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

Introduction

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

$\Longrightarrow$ sound automatic static analysis improves cost-effectiveness!

Astrée

Astrée: *Analyse Statique Temps-RÉEel*

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

AstréeA = Astrée + A

AstréeA: *Analyse statique de programmes temps-réels asynchrones*

- static analyzer for concurrent embedded C codes
- checks for run-time errors and data-races
- **fork** of the Astrée analyzer (around 2007)
  - reuses Astrée's iterator and abstract domains
  - builds on them a thread-modular analysis
  - adds new abstract domains
- as Astrée, aims towards high precision by specialization
- unlike Astrée, still 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

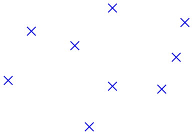
Abstract interpreters

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



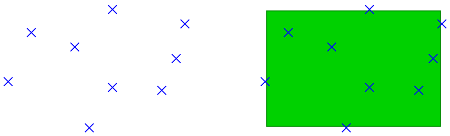
1. State the concrete semantics
    - function from programs to a (rich) mathematical world
    - formalization of the language specification
    - ground truth, immutable
    - not computable!

2. State the class of properties of interest

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

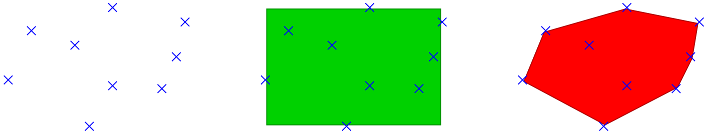
4. Design an analyzer over a computable abstract semantics

"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

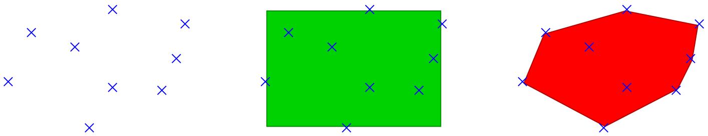
"Classic" abstract interpreter design



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 $\triangledown$
   - abstract domain: data-structures and algorithms

   abstract composition and $\triangledown$ accumulate precision loss
   $\Longrightarrow$ 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 $\rightsquigarrow$ AstréeA

- abstracts away platform details too much
  - arithmetic overflows: non-deterministic $\rightsquigarrow$ modular
    ($\Rightarrow$ more precise analysis)
  - ill-typed dereferences: program halt $\rightsquigarrow$ bit-level type-punning
    ($\Rightarrow$ more behaviors)

  necessary when analyzing non-portable programs

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

Specialized abstract interpreter

Reinvent the concrete semantics

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

- abstractions may still be sound
  e.g., non-deterministic overflow $\rightsquigarrow$ 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

$$\mathcal{X}_1 = I$$
$$\mathcal{X}_2 = \mathcal{X}_1 \cup [\![x \leftarrow x+1]\!]\mathcal{X}_3 \cup [\![x \geq y]\!]\mathcal{X}_2$$
$$\mathcal{X}_3 = [\![x < y]\!]\mathcal{X}_2$$

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

Interpretation by induction

$$[\![\textbf{while} \text{ random } \textbf{do } S]\!]\mathcal{X} \stackrel{\text{def}}{=} \text{lfp } \lambda\mathcal{Y}.\, \mathcal{X} \cup [\![S]\!]\mathcal{Y}$$

$$[\![\textbf{if } x < y \textbf{ then } S]\!]\mathcal{X} \stackrel{\text{def}}{=} [\![S]\!]([\![x < y]\!]\mathcal{X}) \cup [\![x \geq y]\!]\mathcal{X}$$

- 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

$\Rightarrow$ Astrée(A) uses an interpreter by induction

Analyzing concurrent programs

Multi-thread execution model

t_1	t_2
$\underline{1a}$ **while** random **do**	$\underline{1b}$ **while** random **do**
$\underline{2a}$ **if** $x < y$ **then**	$\underline{2b}$ **if** $y < 100$ **then**
$\underline{3a}$ $x \leftarrow x + 1$	$\underline{3b}$ $y \leftarrow y + [1, 3]$

Execution model:

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

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

Product-based analysis

t_1	t_2
$\underline{1a}$ **while** random **do**	$\underline{1b}$ **while** random **do**
$\underline{2a}$ **if** $x < y$ **then**	$\underline{2b}$ **if** $y < 100$ **then**
$\underline{3a}$ $x \leftarrow x + 1$	$\underline{3b}$ $y \leftarrow y + [1,3]$

Product of thread equations, interleaving of instructions:

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

$\mathcal{X}_{1a,1b} = I$

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

$\mathcal{X}_{3a,1b} = [\![x < y]\!] \mathcal{X}_{2a,1b}$

$\mathcal{X}_{2a,2b} = \mathcal{X}_{1a,2b} \cup [\![x \geq y]\!] \mathcal{X}_{2a,2b} \cup [\![x \leftarrow x + 1]\!] \mathcal{X}_{3a,2b} \cup$
$\qquad \mathcal{X}_{2a,1b} \cup [\![y \geq 100]\!] \mathcal{X}_{2a,2b} \cup [\![y \leftarrow y + [1,3]]\!] \mathcal{X}_{2a,3b}$

$\mathcal{X}_{3a,2b} = [\![x < y]\!] \mathcal{X}_{2a,2b} \cup \mathcal{X}_{3a,1b} \cup [\![y \geq 100]\!] \mathcal{X}_{3a,2b} \cup [\![y \leftarrow y + [1,3]]\!] \mathcal{X}_{3a,3b}$

$\mathcal{X}_{2a,3b} = \mathcal{X}_{1a,3b} \cup [\![x \geq y]\!] \mathcal{X}_{2a,3b} \cup [\![x \leftarrow x + 1]\!] \mathcal{X}_{3a,3b} \cup [\![y < 100]\!] \mathcal{X}_{2a,2b}$

$\mathcal{X}_{3a,3b} = [\![x < y]\!] \mathcal{X}_{2a,3b} \cup [\![y < 100]\!] \mathcal{X}_{3a,2b}$

limitations: large number of variables, large equations
no induction on the syntax possible
$\Longrightarrow$ **impractical**

Separate sequential analyses

t_1

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

t_2

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

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

t_1

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

t_2

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

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

  $\Longrightarrow$ one more level of fixpoint iteration

Inferring simple interferences

t_1

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

t_2

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

Analysis of t_1 in isolation

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

(2a): $x = y = 0$ $\mathcal{X}_{2a} = \mathcal{X}_{1a} \cup [\![\, x \leftarrow x + 1 \,]\!] \mathcal{X}_{3a} \cup [\![\, x \geq y \,]\!] \mathcal{X}_{2a}$

(3a): $\perp$ $\mathcal{X}_{3a} = [\![\, x < y \,]\!] \mathcal{X}_{2a}$

Inferring simple interferences

t_1

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

t_2

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

Analysis of t_2 in isolation

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

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

(3*b*): $x = 0,\ y \in [0, 99]$ $\quad\mathcal{X}_{3b} = [\![\, y < 100 \,]\!] \mathcal{X}_{2b}$

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

Inferring simple interferences

t_1

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

t_2

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

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

t_1	t_2
^{1a} **while** random **do**	^{1b} **while** random **do**
^{2a} **if** $x < y$ **then**	^{2b} **if** $y < 100$ **then**
^{3a} $x \leftarrow x + 1$	^{3b} $y \leftarrow y + [1, 3]$

Derived abstract analysis:

- similar to a sequential program analysis, but iterated
  (can be parameterized by arbitrary abstract domains)
- efficient (few reanalyses are required in practice)
- interferences are non-relational and flow-insensitive
  (limit inherited from the concrete semantics)

Limitation:

we get $x, y \in [0, 102]$; we don't get that $x \leq y$
simplistic view of thread interferences (volatile variables)
based on an incomplete concrete semantics!

Rely–guarantee reasoning

checking t_1

t_1	t_2
$\underline{1a}$ **while** random **do**	
$\underline{2a}$ **if** $x < y$ **then**	
$\underline{3a}$ $x \leftarrow x + 1$	

checking t_2

t_1	t_2
	$\underline{1b}$ **while** random **do**
	$\underline{2b}$ **if** $y < 100$ **then**
	$\underline{3b}$ $y \leftarrow y + [1, 3]$

Rely–guarantee: proof method introduced by Jones in 1981

- generalized Hoare logics (by structural induction $\Rightarrow$ thread-modular)

Rely–guarantee reasoning

checking t_1

t_1	t_2
$\underline{1a}$ **while** random **do**	
$\underline{2a}$ **if** $x < y$ **then**	
$\underline{3a}$ $x \leftarrow x + 1$	

(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

t_1	t_2
	$\underline{1b}$ **while** random **do**
	$\underline{2b}$ **if** $y < 100$ **then**
	$\underline{3b}$ $y \leftarrow y + [1, 3]$

(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

Rely–guarantee reasoning

checking t_1

t_1	t_2
1a **while** random **do**	x unchanged
2a **if** $x < y$ **then**	y incremented
3a $x \leftarrow x + 1$	$0 \leq y \leq 102$

(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

t_1	t_2
y unchanged	1b **while** random **do**
$0 \leq x \leq y$	2b **if** $y < 100$ **then**
	3b $y \leftarrow y + [1, 3]$

(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

Rely–guarantee reasoning

checking t_1

t_1	t_2
1a **while** random **do**	x unchanged
2a **if** $x < y$ **then**	y incremented
3a $x \leftarrow x + 1$	$0 \leq y \leq 102$

(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

t_1	t_2
y unchanged	1b **while** random **do**
$0 \leq x \leq y$	2b **if** $y < 100$ **then**
	3b $y \leftarrow y + [1, 3]$

(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 reasoning

checking t_1

t_1	t_2
$\underline{1a}$ **while** random **do**	x unchanged
$\underline{2a}$ **if** $x < y$ **then**	y incremented
$\underline{3a}$ $x \leftarrow x + 1$	$0 \leq y \leq 102$

(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

t_1	t_2
y unchanged	$\underline{1b}$ **while** random **do**
$0 \leq x \leq y$	$\underline{2b}$ **if** $y < 100$ **then**
	$\underline{3b}$ $y \leftarrow y + [1, 3]$

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

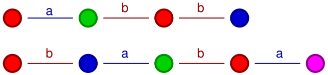
Rely–guarantee: proof method introduced by Jones in 1981

- 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: $\mathcal{F}$

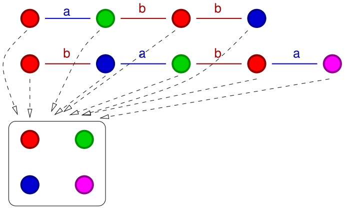
- threads: $\mathcal{T} \stackrel{\text{def}}{=} \{a, b, \dots\}$
- states: $\Sigma \stackrel{\text{def}}{=} \mathcal{C} \times \mathcal{M} = \{\bullet, \bullet, \dots\}$
  - control state: $\mathcal{C} \stackrel{\text{def}}{=} \mathcal{T} \to \mathcal{L}$ (maps threads to locations)
  - memory state: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

partial finite trace semantics in fixpoint form:

$\mathcal{F} \stackrel{\text{def}}{=} \operatorname{lfp} F$ where

$$F \stackrel{\text{def}}{=} \lambda X.\, I \cup \{\, \sigma_0 \xrightarrow{a_1} \cdots \sigma_i \xrightarrow{a_{i+1}} \sigma_{i+1} \mid \sigma_0 \xrightarrow{a_1} \cdots \sigma_i \in X \wedge \sigma_i \xrightarrow{a_{i+1}}_\tau \sigma_{i+1} \,\}$$

Non-modular concrete semantics



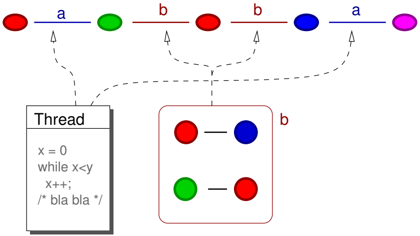
Reachable states: $\mathcal{R}$ (concrete semantics of interest)

Extract the states reached during execution

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

$\alpha^{reach}(T) \stackrel{\text{def}}{=} \{ \sigma \mid \exists \sigma_0 \xrightarrow{a_1} \cdots \sigma_n \in T : \exists i \leq n : \sigma = \sigma_i \}$

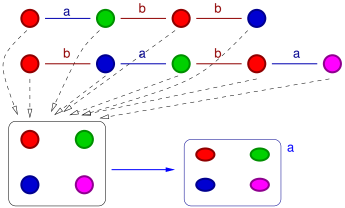
Modularity: main idea



Main idea: separate execution steps

- from the current thread a
  - found by analysis by induction on the syntax of a
- from other threads b
  - given as parameter in the analysis of a
  - inferred during the analysis of b

Trace decomposition



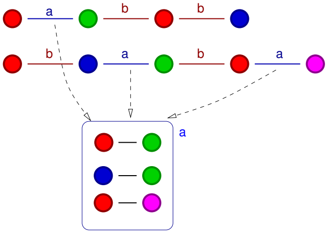
Reachable states projected on thread t: $\mathcal{R}\ell(t)$

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

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

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

Trace decomposition



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

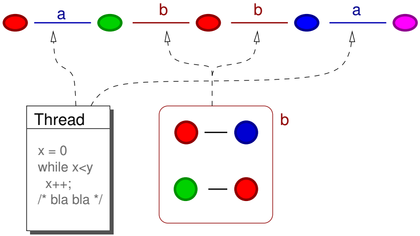
Extract the transitions with action t observed in $\mathcal{F}$

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

$\mathcal{I}(t) \stackrel{\text{def}}{=} \alpha^{itf}(\mathcal{F})(t)$

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

Thread-modular concrete semantics



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

States: $\mathcal{R}\ell$

Interleave:

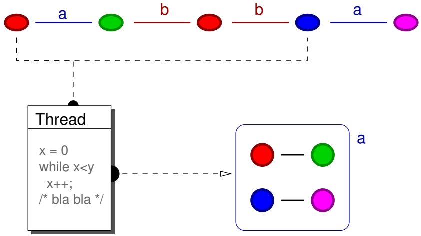
- transitions from the current thread t
- transitions from interferences $\mathcal{I}$ by other threads

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

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

$\Longrightarrow$ similar to reachability for a sequential program, up to $\mathcal{I}$

Thread-modular concrete semantics



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

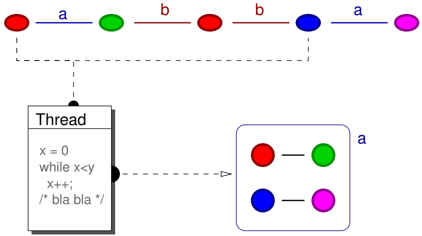
Interferences: $\mathcal{I}$

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

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

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

Thread-modular concrete semantics



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

Recursive definition:

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

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

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

$\Longrightarrow$ iterate analyses with interference

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

Thread-modular abstractions

Retrieving the simple interference-based analysis

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

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

Interference abstraction: remove all control information

$$\alpha^f_{\mathcal{I}}(Y) \stackrel{\text{def}}{=} \{ \langle \rho, \rho' \rangle \mid \exists L, L' \in \mathcal{C} : \langle \langle L, \rho \rangle, \langle L', \rho' \rangle \rangle \in Y \}$$

Note: we lose completeness

we cannot prove that x is bounded in $x \leftarrow x + 1 \mid\mid x \leftarrow x + 1$

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:

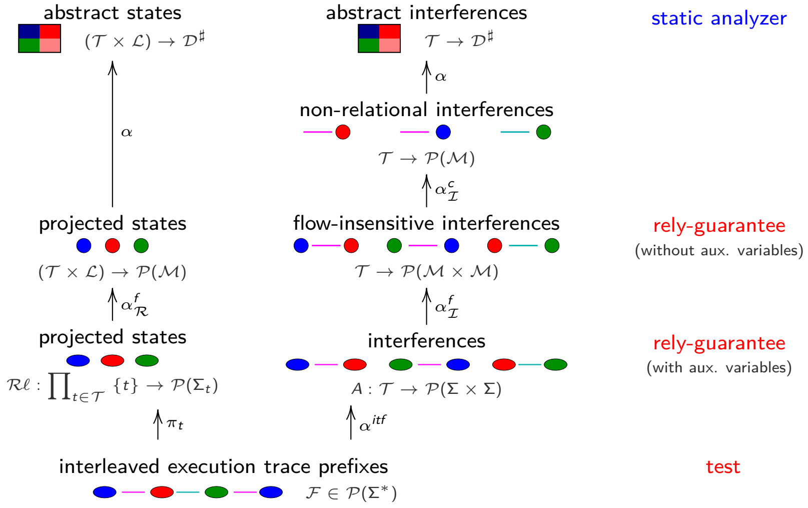
$$\alpha^{c}_{\mathcal{I}}(Y) \stackrel{\text{def}}{=} \lambda V.\, \{\, x \in \mathbb{V} \mid \exists \langle \rho, \rho' \rangle \in Y : \rho(V) \neq x \wedge \rho'(V) = x \,\}$$

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

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

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

From traces to thread-modular analyses



Compare with sequential analyses

abstract states

$\mathcal{L} \to \mathcal{D}^\sharp$ — static analyzer

$\uparrow \alpha$

states

$\mathcal{R} \in \mathcal{P}(\Sigma)$ — reachability

$\uparrow \alpha^{reach}$

execution trace prefixes

$\mathcal{F} \in \mathcal{P}(\Sigma^*)$ — 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

  $\Longrightarrow$ reduce false alarms while keeping scalability

Fully relational interferences

Academic experiment

Fully relational interference abstraction:

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

$\Longrightarrow$ 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 $x' = x + 1 \wedge x \in [0, 10]$

Abstract interpreter:

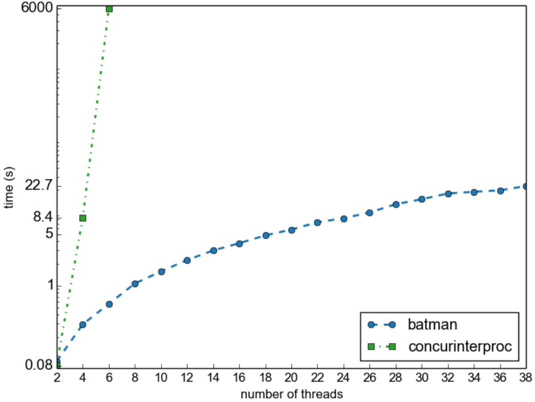
- represent abstract states as polyhedra
- propagate abstract states by induction on thread syntax
- maintain interferences in a thread-wide polyhedron $X^\sharp(t)$
- each assignment in t enriches $X^\sharp(t)$ with new interferences
- apply $(\cup^\sharp_{t' \neq t} X^\sharp(t'))^*$ after each instruction of t

Example analysis

t_1	t_2
while $z < 10000$	**while** $z < 10000$
$z = z + 1$	$z = z + 1$
if $y < c$ **then** $y = y + 1$	**if** $x < y$ **then** $x = x + 1$
done	**done**

- 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 $\mathcal{V}^2$
- able to infer $x \leq y$
- experimental comparison with ConcurInterproc
  (non thread modular, also able to infer $x \leq y$)

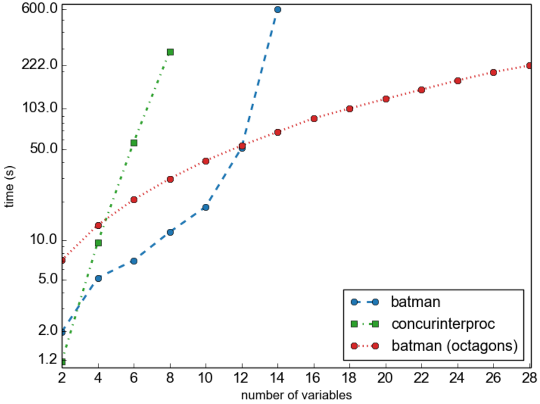
Scalability in threads



- *n* copies of each thread (with varying value for c)
- fixed number of variables

$\Longrightarrow$ much better scalability than non-modular methods!

Scalability in variables

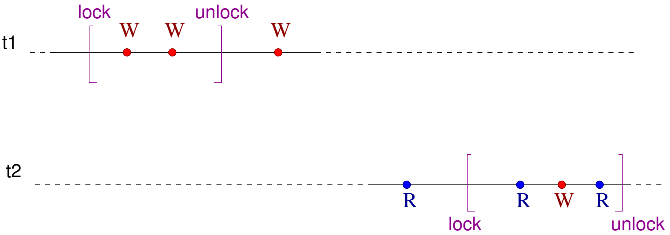


- n copies of each thread
- n copies of each variable

$\Longrightarrow$ scalability issues, packing techniques needed (expected)

Partilly relational interferences in AstréeA

Mutual exclusion locks



Mutexes:

- ensure mutual exclusion

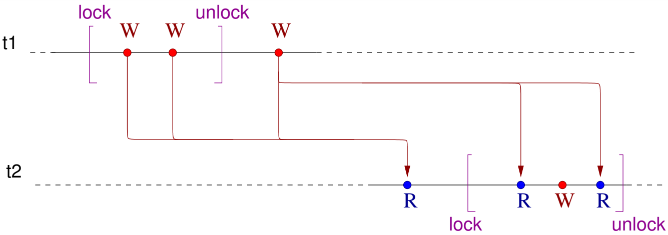
  at each time, each mutex can be locked by a single thread

- enforce memory consistency and atomicity

$\Longrightarrow$ we need to discard spurious interferences, to improve the precision

We assume a fixed, finite number of mutexes

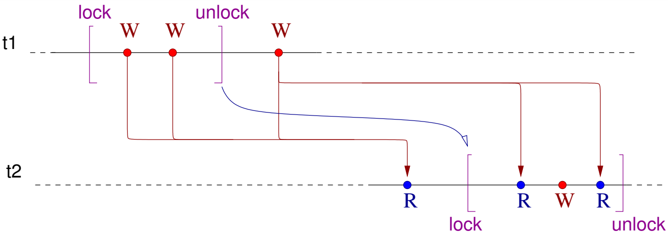
Mutual exclusion locks



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

abstract consumer/producer

N consumers	*N producers*
while random **do**	**while** random **do**
lock(m)	**lock**(m);
if $x > 0$ **then** $x \leftarrow x - 1$ **endif**;	$x \leftarrow x + 1$;
	if $x > 100$ **then** $x \leftarrow 100$ **endif**;
unlock(m)	**unlock**(m)

Assuming we have several (N) producers and consumers:

- no data-race interference (proof of the absence of data-race)
- well-synchronized interferences:

  consumer: $x \leftarrow [0, 99]$

  producer: $x \leftarrow [1, 100]$
- $\Longrightarrow$ we get that $x \in [0, 100]$

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

Locks and priorities

priority-based critical sections

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

Real-time scheduling

- only the highest priority unblocked thread can run
- **lock** and **yeild** may block
- **yeild**ing threads wake up non-deterministically (preempting lower-priority threads)
- explicit synchronisation enforces memory consistency

Locks and priorities

priority-based critical sections

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

Partition interferences and environments wrt. scheduling state

- partition wrt. mutexes tested with **islocked**
- $X \leftarrow$ **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 (**lock**, **yeild**)

Weakly relational interferences

Clock thread

while $Clock < 10^6$ **do**
  $Clock \leftarrow Clock + 1$;
  ...
done

Accumulator thread

while random **do**
  $Prec \leftarrow Clock$;
  ...
  $delta \leftarrow Clock - Prec$;
  if random **then** $x \leftarrow x + delta$ **endif**;
  ...
done

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

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

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

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

Inference and use

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

Relational invariant interferences

Abstraction: keep relations maintained by interferences

- remove control state in interferences ($\alpha_{\mathcal{I}}^{f}$)
- keep mutex state M (set of mutexes held)
- forget input-output relationships
- keep relationships between variables

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

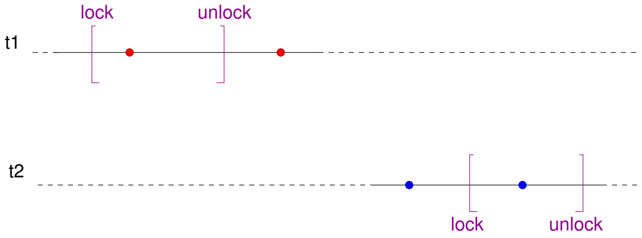
$\langle M, \rho \rangle \in \alpha_{\mathcal{I}}^{inv}(Y) \implies \langle M, \rho \rangle \in \alpha_{\mathcal{I}}^{inv}(Y)$ after any sequence of interferences from Y

Lock invariant:

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

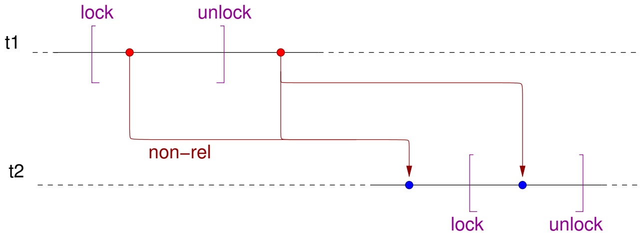
- property maintained outside code protected by m
- possibly broken while m is locked
- restored before unlocking m

Relational lock invariants



Improved interferences: mixing simple interferences and lock invariants

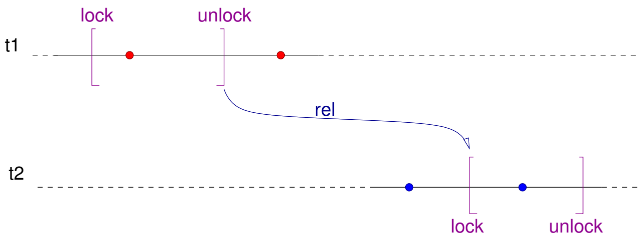
Relational lock invariants



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

t_1	t_2
while random **do**	x unchanged
lock(m);	y incremented
if $x < y$ **then**	$0 \leq y \leq 102$
$x \leftarrow x + 1$;	
unlock(m)	

analyzing t_2

t_1	t_2
y unchanged	**while** random **do**
$0 \leq x, x \leq y$	**lock**(m);
	if $y < 100$ **then**
	$y \leftarrow y + [1, 3]$;
	unlock(m)

Using all three interference abstractions:

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

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

Subsequence interference

t_1 : clock in H	t_2 : sample H into C	t_3 : accumulate elapsed time in T

while random **do**
  if $H < 10,000$ **then**
    $H \leftarrow H + 1$
	while random **do**
	  $C \leftarrow H$
		while random **do**
		  if random **then** $T \leftarrow 0$
		  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 $\not\Longrightarrow$ X is monotonic

$\Longrightarrow$ we infer an additional property implying monotony

Abstraction: subsequence

- $\mathcal{I}^\sharp_{sub}(t)(V) = \{\, W \in \mathcal{V} \mid V$'s values are a subsequence of W's values $\}$
- $\alpha^{sub}_{\mathcal{R}}(X)(V) \stackrel{\text{def}}{=} \{\, W \mid \forall \langle\langle \ell_0, \rho_0 \rangle, \ldots, \langle \ell_n, \rho_n \rangle\rangle \in X : \exists i_0, \ldots, i_n : \forall k : i_k \leq k \wedge i_k \leq i_{k+1} \wedge \forall j : \rho_j(V) = \rho_{i_j}(W) \,\}$

  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

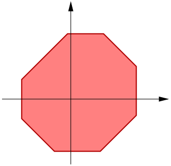
Astrée:

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

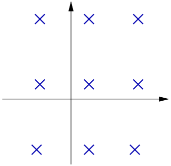
AstréeA:

- 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)

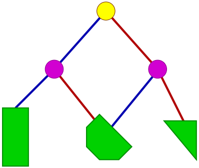
A few abstract domains used in Astrée(A)



octagons
$\pm X \pm Y \leq c$
[Miné 2006]

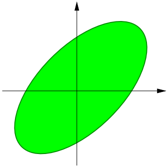


congruences
$X \equiv a\,[b]$
[Granger 1989]

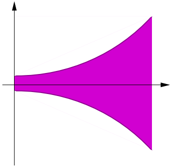


boolean decision trees

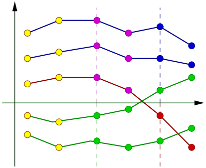
[Mauborgne]



ellipsoids
digital filters
[Feret 2005]



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

$\downarrow$

syntax iterator

$\downarrow$

trace partitioning domain

$\updownarrow$

memory domain

$\updownarrow$

pointer domain

$\updownarrow$

(reduced product of) numerical abstract domains

$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\vdots$
intervals	octagons	decision trees	filters	...

Astrée's abstract interpreter layers

↓ general C code

syntax iterator

↓ side-effect-free C assignments & tests

trace partitioning domain

↕

memory domain

↕ scalar assignments & tests

pointer domain

↕ numeric assignments & tests

(reduced product of) numerical abstract domains

↕	↕	↕	↕	⋮
intervals	octagons	decision trees	filters	...

Astrée's abstract interpreter layers

↓	`for (i=0;...)  a[i] = *p;`
syntax iterator	
↓	`a[i] = *p`
trace partitioning domain	
↕	`a[0] = *p, a[1] = *p, ...`
memory domain	
↕	`a@0 = x, a@4 = x`
pointer domain	
↕	`a@0 = x, a@4 = x`

(reduced product of) numerical abstract domains

↕	↕	↕	↕	⋮	`a@0 = x, a@4 = x`
intervals	octagons	decision trees	filters	...	

AstréeA's abstract interpreter layers

thread iterator

$\downarrow$

syntax iterator

$\downarrow$

trace partitioning domain

$\downarrow$

lock partitioning domain

$\updownarrow$

memory domain

$\updownarrow$ $\quad$ $\uparrow$

interference domain $\quad \vdots$

$\updownarrow$ $\quad$ $\downarrow$

pointer domain

$\updownarrow$

(reduced product of) numerical abstract domains

$\updownarrow$ $\quad$ $\updownarrow$ $\quad$ $\updownarrow$ $\quad$ $\updownarrow$ $\quad$ $\vdots$

intervals $\quad$ octagons $\quad$ decision trees $\quad$ filters $\quad$ ...

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

Analysis context

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)

Analysis context

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)

$\Longrightarrow$ analyze stand-alone "C" programs, with no undefined symbol

Example stub

```
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

size	OS	stub	selectivity	time	memory
1.9 M	ARINC	2.4 K	99.56%	154 h	18 GB
2.2 M	POSIX	2.3 K	99.52%	160 h	23 GB
31.8 K	POSIX	2.2 K	99.28%	50 mn	0.6 GB
33.1 K	POSIX	1.2 K	97.18%	35 h	2.5 GB

selectivity only slightly worse than for the main case study
$\Longrightarrow$ towards a cost-effective industrial use of AstréeA

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)

$\Longrightarrow$ an analysis sound only for sequential consistency may not be sound for the actual memory model!

(example: $y \leftarrow 1$; **if** $x = 0$ **then** $\cdots \parallel x \leftarrow 1$; **if** $y = 0$ **then** $\cdots$)

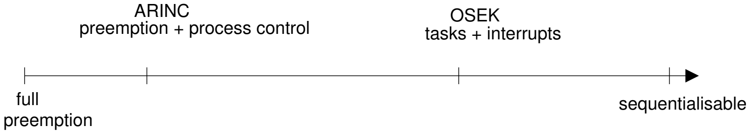
Result: The flow-insensitive non-relational analysis is sound wrt. a large set of weak memory models

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