Thread-Modular Abstract Interpretation of Concurrent Software

Antoine Miné

LIP6
University Pierre and Marie Curie
Paris, France

Static Analysis Symposium
11 September 2015
Saint-Malo, France
**Goal:** static analysis of concurrent programs

Discover properties of the dynamic behaviors of programs:
- directly on the source code (not a model)
- in an **automated** way (not interactive)
- in a terminating and **efficient** way
- with approximations (computability and efficiency)
- soundly (full coverage of all behaviors)
- with customizable precision control (global and local control)

We use the theory of **abstract interpretation** [Cousot and Cousot]

We developed the **AstréeA** static analyzer
(an extension of the Astrée analyzer to concurrent embedded software)
Certification in the avionics industry

Critical avionics software are subject to certification:
- more than half the development cost
- regulated by international standards (DO-178B, DO-178C)
- mostly based on massive test campaigns & intellectual reviews

Current trend:
- use of formal methods now acknowledged (DO-178C, DO-333)
  - at the binary level, to replace testing
  - at the source level, to replace intellectual reviews, and testing when using a certified compiler (CompCert)
  - to check robustness, RTE-freedom, WCET, etc.
- static analysis can be used provided it is sound

⇒ sound automatic static analysis improves cost-effectiveness!
Astrée: Analyse Statique Temps-RÉel

Features:
- checks statically for the absence of run-time errors (RTE)
- supports a large subset of C (targeting embedded software)
- specialized for synchronous reactive codes (e.g., avionics)
- fast, sound and precise (aims at 0 alarm)
- limited to sequential software (no concurrency)

Time-line:
- 2001 Astrée project starts
- 2003 0 alarm on A340 primary control software
- 2005 0 alarm on A380 primary control software
- 2009 industrialization by AbsInt

Development team: ÉNS, Paris, France
B. Blanchet, P. Cousot, R. Cousot, L. Mauborgne,
D. Monniaux, J. Feret, A. Miné, X. Rival
**Concurrent software**

**Concurrent programming:**
decompose a program into a set of **loosely interacting processes**
- exploit parallelism in computers *(multi-cores, distributed computing)*
- logical decomposition into asynchronous tasks *(servers, GUI, reactive programs)*

Use in avionics software: **Integrated Modular Avionics**
- integrate functionnalities (less CPUs)
- replace buses with shared memory communications
- limited to less critical software *(DAL C–E, less stringent certification)*
- static resource allocation *(threads, locks, memory)*

**Issues:**
- concurrent software are more difficult to design correctly
- and more difficult to validate and verify
- test is ineffective, formal methods are nonexistent
\textbf{AstréeA:} Analyse statique de programmes temps-réels asynchrones

- static analyzer for \textit{concurrent} \textit{embedded} C codes
- checks for run-time errors and data-races
- \textbf{fork} of the Astrée analyzer (around 2007)
  - reuses Astrée's iterator and abstract domains
  - builds on them a thread-modular analysis
  - adds \textit{new} abstract domains
- as Astrée, aims towards high precision by \textit{specialization}
- unlike Astrée, still \textit{many false alarms} on target code
  but already usable (industrialization in progress)
Talk overview

- From sequential to concurrent abstract interpreters
  - specialized analyzers
  - iterated sequential analysis with simple interference
- Abstract rely-guarantee
  - a complete concrete semantics
  - retrieving simple interferences by abstraction
  - novel abstractions of interferences
- Experiments
  - academic experiments
  - industrial experiments
- Conclusion and future challenges
Abstract interpreters
Classic and specialized interpreters
“Classic” abstract interpreter design

1. State the concrete semantics
2. State the class of properties of interest
3. Fix the class of properties that actually need to be inferred
4. Design an analyzer over a computable abstract semantics
“Classic” abstract interpreter design

1. State the concrete semantics
   - function from programs to a (rich) mathematical world
   - formalization of the language specification
   - ground truth, immutable
   - not computable!

2. State the class of properties of interest

3. Fix the class of properties that actually need to be inferred

4. Design an analyzer over a computable abstract semantics
“Classic” abstract interpreter design

1. State the concrete semantics

2. State the class of properties of interest
   - e.g.: variable bounds $X \in [a, b]$

3. Fix the class of properties that actually need to be inferred

4. Design an analyzer over a computable abstract semantics
“Classic” abstract interpreter design

1. State the concrete semantics

2. State the class of properties of interest

3. Fix the class of properties that actually need to be inferred
   - e.g.: linear constraints $\alpha X + \beta Y \leq \gamma$
   - generally richer than the properties of interest
     need to represent intermittent assertions, inductive loop invariants
   - may depend on the class of analyzed programs

4. Design an analyzer over a computable abstract semantics
“Classic” abstract interpreter design

1. State the concrete semantics
2. State the class of properties of interest
3. Fix the class of properties that actually need to be inferred
4. Design an analyzer over a computable abstract semantics
   - derive or invent abstract operators (e.g., interval arithmetic)
   - invent acceleration operators $\nabla$
   - abstract domain: data-structures and algorithms
     - abstract composition and $\nabla$ accumulate precision loss
     $\Rightarrow$ we will not find the most precise property in the class
Specialized abstract interpreter
Refine the abstraction

1. Start with a **simple and fast analyzer** *(intervals)* and a representative program in a class of programs of interest

2. **Refine by hand** the analyzer until 0 false alarm
   - determine which intermittent properties are missed
   - **add a new domain** *(if the property is not expressible)*
     - employ fast transfer functions, if possible
     - limit the activation scope *(variables, program part)* to keep scalability
     - connect to existing domains through partial reductions
   - refine transfer functions
   - add communications *(reductions)*
   - adjust precision parameters
     - activation scope, iteration parameters, . . .
     - *(available to end-users)*
**Specialized abstract interpreter**
Refine the **abstraction**

**Result**
- **sound** by construction
- **efficient** by parsimony
- 0 false alarm on the target program by refinement
- encourages modular design, reusable abstractions

**Rationale:**
- For each program and property, an adequate domain exists
  but its construction is generally not mechanizable
- A domain succeeds on infinitely many programs
- Any combination of domains fails on infinitely many programs

In practice, the analyzer is precise on a whole class of programs
False alarm reduction requires per-program tuning of parameters
(available to end-users, unlike domain refinement)
Specialized abstract interpreter
Reinvent the concrete semantics

We may also need to change the concrete semantics!

Reasons: the original concrete semantics

- does not support some constructions
e.g., concurrency: Astrée ⇝ AstréeA

- abstracts away platform details too much
  - arithmetic overflows: non-deterministic ⇝ modular
    (⇒ more precise analysis)
  - ill-typed dereferences: program halt ⇝ bit-level type-punning
    (⇒ more behaviors)
  necessary when analyzing non-portable programs

- is incomplete (hard limit on the precision of any abstraction)
  (e.g., simple interference semantics)
Abstract interpreters

Specialized abstract interpreter
Reinvent the concrete semantics

Even though the concrete semantics has changed, the abstract domains can be reused:

- Abstractions may still be sound
  - e.g., non-deterministic overflow $\leadsto$ modular overflow
  - Completed by new abstractions sound only in the refined semantics

- Thread semantics as sequential program semantics
  - Slightly modified with interferences
Abstract interpretation of sequential programs

Two approaches

Sequential program exemple

1. while random do
2. if $x < y$ then
3. $x \leftarrow x + 1$

Equation solving

\[ x_1 = I \]
\[ x_2 = x_1 \cup [x \leftarrow x + 1] x_3 \cup [x \geq y] x_2 \]
\[ x_3 = [x < y] x_2 \]

Interpretation by induction

\[ [\text{while random do } S] X \end{align*} \]
\[ \text{def} \]
\[ \text{ifp} \ \lambda \ Y. X \cup [S] Y \]
\[ [\text{if } x < y \text{ then } S] X \end{align*} \]
\[ \text{def} \]
\[ [S] ([x < y] X) \cup [x \geq y] X \]

• linear memory in program length
• flexible solving strategy
• flexible context sensitivity
• easy to adapt to concurrency

• linear memory in program depth
• fixed iteration strategy
• fixed context sensitivity
(follows the program structure)

for scalability on large programs, memory is a limiting factor
\[ \Rightarrow \text{Astrée(A) uses an interpreter by induction} \]
Analyzing concurrent programs
Multi-thread execution model

**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 supposed atomic)

\[
\begin{array}{c|c|c|c}
\hline
& t_1 & t_2 \\
\hline
1a & \textbf{while random do} & 1b & \textbf{while random do} \\
2a & \textbf{if } x < y \textbf{ then} & 2b & \textbf{if } y < 100 \textbf{ then} \\
3a & x \leftarrow x + 1 & 3b & y \leftarrow y + [1, 3] \\
\hline
\end{array}
\]

\(\implies\) we have the global invariant \(0 \leq x \leq y \leq 102\)
### Product-based analysis

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a) while random do</td>
<td>1b) while random do</td>
</tr>
<tr>
<td>2a) if $x &lt; y$ then</td>
<td>2b) if $y &lt; 100$ then</td>
</tr>
<tr>
<td>3a) $x \leftarrow x + 1$</td>
<td>3b) $y \leftarrow y + [1, 3]$</td>
</tr>
</tbody>
</table>

Product of thread equations, interleaving of instructions:

$X_{\ell, \ell'} \subseteq \mathbb{Z}^2$, $\ell \in \{1a, 2a, 3a\}$, $\ell' \in \{1b, 2b, 3b\}$

- $X_{1a,1b} = I$
- $X_{2a,1b} = X_{1a,1b} \cup [x \geq y]X_{2a,1b} \cup [x \leftarrow x + 1]X_{3a,1b}$
- $X_{3a,1b} = [x < y]X_{2a,1b}$
- $X_{2a,2b} = X_{1a,2b} \cup [x \geq y]X_{2a,2b} \cup [x \leftarrow x + 1]X_{3a,2b} \cup [y \geq 100]X_{2a,2b} \cup [y \leftarrow y + [1, 3]]X_{2a,3b}$
- $X_{3a,2b} = [x < y]X_{2a,2b} \cup X_{3a,1b} \cup [y \geq 100]X_{3a,2b} \cup [y \leftarrow y + [1, 3]]X_{3a,3b}$
- $X_{2a,3b} = X_{1a,3b} \cup [x \geq y]X_{2a,3b} \cup [x \leftarrow x + 1]X_{3a,3b} \cup [y < 100]X_{2a,2b}$
- $X_{3a,3b} = [x < y]X_{2a,3b} \cup [y < 100]X_{3a,2b}$

**limitations:** large number of variables, large equations
no induction on the syntax possible

$\implies$ **impractical**
Abstract interpreters
Analyzing concurrent programs

Separate sequential analyses

$t_1$

1a while random do
2a if $x < y$ then
3a $x \leftarrow x + 1$

$t_2$

1b while random do
2b if $y < 100$ then
3b $y \leftarrow y + [1, 3]$

Our wish: analyze each thread separately
- **scale** linearly in program size
- **reuse** the interpreter by induction on each thread
- **unsound** if we don’t take thread interferences into account

Poor’s man concurrent analysis:
- consider each shared variable as **volatile input**
- rely on the **user** to list shared variables
- rely on the **user** to provide ranges on shared variables

$\Longrightarrow$ huge human cost, drop in analysis confidence
Inferring simple interferences

**Principle:** [Miné 2010, Carré & Hymans 2009]

- analyze each thread in isolation
  but also gather interferences
    (abstraction of) the values stored into each variable by each thread
- re-analyze the threads taking interferences into account
  (variable read returns the last value written, or an interference)
  gather new sets of interferences
- iterate until stabilization
  \( \Rightarrow \) one more level of fixpoint iteration

---

\[ t_1 \]

1\textsubscript{a} \hspace{1cm} \textbf{while} random \hspace{0.5cm} \textbf{do}

2\textsubscript{a} \hspace{1cm} \textbf{if} \hspace{0.5cm} x < y \hspace{0.5cm} \textbf{then}

3\textsubscript{a} \hspace{1cm} x \leftarrow x + 1

\[ t_2 \]

1\textsubscript{b} \hspace{1cm} \textbf{while} random \hspace{0.5cm} \textbf{do}

2\textsubscript{b} \hspace{1cm} \textbf{if} \hspace{0.5cm} y < 100 \hspace{0.5cm} \textbf{then}

3\textsubscript{b} \hspace{1cm} y \leftarrow y + [1, 3]
Inferring simple interferences

**Analysis of** $t_1$ **in isolation**

(1a): $x = y = 0$  
$\mathcal{X}_{1a} = I$

(2a): $x = y = 0$  
$\mathcal{X}_{2a} = \mathcal{X}_{1a} \cup \llbracket x \leftarrow x + 1 \rrbracket \mathcal{X}_{3a} \cup \llbracket x \geq y \rrbracket \mathcal{X}_{2a}$

(3a): $\bot$  
$\mathcal{X}_{3a} = \llbracket x < y \rrbracket \mathcal{X}_{2a}$

**Analysis of** $t_2$ **in isolation**

(1b): $x = y = 0$  
$\mathcal{X}_{1b} = I$

(2b): $x = y = 0$  
$\mathcal{X}_{2b} = \mathcal{X}_{1b} \cup \llbracket x \leftarrow x + 1 \rrbracket \mathcal{X}_{3b} \cup \llbracket x \geq y \rrbracket \mathcal{X}_{2b}$

(3b): $y \leftarrow y + [1, 3]$  
$\mathcal{X}_{3b} = \llbracket x < y \rrbracket \mathcal{X}_{2b}$
## Inferring simple interferences

### t₁

1a. `while random do`
2a. `if x < y then`
3a. `x ← x + 1`

### t₂

1b. `while random do`
2b. `if y < 100 then`
3b. `y ← y + [1, 3]`

### Analysis of t₂ in isolation

1b: \(x = y = 0\) \(\mathcal{X}_{1b} = I\)

2b: \(x = 0, y \in [0, 102]\) \(\mathcal{X}_{2b} = \mathcal{X}_{1b} \cup [y \leftarrow y + [1, 3]] \mathcal{X}_{3b} \cup [y \geq 100] \mathcal{X}_{2b}\)

3b: \(x = 0, y \in [0, 99]\) \(\mathcal{X}_{3b} = [y \leftarrow 100] \mathcal{X}_{2b}\)

output interferences: \(y \leftarrow [1, 102]\)
Inferring simple interferences

Re-analysis of $t_1$ with interferences from $t_2$

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

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

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

subsequent re-analyses are identical (fixpoint reached)
Inferring simple interferences

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!
### Rely–guarantee reasoning

**Abstract interpreters**

**Analyzing concurrent programs**

**Rely–guarantee reasoning**

<table>
<thead>
<tr>
<th>checking $t_1$</th>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1^a$</td>
<td>while random do</td>
<td></td>
</tr>
<tr>
<td>$2^a$</td>
<td>if $x &lt; y$ then</td>
<td></td>
</tr>
<tr>
<td>$3^a$</td>
<td>$x \leftarrow x + 1$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>checking $t_2$</th>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1^b$</td>
<td>while random do</td>
<td></td>
</tr>
<tr>
<td>$2^b$</td>
<td>if $y &lt; 100$ then</td>
<td></td>
</tr>
<tr>
<td>$3^b$</td>
<td>$y \leftarrow y + [1, 3]$</td>
<td></td>
</tr>
</tbody>
</table>

**Rely–guarantee:** proof method introduced by Jones in 1981
- generalized Hoare logics (by structural induction $\Rightarrow$ thread-modular)
Abstract interpreters
Analyzing concurrent programs

Rely–guarantee reasoning

checking \( t_1 \)

\[
\begin{array}{c|c|c}
& t_1 & t_2 \\
\hline
1a & \text{while random do} & \text{while random do} \\
2a & \text{if } x < y \text{ then} & \text{if } y < 100 \text{ then} \\
3a & x \leftarrow x + 1 & y \leftarrow y + [1, 3] \\
\end{array}
\]

(1a) : \( x = y = 0 \)
(2a) : \( x, y \in [0, 102], x \leq y \)
(3a) : \( x \in [0, 101], y \in [1, 102], x < y \)

checking \( t_2 \)

\[
\begin{array}{c|c|c}
& t_1 & t_2 \\
\hline
1b & \text{while random do} & \text{while random do} \\
2b & \text{if } y < 100 \text{ then} & \text{if } y < 100 \text{ then} \\
3b & y \leftarrow y + [1, 3] & y \leftarrow y + [1, 3] \\
\end{array}
\]

(1b) : \( x = y = 0 \)
(2b) : \( x, y \in [0, 102], x \leq y \)
(3b) : \( x, y \in [0, 99], x \leq y \)

Rely–guarantee: proof method introduced by Jones in 1981

- generalized Hoare logics (by structural induction \( \Rightarrow \) thread-modular)
- requires thread-local invariant assertions
Abstract interpreters  Analyzing concurrent programs

Rely–guarantee reasoning

checking \( t_1 \)

\[
\begin{array}{c|c}
 t_1 & t_2 \\
\hline
1a & \text{while random do} \\
2a & \text{if } x < y \text{ then} \\
3a & x \leftarrow x + 1 \\
\end{array}
\]

\( x \) unchanged

\( y \) incremented

\( 0 \leq y \leq 102 \)

\( 0 \leq x \leq y \)

\( x \leq y \)

\( x < y \)

\( x = y = 0 \)

\( x, y \in [0, 102], x \leq y \)

\( x \in [0, 101], y \in [1, 102], x < y \)

Rely–guarantee: proof method introduced by Jones in 1981

- generalized Hoare logics (by structural induction ⇒ thread-modular)
- requires thread-local invariant assertions
- requires guarantees on transitions generated by other threads
Rely–guarantee reasoning

Rely–guarantee: proof method introduced by Jones in 1981

- generalized Hoare logics (by structural induction ⇒ thread-modular)
- requires thread-local invariant assertions
- requires guarantees on transitions generated by other threads
- checks each thread against an abstraction of the other threads
- allows proving that $x \leq y$ holds!
Rely–guarantee reasoning

**checking** $t_1$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a) while random do</td>
<td>$x$ unchanged</td>
</tr>
<tr>
<td>2a) if $x &lt; y$ then</td>
<td>$y$ incremented</td>
</tr>
<tr>
<td>3a) $x \leftarrow x + 1$</td>
<td>$0 \leq y \leq 102$</td>
</tr>
</tbody>
</table>

(1a) : $x = y = 0$

(2a) : $x, y \in [0, 102], x \leq y$

(3a) : $x \in [0, 101], y \in [1, 102], x < y$

**checking** $t_2$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$ unchanged</td>
<td>1b) while random do</td>
</tr>
<tr>
<td>0 $\leq x \leq y$</td>
<td>2b) if $y &lt; 100$ then</td>
</tr>
<tr>
<td></td>
<td>3b) $y \leftarrow y + [1, 3]$</td>
</tr>
</tbody>
</table>

(1b) : $x = y = 0$

(2b) : $x, y \in [0, 102], x \leq y$

(3b) : $x, y \in [0, 99], x \leq y$

**Rely–guarantee:** proof method introduced by Jones in 1981

- generalized Hoare logics (by structural induction $\Rightarrow$ thread-modular)
- requires thread-local invariant assertions
- requires guarantees on transitions generated by other threads
- checks each thread against an abstraction of the other threads
- allows proving that $x \leq y$ holds!
Rely–guarantee in abstract interpretation form
Thread-modular concrete fixpoint semantics
Non-modular concrete semantics

Concrete trace semantics: \( \mathcal{F} \)

- threads: \( T \overset{\text{def}}{=} \{a, b, \ldots\} \)
- states: \( \Sigma \overset{\text{def}}{=} \mathcal{C} \times \mathcal{M} = \{\bullet, \bullet, \ldots\} \)
  - control state: \( \mathcal{C} \overset{\text{def}}{=} T \rightarrow \mathcal{L} \) (maps threads to locations)
  - memory state: \( \mathcal{M} \overset{\text{def}}{=} \mathcal{V} \rightarrow \mathcal{V} \) (maps variables to values)
- transition relation: \( \tau \subseteq \Sigma \times T \times \Sigma: \sigma \overset{a}{\rightarrow}_\tau \sigma' \)

Partial finite trace semantics in fixpoint form:

\[
\mathcal{F} \overset{\text{def}}{=} \text{lfp } F \text{ where } \\
F \overset{\text{def}}{=} \lambda X. I \cup \{ \sigma_0 \overset{a_1}{\rightarrow} \cdots \sigma_i \overset{a_{i+1}}{\rightarrow} \sigma_{i+1} \mid \sigma_0 \overset{a_1}{\rightarrow} \cdots \sigma_i \in X \land \sigma_i \overset{a_{i+1}}{\rightarrow}_\tau \sigma_{i+1} \}
\]
Reachable states:  $\mathcal{R}$  (concrete semantics of interest)

Extract the states reached during execution

$\mathcal{R} = \alpha^{\text{reach}}(\mathcal{F}) \subseteq \Sigma$  where

$\alpha^{\text{reach}}(T) \overset{\text{def}}{=} \{ \sigma \mid \exists \sigma_0 \xrightarrow{a_1} \cdots \sigma_n \in T : \exists i \leq n : \sigma = \sigma_i \}$
Rely–guarantee in abstract interpretation form

Thread-modular concrete fixpoint semantics

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

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

\[
\mathcal{RL}(t) \overset{\text{def}}{=} \pi_t(\mathcal{R}) \subseteq \mathcal{L} \times (\mathcal{V} \cup \{ pc_{t'} \mid t \neq t' \in \mathcal{T} \}) \rightarrow \mathcal{V}
\]

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

Interferences generated by $t$: $\mathcal{I}(t)$ (≈ guarantees on transitions)

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

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

\[
\mathcal{I}(t) \overset{\text{def}}{=} \alpha_{itf}(\mathcal{F})(t)
\]

where $\alpha_{itf}(X)(t) \overset{\text{def}}{=} \{ \langle \sigma_i, \sigma_{i+1} \rangle | \exists \sigma_0 \xrightarrow{a_1} \sigma_1 \cdots \xrightarrow{a_n} \sigma_n \in X : a_{i+1} = t \}$
**Thread-modular concrete semantics**

**Principle:** express $R_\ell(t)$ and $I(t)$ directly, without computing $F$

**States:** $R_\ell$

Interleave:

- transitions from the current thread $t$
- transitions from interferences $I$ by other threads

$$R_\ell(t) = \text{lfp } R_t(I)$$

where

$$R_t(Y)(X) \overset{\text{def}}{=} \pi_t(I) \cup \{ \pi_t(\sigma') | \exists \pi_t(\sigma) \in X : \sigma \overset{t}{\rightarrow}_r \sigma' \} \cup \{ \pi_t(\sigma') | \exists \pi_t(\sigma) \in X : \exists t' \neq t : \langle \sigma, \sigma' \rangle \in Y(t') \}$$

$\implies$ similar to reachability for a sequential program, up to $I$
Thread-modular concrete semantics

**Principle:** express $R\ell(t)$ and $I(t)$ directly, without computing $F$

**Interferences:** $I$

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

$I(t) = B(R\ell)(t)$, where

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

**Principle:** express $R_\ell(t)$ and $I(t)$ directly, without computing $F$

Recursive definition:

- $R_\ell(t) = \text{lfp } R_t(I)$
- $I(t) = B(R_\ell)(t)$

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

$$R_\ell = \text{lfp } \lambda Z. \lambda t. \text{lfp } R_t(B(Z))$$

$\Rightarrow$ iterate analyses with interference

**Completeness:** $\forall t : R_\ell(t) \simeq R \quad (\pi_t \text{ is bijective thanks to auxiliary variables})$
Thread-modular abstractions
Flow-insensitive abstraction:
- reduce as much control information as possible
- but keep flow-sensitivity on each thread’s control location

State abstraction: remove auxiliary variables
\[ \alpha^f_R(X) \overset{\text{def}}{=} \{ \langle \ell, \rho|_Y \rangle \mid \langle \ell, \rho \rangle \in X \} \]

Interference abstraction: remove all control information
\[ \alpha^f_I(Y) \overset{\text{def}}{=} \{ \langle \rho, \rho' \rangle \mid \exists L, L' \in C : \langle \langle L, \rho \rangle, \langle L', \rho' \rangle \rangle \in Y \} \]

Note: we lose completeness
we cannot prove that \( x \) is bounded in \( x \leftarrow x + 1 \parallel x \leftarrow x + 1 \)
Retrieving the simple interference-based analysis

**Cartesian abstraction:** on interferences

- forget the relations between variables
- forget the relations between values before and after transitions (input-output relationship)
- only remember which variables are modified and their new value:

\[
\alpha^C(Y) \overset{\text{def}}{=} \lambda V. \{ x \in V | \exists \langle \rho, \rho' \rangle \in Y : \rho(V) \neq x \land \rho'(V) = x \}
\]

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

\[\implies\] we get back our simple interference analysis!

Finally, use a numeric abstract domain \( \alpha : \mathcal{P}(V \to V) \to \mathcal{D}^\# \)
(for interferences, \( V \to \mathcal{P}(V) \) is abstracted as \( V \to \mathcal{D}^\# \))
From traces to thread-modular analyses

abstract states
$$(T \times L) \rightarrow D^\#$$

abstract interferences
$$T \rightarrow D^\#$$

static analyzer

non-relational interferences
$$T \rightarrow \mathcal{P}(M)$$

rely-guarantee
(without aux. variables)

flow-insensitive interferences
$$T \rightarrow \mathcal{P}(M \times M)$$

rely-guarantee
(with aux. variables)

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

interleaved execution trace prefixes
$$F \in \mathcal{P}(\Sigma^*)$$

projected states
$$(T \times L) \rightarrow \mathcal{P}(M)$$

projected states
$$(T \times L) \rightarrow \mathcal{P}(\Sigma_t)$$

$R^\ell : \prod_{t \in T} \{t\} \rightarrow \mathcal{P}(\Sigma_t)$$

$\alpha^f_R$
Compare with sequential analyses

abstract states
\[ L \rightarrow D^# \]

states
\[ R \in \mathcal{P}(\Sigma) \]

reachability
\[ \alpha^{reach} \]

execution trace prefixes
\[ F \in \mathcal{P}(\Sigma^*) \]

static analyzer

reachability

test
Beyond simple interferences

- **Academic experiment** (internship of Raphaël Monat, 2015)
  - fully-relational flow-insensitive interferences
  - academic prototype, no concern for scalability

- **Industry-targeted experiments** (AstréeA, 2011–)
  - AstréeA was initially based on simple interferences (flow-insensitive, non-relational)
  - now specialize AstréeA with partially relational interferences
  - support for locks and priorities
  - reduce false alarms while keeping scalability
Fully relational interferences
Full relational interference abstraction:

- $\alpha^f_I(Y) \overset{\text{def}}{=} \{ \langle \rho, \rho' \rangle \mid \exists L, L' \in \mathcal{C} : \langle \langle L, \rho \rangle, \langle L', \rho' \rangle \rangle \in Y \}$
- $\alpha^f_I(Y) \in \mathcal{M} \times \mathcal{M}$: relation between states

$\implies$ can be abstracted in a numeric abstract domain over $\mathcal{V}^2$ (e.g., polyhedra)

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

prototype “batman” [Monat 2015] in OCaml
supporting a small imperative language

abstractions based on Apron (polyhedra and octagons)

interference operations simulated with state operations on $V^2$

able to infer $x \leq y$

experimental comparison with ConcurInterproc
(non thread modular, also able to infer $x \leq y$)
Rely–guarantee in abstract interpretation form

Scalability in threads

- \( n \) copies of each thread \((\text{with varying value for c})\)
- fixed number of variables

\(\Rightarrow\) much better scalability than non-modular methods!
Scalability in variables

- $n$ copies of each thread
- $n$ copies of each variable

$\Rightarrow$ scalability issues, packing techniques needed (expected)
Partilly relational interferences in AstréeA
Mutual exclusion locks

Mutexes:
- ensure mutual exclusion
  at each time, each mutex can be locked by a single thread
- enforce memory consistency and atomicity

⇒ we need to discard spurious interferences, to improve the precision

We assume a fixed, finite number of mutexes
Mutual exclusion locks

Data-race interferences:
- across read / write not protected by a mutex
Mutual exclusion locks

Data-race interferences:
- across read / write not protected by a mutex

Well-synchronized interferences:
- last write before an unlock in $t_1$
  influence reads between lock and first write in $t_2$

We partition interferences by the set of mutexes held.
Example

Assuming we have several \((N)\) producers and consumers:

- no data-race interference
- well-synchronized interferences:
  - \textit{consumer}: \(x \leftarrow [0, 99]\)
  - \textit{producer}: \(x \leftarrow [1, 100]\)

\[\implies\text{we get that } x \in [0, 100]\]

(proof of the absence of data-race)
## Locks and priorities

### Priority-based critical sections

<table>
<thead>
<tr>
<th>high thread</th>
<th>low thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \leftarrow \text{islocked}(m);$</td>
<td>$\text{lock}(m);$</td>
</tr>
<tr>
<td><strong>if</strong> $L = 0$ <strong>then</strong></td>
<td>**Z \leftarrow Y;$</td>
</tr>
<tr>
<td>$Y \leftarrow Y + 1;$</td>
<td>**Y \leftarrow 0;$</td>
</tr>
<tr>
<td>yeild</td>
<td>unlock(m)</td>
</tr>
</tbody>
</table>

### Real-time scheduling

- only the **highest priority** unblocked thread can run
- **lock** and **yeild** may block
- **yeilding** threads *wake up non-deterministically* (preempting lower-priority threads)
- **explicit** synchronisation enforces memory consistency
Rely–guarantee in abstract interpretation form

Partilly relational interferences in AstréeA

Locks and priorities

<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>\textbf{if} $L = 0$ \textbf{then}</td>
</tr>
<tr>
<td>\textbf{Y} $\leftarrow$ \textbf{Y} + 1;</td>
</tr>
<tr>
<td>\textbf{yield}</td>
</tr>
</tbody>
</table>

Partition interferences and environments wrt. scheduling state

- partition wrt. mutexes tested with \textbf{islocked}
- $X \leftarrow \text{islocked}(m)$ creates two partitions
  - $P_0$ where $X = 0$ and $m$ is free
  - $P_1$ where $X = 1$ and $m$ is locked
- $P_0$ handled as if $m$ where locked
- blocking primitives merge $P_0$ and $P_1$ ($\text{lock}$, \textbf{yield})
Weakly relational interferences

Clock thread

```plaintext
while Clock < 10^6 do
    Clock ← Clock + 1;
    ...
done
```

Accumulator thread

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

- *Clock* is a global, increasing clock
- *x* accumulates periods of time
- no overflow on *Clock − Prec* nor *x ← x + delta*

To prove this we need:

- relational abstractions of interferences
  (keep input-output relationships)
- hypotheses on memory consistency
  (e.g., partial store ordering)
Monotonicity abstraction

**Abstraction:**
map variables to ↑ monotonic or ⊤ don’t know

\[ \alpha_{\mathcal{I}}^{\text{mon}}(Y) \overset{\text{def}}{=} \lambda V. \text{if } \forall \langle \rho, \rho' \rangle \in Y : \rho(V) \leq \rho'(V) \text{ then } \uparrow \text{ else } \top \]

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

**Inference and use**

- **gather:**
  \[ \mathcal{I}_{\text{mon}}(t)(V) = \uparrow \iff \text{all assignments to } V \text{ in } t \text{ have the form } V \leftarrow V + e, \text{ with } e \geq 0 \]

- **use:** combined with non-relational interferences
  
  if \( \forall t : \mathcal{I}_{\text{mon}}(t)(V) = \uparrow \)
  then any test with non-relational interference \[ X \leq (V | [a, b]) \] can be strengthened into \[ X \leq V \]
Relational invariant interferences

**Abstraction:** keep relations maintained by interferences

- remove control state in interferences
- keep mutex state $M$
- forget input-output relationships
- keep relationships between variables

$$\alpha^\text{inv}_I(Y) \buildrel \text{def} \over = \{ \langle M, \rho \rangle \mid \exists \rho' : \langle \langle M, \rho \rangle, \langle M, \rho' \rangle \rangle \in Y \lor \langle \langle M, \rho' \rangle, \langle M, \rho \rangle \rangle \in Y \}$$

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

**Lock invariant:**

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

- 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
**Weakly relational interference example**

| analyzing \( t_1 \) | \( \begin{array}{c|c}
\text{\( t_1 \)} & \text{\( t_2 \)} \\
\hline
\text{while random do} & \text{x unchanged} \\
\text{lock(\( m \));} & \text{y incremented} \\
\text{if \( x < y \) then} & \text{0 \( \leq \) y \( \leq \) 102} \\
\text{x \( \leftarrow \) x + 1;} & \\
\text{unlock(\( m \))} & \\
\end{array} \) | \\
| \( t_1 \) | \( t_2 \) | \\
| \text{y unchanged} | \text{while random do} | \\
| \text{0 \( \leq \) x, x \( \leq \) y} | \text{lock(\( m \));} | \\
| | \text{if \( y < 100 \) then} | \\
| | \text{y \( \leftarrow \) y + [1, 3];} | \\
| | \text{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** \((y \text{ monotonic})\)

we can prove automatically that \( x \leq y \) holds
Subsequence interference

\[ t_1 : \text{clock in } H \]
\[ \text{while random do} \]
\[ \text{if } H < 10,000 \text{ then} \]
\[ H \leftarrow H + 1 \]
\[ t_2 : \text{sample } H \text{ into } C \]
\[ \text{while random do} \]
\[ C \leftarrow H \]
\[ t_3 : \text{accumulate elapsed time in } T \]
\[ \text{while random do} \]
\[ \text{if random then } T \leftarrow 0 \]
\[ \text{else } T \leftarrow T + (C - L) \]
\[ L \leftarrow C \]

**Problem:** we wish to prove that \( T \leq L \leq C \leq H \)

it is sufficient to prove the monotony of \( H, C, \) and \( L \)

but **monotony is not transitive**

\( X \) is only assigned monotonic variables \( \not\implies X \) is monotonic

\( \implies \) we infer an **additional property** implying monotony

**Abstraction:** subsequence

\[ I^\sharp_{\text{sub}}(t)(V) = \{ W \in V \mid V \text{'s values are a subsequence of } W \text{'s values} \} \]

\[ \alpha_{\mathcal{R}}^{\text{sub}}(X)(V) \overset{\text{def}}{=} \{ W \mid \forall \langle \ell_0, \rho_0 \rangle, \ldots, \langle \ell_n, \rho_n \rangle \in X : \exists i_0, \ldots, i_n : \]
\[ \forall k : i_k \leq k \land i_k \leq i_{k+1} \land \forall j : \rho_j(V) = \rho_{i_j}(W) \} \]

based on a **trace version** of the modular semantics
AstréeA
Sources for Astrée(A)’s concrete semantics

Concrete semantics: defined through
- C99 norm (portable programs)
- IEEE 754-1985 norm (floating-point arithmetic)
- architecture parameters (sizeof, endianess, struct, etc.)
- compiler and linker parameters (initialization, etc.)

Properties of interest: absence of run-time error
- no integer nor float numeric overflow
- no invalid arithmetic operation (/0, << 33)
- no invalid memory access (arrays [], pointers *)
- respect the constraints put by the programmer (assert)

i.e., reachability of bad memory states
Astrée(A)’s targets

**Analyzed programs:** embedded critical C codes
- no dynamic memory allocation
- no recursivity

**Astrée:**
- no concurrency
- tuned for *synchronous control/command* software
  (numeric & boolean; no string, list, etc.)
  but sound on all accepted programs

**AstréeA:**
- supports shared-memory *concurrency* (statically allocated threads)
- supports *operating systems* (through externally provided stub models)
- on-going support for data-structures
  (strings, arrays; static allocation but dynamic usage)
A few abstract domains used in Astrée(A)

- **Octagons**
  \[ \pm X \pm Y \leq c \]
  [Miné 2006]

- **Congruences**
  \[ X \equiv a [b] \]
  [Granger 1989]

- **Boolean decision trees**

- **Ellipsoids**
  [Feret 2005]

- **Digital filters**

- **Exponentials**
  \[ X \leq (1 + \alpha)^{\beta t} \]
  [Feret 2005]

- **Trace partitions**
  [Mauborgne Rival 2005]

**Relational** domains are often required to find inductive invariants for scalability, they are limited to small variable packs, selected by syntactic heuristics.
Astrée’s abstract interpreter layers

↓
syntax iterator

↓
trace partitioning domain

↕
memory domain

↕
pointer domain

↕
(reduced product of) numerical abstract domains

↕ ↕ ↕ ↕
intervals octagons decision trees filters ...
Astrée’s abstract interpreter layers

↓
syntax iterator
↓
trace partitioning domain
↕
memory domain
↕
pointer domain
↕
(reduced product of) numerical abstract domains

↓     ↓     ↓     ↓     ↓     ↓
intervals octagons decision trees filters ...
Astrée’s abstract interpreter layers

\[
\text{for (i=0;...) } a[i] = *p; \\
\]

\[
\text{a[i] = *p} \\
\]

\[
\text{a[0] = *p, a[1] = *p, ...} \\
\]

\[
\text{a@0 = x, a@4 = x} \\
\]

\[
\text{a@0 = x, a@4 = x} \\
\]

\[
\text{(reduced product of) numerical abstract domains} \\
\]

\[
\text{intervals octagons decision trees filters ...} \\
\]
AstréeA’s abstract interpreter layers

thread iterator
↓
syntax iterator
↓
trace partitioning domain
↓
lock partitioning domain
↕
memory domain
↕
interference domain
↓
pointer domain
↕
(reduced product of) numerical abstract domains

↓ ↓ ↓ ↓
intervals octagons decision trees filters ...
Main case study

Specialization process:

- choose one representative target industrial application
- refine the domains until zero (or few) alarms
- extend to other targets

performed in an academic settings
(requires modifying the analyzer)

Core target application:

- embedded avionic code (DAL C)
- 2.1 Mloc (2 Mloc generated)
- 15 threads, shared memory, locks
- preemptive real-time scheduling on a single processor
- reactive code + network code + lists, strings, pointers
- many variables, large arrays, many loops, shallow call graph
- no dynamic memory allocation, no recursivity
Concrete execution context:

- **Target application, in C**
- Other applications
- ARINC 653 operating system, in C+asm
- Hardware

The target application:

- runs concurrently with other applications (memory separation)
- interacts dynamically with an ARINC 653 operating system (thread control operations, mutex lock and unlock, communication services)
- interacts with other applications through the OS
- creates system objects only during an initialization phase (the set of objects is inferred by the analysis)
Abstract analysis context:

- Target application, in C
- ARINC 653 model, in C + built-ins

The target application is enriched with a hand-written model of the OS

- 5.2 Kloc of C + low-level AstréeA built-ins
- stub and simulate all OS system calls
- manage (fat) OS objects, mapped to (thin) AstréeA objects
  (e.g., AstréeA’s locks are simple integers, ARINC 653’s locks have a string name)

⇒ analyze stand-alone “C” programs, with no undefined symbol
Example stub

```c
void WAIT_SEMAPHORE(
    SEMAPHORE_ID_TYPE SEMAPHORE_ID, SYSTEM_TIME_TYPE TIMEOUT,
    RETURN_CODE_TYPE * RETURN_CODE)
{
    *RETURN_CODE = NO_ERROR;
    if (SEMAPHORE_ID < 0 || SEMAPHORE_ID >= NB_SEMAPHORE) {
        __ASTREE_error("invalid semaphore");
        *RETURN_CODE = INVALID_PARAM;
    }
    else if (TIMEOUT > 0) {
        if (TIMEOUT == INFINITE_SYSTEM_TIME_VALUE || __ASTREE_rand())
            __ASTREE_lock_mutex(SEMAPHORE_ID);
    }
    else {
        __ASTREE_yield();
        *RETURN_CODE = TIMED_OUT;
    }
}
else {
    if (__ASTREE_rand()) *RETURN_CODE = NOT_AVAILABLE;
    else __ASTREE_lock_mutex(SEMAPHORE_ID);
}
```
Results

**Precision:** achieved by specialization
- 2010: 12,257 alarms
- 2015: 1,195 alarms (60% on hand-written code)
99.94% selectivity (% of lines without alarm)

**Efficiency:**
- on an intel i7 2.90 GHz workstation (1 core used)
- computation time: 24h
- analysis iterations: 6 (no widening needed on interferences)
- 27 GB RAM

Achieved through:
- well-synchronized interferences with lock partitioning
- relational interference domains
- additional state abstract domains
  (offset domains, bit-level float manipulation, memory domains)
- limiting the scope of relational domains (variable packing)
Industrial case studies

Additional case studies performed by industrials
study headed by David Delmas (Airbus)
on DAL C & DAL E avionics software

- enrich ARINC stubs with newly used functions
- design a full set of POSIX threads stubs
- analysis precision tuning through end-user directives
- no modification of the analyzer

<table>
<thead>
<tr>
<th>size</th>
<th>OS</th>
<th>stub</th>
<th>selectivity</th>
<th>time</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.9 M</td>
<td>ARINC</td>
<td>2.4 K</td>
<td>99.56%</td>
<td>154 h</td>
<td>18 GB</td>
</tr>
<tr>
<td>2.2 M</td>
<td>POSIX</td>
<td>2.3 K</td>
<td>99.52%</td>
<td>160 h</td>
<td>23 GB</td>
</tr>
<tr>
<td>31.8 K</td>
<td>POSIX</td>
<td>2.2 K</td>
<td>99.28%</td>
<td>50 mn</td>
<td>0.6 GB</td>
</tr>
<tr>
<td>33.1 K</td>
<td>POSIX</td>
<td>1.2 K</td>
<td>97.18%</td>
<td>35 h</td>
<td>2.5 GB</td>
</tr>
</tbody>
</table>

selectivity only slightly worse than for the main case study
⇒ towards a cost-effective industrial use of AstréeA
Conclusion
Conclusion

Summary

We proposed a static analysis framework for concurrent programs:

- **sound** for all interleavings
- **thread-modular**
  scalable, able to reuse existing analyzers
- **parameterized** by abstract domains
  able to reuse existing domains
- **constructed by** abstraction of a complete method
  enable refinement to arbitrary precision
- **generalized** previous simple interference analysis
- **defined** novel **relational interference domains**
- **presented** encouraging experimental results
Future challenges: towards zero alarm

**Precision target** to be usable in avionics certification:

- 99.80% selectivity on hand-written code  
  (currently: 95.97% to 99.2%)
- 99.99% selectivity on automatically generated code  
  (currently: 99.78% to 99.98%)

**Planned improvements:**

- specialized relational interference domains
- memory domains  
  (segmented array domain, specialization to strings)
- automate precision-control heuristics

On-going industrialization towards AbsInt:  
merging with commercial Astrée
Future challenges: weak memory

The interleaving semantics (sequential consistency) is not realistic. Actual languages and CPU obey relaxed memory models due to:

- CPU-level optimizations (memory buffers, instruction reordering)
- Compiler-level optimizations (allowed by language specifications)

⇒ an analysis sound only for sequential consistency may not be sound for the actual memory model!

(Example:  
\[
y \leftarrow 1; \text{if } x = 0 \text{ then } \cdots \text{|| } x \leftarrow 1; \text{if } y = 0 \text{ then } \cdots
\]

Result: The flow-insensitive non-relational analysis is sound wrt. a large set of weak memory models

Rationale: flow-insensitive non-relational interferences are insensitive to the reordering of reads and writes [Miné 2011, Alglave et al. 2011]

Challenges:

- flow-sensitivity and relationality despite weak memory
- Specialization to realistic memory models
Future challenges: inter-thread flow-sensitivity

Preemptive vs. sequential:
- AstréeA started with a fully preemptive semantics (allow all interleavings)
- refined to take into account locks and priorities (mutual exclusion)

Future work:
- take into account inter-thread flow more precisely (almost sequential initialization, process and scheduling control)
- more precise support for OSEK/AUTOSAR (low preemption scheduling, explicit task switching)

full preemption is a sound (but coarse) abstraction of all other scheduling