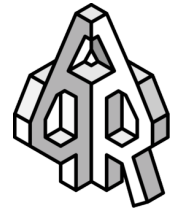
Verification of Concurrent Embedded Software by Abstract Interpretation

Antoine Miné

LIP6
University Pierre and Marie Curie
Paris, France

LaMHA/LTP Day
11 October 2017



Static analysis

Goal: program verification by static analysis

source

```
int search(int* t, int n) {
  int i;
  for (i=0; i<n; i++) {
    if (t[i]) break;
  }
  return t[i];
}
```

$\Longrightarrow$

analysis result

```
int search(int* t, int n) {
  int i;
  for (i=0; i<n; i++) {
    // 0 ≤ i < n
    if (t[i]) break;
  }
  // 0 ≤ i ≤ n ∨ n < 0
  return t[i];
}
```

✓

✗

- work directly on the source code
- infer properties on program executions
- automatically (cost effective)
- by constructing dynamically a semantic abstraction of the program
- deduce program correctness or raise alarms
  implicit specification: absence of RTE; or user-defined properties: contracts
- with approximations (efficient, but possible false alarms)
- soundly (no false positive)

We use the abstract interpretation theory.

Outline

More specifically:

- focus on accessibility and numeric properties of program variables
- with application to validation
  proof of absence of arithmetic overflow, invalid operation, illegal memory access, etc.
- on concurrent embedded programs
  adding: scheduling, process priorities, proof of absence of data-race or deadlock

Outline:

- Abstract interpretation primer
- The Astrée analyzer for embedded synchronous C code
- The AstréeA extension to embedded coucurrent C code
- Future directions

Abstract Interpretation

Abstract interpretation

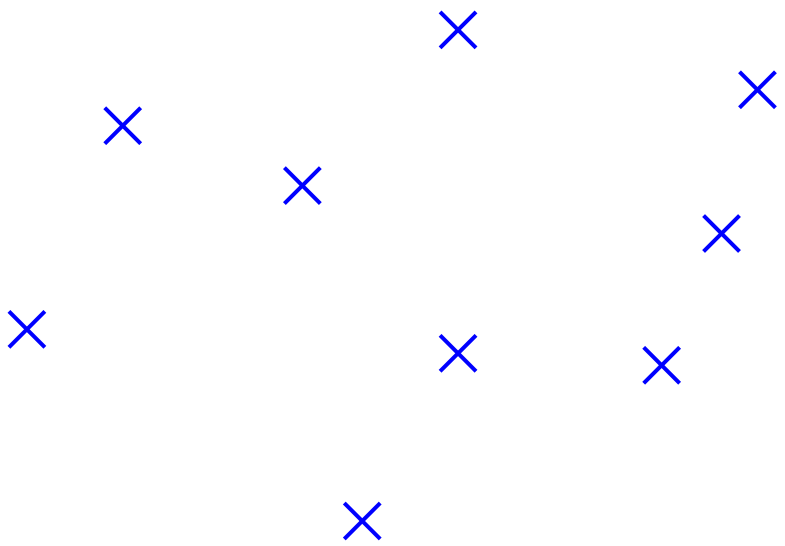
Abstract interpretation: theory of the approximation of (program) semantics

Principle: be tractable by reasoning at an abstract level

Abstract interpretation

Abstract interpretation: theory of the approximation of (program) semantics

Principle: be tractable by reasoning at an abstract level

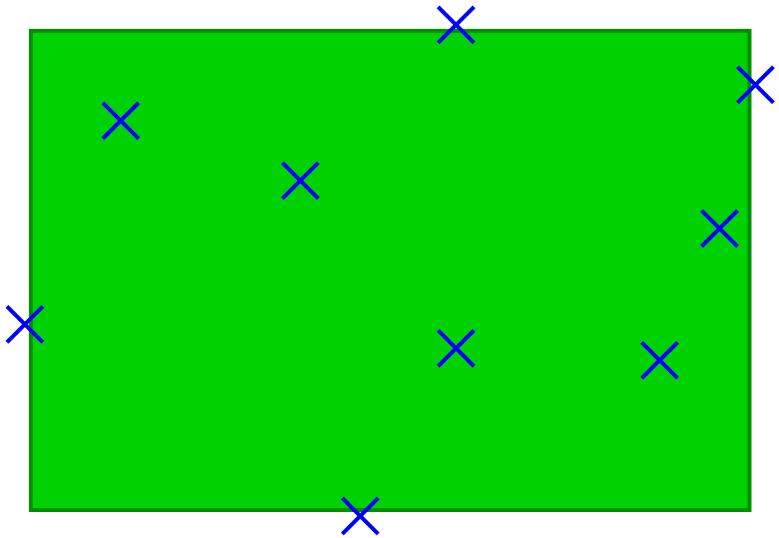


concrete executions : $\{(0,3),(5.5,0),(12,7),\ldots\}$ (not computable)

Abstract interpretation

Abstract interpretation: theory of the approximation of (program) semantics

Principle: be tractable by reasoning at an abstract level



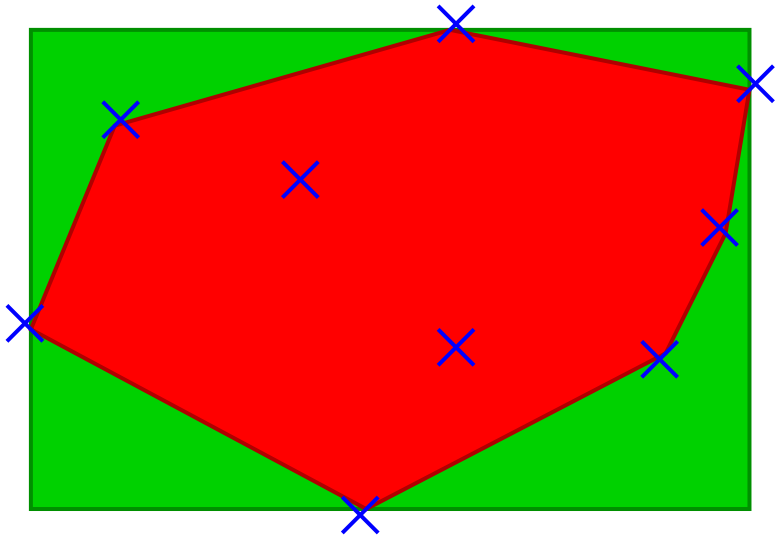
concrete executions : $\{(0,3),(5.5,0),(12,7),\ldots\}$ (not computable)

box domain : $X \in [0,12] \wedge Y \in [0,8]$ (linear cost)

Abstract interpretation

Abstract interpretation: theory of the approximation of (program) semantics

Principle: be tractable by reasoning at an abstract level



concrete executions :	$\{(0,3),(5.5,0),(12,7),\ldots\}$	(not computable)
box domain :	$X \in [0,12] \wedge Y \in [0,8]$	(linear cost)
polyhedra domain :	$6X + 11Y \geq 33 \wedge \cdots$	(exponential cost)

$\Longrightarrow$ various abstractions, trade-off cost vs. precision and expressiveness

Abstract computations

Define an interpretation of atomic statements in the abstract domain(s), compose them to analyze the program

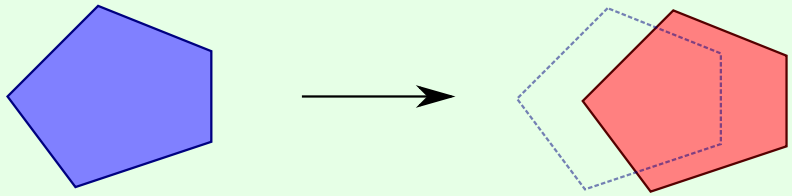
- by propagation on the control-flow graph edges (à la data-flow)
- or by induction on the syntax of programs (interpretation)

Example in the polyhedra domain

Assignment:

• `X = X + 1` •

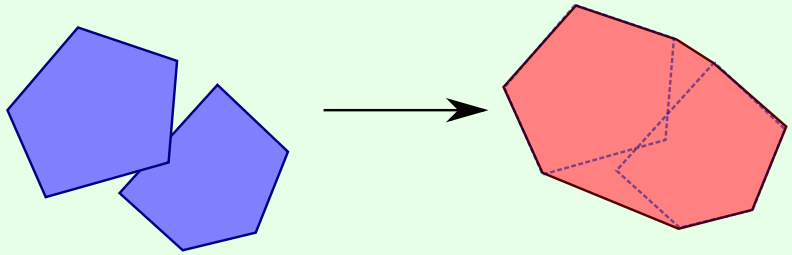
translation



Join:

if $\cdots$ **then** $\cdots$ • **else** $\cdots$ • **fi** •

convex hull



Loop invariants

Loops are difficult to analyze, due to large (unbounded) behaviors!

Principle: find an inductive invariant summarizing all loop iterates

- true when first entering the loop
- stable by a loop iteration (invariance proof by induction on loop iterations)

Loop invariants

Loops are difficult to analyze, due to large (unbounded) behaviors!

Principle: find an inductive invariant summarizing all loop iterates

- true when first entering the loop
- stable by a loop iteration (invariance proof by induction on loop iterations)

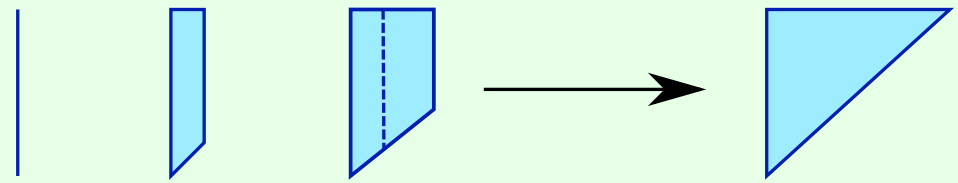
In abstract interpretation, we find an abstract inductive invariant

- iterate the loop in the abstract
- accelerate loop convergence to ensure analysis termination
  $\Longrightarrow$ drop unstable constraints and bounds (widening $\triangledown$)

Simple loop

```
assume N in [0,1000];
I = 0;
while • (i < N)
    I = I + 1;
```

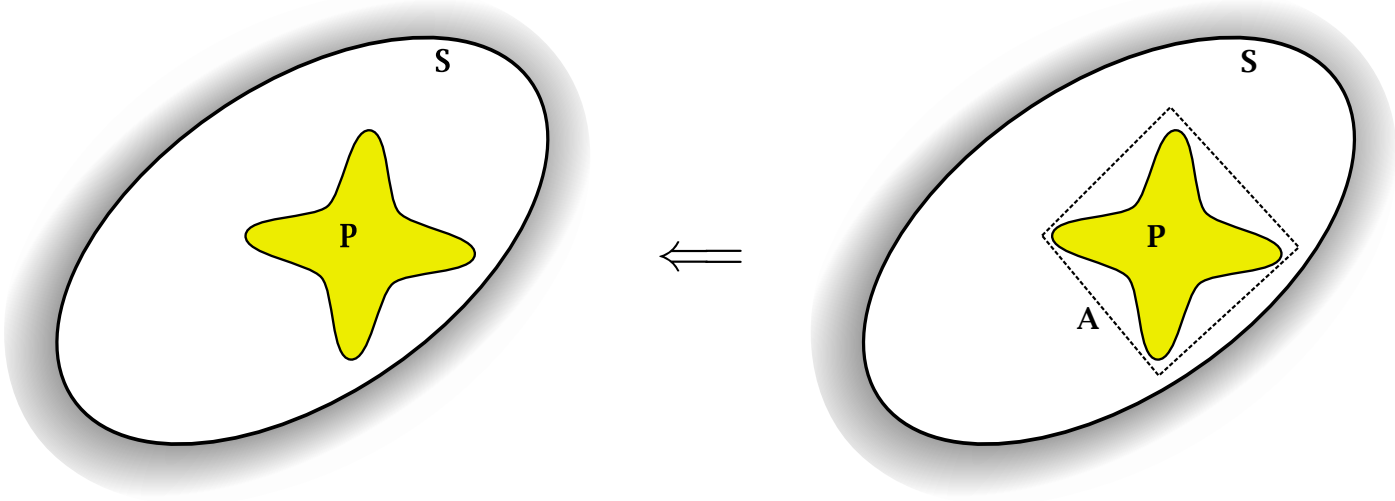
Iterations with widening on polyhedra



to infer a precise box at the end of a loop (e.g., $\mathtt{I} \leq 1000$),
we may need to infer a relational loop invariant! (e.g., $\mathtt{I} \leq \mathtt{N}$)

Widenings offer a general method to approximate fixpoints.
Useful for loops, recursive calls, cycles in the CFG, etc.

Soundness and false alarms



$P \subseteq S$

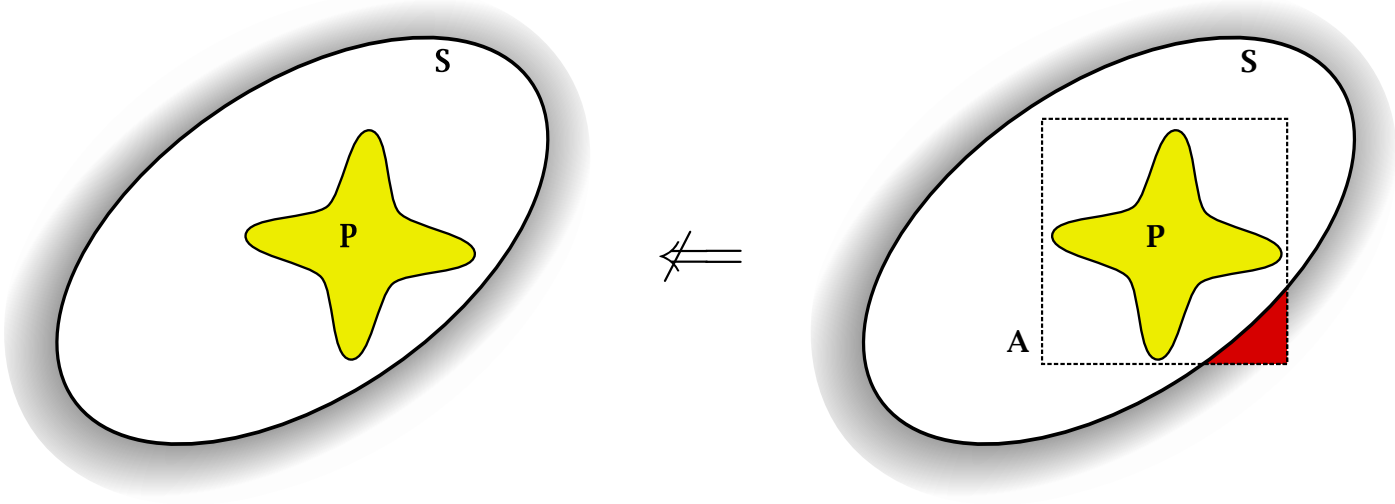
$A \subseteq S$
program proved

Goal : prove that a program P satisfies its specification S

We collect (an abstraction of) the reachable states P and compare to states S

A polyhedral abstraction A can prove the correctness

Soundness and false alarms



$P \subseteq S$

$A \not\subseteq S$
false alarm

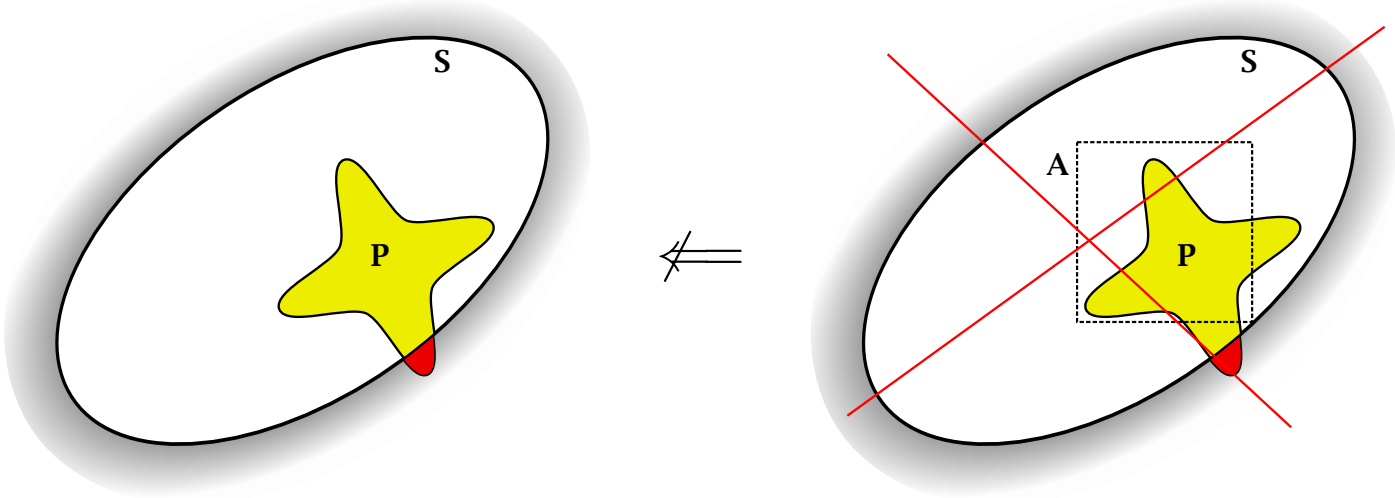
Goal : prove that a program P satisfies its specification S

We collect (an abstraction of) the reachable states P and compare to states S

A polyhedral abstraction A can prove the correctness

A box abstraction cannot prove the correctness $\Longrightarrow$ false alarm

Soundness and false alarms



$P \not\subseteq S$

$A \subseteq S$
false negative
cannot occur

Goal : prove that a program P satisfies its specification S

We collect (an abstraction of) the reachable states P and compare to states S

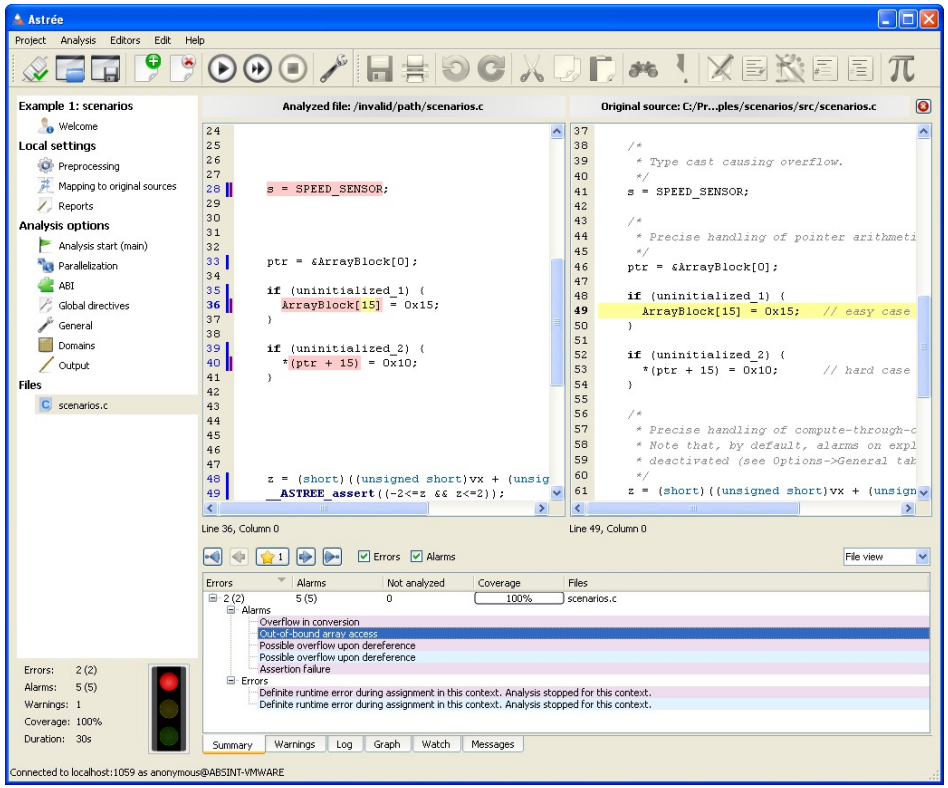
A polyhedral abstraction A can prove the correctness

A box abstraction cannot prove the correctness $\Longrightarrow$ false alarm

The analysis is sound: no false negative reported!

The Astrée Analyzer

Example static analyzer: Astrée



- checks for absence of run-time error in embedded synchronous C code
- prototype started at ENS in 2001
- industrialized by AbsInt in 2009
- used in production e.g. at Airbus

www.absint.com

Construction by refinement

Theoretical completeness:

- for each program and property, an abstract domain exists
- but the construction is *generally* not mechanizable
  (except in limited cases: finite restrictions of infinite domains)

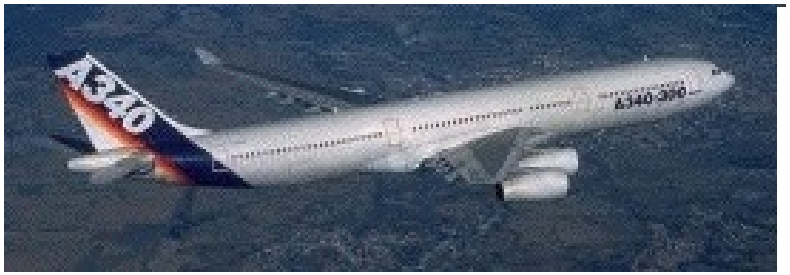
Practical approach used in Astrée

- build a simple and fast analyzer (intervals)
- iterate: refine by hand the analyzer until 0 false alarm:
  - determine which necessary properties are missed
  - add / refine an abstract domain to infer it
  - improve the widening ∇
  - improve the communications between domains

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

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

Astrée results



Airbus A340-300 (2003)

Airbus A380 (2004)

Success: on specific industrial applications

- size: from 70 000 to 860 000 lines of embedded reactive C
- analysis time: from 45mn to $\simeq$40h
- 0 alarm: proof of absence of run-time error
  $\Longrightarrow$ usable for software validation

Now available commercially through AbsInt (2009)

usable by industrial customers in the embedded world

Specialisation example: Low-level memory abstraction

C union types

```
union {
  struct { uint8 al,ah,bl,bh } b;
  struct { uint16 ax,bx } w;
} r;
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
```

C standard: ill-typed programs, undefined behavior

In practice:

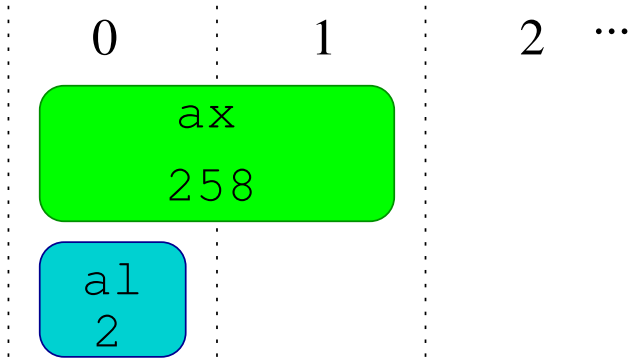
- there is no error
- the semantics is well-defined

(ABI specification)

Specialisation example: Low-level memory abstraction

C union types

```
union {
  struct { uint8 al,ah,bl,bh } b;
  struct { uint16 ax,bx } w;
} r;
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
```



C standard: ill-typed programs, undefined behavior

In practice:

- there is no error
- the semantics is well-defined (ABI specification)

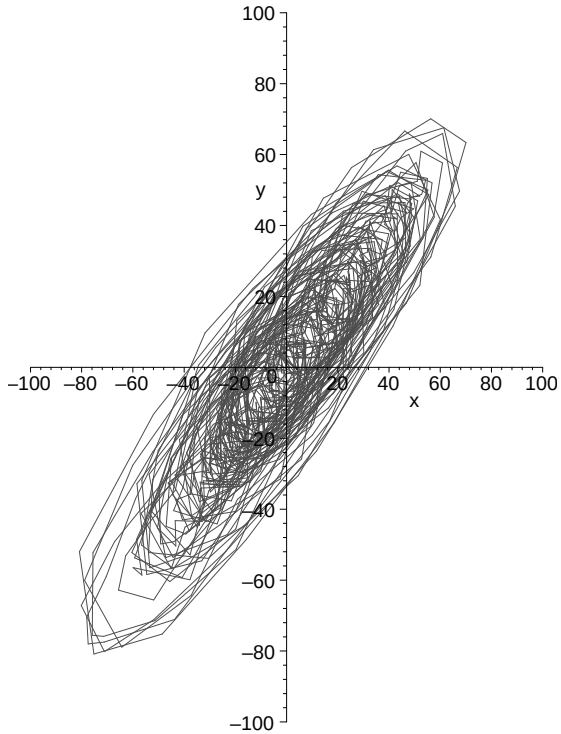
$\Longrightarrow$ develop abstractions for overlapping memory cells
– creating cells dynamically, on-demand
– parameterized by (memory-unaware) numeric abstractions

also works for type-punning constructions, without any static type information

Specialisation example: Domain-specific abstraction

Digital 2nd order filter

```
while (1) {
    X = input();
    P = 0.46*X - 0.77*E1 + 0.43*E2;
    P = P + 1.54*S1 - 0.67*S2;
    E2 = E1; E1 = X;
    S2 = S1; S1 = P;
}
```



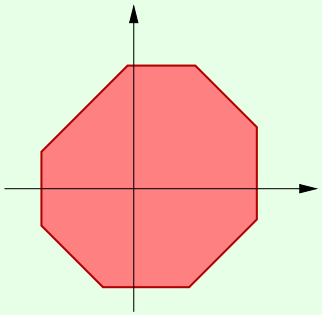
No box over (S1,S2) is an inductive invariant

$\Longrightarrow$ to infer variable bounds, we need strictly more expressive abstract domains

Ellipsoid domain: $Y^2 - aXY - bX^2 \leq c$ (Feret 2005)

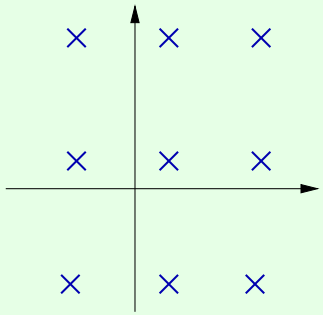
Composing abstractions

A few of the abstract domains used in Astrée



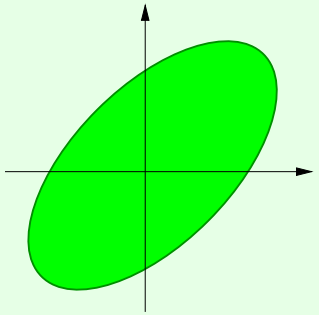
octagons

$\pm X \pm Y \leq c$



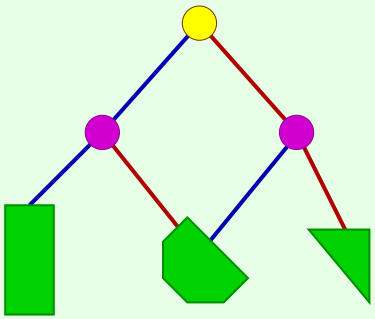
congruences

$X \equiv a\,[b]$

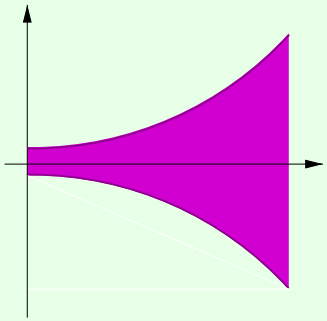


ellipsoids

digital filters

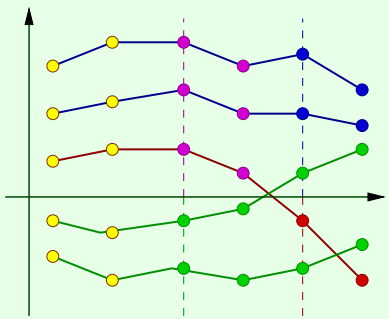


boolean decision trees



exponentials

$X \leq (1+\alpha)^{\beta t}$



trace partitions

Combine the strength of different abstractions!

The AstréeA Analyzer

Example static analyzer: AstréeA

Extension of Astrée to embedded concurrent software

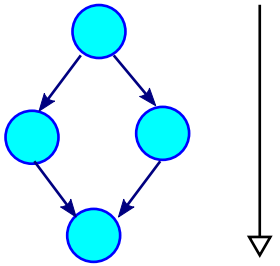
Concurrency model:

- fixed number of threads
- preemptive real-time scheduling on a single processor
- shared memory, locks

Target application:

- embedded avionic code
- 2 MIoc of C, 15 threads
- reactive code + network code + lists, strings, pointers
- many variables, large arrays, many loops, shallow call graph
- no dynamic memory allocation, no recursivity
- 1100 alarms, in 30h analysis time

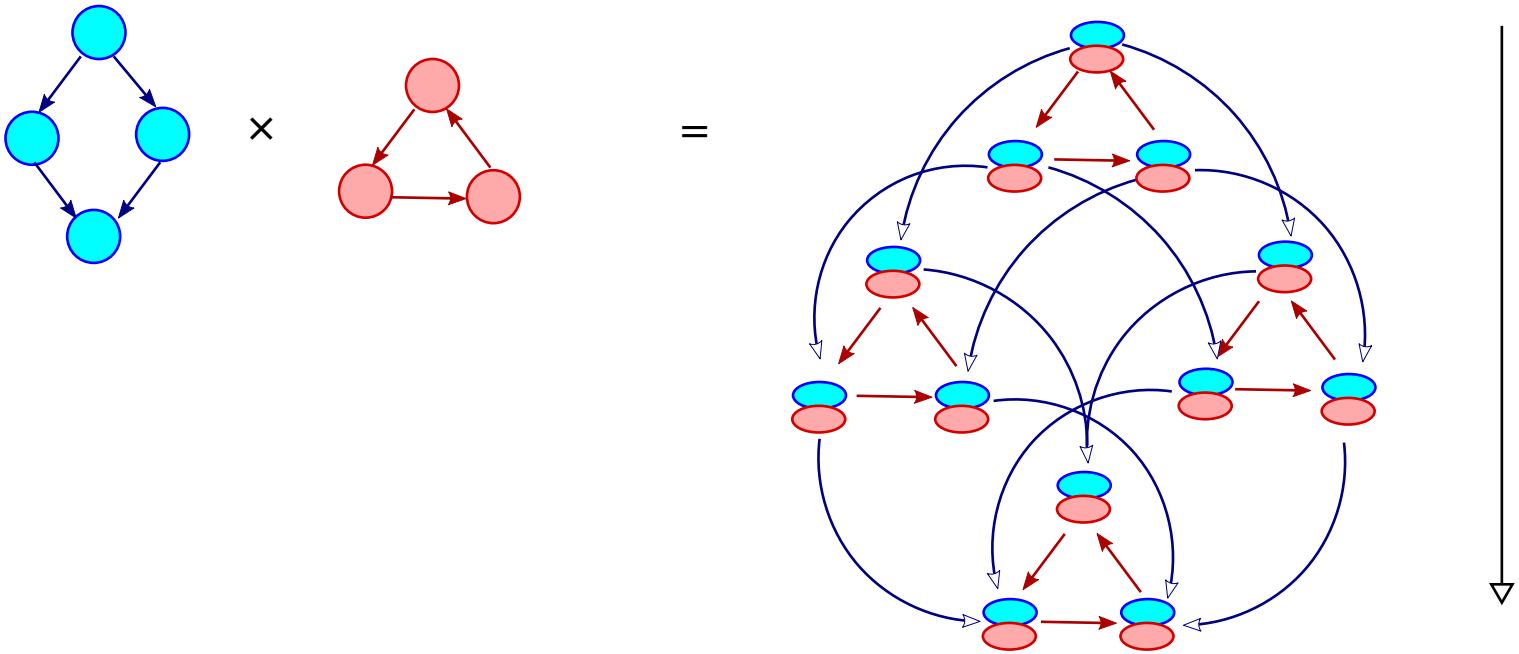
Non-thread modular analysis



Sequential analysis:

- one abstract state per program point
- one transfer function per instruction
- various iteration schemes with widening

Non-thread modular analysis



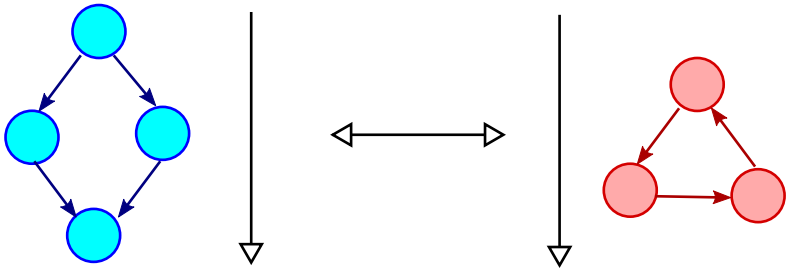
Natural extension to multi-thread: CFG product

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

Not practical for high scalability...

Beyond partial-order reduction: we need abstraction *a priori*

Thread-modular analysis



Thread-modular analysis:

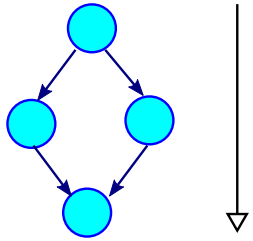
- analyze each thread separately
- also analyze their interaction

$\Longrightarrow$ no product, more efficient!

Soundness : do we still cover all possible interleavings?

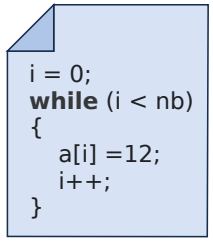
CFG-based vs. syntax-based

CFG-based:



$$\begin{cases} X_1 = \top \\ X_2 = F_2(X_1) \\ X_3 = F_3(X_1) \\ X_4 = F_4(X_3, X_4) \end{cases}$$

Syntax-based:



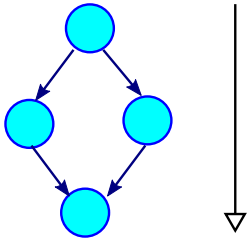
$$[\![\textbf{while } c \textbf{ do } b]\!] X \stackrel{\text{def}}{=} [\![\neg c]\!] (\text{lfp } \lambda Y.\, X \cup [\![b]\!]([\![c]\!] Y))$$

$$[\![\textbf{if } c \textbf{ then } t]\!] X \stackrel{\text{def}}{=} [\![t]\!] ([\![c]\!] X) \cup [\![\neg c]\!] X$$

$\ldots$

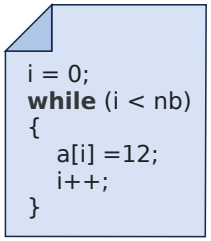
CFG-based vs. syntax-based

CFG-based:



$$\begin{cases} X_1 = \top \\ X_2 = F_2(X_1) \\ X_3 = F_3(X_1) \\ X_4 = F_4(X_3, X_4) \end{cases}$$

Syntax-based:

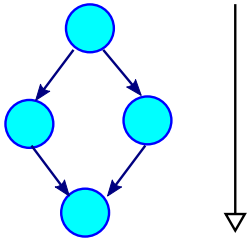


$$[\![\textbf{while } c \textbf{ do } b]\!] X \stackrel{\text{def}}{=} [\![\neg c]\!] (\text{lfp } \lambda Y.\, X \cup [\![b]\!]([\![c]\!] Y))$$
$$[\![\textbf{if } c \textbf{ then } t]\!] X \stackrel{\text{def}}{=} [\![t]\!] ([\![c]\!] X) \cup [\![\neg c]\!] X$$
$$\ldots$$

- linear memory in program length
- flexible solving strategy
  flexible context sensitivity
- easy to adapt to concurrency,
  both in thread-modular and CFG
  product way

CFG-based vs. syntax-based

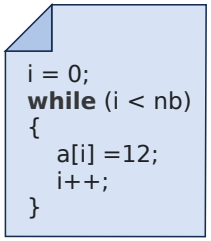
CFG-based:



$$\begin{cases} X_1 = \top \\ X_2 = F_2(X_1) \\ X_3 = F_3(X_1) \\ X_4 = F_4(X_3, X_4) \end{cases}$$

- linear memory in program length
- flexible solving strategy
  flexible context sensitivity
- easy to adapt to concurrency,
  both in thread-modular and CFG product way

Syntax-based:

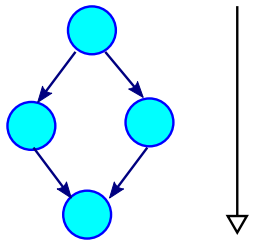


$$[\![\textbf{while } c \textbf{ do } b]\!] X \overset{\text{def}}{=} [\![\neg c]\!] (\text{lfp } \lambda Y.\, X \cup [\![b]\!]([\![c]\!] Y))$$
$$[\![\textbf{if } c \textbf{ then } t]\!] X \overset{\text{def}}{=} [\![t]\!] ([\![c]\!] X) \cup [\![\neg c]\!] X$$
$$\ldots$$

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

CFG-based vs. syntax-based

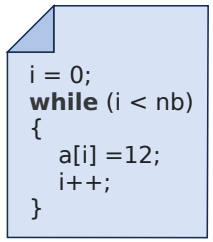
CFG-based:



$$\begin{cases} X_1 = \top \\ X_2 = F_2(X_1) \\ X_3 = F_3(X_1) \\ X_4 = F_4(X_3, X_4) \end{cases}$$

- linear memory in program length
- flexible solving strategy
  flexible context sensitivity
- easy to adapt to concurrency,
  both in thread-modular and CFG product way

Syntax-based:

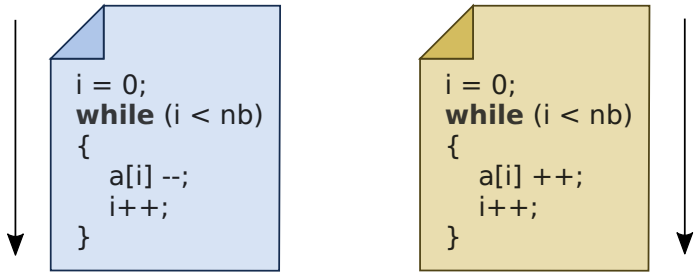


$$[\![\textbf{while } c \textbf{ do } b]\!] X \overset{\text{def}}{=} [\![\neg c]\!] (\text{lfp } \lambda Y.\, X \cup [\![b]\!]([\![c]\!] Y))$$
$$[\![\textbf{if } c \textbf{ then } t]\!] X \overset{\text{def}}{=} [\![t]\!] ([\![c]\!] X) \cup [\![\neg c]\!] X$$
$$\ldots$$

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

for scalability on large programs, memory is a limiting factor
$\Rightarrow$ **we use an interpreter by induction on the syntax**

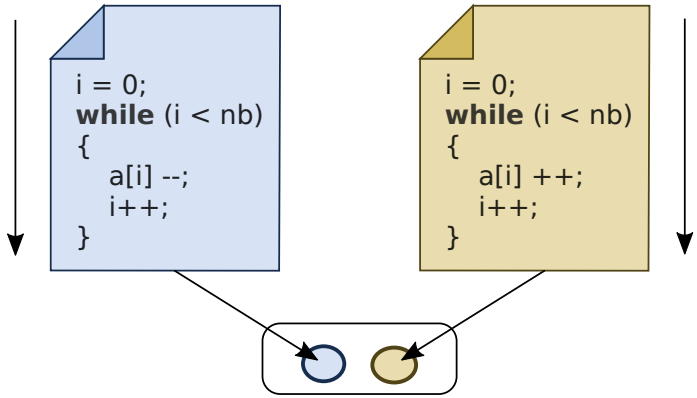
Thread-modular analysis with simple interferences



Principle:

- analyze each thread in isolation

Thread-modular analysis with simple interferences

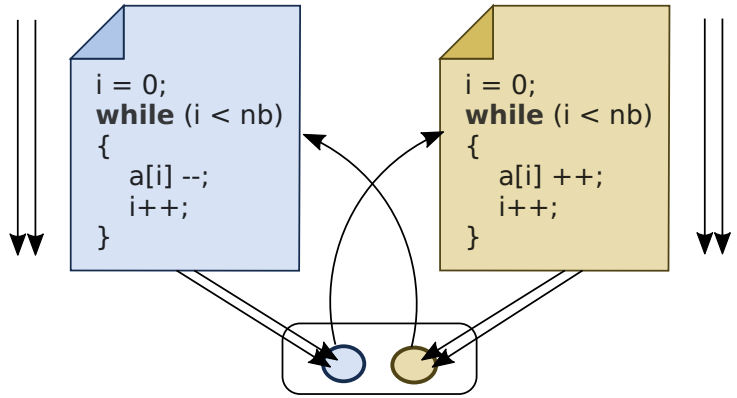


Principle:

- analyze each thread in isolation
- gather the values written into each variable by each thread
  $\Longrightarrow$ so-called interferences

  suitably abstracted in an abstract domain, such as intervals

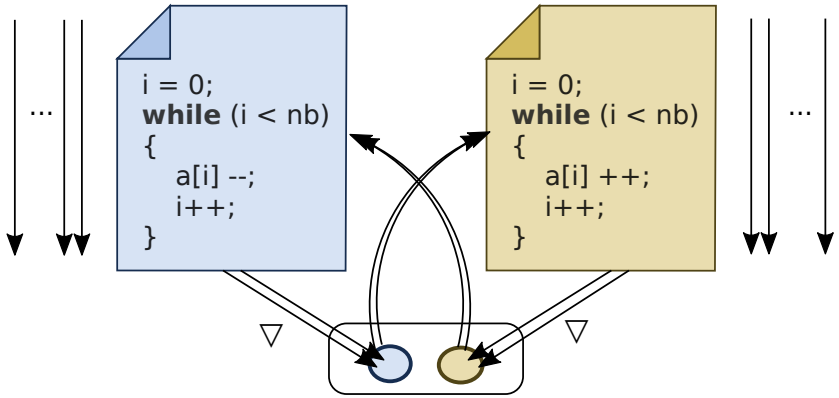
Thread-modular analysis with simple interferences



Principle:

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

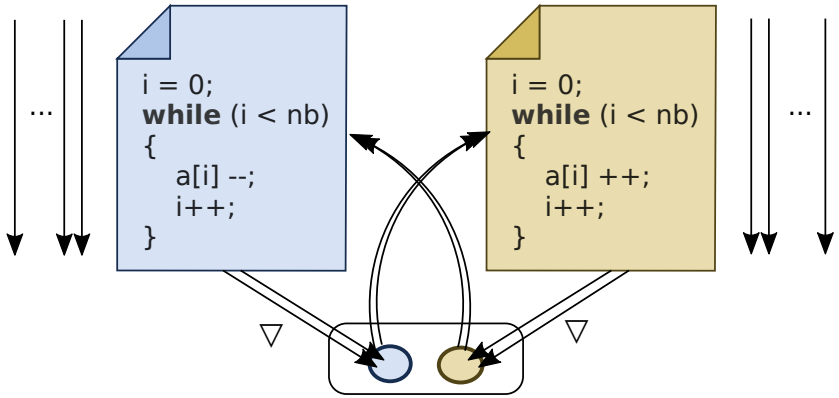
Thread-modular analysis with simple interferences



Principle:

- analyze each thread in isolation
- gather the values written into each variable by each thread
  $\Longrightarrow$ so-called interferences
  suitably abstracted in an abstract domain, such as intervals
- reanalyze threads, injecting these values at each read
- iterate until stabilization while widening interferences

Thread-modular analysis with simple interferences



Principle:

- analyze each thread in isolation
- gather the values written into each variable by each thread
  $\Longrightarrow$ so-called interferences
  suitably abstracted in an abstract domain, such as intervals
- reanalyze threads, injecting these values at each read
- iterate until stabilization while widening interferences

Benefits:

- very similar to a sequential analysis (high reusability)
- efficient
- sound: takes all thread interleavings into account

Simple abstract interferences: Example

Simple interference analysis: with interval abstraction

t_1	t_2
while random **do**	**while** random **do**
if $x < y$ **then**	**if** $y < 100$ **then**
$x \leftarrow x + 1$	$y \leftarrow y + [1, 3]$
done	**done**

Simple abstract interferences: Example

Simple interference analysis: with interval abstraction

t_1	t_2
while random **do**	**while** random **do**
if $x < y$ **then**	**if** $y < 100$ **then**
$x \leftarrow x + 1$	$y \leftarrow y + [1, 3]$
done	**done**

iteration	t_1	t_2
1	$\emptyset$	$\emptyset$

Analysis as separate sequential processes, without interferences

$\Longrightarrow$ t_2 writes $[1, 102]$ into y

Simple abstract interferences: Example

Simple interference analysis: with interval abstraction

t_1	t_2
while random **do**	**while** random **do**
if $x < y$ **then**	**if** $y < 100$ **then**
$x \leftarrow x + 1$	$y \leftarrow y + [1, 3]$
done	**done**

iteration	t_1	t_2
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$y \mapsto [1, 102]$

add the information that t_2 writes $[1, 102]$ into y,
for t_1 to use

$\Longrightarrow$ t_1 now writes into x

Simple abstract interferences: Example

Simple interference analysis: with interval abstraction

t_1	t_2
while random **do**	**while** random **do**
if $x < y$ **then**	**if** $y < 100$ **then**
$x \leftarrow x + 1$	$y \leftarrow y + [1, 3]$
done	**done**

iteration	t_1	t_2	
1	$\emptyset$	$\emptyset$	
2	$\emptyset$	$y \mapsto [1, 102]$	
3	$x \mapsto [1, 102]$	$y \mapsto [1, 102]$	(stable)

t_1 writes into x, but this is not visible by t_2;
we reach a stable point

$\Longrightarrow$ program invariant: $x, y \in [0, 102]$

Limitation of simple interferences

t_1	t_2
while random **do**	**while** random **do**
if $x < y$ **then**	**if** $y < 100$ **then**
$x \leftarrow x + 1$	$y \leftarrow y + [1, 3]$
done	**done**

- the analysis finds $x, y \in [0, 102]$
- but, in fact, $0 \leq x \leq y \leq 102$

Cause:

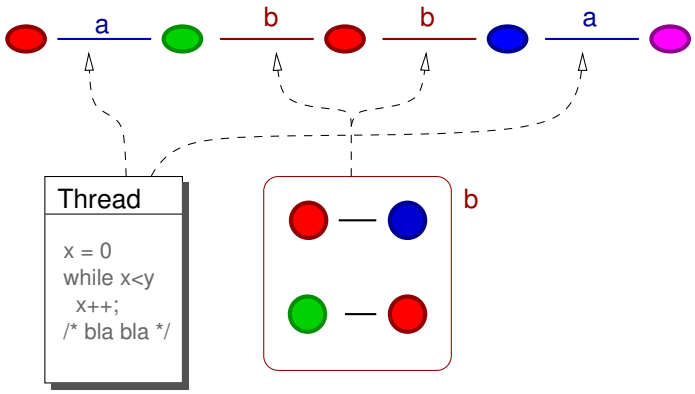
Transporting abstractions of sets of variable values is insufficient.
even if we transport the exact set instead of an interval abstraction, we lose precision!

Simple interferences perform a flow-insensitive, non-relational abstraction

$\Rightarrow$

we need to transport flow-sensitive relations
we will develop a new thread-modular concrete semantics with is complete

A thread-modular concrete semantics (1/3)



Intuition :

During the analysis of a single thread a :

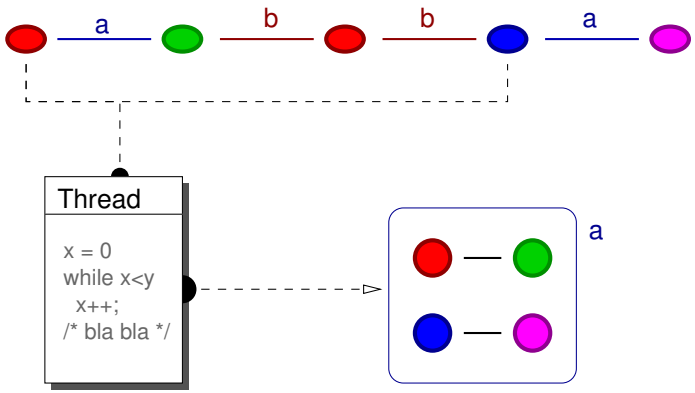
- in order to reconstruct whole program executions
- it is sufficient to know the transitions the other threads b can apply $I(b)$

$\Longrightarrow$

interleaving of the current thread and the environment (other threads)
we can then extract all the reachable states $R(a)$

- $R(x) \subseteq control \times memory$
- $I(x) \subseteq (control \times memory)^2$
- with auxiliary variables! (control and memory of all threads visible by every thread)

A thread-modular concrete semantics (2/3)



Intuition :

During the analysis of a single thread :

- when computing the reachable states $R(a)$
- we can extract the effective state transitions $I(a)$ it performs

$\Longrightarrow$ this information is required by the other threads...

A thread-modular concrete semantics (3/3)

Problem: mutually recursive equations

- reachability $R(a)$ depends on $\{ I(x) \mid x \neq a \}$
- interference $I(a)$ depends on $R(a)$

Solution: fixpoint computed by iteration

- start with $\forall x : I(x) = \emptyset$
- compute a first approximation of $R(x)$ by analyzing every thread x
- deduce an approximation of $I(x)$ for every x
- analyze again each thread to get a more realistic $R(x)$
- deduce an enriched $I(x)$
- ...
- iterate until reaching a fixpoint (may be infinite...)

This uncomputable concrete semantic is complete
and in thread-modular form.

A thread-modular abstract semantics

Principle: Mimic the concrete computation

- using a state abstract domain to approximate $R(x)$
- using a relation abstract domain to approximate $I(x)$
- iterate with widening on $I(x)$

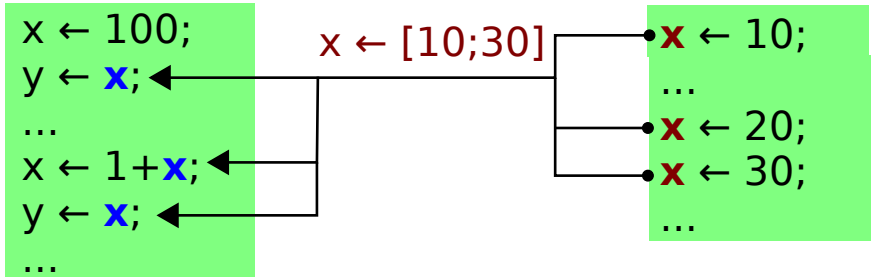
$\Longrightarrow$ computable abstract semantics

Retrieving the simple interference abstraction :

- $R(x)$: forget the control information of all other threads $y \neq x$
  abstract the memory in a numeric abstract domain
- $I(x)$:
  forget all the control information (flow insensitivity)
  remember only the image of the relation $\{ w \mid \exists v : (v, w) \in I(x) \}$
  abstract the image in a non-relational domain (non-relationality)

We can do better by keeping some relationality and flow-sensitivity!

Application: Lock-partitioning of simple interferences



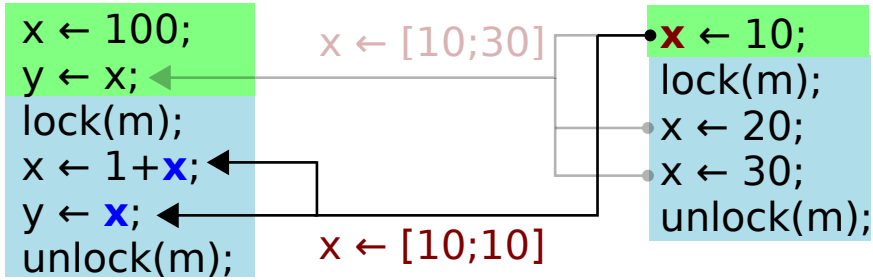
Without lock:

- all writes into x on the right affect all reads from x on the left
- interferences taken into account through expression injection

  $y \leftarrow x$ becomes $y \leftarrow x \cup [10;30]$
  $x \leftarrow 1 + x$ becomes $x \leftarrow 1 + (x \cup [10;30])$

  and then use a regular transfer function
- we detect the presence of data-races

Application: Lock-partitioning of simple interferences

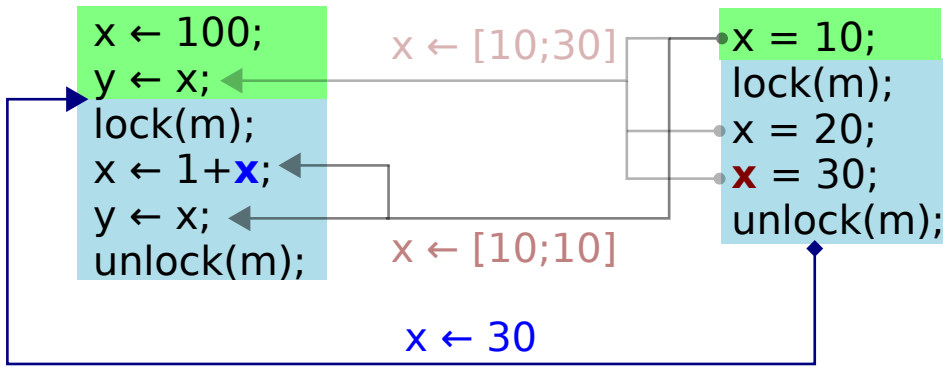


With locks:

- partition interferences wrt. locks held
- the first $y \leftarrow x$ is still $y \leftarrow x \cup [10; 30]$
  the second $y \leftarrow x$ is now $y \leftarrow x \cup [10; 10]$

  these interferences are caused by data-races

Application: Lock-partitioning of simple interferences



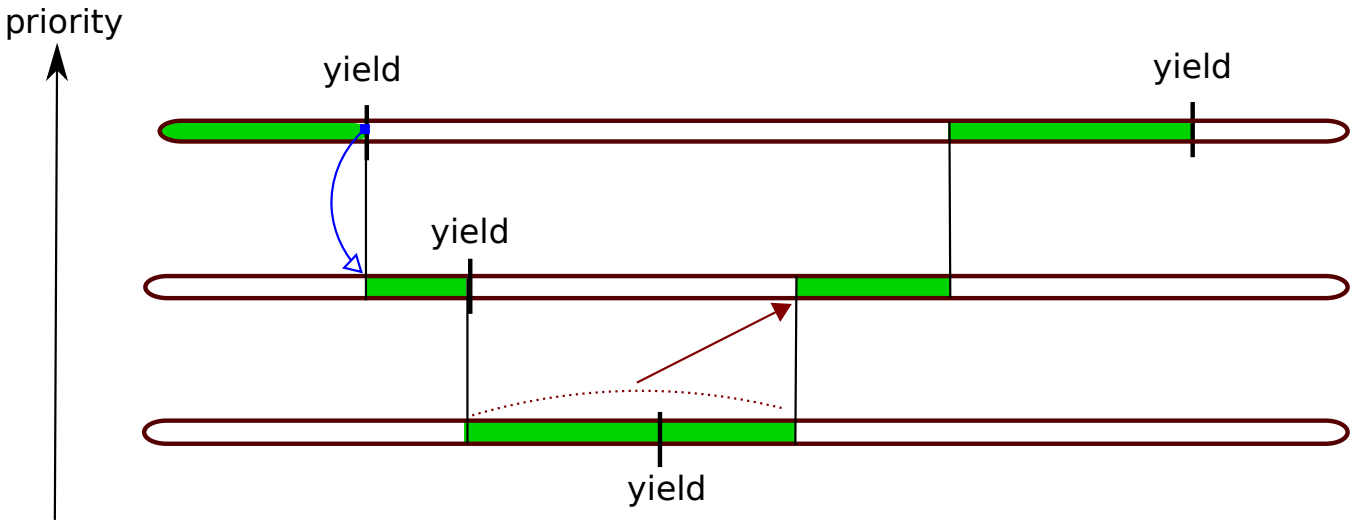
With locks:

- partition interferences wrt. locks held
- the first $y \leftarrow x$ is still $y \leftarrow x \cup [10; 30]$
  the second $y \leftarrow x$ is now $y \leftarrow x \cup [10; 10]$

  these interferences are caused by data-races
- the last write to x before unlock
  influences all reads from x between lock and update of x
  $\Longrightarrow$ we transfer the values of x from unlock to lock instruction

  these are well-synchronized interferences

Application: Priority-based scheduling



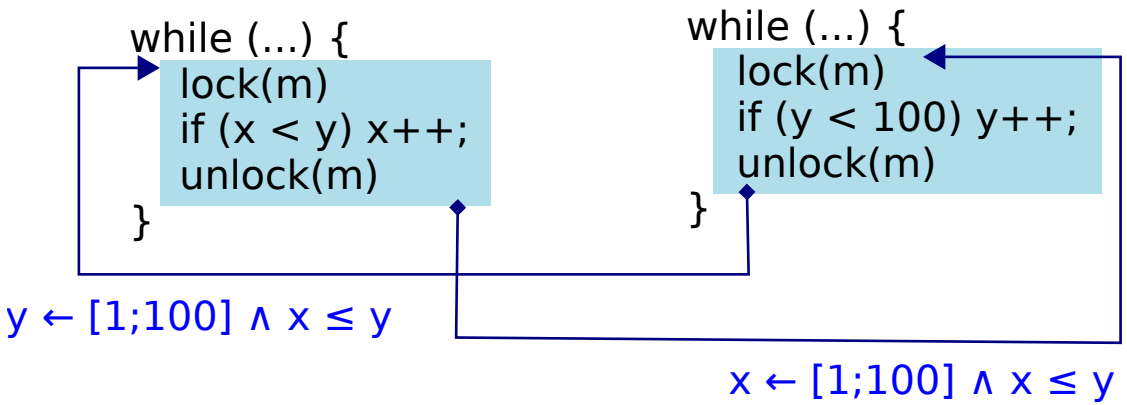
Real-time scheduling:

- priorities are strict (but possibly dynamic)
- a process can only be preempted by a process of strictly higher priority
- a process can block for an indeterminate amount of time (yield, lock)

Analysis: refined transfer of interference based on priority

- partition interferences wrt. thread and priority
  support for manual priority change, and for priority ceiling protocol
- higher priority processes inject state from yield into every point
- lower priority processes inject data-race interferences into yield

Application: Relational lock invariants



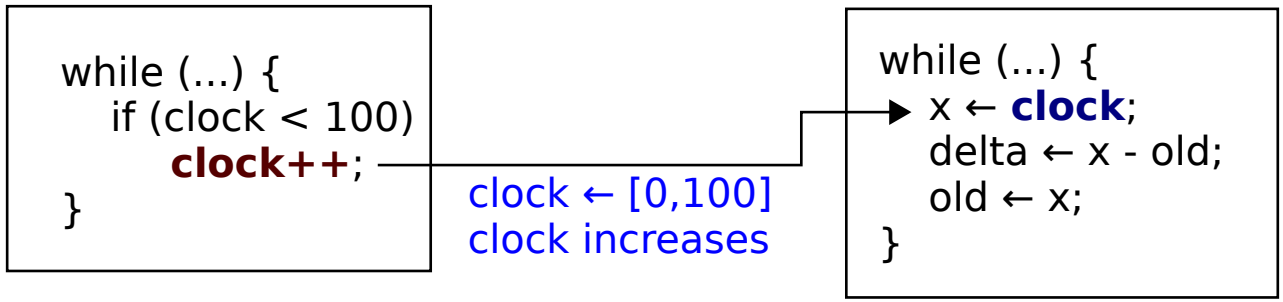
Idea: use (costly) relational interferences only at lock instructions

Rationale: locks often protect important, complex invariants

- data-race interference unchanged (here, $\emptyset$, as there is no data-race)
- well-synchronized interferences now carry:
  - a set of written values
  - a state property left invariant by the block
    (intersection of state at lock and at unlock point)

we don't keep input/output relation

Application: Monotonicity interference



Idea: specialized domain to keep simple input/output relations

- clock is only increased (i.e., monotonic)
  - easy to infer (check all assignments)
  - easy to represent (one flow-insensitive flag per variable)
  - easy to exploit: new value of clock - old value of clock ≥ 0

very common pattern in control-command software

Weak memory consistency

Multi-core CPU and optimizing compilers enforce weak memory consistency

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

Soundness argument: on a per abstraction basis

- simple interferences: sound for reordering of independent R/W
  (includes PSO, TSO, traditional compiler optimization)
- monotonicity abstraction: sound for TSO & PSO
- relational lock invariants: sound for DRF guarantee
  if no data-race!
  (includes C, C++, Java)
- full relational interferences: sound for SC only

Application to AstréeA

monotonicity domain	relational lock invariants	analysis time	memory	iterations	alarms
×	×	25h 26mn	22 GB	6	4616
✓	×	30h 30mn	24 GB	7	1100
✓	✓	110h 38mn	90 GB	7	1009

We only integrated into AstréeA a part of the proposed abstractions

Still scalability concerns with relational lock invariants (packing needed)

Reminder: embedded ARINC 653 C application with 15 threads, 2 Mlines

Conclusion

Summary

We proposed a static analysis framework for concurrent programs:

- sound for all interleavings
  and in some cases weakly consistent memories
- thread-modular
  scalable, ability to reuse existing analyzers
- parameterized by abstract domains
  ability to reuse existing domains
- constructed by abstraction of a complete method
  enable refinement to arbitrary precision
- presented several abstraction instances (relational, flow-sensitive)

Future work:

- specialization of state and interference domains for AstréeA
- bridge the gap between full relational and non-relational inteferences
- bridge the gap between arbitrary preemption and sequentializable
  flow-sensitive or even history-sensitive interference abstraction, e.g.: initialization

Perspectives: Open-source static analysis

Static analysis by abstract interpretation

- has had some success in verification
- is becoming more popular, especially in industry
- but there are still some limits to its adoption;

  tools do not offer at the same time:

  - generality
  - soundness
  - scalability
  - precision

MOPSA Project (2016–2021)

- develop a new open-source platform for static analysis
- sound, modular, extensible, multi-language, multi-abstraction
- demonstrate on the analysis of important open-source software (e.g., name servers)
- provide a platform for research and collaboration
- provide a usable tool and increase awareness for developers

Thank you