Static analysis by abstract interpretation of concurrent programs

Antoine Miné

Habilitation
École normale supérieure
Paris, France

28 May 2013
**Cause:** software error

- arithmetic overflow in unprotected data conversion from 64-bit float to 16-bit integer
- uncaught software exception $\Rightarrow$ 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

\[
i \leftarrow 2 \\
\text{n} \leftarrow \text{input} [\!-100, 100] \\
\text{while } i \leq n \text{ do} \\
\quad \text{if random()} \text{ then} \\
\quad \quad i \leftarrow i + 2
\]

**Program proof:** deductive method on a logic of programs

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

Reasoning about programs

Example

\{i=0, n=0\}
\i \leftarrow 2 \ {i=2, n=0}
n \leftarrow \text{input} \ [-100, 100] \ {i=2, -100 \leq n \leq 100]}
\textbf{while} \ {i \geq 2, i \leq \max(2, n+2), -100 \leq n \leq 100} \ i \leq n \ \textbf{do}
{\i \geq 2, i \leq n, 2 \leq n \leq 100}
\textbf{if} \ \text{random()} \ \text{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

\[\{i=0, n=0\}\]

\[i \leftarrow 2 \quad \{i=2, n=0\}\]

\[n \leftarrow \text{input } [-100, 100] \quad \{i=2, -100 \leq n \leq 100\}\]

\[\textbf{while} \quad \{i \geq 2, i \leq \max(2,n+2), -100 \leq n \leq 100\} \quad i \leq n \quad \textbf{do}\]

\[\{i \geq 2, i \leq n, 2 \leq n \leq 100\}\]

\[\textbf{if random()} \textbf{then}\]

\[i \leftarrow i + 2\]

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

how can we \textbf{infer} invariants?

(especially \textbf{loop invariants})

generally \textbf{undecidable}

\[\implies \text{use approximations}\]
**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
The program is correct \((\text{blue} \cap \text{red} = \emptyset)\)
The program is **correct** \((\text{blue} \cap \text{red} = \emptyset)\)

A polyhedral abstraction **can prove the correctness** \((\text{cyan} \cap \text{red} = \emptyset)\)
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** \((\text{green} \cap \text{red} \neq \emptyset, \text{false alarm})\)
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 \((\times 2\) transistors every 2 years)

- can exploit several computers
  (distributed computing)

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

\[\Rightarrow\] found in embedded critical applications (event-driven)
Introduction

Concurrent programs verification

Concurrent programs are hard to design and hard to verify:

- programs are highly non-deterministic
  (many possible scheduling, execution interleavings)
  $\implies$ 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** \((\Sigma, \tau, I)\)

- \(\Sigma\): set of program states
- \(\tau \subseteq \Sigma \times \Sigma\): transition relation, \(\sigma \rightarrow \sigma'\) (execution step)
- \(I \subseteq \Sigma\): set of initial states
Transition systems

**Formal model of programs** \((\Sigma, \tau, I)\)

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

**Example**

1. \(i \leftarrow 2\)
2. \(n \leftarrow \text{input} \quad [−100, 100]\)
3. while \(i \leq n\) do
   - if random() then
     - \(i \leftarrow i + 2\)
4. \(i \leftarrow i + 2\)

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

\(I = \{(1, 0, 0)\}\)
**Partial execution traces** \( \mathbb{T} \)

- set of execution traces, in \( \mathcal{P}(\Sigma^*) \)

\[ \mathbb{T} \overset{\text{def}}{=} \text{lfp } F \text{ where } \]
\[ F(\mathbb{T}) \overset{\text{def}}{=} I \cup \{ \langle \sigma_0, \ldots, \sigma_{n+1} \rangle \mid \langle \sigma_0, \ldots, \sigma_n \rangle \in \mathbb{T} \land \sigma_n \rightarrow \sigma_{n+1} \} \]

**Expressiveness:**

computing \( \mathbb{T} \) is equivalent to **exhaustive test**

\( \implies \) can answer question about program safety

**Cost:**

\( \mathbb{T} \) is often very large or unbounded

\( \implies \) well-defined mathematically but **not computable**
State semantics $\mathcal{S}$:
- set of reachable states, in $\mathcal{P}(\Sigma)$
- $\mathcal{S} \overset{\text{def}}{=} \text{lfp } G$ where $G(\mathcal{S}) \overset{\text{def}}{=} I \cup \{ \sigma \mid \exists \sigma' \in \mathcal{S} : \sigma' \rightarrow \sigma \}$

Abstraction of the trace semantics:
- $\mathcal{S} = \alpha_{\text{state}}(T)$ where
  $\alpha_{\text{state}}(T) \overset{\text{def}}{=} \{ \sigma_i \mid \exists \langle \sigma_0, \ldots, \sigma_n \rangle \in T : i \in [0, n] \}$

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

Abstract interpretation

Concrete semantics

Language syntax

\[
\text{stat} ::= \quad X \leftarrow \text{expr} \quad \text{(assignment)}
\]

\[
| \quad \text{if } \text{expr} \ni 0 \text{ then } \text{stat} \quad \text{(conditional)}
\]

\[
| \quad \text{while } \text{expr} \ni 0 \text{ do } \text{stat} \quad \text{(loop)}
\]

\[
| \quad \text{stat}; \text{stat} \quad \text{(sequence)}
\]

\[
\text{expr} ::= \quad X \mid [c_1, c_2] \mid \text{expr} \diamond \ell \text{ expr} \mid \cdots
\]

\[
X \in V \quad \text{finite set of variables}
\]

\[
c_1, c_2 \in \mathbb{R}, \diamond \in \{+, -, \times, /\}, \ni \in \{=, >, \geq, <, \leq\}
\]

Idealized language:

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

**States:** \[ \Sigma \overset{\text{def}}{=} \mathcal{L} \times \mathcal{E} \]
- control state \( \ell \in \mathcal{L} \) (syntactic location)
- memory state \( \sigma \in \mathcal{E} \overset{\text{def}}{=} \mathcal{V} \rightarrow \mathbb{R} \) (maps variables to values)

**Expression semantics:** \[ E[\text{expr}] : \mathcal{E} \rightarrow \mathcal{P}(\mathbb{R}) \]
- \[ E[\left[c_1, c_2\right]] \rho \overset{\text{def}}{=} \{ v \in \mathbb{R} | c_1 \leq v \leq c_2 \} \]
- \[ E[X] \rho \overset{\text{def}}{=} \{ \rho(X) \} \]
- \[ E[-e_1] \rho \overset{\text{def}}{=} \{ -v | v \in E[e_1] \} \]
- \[ E[e_1 \odot e_2] \rho \overset{\text{def}}{=} \{ v_1 \odot v_2 | v_i \in E[e_i] \rho, \odot \neq / \lor v_2 \neq 0 \} \]

**Command semantics:** \[ C[\text{stat}] : \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathcal{E}) \]
- \[ C[V \leftarrow e] R \overset{\text{def}}{=} \{ \rho[V \leftarrow v] | \rho \in R, v \in e[\rho] \} \]
- \[ C[e \triangleright 0] R \overset{\text{def}}{=} \{ \rho | \rho \in R, \exists v \in e[\rho] : v \triangleright 0 \} \]
State semantic as equation systems

1. $i \leftarrow 2$
2. $n \leftarrow \text{input} \, [-100, 100]$
3. while $i \leq n$ do
4. 5. if random() then
8. 6. $i \leftarrow i + 2$
9. $X_1 = \{ (0, 0) \}$
10. $X_2 = C[i\leftarrow 2 \,] \, X_1$
11. $X_3 = C[n\leftarrow [-100, 100] \,] \, X_2$
12. $X_4 = X_3 \cup X_6$
13. $X_5 = C[i\leq n \,] \, X_4$
14. $X_6 = X_5 \cup C[i\leftarrow i+2 \,] \, X_5$
15. $X_7 = C[i> n \,] \, X_4$

where:
- $\forall \ell \in L:\, X_0^\ell = \emptyset$ (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 $\Rightarrow$ most precise invariant)
- it can be solved by increasing iteration:
  $\forall \ell \in L:\, X_0^\ell = \emptyset$, $\forall i > 0:\, X_{i+1}^\ell = F_\ell(X_1^i, \ldots, X_{|L|}^i)$
  (may require transfinite iterations! $\Rightarrow$ not computable)
Abstract interpretation

Abstract numeric semantics

Numeric domains

We abstract $P(E) \cong P(R | V|)$ further concrete sets, in $P(E)$:

$$\{\langle 0, 3 \rangle, \langle 5.5, 0 \rangle, \langle 12, 7 \rangle, \ldots \}$$

(not computable)

polyhedra:

$$6X + 11Y \geq 33 \land \cdots$$

(integral cost)

intervals:

$$X \in [0, 12] \land Y \in [0, 8]$$

(linear cost)

octagons:

$$X + Y \geq 3 \land Y \geq 0 \land \cdots$$

(cubic cost)

Trade-off between cost and expressiveness / precision
Abstract interpretation

Numeric domains

We abstract $\mathcal{P}(\mathcal{E}) \simeq \mathcal{P}(\mathbb{R}^{|V|})$ 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: $6X + 11Y \geq 33 \land \cdots$ (exponential cost)
We abstract $\mathcal{P}(\mathcal{E}) \simeq \mathcal{P}(\mathbb{R}^{\vert V \vert})$ 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: $6X + 11Y \geq 33 \land \cdots$ (exponential cost)

Intervals: $X \in [0, 12] \land Y \in [0, 8]$ (linear cost)
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: \[6X + 11Y \geq 33 \land \ldots\] \text{(exponential cost)}

intervals: \[X \in [0, 12] \land Y \in [0, 8]\] \text{(linear cost)}

octagons: \[X + Y \geq 3 \land Y \geq 0 \land \ldots\] \text{(cubic cost)}

Trade-off between cost and expressiveness / precision
Abstract interpretation

Abstract numeric semantics

Static analysis

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

$\mathcal{X}_{1}^{i+1} \overset{\text{def}}{=} \{ (0, 0) \}^{\#}$
$\mathcal{X}_{2}^{i+1} \overset{\text{def}}{=} C^{\#} \langle i \leftarrow 2 \rangle \mathcal{X}_{1}^{i}$
$\mathcal{X}_{3}^{i+1} \overset{\text{def}}{=} C^{\#} \langle n \leftarrow [-100, 100] \rangle \mathcal{X}_{2}^{i}$
$\mathcal{X}_{4}^{i+1} \overset{\text{def}}{=} \mathcal{X}_{4}^{i} \triangledown (\mathcal{X}_{3}^{i} \cup \mathcal{X}_{6}^{i})$
$\mathcal{X}_{5}^{i+1} \overset{\text{def}}{=} C^{\#} \langle i \leq n \rangle \mathcal{X}_{4}^{i}$
$\mathcal{X}_{6}^{i+1} \overset{\text{def}}{=} \mathcal{X}_{5}^{i} \cup C^{\#} \langle i \leftarrow i + 2 \rangle \mathcal{X}_{5}^{i}$
$\mathcal{X}_{7}^{i+1} \overset{\text{def}}{=} C^{\#} \langle i > n \rangle \mathcal{X}_{4}^{i}$

- abstract variables $\mathcal{X}_{\ell}^{\#} \in \mathcal{E}^{\#}$ replace concrete ones $\mathcal{X}_{\ell} \in \mathcal{P}(\mathcal{E})$
- abstract operators are used: $C^{\#} \langle \cdot \rangle : \mathcal{E}^{\#} \rightarrow \mathcal{E}^{\#}$, $\cup^{\#} : \mathcal{E}^{\#} \times \mathcal{E}^{\#} \rightarrow \mathcal{E}^{\#}$
- the system is solved by iterations
  $\mathcal{X}_{\ell}^{\#0} \overset{\text{def}}{=} \emptyset^{\#}$, $\mathcal{X}_{\ell}^{\#i+1} \overset{\text{def}}{=} F_{\ell}^{\#}(\mathcal{X}_{1}^{\#i}, \ldots, \mathcal{X}_{|L|}^{\#i})$
- widening $\triangledown$ is used to force convergence in finite time
  (e.g.: put unstable bounds to $\infty$)
  $\implies$ effective, terminating, sound static analyzer
Abstract interpretation

Abstract numeric semantics

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)
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
  \[
  \text{round}(X) \leadsto X + [-\epsilon, \epsilon]X + [-\epsilon, \epsilon]
  \]

**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
Abstract interpretation

Abstract numeric semantics

Abstraction summary for sequential programs

abstract states
\[ x^\# \in \mathcal{L} \rightarrow \mathcal{E}^\# \]
states
\[ S \in \mathcal{P}(\mathcal{L} \times \mathcal{E}) \]
execution traces
\[ T \in \mathcal{P}((\mathcal{L} \times \mathcal{E})^*) \]
(abstract invariants)
\[ \alpha_{\text{val}} \]
implementable data-structures + algorithms
(mathematical non-computable)

\[ \alpha_{\text{state}} \]
Static analysis of concurrent software
Concurrent language

Language extension:

- finite, fixed set of threads $\text{stat}_t$, $t \in \mathcal{T}$
- 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 $\Sigma \overset{\text{def}}{=} (\mathcal{T} \rightarrow \mathcal{L}) \times \mathcal{E}$
  (thread-local control state in $\mathcal{T} \rightarrow \mathcal{L}$, shared memory in $\mathcal{E}$)
- labelled transitions $\sigma \xrightarrow{t} \sigma'$, $t \in \mathcal{T}$

$$\langle L[t \mapsto \ell], \rho \rangle \xrightarrow{t} \langle L[t \mapsto \ell'], \rho' \rangle \iff \langle \ell, \rho \rangle \rightarrow_{\text{stat}_t} \langle \ell', \rho' \rangle$$
(derived from the transitions of individual threads)
Trace and state semantics

**Labelled trace semantics:**
- set of interleaved execution traces, with thread labels
- \( \mathbb{T} \overset{\text{def}}{=} \text{lfp } F \) where
  \[
  F(\mathbb{T}) \overset{\text{def}}{=} I \cup \{ \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_i} \sigma_{i+1} \mid \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{i-1}} \sigma_i \in \mathbb{T} \land \sigma_i \xrightarrow{t_i} \sigma_{i+1} \}
  \]

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

**Idea:**
- forget about threads and labels
- analyze as a sequential program interleaving thread statements
Example: inferring $0 \leq x \leq y \leq 10$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>while</strong> $1$ true <strong>do</strong></td>
<td><strong>while</strong> $4$ true <strong>do</strong></td>
</tr>
<tr>
<td>$2$ if $x &lt; y$ <strong>then</strong></td>
<td>$5$ if $y &lt; 10$ <strong>then</strong></td>
</tr>
<tr>
<td>$3$ $x \leftarrow x + 1$</td>
<td>$6$ $y \leftarrow y + 1$</td>
</tr>
</tbody>
</table>

- attach variables $x_L \in \mathcal{P}(E)$ to control locations $L \in \mathcal{T} \rightarrow \mathcal{L}$
- synthesize equations $x_L = F_L(x_{(1,\ldots,1)}, \ldots, x_{(|L|,\ldots,|L|)})$
  from thread equations $x_{\ell,t} = F_{\ell,t}(x_{1,t}, \ldots, x_{|L|,t})$
Equational state semantics example

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

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>while</strong> $^1$ true <strong>do</strong></td>
<td><strong>while</strong> $^4$ true <strong>do</strong></td>
</tr>
<tr>
<td>$^2$ <strong>if</strong> $x &lt; y$ <strong>then</strong></td>
<td>$^5$ <strong>if</strong> $y &lt; 10$ <strong>then</strong></td>
</tr>
<tr>
<td>$^3$ $x \leftarrow x + 1$</td>
<td>$^6$ $y \leftarrow y + 1$</td>
</tr>
</tbody>
</table>

(Simplified) concrete equation system:

\[
\begin{align*}
\mathcal{X}_{1,4} &= I \cup C[x \leftarrow x + 1] \mathcal{X}_{3,4} \cup C[x \geq y] \mathcal{X}_{2,4} \\
& \quad \cup C[y \leftarrow y + 1] \mathcal{X}_{1,6} \cup C[y \geq 10] \mathcal{X}_{1,5} \\
\mathcal{X}_{2,4} &= \mathcal{X}_{1,4} \cup C[y \leftarrow y + 1] \mathcal{X}_{2,6} \cup C[y \geq 10] \mathcal{X}_{2,5} \\
\mathcal{X}_{3,4} &= C[x < y] \mathcal{X}_{2,4} \cup C[y \leftarrow y + 1] \mathcal{X}_{3,6} \cup C[y \geq 10] \mathcal{X}_{3,5} \\
\mathcal{X}_{1,5} &= C[x \leftarrow x + 1] \mathcal{X}_{3,5} \cup C[x \geq y] \mathcal{X}_{2,5} \cup \mathcal{X}_{1,4} \\
\mathcal{X}_{2,5} &= \mathcal{X}_{1,5} \cup \mathcal{X}_{2,4} \\
\mathcal{X}_{3,5} &= C[x < y] \mathcal{X}_{2,5} \cup \mathcal{X}_{3,4} \\
\mathcal{X}_{1,6} &= C[x \leftarrow x + 1] \mathcal{X}_{3,6} \cup C[x \geq y] \mathcal{X}_{2,6} \cup C[y < 10] \mathcal{X}_{1,5} \\
\mathcal{X}_{2,6} &= \mathcal{X}_{1,6} \cup C[y < 10] \mathcal{X}_{2,5} \\
\mathcal{X}_{3,6} &= C[x < y] \mathcal{X}_{2,6} \cup C[y < 10] \mathcal{X}_{3,5}
\end{align*}
\]
Rely/guarantee proof method

Modular proof method introduced by [Jones 1981]

<table>
<thead>
<tr>
<th>checking $t_1$</th>
<th>checking $t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>while</strong> $^1$ true do</td>
<td><strong>while</strong> $^4$ true do</td>
</tr>
<tr>
<td>$^2$ if $x &lt; y$ then</td>
<td>$^5$ if $y &lt; 10$ then</td>
</tr>
<tr>
<td>$^3$ $x \leftarrow x + 1$</td>
<td>$^6$ $y \leftarrow y + 1$</td>
</tr>
</tbody>
</table>

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

at $3 : 0 \leq x < y \leq 10$

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

For each thread, prove that local invariants hold
Rely/guarantee proof method

Modular proof method introduced by [Jones 1981]

<table>
<thead>
<tr>
<th>Checking $t_1$</th>
<th>Checking $t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>while</strong> 1 <em>true do</em></td>
<td><strong>while</strong> 4 <em>true do</em></td>
</tr>
<tr>
<td>2 <em>if</em> $x &lt; y$ <em>then</em></td>
<td>5 <em>if</em> $y &lt; 10$ <em>then</em></td>
</tr>
<tr>
<td>3 $x \leftarrow x + 1$</td>
<td>6 $y \leftarrow y + 1$</td>
</tr>
</tbody>
</table>

### Checking $t_1$

1. $x$ unchanged
2. $y$ incremented
3. $y \leq 10$

### Checking $t_2$

5. $y$ unchanged
6. $y \leq 10$

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

$\Rightarrow$ 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 $t \in \mathcal{T}$
  $$\mathcal{S}_t \overset{\text{def}}{=} \pi_t(\alpha_{\text{state}}(\mathbb{T}))$$
  where
  $$\pi_t\langle L, \rho \rangle \overset{\text{def}}{=} \langle L(t), \rho[\forall t' \neq t: pc_t' \mapsto L(t')] \rangle \in \mathcal{P}(\mathcal{L} \times \mathcal{E}_t)$$
  (keep other threads’ location in auxiliary variables)

- interferences generated by $t \in \mathcal{T}$
  $$\mathcal{A}_t \overset{\text{def}}{=} \{ \langle \sigma_i, \sigma_{i+1} \rangle | \exists \sigma_i \overset{t}{\rightarrow} \sigma_{i+1} \cdots : \in \mathbb{T} \}$$
  transitions from $\tau$ actually observed in execution traces
  (relational and flow-sensitive information)
Nested fixpoint form: for the state semantics $S$

\[
S = \text{lfp } G \text{ where }
\]
\[
G_t(S) \overset{\text{def}}{=} \text{lfp } H_t(\lambda t'. \{ \langle \sigma, \sigma' \rangle | \sigma \in S_{\tau'}, \sigma \xrightarrow{t'} \sigma' \} )
\]
\[
H_t(A)(S) \overset{\text{def}}{=} \pi_t (I \cup \{ \sigma' | \exists \pi_t(\sigma) \in S: \sigma \xrightarrow{t} \sigma' \lor \exists t' \neq t: (\sigma, \sigma') \in A_{\tau'} \} )
\]

- $H_t(A)$: execute one step, in thread $t$ or interferences $A$
- $G_t(S) \simeq \text{lfp } H_t$: analyze thread $t$ completely with fixed interferences (spawned from $S$)
- $\text{lfp } G$: 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
  \[ \alpha_{aux}(X) \overset{\text{def}}{=} \{ \langle \ell, \rho|_\varepsilon \rangle | \langle \ell, \rho \rangle \in X \} \in \mathcal{P}(\mathcal{L} \times \mathcal{E}) \]
  (allows uniform analyses of threads with unbounded instances)

**Interference abstractions:**

- flow-insensitive abstraction:
  \[ \alpha_{flow}(X) \overset{\text{def}}{=} \{ \langle \rho, \rho' \rangle | \exists L, L': \langle \langle L, \rho \rangle, \langle L', \rho' \rangle \rangle \in X \} \]
  (infer global interferences)

- input-insensitive abstraction:
  \[ \alpha_{out}(X) \overset{\text{def}}{=} \{ \rho' | \exists \rho: \langle \rho, \rho' \rangle \in X \} \in \mathcal{P}(\mathcal{E}) \]

- non-relational abstraction:
  \[ \alpha_{val}(X) \overset{\text{def}}{=} \lambda V \in \mathcal{V}. \{ \rho(V) | \rho \in X \} \in \mathcal{V} \rightarrow \mathcal{P}(\mathbb{R}) \]

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**

\[ I \overset{\text{def}}{=} \mathcal{T} \times \mathcal{V} \times \mathbb{R} \]

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

**Modified semantic of expressions and commands:**

\[
E_t[X] \langle \rho, I \rangle \overset{\text{def}}{=} \{ \rho(X) \} \cup \{ v | \exists t' \neq t: \langle t', X, v \rangle \in I \}
\]

\[
C_t[X \leftarrow e] \langle R, I \rangle \overset{\text{def}}{=} \langle \{ \rho[X \mapsto v] | \rho \in R, v \in V_\rho \} \cup \{ \langle t, X, v \rangle | \rho \in R, v \in V_\rho \} \rangle
\]

where \( V_\rho \overset{\text{def}}{=} E_t[e] \langle \rho, I \rangle \)

- analyze each thread as a sequential program with interferences \( I \subseteq I \)
- a thread analysis infers new interferences
- iterate (with widening \( \nabla \)) until stabilization
Simple interference analysis: example

**Example**

<table>
<thead>
<tr>
<th></th>
<th>$t_1$</th>
<th></th>
<th>$t_2$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>while true do</td>
<td>4</td>
<td>while true do</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>if x &lt; y then</td>
<td>5</td>
<td>if y &lt; 10 then</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>x ← x + 1</td>
<td>6</td>
<td>y ← y + 1</td>
<td></td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>t₁</th>
<th>t₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>while ¹ true do</td>
<td>while ⁴ true do</td>
</tr>
<tr>
<td>² if x &lt; y then</td>
<td>⁵ if y &lt; 10 then</td>
</tr>
<tr>
<td>³ x ← x + 1</td>
<td>⁶ y ← y + 1</td>
</tr>
</tbody>
</table>

**Interference semantics:**

iteration 2

\[ I = \{ \langle t₂, y, 1 \rangle, \ldots, \langle t₂, y, 10 \rangle \} \]

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

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

new \( I = \{ \langle t₁, x, 1 \rangle, \ldots, \langle t₁, x, 10 \rangle, \langle t₂, y, 1 \rangle, \ldots, \langle t₂, y, 10 \rangle \} \)
Simple interference analysis: example

**Interference semantics:**

iteration 3
\[ l = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \} \]

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

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

new \( l = \{ \langle t_1, x, 1 \rangle, \ldots, \langle t_1, x, 10 \rangle, \langle t_2, y, 1 \rangle, \ldots, \langle t_2, y, 10 \rangle \} \)
Simple interference analysis: example

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

abstract states
\[ x^\# \in \mathcal{L} \rightarrow \mathcal{E}^\# \]

states
\[ S \in \mathcal{P}(\mathcal{L} \times \mathcal{E}) \]

execution traces
\[ T \in \mathcal{P}((\mathcal{L} \times \mathcal{E})^*) \]

(abstract invariants)

\[ \alpha_{val} \]

\[ \alpha_{state} \]

implementable data-structures + algorithms

mathematical non-computable
Abstraction summary for concurrent programs

abstract states
\[ \mathcal{T} \to \mathcal{L} \to \mathcal{E}^\# \]

abstract interferences
\[ \mathcal{T} \to \mathcal{E}^\# \]

input-insensitive interferences
\[ \mathcal{T} \to \mathcal{P}(\mathcal{E}) \]

flow-insensitive interferences
\[ \mathcal{T} \to \mathcal{P}(\mathcal{E} \times \mathcal{E}) \]

interferences
\[ \mathcal{T} \to \mathcal{P}(((\mathcal{T} \to \mathcal{L}) \times \mathcal{E}) \times ((\mathcal{T} \to \mathcal{L}) \times \mathcal{E})) \]

interleaved execution traces
\[ T \in \mathcal{P}(((\mathcal{T} \to \mathcal{L}) \times \mathcal{E})^*) \]

static analyzer

rely/guarantee
(without aux. variables)

rely/guarantee
(with aux. variables)

concrete executions
Weak memory consistency

program written

\[ F_1 \leftarrow 1; \]
\[ \text{if } F_2 = 0 \text{ then } \]
\[ S_1 \]
\[ F_2 \leftarrow 1; \]
\[ \text{if } F_1 = 0 \text{ then } \]
\[ S_2 \]

(simplified Dekker mutual exclusion algorithm)

\( S_1 \) and \( S_2 \) cannot execute simultaneously
Weak memory consistency

<table>
<thead>
<tr>
<th>Program written</th>
<th>Program executed</th>
</tr>
</thead>
</table>
| \( F_1 \leftarrow 1; \)  
  if \( F_2 = 0 \) then  
  \( S_1 \)  
  \( F_2 \leftarrow 1; \)  
  if \( F_1 = 0 \) then  
  \( S_2 \) | \( F_1 \leftarrow 1; \)  
  if \( F_2 = 0 \) then  
  \( S_1 \)  
  \( F_2 \leftarrow 1; \)  
  if \( F_1 = 0 \) then  
  \( S_2 \) |

(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

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

(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

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
  \[ T \times V \times R \xrightarrow{} T \times P (\text{mutexes}) \times V \times R \]
- extract / apply interferences at critical section boundaries
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
  \[ T \times V \times R \leadsto T \times \mathcal{P}(\text{mutexes}) \times V \times R \]
- extract / apply interferences at critical section boundaries
Priority-based scheduling

<table>
<thead>
<tr>
<th>priority-based critical sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>high thread</td>
</tr>
<tr>
<td>$L \leftarrow \text{islocked}(m)$;</td>
</tr>
<tr>
<td>if $L = 0$ then</td>
</tr>
<tr>
<td>$Y \leftarrow Y + 1$;</td>
</tr>
<tr>
<td>yield</td>
</tr>
</tbody>
</table>

**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

$\Rightarrow$ predictable scheduling, but not fixed

**Static analysis:**

Partition wrt. **enriched** scheduling state
Relational lock invariants
Work in progress

example

```plaintext
while true do
    lock(m);
    if X > 0 then
        X ← X − 1;
        Y ← Y − 1;
    unlock(m)
```  

```plaintext
while true do
    lock(m);
    if X < 10 then
        X ← X + 1;
        Y ← 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

```plaintext
while true do
    lock(m);
    if X > 0 then
        X ← X − 1;
        Y ← Y − 1;
    else
        X ← X + 1;
        Y ← 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

$$\alpha_{rel}(X) \overset{\text{def}}{=} \{ \rho \mid \exists \rho' : \langle \rho, \rho' \rangle \in X \lor \langle \rho', \rho \rangle \in X \} \in \mathcal{P}(\mathcal{E})$$
(keep only constraints that are respected by the critical section)
Lack of inter-process flow-sensitivity

Future work

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 ← X + 1;
  unlock(m);
  lock(m);
  X ← X - 1;
  unlock(m)

while true do
  lock(m);
  X ← X + 1;
  unlock(m);
  lock(m);
  X ← 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)

⇒ 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
Applications

Astrée applications


(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

Compute all 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

<table>
<thead>
<tr>
<th>lines</th>
<th># threads</th>
<th># iters.</th>
<th>time</th>
<th># alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 K</td>
<td>5</td>
<td>4</td>
<td>46 mn</td>
<td>64</td>
</tr>
<tr>
<td>1.6 M</td>
<td>15</td>
<td>6</td>
<td>43 h</td>
<td>1 208</td>
</tr>
</tbody>
</table>

efficiency on par with analyses of synchronous code

- few thread reanalyses
- few partitions

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)
  \((\Rightarrow\) 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