y + 1$;	$Y \leftarrow 0$;
$\quad$**yield**	**unlock**(m)

Real-time scheduling:

- the runnable thread of highest priority always runs
- threads can yield for a non-deterministic time and preempt lower priority threads when waking up

$\Longrightarrow$ predictable scheduling, but not fixed

Static analysis:

Partition wrt. enriched scheduling state

Relational lock invariants

Work in progress

example

while true **do**
    lock(m);
    if $X > 0$ **then**
        $X \leftarrow X - 1;$
        $Y \leftarrow Y - 1;$
    unlock(m)
	while true **do**
	    lock(m);
	    if $X < 10$ **then**
	        $X \leftarrow X + 1;$
	        $Y \leftarrow Y + 1;$
	    unlock(m)

Non-relational interferences find $X \in [0, 10]$, but no bound on Y
Actually, $Y \in [0, 10]$

Relational lock invariants

Work in progress

example

while true **do**
    lock(m);
    if $X > 0$ **then**
        $X \leftarrow X - 1$;
        $Y \leftarrow Y - 1$;
    unlock(m)
	while true **do**
	    lock(m);
	    if $X < 10$ **then**
	        $X \leftarrow X + 1$;
	        $Y \leftarrow Y + 1$;
	    unlock(m)

Non-relational interferences find $X \in [0, 10]$, but no bound on Y
Actually, $Y \in [0, 10]$

Solution: infer the relational invariant $X = Y$ at lock boundaries

$\alpha_{rel}(X) \stackrel{\text{def}}{=} \{ \rho \mid \exists \rho' : \langle \rho, \rho' \rangle \in X \vee \langle \rho', \rho \rangle \in X \} \in \mathcal{P}(\mathcal{E})$
(keep only constraints that are respected by the critical section)

Lack of inter-process flow-sensitivity

Future work

a more difficult example

while true **do**
  lock(m);
  $X \leftarrow X + 1$;
  unlock(m);
  lock(m);
  $X \leftarrow X - 1$;
  unlock(m)
	while true **do**
	  lock(m);
	  $X \leftarrow X + 1$;
	  unlock(m);
	  lock(m);
	  $X \leftarrow X - 1$;
	  unlock(m)

Our analysis finds no bound on X

Actually $X \in [-2, 2]$ at all program points

Lack of inter-process flow-sensitivity

Future work

a more difficult example

while true **do**
    lock(m);
    $X \leftarrow X + 1$;
    unlock(m);
    lock(m);
    $X \leftarrow X - 1$;
    unlock(m)
	while true **do**
	    lock(m);
	    $X \leftarrow X + 1$;
	    unlock(m);
	    lock(m);
	    $X \leftarrow X - 1$;
	    unlock(m)

Our analysis finds no bound on X
Actually $X \in [-2, 2]$ at all program points

To prove this, we need to infer an
invariant on the history of interleaved executions:
at most two incrementations (resp. decrementation) can occur without a decrementation (resp. incrementation)

Applications

Specialized static analyzers

Design by refinement:

- focus on a specific family of programs and properties
- start with a fast and coarse analyzer (intervals)
- while the precision is insufficient (too many false alarms)
  - add new abstract domains (generic or application-specific)
  - refine existing domains (better transfer functions)
  - improve communication between domains (reductions)

$\Longrightarrow$ analyzer specialized for a (infinite) class of programs

- efficient and precise
- parametric (by end-users, to analyze new programs in the family)
- extensible (by developers, to analyze related families)

The Astrée static analyzer

Analyseur statique de programmes temps-réels embarqués

(static analyzer for real-time embedded software)

- developed at ENS (since 2001)
  - B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, D. Monniaux, A. Miné, X. Rival
- industrialized and made commercially available by AbsInt (since 2009)

Astrée

`www.astree.ens.fr`

AbsInt

`www.absint.com`

Astrée specialization

Specialized:

- for the analysis of run-time errors
  (arithmetic overflows, array overflows, divisions by 0, etc.)
- on embedded critical C software
  (no dynamic memory allocation, no recursivity)
- in particular on control / command software
  (reactive programs, intensive floating-point computations)
- intended for validation
  (does not miss any error and tries to minimise false alarms)

Astrée specialization

Specialized:

- for the analysis of run-time errors
  (arithmetic overflows, array overflows, divisions by 0, etc.)
- on embedded critical C software
  (no dynamic memory allocation, no recursivity)
- in particular on control / command software
  (reactive programs, intensive floating-point computations)
- intended for validation
  (does not miss any error and tries to minimise false alarms)

Approximately 40 abstract domains are used at the same time:

- numeric domains (intervals, octagons, ellipsoids, etc.)
- boolean domains
- domains expressing properties on the history of computations

Astrée applications

Airbus A340-300 (2003)

Airbus A380 (2004)

(case study for) ESA ATV (2008)

- size: from 70 000 to 860 000 lines of C
- analysis time: from 45mn to ≃40h
- alarm(s): 0 (proof of absence of run-time error)

AstréeA project

Goal: Astrée for asynchronous programs

Target programs: large embedded avionic C software

Scope: ARINC 653 real-time operating system

- several concurrent threads, one a single processor
- shared memory (implicit communications)
- synchronisation primitives (mutexes)
- real-time scheduling (priority-based)
- fixed set of threads and mutexes, fixed priorities
- no dynamic memory allocation, no recursivity

Computeall run-time errors in a sound way:

- classic C run-time errors (overflows, invalid pointers, etc.)
- data-races (report & factor in the analysis)

but not deadlocks, livelocks, nor priority inversions

Abstract interpreter
Astrée

syntax iterator

↕

trace partitioning domain

↕

memory domain

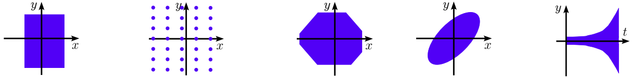
↕

pointer domain

↕

(reduced product of) numerical abstract domains

↕	↕	↕	↕	↕	⋮
intervals	congruences	octagons	filters	exponentials	...



Abstract interpreter

AstréeA

thread iterator

↕

syntax iterator

↕

trace partitioning domain

↕

scheduler partitioning domain

↕

memory domain

↕ ↑

interference domain ⋮

↕ ↓

pointer domain

↕

(reduced product of) numerical abstract domains

↕ ↕ ↕ ↕ ↕ ⋮

intervals congruences octagons filters exponentials . . .

Target system

- embedded avionic code
- 1.6 Mloc of C, 15 threads
  + 2.6 Kloc (hand-written) OS model (ARINC 653)
- many variables, large arrays, many loops
- reactive code + network code + lists, strings. pointers
- initialization phase, followed by a multithreaded phase

Analysis results

Analysis on our intel 64-bit 2.66 GHz server, 64 GB RAM

Analysis results

lines	# threads	# iters.	time	# alarms
100 K	5	4	46 mn	64
1.6 M	15	6	43 h	1 208

efficiency on par with analyses of synchronous code

- few thread reanalyses (time efficiency)
- few partitions (memory efficiency)

but still many alarms

Conclusion

Summary

A method to analyze concurrent programs:

- sound for all interleavings
- sound for weakly consistent memory semantics
- taking synchronization into account
- thread-modular
- parametrized by abstract domains
- exploits directly existing non-parallel analyzers
- efficient (on par with non-parallel analyses)
- abstraction of a semantics complete for safety (rely/guarantee)
  ($\Longrightarrow$ wide range of trade-offs between cost and precision)

Encouraging experimental results
on embedded real-time concurrent programs

Future work

Ongoing work:

- new classes of interference abstractions
  (relational and history-sensitive interferences)
- dynamic threads
  (thread creation, dynamic priorities)
- refined weakly consistent memory models (TSO)
- improve AstréeA (zero false alarm goal)
- extend to other synchronization mechanisms and OS kinds
  (towards industrialization)

Long-term challenges:

- functional, time-related, and security properties
- liveness proofs under fairness conditions