Static Analysis by Abstract Interpretation and Decision Procedures

Julien Henry

October 13, 2014

Jury

David Monniaux	Director	CNRS
Matthieu Moy	Co-Advisor	Grenoble INP
Antoine Miné	Reviewer	ENS Paris
Cesare Tinelli	Reviewer	University of Iowa
Hugues Cassé	Examiner	IRIT
Roland Groz	Examiner	Grenoble INP
Andreas Podelski	Examiner	University of Freiburg

Static Analysis

- Used in Embedded and Safety Critical Systems (Astrée)
- Require strong guarantees that programs behave correctly


Principle of **Static Analysis**:

Look at the source code
Discover properties on programs that <u>always</u> hold (**invariants**)

[Spoiler Alert] - PAGAI Screenshot

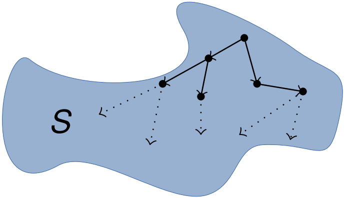
File Edit View Search Terminal Help

```
int bicycle() {
  int count=0, phase=0;
  for(int i=0; i<10000; i++) {
    if (phase == 0) {
      count += 2; phase = 1;
    } else if (phase == 1) {
      count += 1; phase = 0;
    }
  }
  assert(count <= 15000);
  return count;
}
```

File Edit View Search Terminal Help

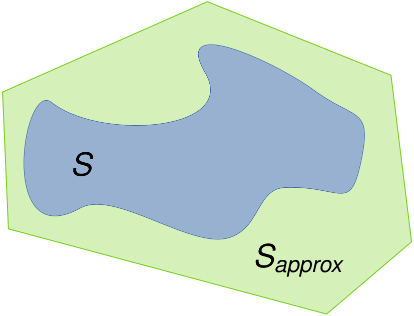
```
int bicycle() {
  int /* reachable */
      count=0, phase=0;
  for(int i=0; i<10000; // safe
                      i++) {
    /* invariant:
    -2*count+phase+3*i = 0
    14998-count+phase >= 0
    1-phase >= 0
    phase >= 0
    count-2*phase >= 0
    */
    if (phase == 0) {
      // safe
      count += 2; phase = 1;
    } else if (phase == 1) {
      // safe
      count += 1; phase = 0;
    }
  }
  /* assert OK */
  assert(count <= 15000);
  /* invariant:
  -15000+count = 0
  */
  return count;
}
```

Static Analysis Uses Over-Approximations



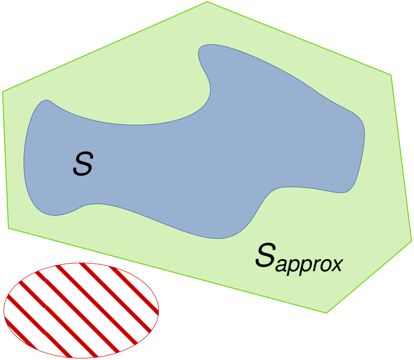
Rice's Theorem: computing the exact blue set is undecidable

Static Analysis Uses Over-Approximations



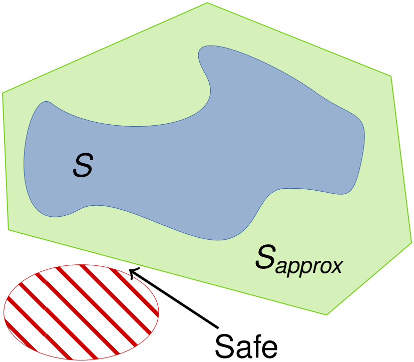
Rice's Theorem: computing the exact blue set is undecidable

Static Analysis Uses Over-Approximations



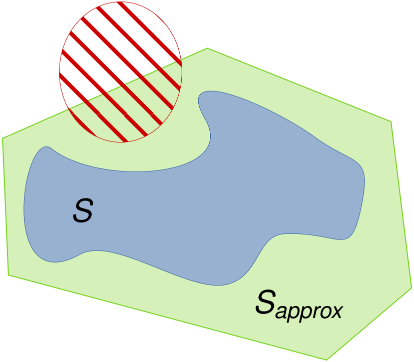
Rice's Theorem: computing the exact blue set is undecidable

Static Analysis Uses Over-Approximations



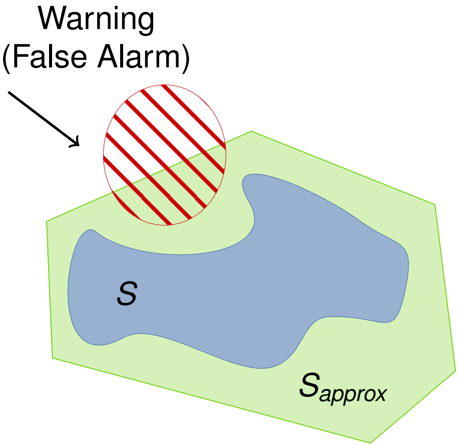
Rice's Theorem: computing the exact blue set is undecidable

Static Analysis Uses Over-Approximations



Rice's Theorem: computing the exact blue set is undecidable

Static Analysis Uses Over-Approximations



Rice's Theorem: computing the exact blue set is undecidable

Several Approaches to Formal Verification

Model Checking

Explores the "exact" set of states

Precise
Costly

↔

Abstract Interpretation

Compute "tractable" over-approximations

Fast
Imprecise

Several Approaches to Formal Verification

Model Checking

Explores the “exact” set of states

Precise
Costly

Abstract Interpretation

Compute “tractable” over-approximations

Fast
Imprecise

This Thesis

Use Model Checking techniques in Abstract Interpretation

Summary

1. Introduction
2. Improving Abstract Interpretation with Decision Procedures
3. Modular Static Analysis
4. Implementation: The PAGAI Static Analyzer
5. Application: Bounding Worst-Case Execution Time (WCET)

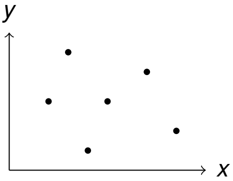
Summary

1. Introduction
2. Improving Abstract Interpretation with Decision Procedures
3. Modular Static Analysis
4. Implementation: The PAGAI Static Analyzer
5. Application: Bounding Worst-Case Execution Time (WCET)

Abstract Interpretation

[Cousot & Cousot, 1977]

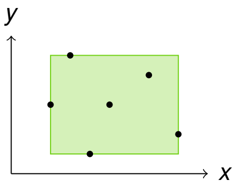
Abstract domain to over-approximate sets of states:



Abstract Interpretation

[Cousot & Cousot, 1977]

Abstract domain to over-approximate sets of states:



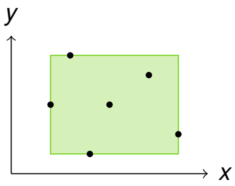
BOXES:

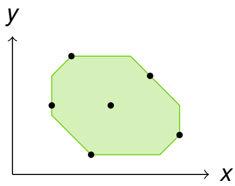
$4 \leq x \leq 17$
$2 \leq y \leq 12$

Abstract Interpretation

[Cousot & Cousot, 1977]

Abstract domain to over-approximate sets of states:





BOXES:

$$4 \leq x \leq 17$$
$$2 \leq y \leq 12$$

OCTAGONS:

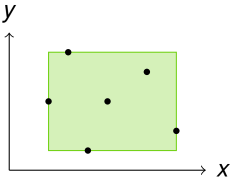
$$\begin{array}{rcl} 10 \leq & x+y & \leq 24 \\ -6 \leq & x-y & \leq 13 \\ 4 \leq & x & \leq 17 \\ 2 \leq & y & \leq 12 \end{array}$$

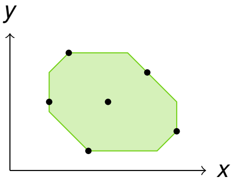
[MinéPhD04]

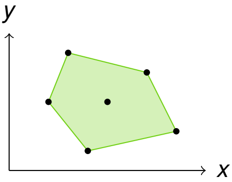
Abstract Interpretation

[Cousot & Cousot, 1977]

Abstract domain to over-approximate sets of states:







BOXES:

$$4 \leq x \leq 17$$
$$2 \leq y \leq 12$$

OCTAGONS:

$$\begin{array}{rcl} 10 \leq & x+y & \leq 24 \\ -6 \leq & x-y & \leq 13 \\ 4 \leq & x & \leq 17 \\ 2 \leq & y & \leq 12 \end{array}$$

[MinéPhD04]

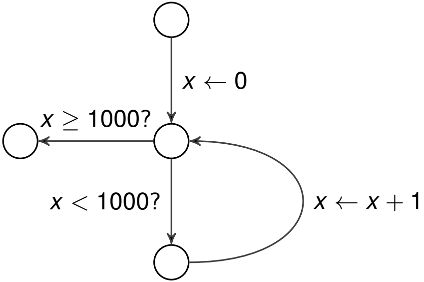
CONVEX POLYHEDRA:

$$\begin{array}{rcl} 5x-2y & \geq & 6 \\ 2x+y & \leq & 38 \\ & \vdots & \end{array}$$

[CH78]

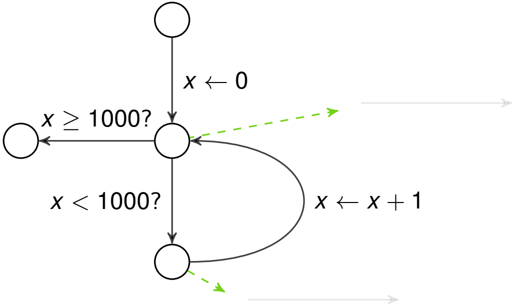
Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```



Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

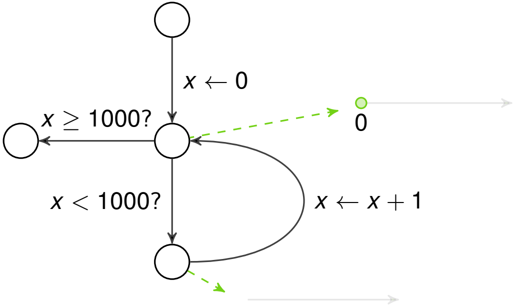


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

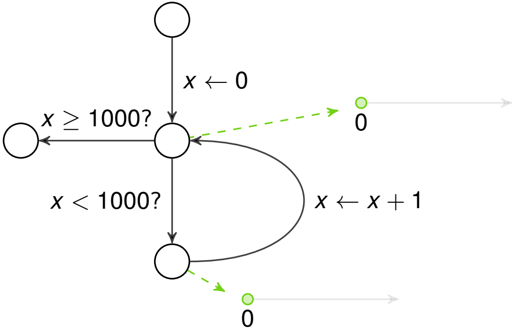


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

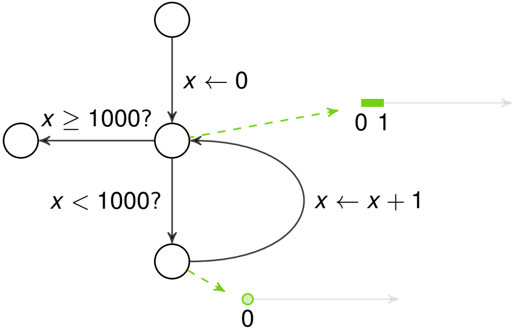


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

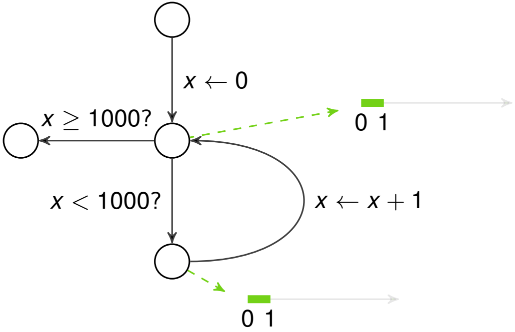


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

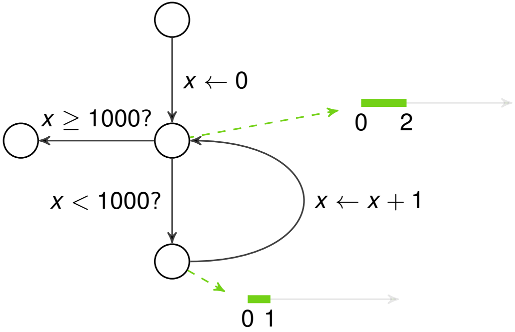


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

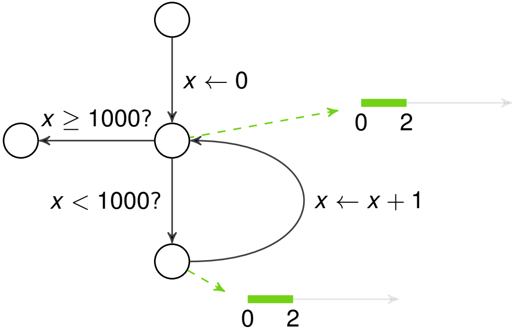


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

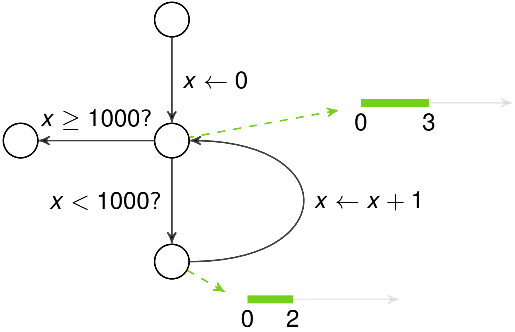


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

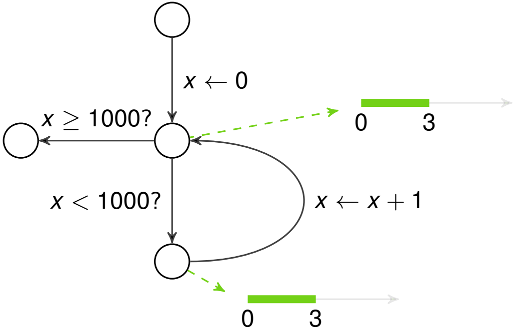


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

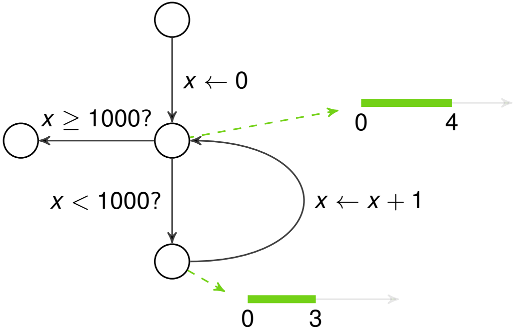


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

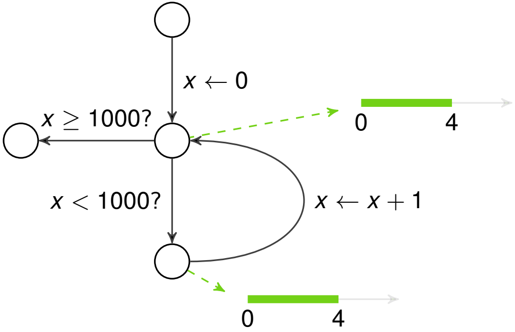


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

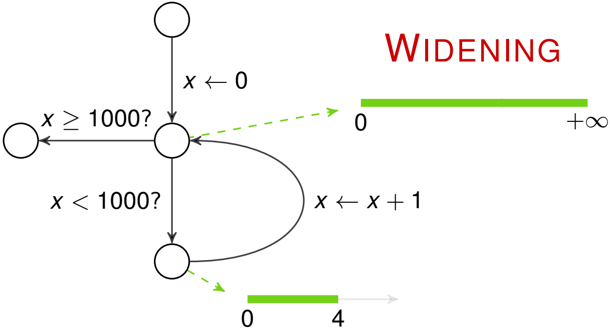


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

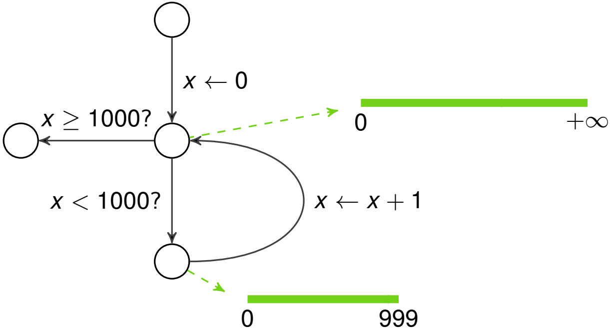


Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Abstract Interpretation

```
x = 0;
while (x < 1000)
{
    x++;
}
```

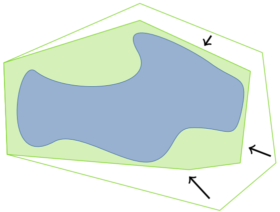


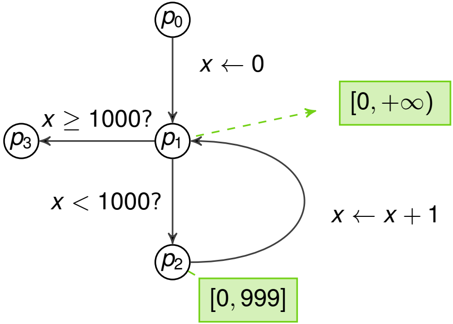
Fixpoint computation:

- All abstract values (**intervals**) initialized to $\emptyset$
- Update until there is no more element to add

Descending Sequence

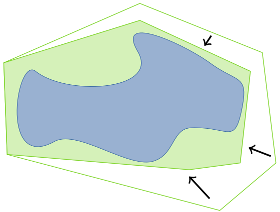
Recover precision **after** an invariant is reached

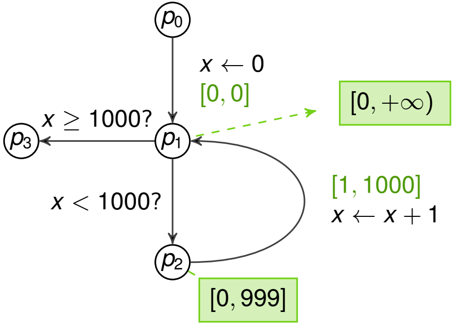




Descending Sequence

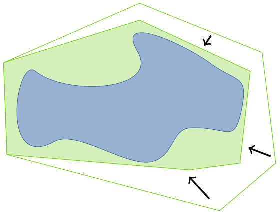
Recover precision **after** an invariant is reached

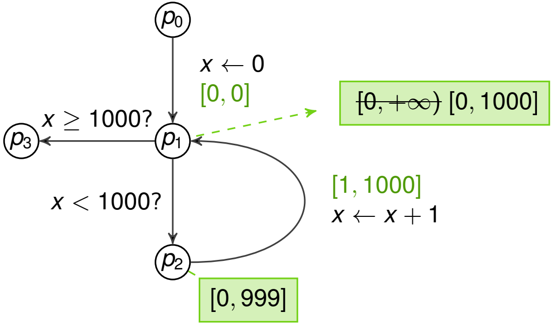




Descending Sequence

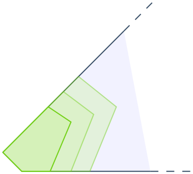
Recover precision **after** an invariant is reached





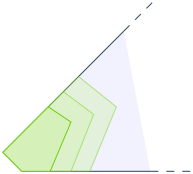
Some Sources of Imprecision

- Widening operator
  - Ensures termination, degrades precision
  - Descending sequence sometimes helps…

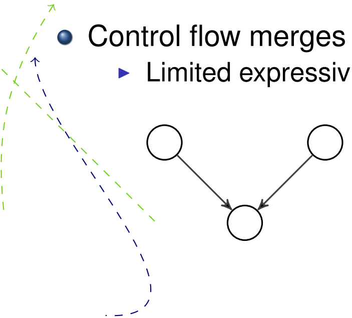


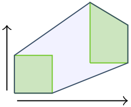
Some Sources of Imprecision

- Widening operator
  - Ensures termination, degrades precision
  - Descending sequence sometimes helps…



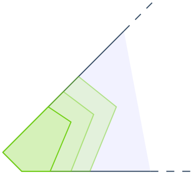
- Control flow merges
  - Limited expressivity of the abstract domain



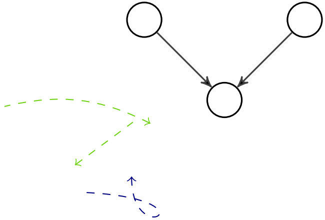


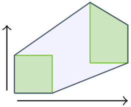
$\bigsqcup$ = Least Upper Bound

Some Sources of Imprecision



- **Widening operator**
  - Ensures termination, degrades precision
  - Descending sequence sometimes helps. . .
- **Control flow merges**
  - Limited expressivity of the abstract domain





$\bigsqcup$ = Least Upper Bound

In this thesis

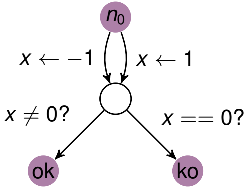
Limit the bad effects of **widenings** and **least upper bounds**

Summary

1. Introduction
2. Improving Abstract Interpretation with Decision Procedures
3. Modular Static Analysis
4. Implementation: The PAGAI Static Analyzer
5. Application: Bounding Worst-Case Execution Time (WCET)

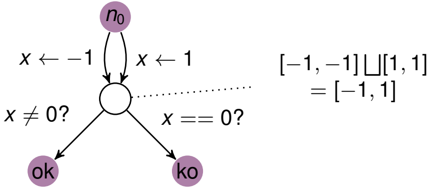
Least Upper Bound yields Imprecision

```
if (input())
    x = 1;
else
    x = -1;
    // (here)
if (x == 0)
    error();
y = 1 / x;
```



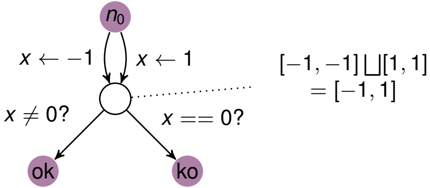
Least Upper Bound yields Imprecision

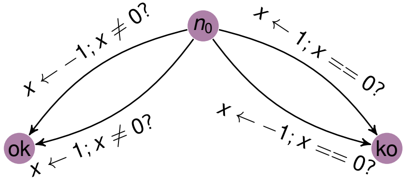
```
if (input())
    x = 1;
else
    x = -1;
    // (here)
if (x == 0)
    error();
y = 1 / x;
```



Least Upper Bound yields Imprecision

```
if (input())
    x = 1;
else
    x = -1;
    // (here)
if (x == 0)
    error();
y = 1 / x;
```



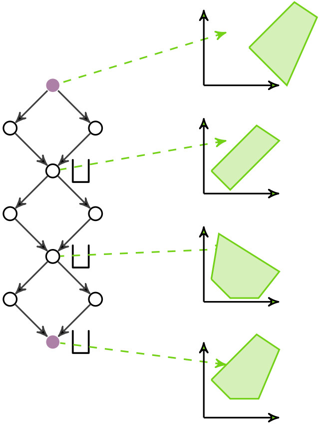


Trace Partitioning [Mauborgne & Rival]
Large Block Encoding [Beyer & al.]

Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

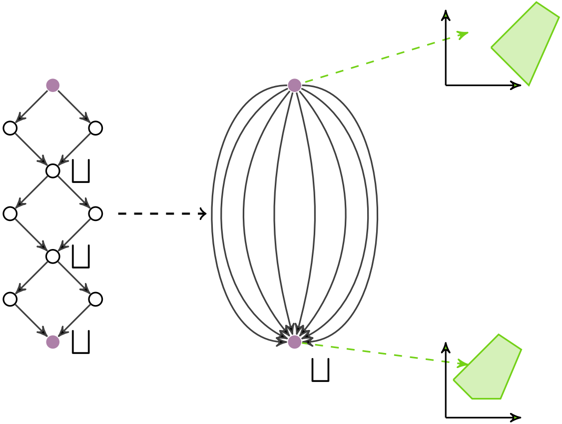
Idea: delay control-flow merges



Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

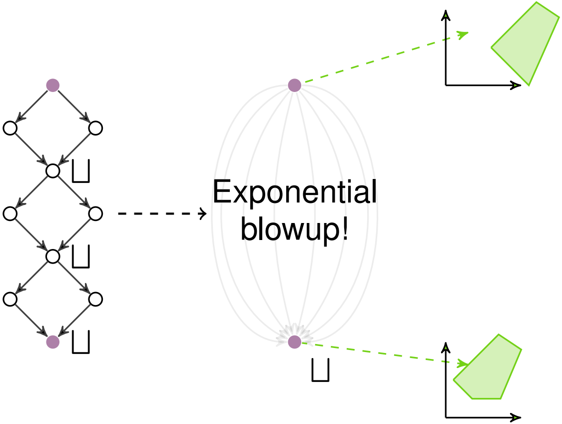
Idea: delay control-flow merges



Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

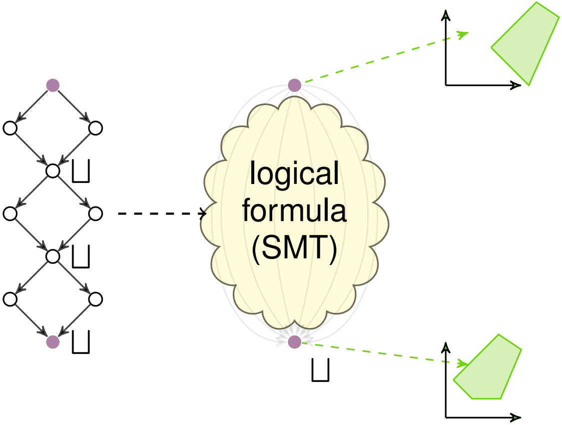
Idea: delay control-flow merges



Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Idea: delay control-flow merges



Satisfiability Modulo Theory (SMT)

Boolean SATISFIABILITY (SAT):

$$b_1 \wedge ((b_2 \wedge b_3) \vee (b_4))$$

Satisfiability Modulo Theory (SMT)

Boolean SATISFIABILITY (SAT):

$$b_1 \wedge ((b_2 \wedge b_3) \vee (b_4))$$

$$x \geq 0 \wedge ((y \geq x + 10 \wedge b_3) \vee (x + 1 \geq 0))$$

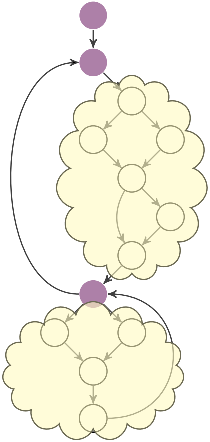
MODULO THEORY: Atoms can be interpreted in a given decidable theory

e.g. Linear Integer Arithmetic

Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

In practice: distinguish every paths **between loop heads**

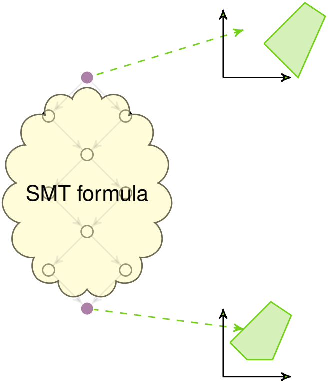


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only "interesting" paths are those that make this invariant computation **progress**.

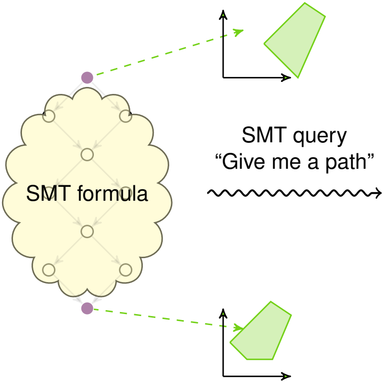


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only “interesting” paths are those that make this invariant computation **progress**.

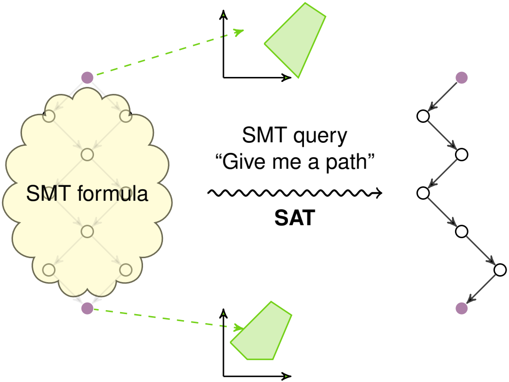


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only "interesting" paths are those that make this invariant computation **progress**.

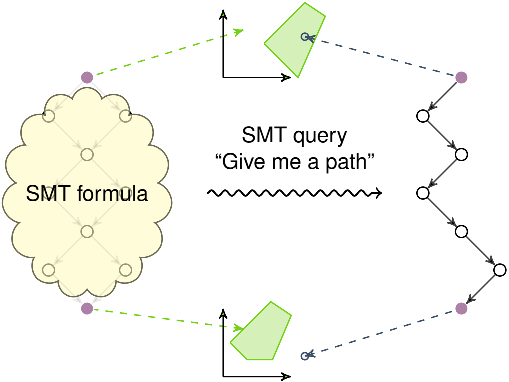


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only "interesting" paths are those that make this invariant computation **progress**.

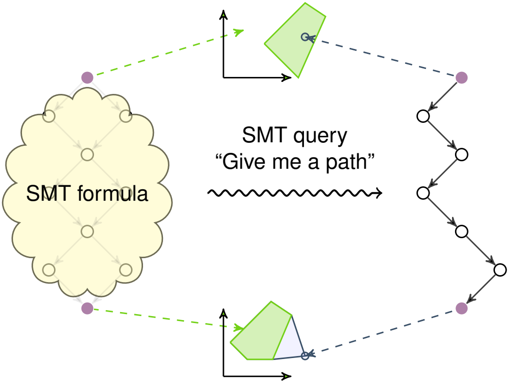


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only “interesting” paths are those that make this invariant computation **progress**.

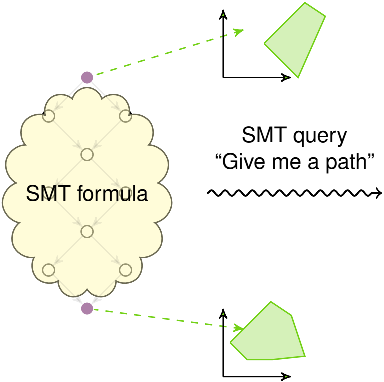


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only “interesting” paths are those that make this invariant computation **progress**.

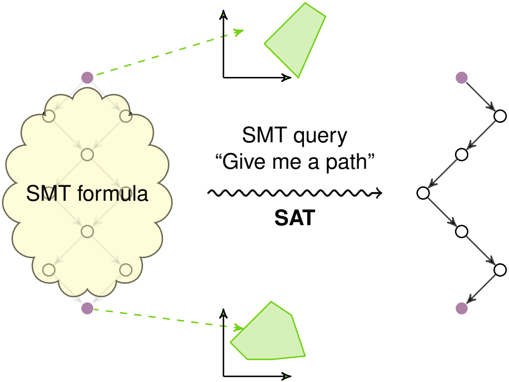


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only “interesting” paths are those that make this invariant computation **progress**.

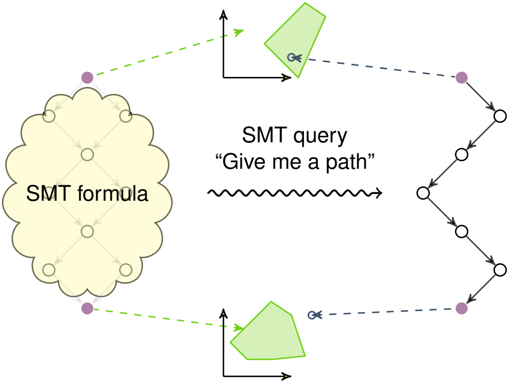


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only "interesting" paths are those that make this invariant computation **progress**.

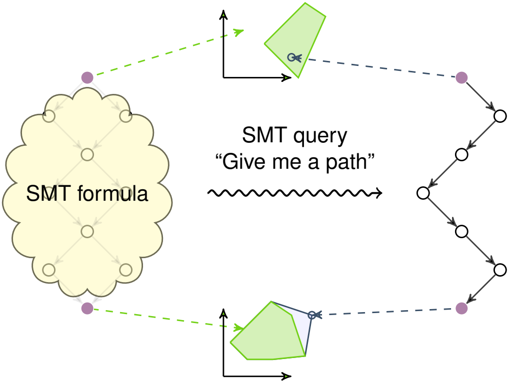


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only “interesting” paths are those that make this invariant computation **progress**.

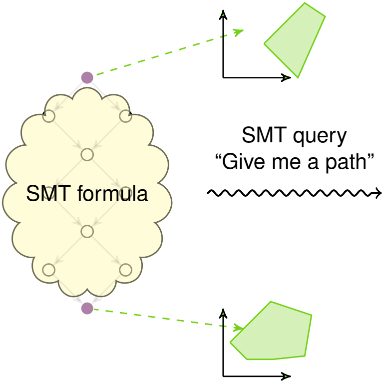


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only "interesting" paths are those that make this invariant computation **progress**.

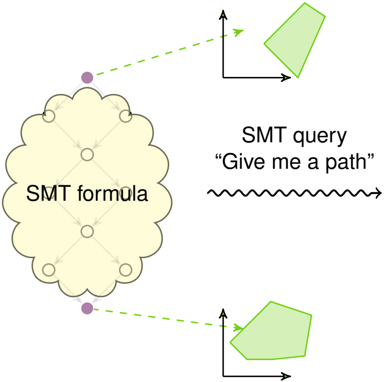


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only “interesting” paths are those that make this invariant computation **progress**.

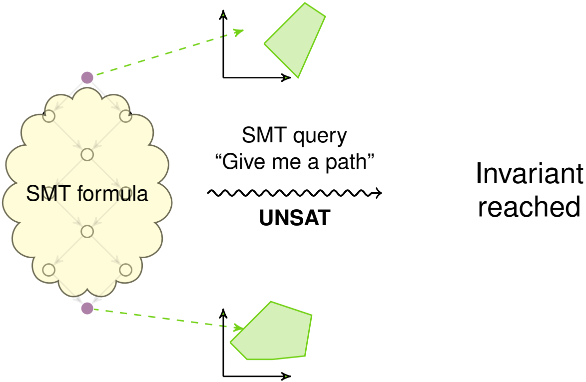


Path Focusing

[Monniaux & Gonnord, SAS11], [Henry & al, TAPAS12]

Usual algorithm: update an abstract value until it is an inductive invariant.

The only “interesting” paths are those that make this invariant computation **progress**.



CONTRIBUTION: Guided Path Analysis

[Henry & Monniaux & Moy, SAS12]

Observation: Imprecision due to widening spreads

Intuition: Widenings might enable paths that were previously infeasible

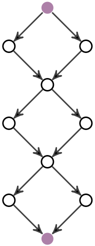
```
x = 0;
while (x < 1000) {
    if (x > 2000) { ... }
    x++;
}
```

[Gopan & Reps, SAS07] :

- Do not consider these “spurious” transitions
- Eliminate them using **descending sequences**

CONTRIBUTION: Guided Path Analysis

[Henry & Monniaux & Moy, SAS12]



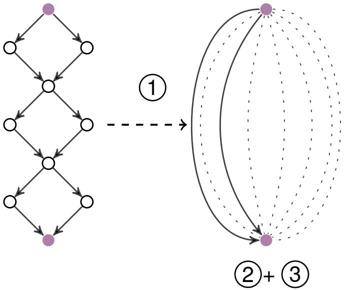
Compute **precise** invariants for a sequence of sub-programs

ALGORITHM:

1. Choose sub-program (= set of paths directly feasible)
2. Ascending iterations
3. **Descending iterations**

CONTRIBUTION: Guided Path Analysis

[Henry & Monniaux & Moy, SAS12]



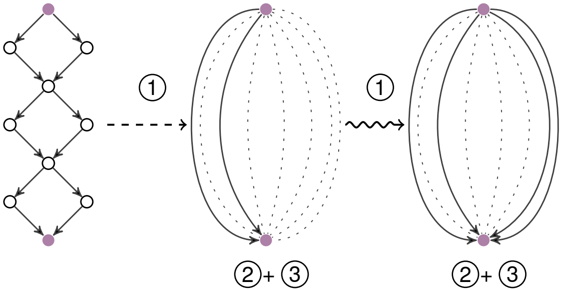
Compute **precise** invariants for a sequence of sub-programs

ALGORITHM:

1. Choose sub-program (= set of paths directly feasible)
2. Ascending iterations
3. **Descending iterations**

CONTRIBUTION: Guided Path Analysis

[Henry & Monniaux & Moy, SAS12]



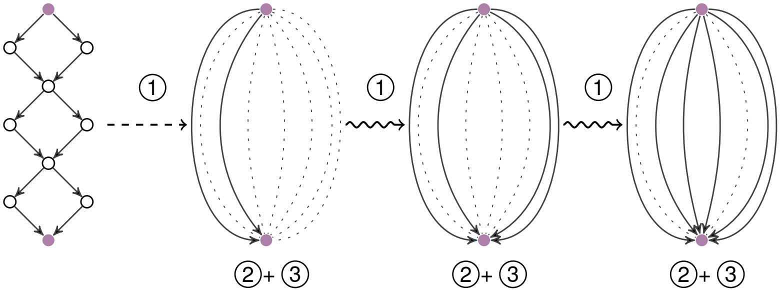
Compute **precise** invariants for a sequence of sub-programs

ALGORITHM:

1. Choose sub-program (= set of paths directly feasible)
2. Ascending iterations
3. **Descending iterations**

CONTRIBUTION: Guided Path Analysis

[Henry & Monniaux & Moy, SAS12]



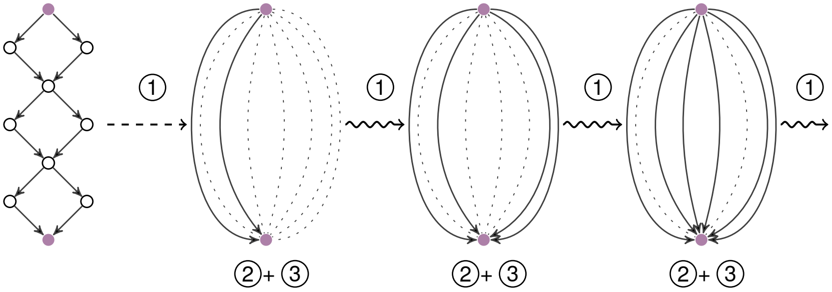
Compute **precise** invariants for a sequence of sub-programs

ALGORITHM:

1. Choose sub-program (= set of paths directly feasible)
2. Ascending iterations
3. **Descending iterations**

CONTRIBUTION: Guided Path Analysis

[Henry & Monniaux & Moy, SAS12]



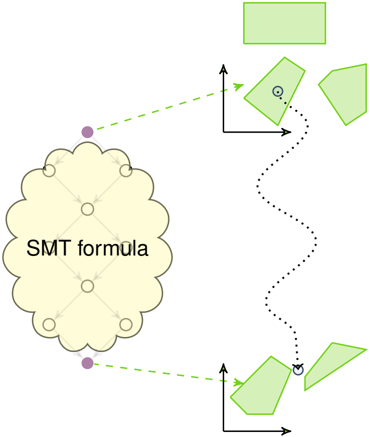
Compute **precise** invariants for a sequence of sub-programs

ALGORITHM:

1. Choose sub-program (= set of paths directly feasible)
2. Ascending iterations
3. **Descending iterations**

Extension 1: Disjunctive Invariants

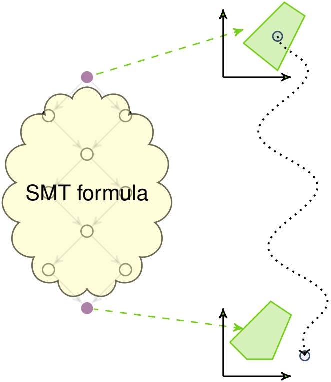
Allow disjunctions of abstract values



Extension 2: Less SMT Queries

In the worst case, 2^n paths $\Longrightarrow$ exponential number of SMT queries

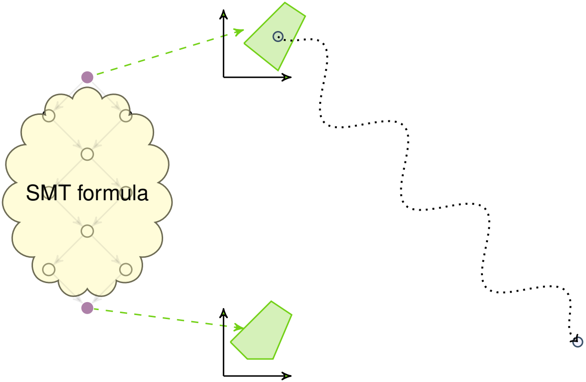
"Interesting traces" **far** from the current abstract value



Extension 2: Less SMT Queries

In the worst case, 2^n paths $\Longrightarrow$ exponential number of SMT queries

“Interesting traces” **far** from the current abstract value



Intermediate Conclusion

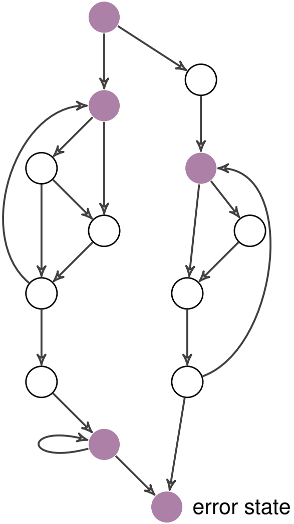
- Abstract Interpretation can be parametrized in many ways
- From very cheap to very expensive
  - fixpoint computation techniques
  - abstract domains

Run cheap techniques first, and refine program portions if needed (CEGAR)

Summary

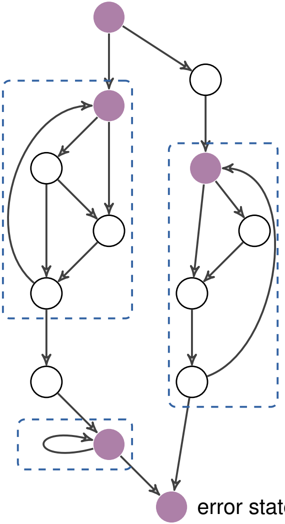
1. Introduction
2. Improving Abstract Interpretation with Decision Procedures
3. Modular Static Analysis
4. Implementation: The PAGAI Static Analyzer
5. Application: Bounding Worst-Case Execution Time (WCET)

CONTRIBUTION: Modular Static Analysis



Input:
Complicated CFG

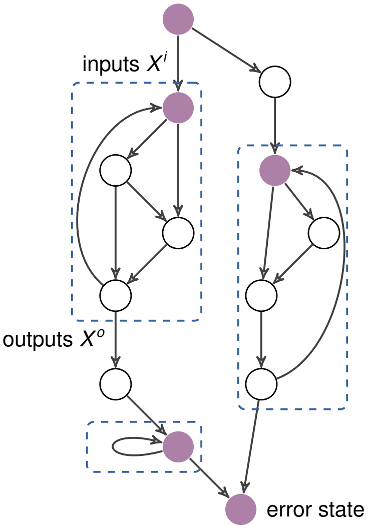
CONTRIBUTION: Modular Static Analysis



Select blocks/portions to be abstracted:

- Loops,
- Function calls,
- Complicated program portions

CONTRIBUTION: Modular Static Analysis

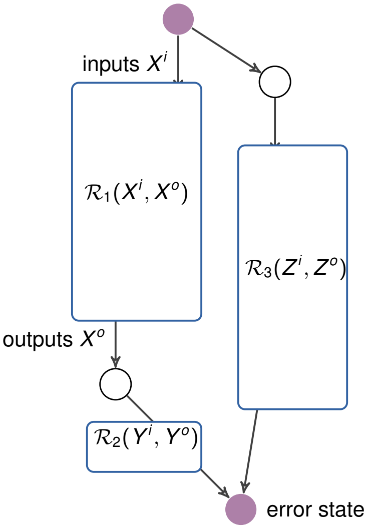


Select blocks/portions to be abstracted:

- Loops,
- Function calls,
- Complicated program portions

Each block has **input** and **output** variables

CONTRIBUTION: Modular Static Analysis

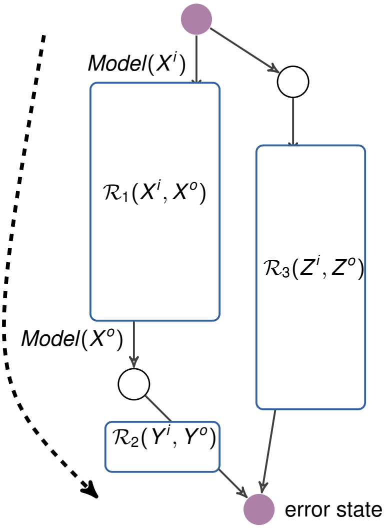


Abstract each block with a logical formula.

$\mathcal{R}_1(x^i, x^o)$ involves **inputs** and **outputs**
Example: $x^i > 0 \Rightarrow x^o = x^i + 1$

$\mathcal{R}_i$ initialized to **true**
(= safe over-approximation)

CONTRIBUTION: Modular Static Analysis



SMT query:

"Is there a path to the error state ?"

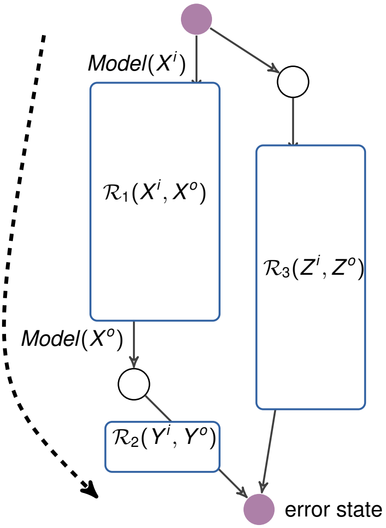
YES:

$Model(x^i) = (x_i = 10)$
$Model(x^o) = (x_o = 12)$

$$x_i = 10 \wedge \mathcal{R}_1(x^i, x^o) \wedge x_o = 12$$

is SAT

CONTRIBUTION: Modular Static Analysis



SMT query:

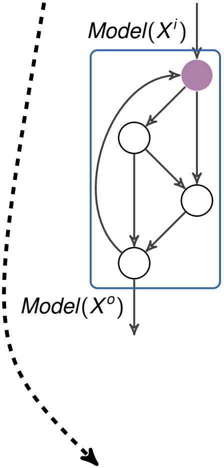
"Is there a path to the error state ?"

YES:
$Model(x^i) = (x_i = 10)$
$Model(x^o) = (x_o = 12)$

$$x_i = 10 \wedge \mathcal{R}_1(x^i, x^o) \wedge x_o = 12$$
is SAT

$\rightarrow$ Improve precision of $\mathcal{R}_1(x^i, x^o)$ s.t. the formula becomes UNSAT

CONTRIBUTION: Modular Static Analysis

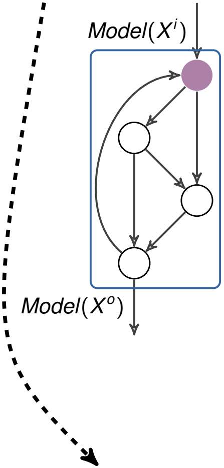


Compute a new relation input context $x_i = 10$

Example:

$$x_i = 10 \Rightarrow x_o \leq x_i$$

CONTRIBUTION: Modular Static Analysis



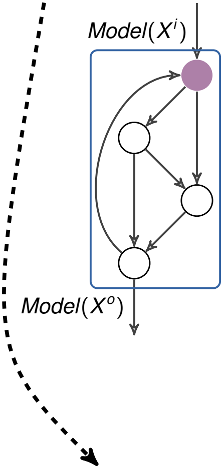
Compute a new relation input context $x_i = 10$

Example:

$$x_i = 10 \Rightarrow x_o \leq x_i$$

Not very general...

CONTRIBUTION: Modular Static Analysis



Compute a new relation
input context $x_i = 10$

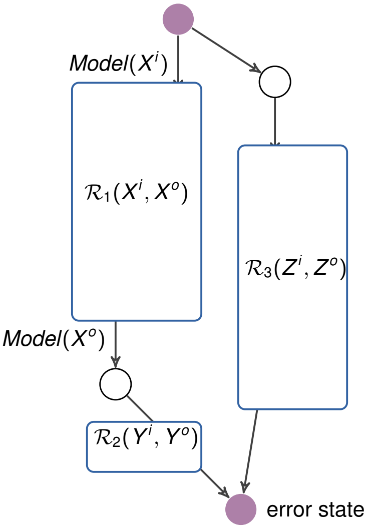
Example:

$x_i = 10 \Rightarrow x_o \leq x_i$

$x_i > 0 \Rightarrow x_o \leq x_i$

$x_i = 10 \wedge (x_o \leq x_i) \wedge x_o = 12$
no more possible

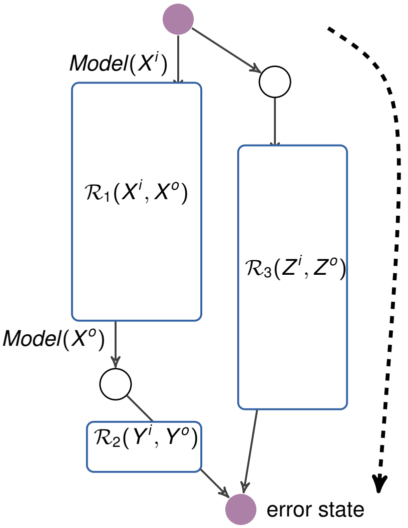
CONTRIBUTION: Modular Static Analysis



Search for new error trace

Until no trace is found...

CONTRIBUTION: Modular Static Analysis



Search for new error trace

Until no trace is found. . .

Summary

1. Introduction
2. Improving Abstract Interpretation with Decision Procedures
3. Modular Static Analysis
4. Implementation: The PAGAI Static Analyzer
5. Application: Bounding Worst-Case Execution Time (WCET)

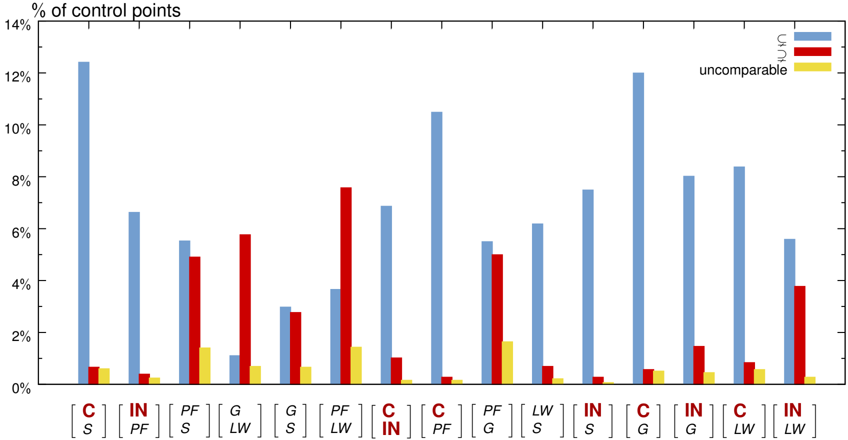
CONTRIBUTION: PAGAI Static Analyzer

[Henry & Monniaux & Moy, TAPAS12]

Static analyzer for C/C++/Ada/Fortran/...
Written in C++, > 20,000 LOC

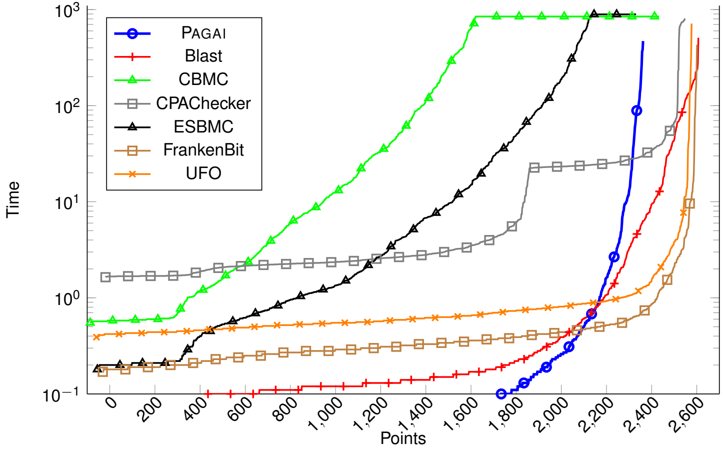
- Uses the LLVM compiler infrastructure
- Most approaches described here are implemented
- **Numerical invariants**
- PAGAI checks:
  - array out-of-bound accesses
  - integer overflows
  - `assert` over numerical variables
- Handles **real-life** programs
- Already used outside Verimag (Spain, India, ...)

Comparisons of Various Techniques



Experiments on GNU projects: libgsl, libjpeg, libpng, gnugo, tar, ...

Software-Verification Competition



Summary

1 Introduction

2 Improving Abstract Interpretation with Decision Procedures

3 Modular Static Analysis

4 Implementation: The PAGAI Static Analyzer

5 Application: Bounding Worst-Case Execution Time (WCET)

Target: Reactive Control Systems

```
void main() {
    while (1) {
        READ_INPUTS();
        COMPUTE();
        WRITE_OUTPUTS();
    }
}
```

1 “big” infinite loop

$\sim$ Loop-free body

Goal: WCET for 1 loop iteration $<$ some bound

CONTRIBUTION: Estimating WCET using SMT

[Henry & Asavoae & Monniaux & Maiza, LCTES14]

Input:

- **Loop-free** control-flow graph of the loop body
- local timings for basic blocks (# clock cycles)
  - given by an external tool, e.g. OTAWA
  - runs a panel of static analysis, sensitive to micro-architecture

Principle: Encode the problem into SMT and optimize a cost function

Output:

- WCET for the entire CFG + Worst Case path

Computing the WCET

Optimization modulo Theory:
We search for the trace maximizing the variable cost.
cost = execution time for the trace

Using any off-the-shelf SMT solver

Computing the WCET

Optimization modulo Theory:
We search for the trace maximizing the variable cost.
cost = execution time for the trace

Using any off-the-shelf SMT solver

Binary Search strategy:
Maintain an interval containing the WCET

- Initial interval $[0, 100]$

Computing the WCET

Optimization modulo Theory:
We search for the trace maximizing the variable cost.
cost $=$ execution time for the trace

Using any off-the-shelf SMT solver

Binary Search strategy:
Maintain an interval containing the WCET

- Initial interval $[0, 100]$
- Is there a trace where *cost* > 50? Yes, 70

Computing the WCET

Optimization modulo Theory:
We search for the trace maximizing the variable cost.
cost = execution time for the trace

Using any off-the-shelf SMT solver

Binary Search strategy:
Maintain an interval containing the WCET

- Initial interval $[0, 100]$
- Is there a trace where *cost* > 50? Yes, 70
- new interval $[70, 100]$

Computing the WCET

Optimization modulo Theory:
We search for the trace maximizing the variable cost.
cost $=$ execution time for the trace

Using any off-the-shelf SMT solver

Binary Search strategy:
Maintain an interval containing the WCET

- Initial interval $[0, 100]$
- Is there a trace where *cost* $>$ 50? Yes, 70
- new interval $[70, 100]$
- Is there a trace where *cost* $>$ 85? No

Computing the WCET

Optimization modulo Theory:
We search for the trace maximizing the variable cost.
cost $=$ execution time for the trace

Using any off-the-shelf SMT solver

Binary Search strategy:
Maintain an interval containing the WCET

- Initial interval $[0, 100]$
- Is there a trace where $cost > 50$? Yes, 70
- new interval $[70, 100]$
- Is there a trace where $cost > 85$? No
- new interval $[70, 85]$

Computing the WCET

Optimization modulo Theory:
We search for the trace maximizing the variable cost.
cost $=$ execution time for the trace

Using any off-the-shelf SMT solver

Binary Search strategy:
Maintain an interval containing the WCET

- Initial interval $[0, 100]$
- Is there a trace where $\mathit{cost} > 50$? Yes, 70
- new interval $[70, 100]$
- Is there a trace where $\mathit{cost} > 85$? No
- new interval $[70, 85]$
- ...

Computing the WCET

Optimization modulo Theory:
We search for the trace maximizing the variable <u>cost</u>.
<u>cost</u> $=$ execution time for the trace

Using any off-the-shelf SMT solver

Binary Search strategy:
Maintain an interval containing the WCET

- Initial interval $[0, 100]$
- Is there a trace where $cost > 50$? Yes, 70
- new interval $[70, 100]$
- Is there a trace where $cost > 85$? No
- new interval $[70, 85]$
- $\ldots$

Approach Fails on Simple Examples

$b_1, \ldots, b_n$ unconstrained Booleans, **ci** and **ci'** are the timing costs

```
if (b1) { /*c1=2*/ } else { /*c1=3*/ } //cost c1
if (b1) { /*c1'=3*/ } else { /*c1'=2*/ } //cost c1'
...
if (bn) { /*cn=2*/ } else { /*cn=3*/ } //cost cn
if (bn) { /*cn'=3*/ } else { /*cn'=2*/ } //cost cn'
```

"Obviously" all traces take time $(3 + 2)n = 5n$.

Approach Fails on Simple Examples

$b_1, \ldots, b_n$ unconstrained Booleans, **ci** and **ci'** are the timing costs

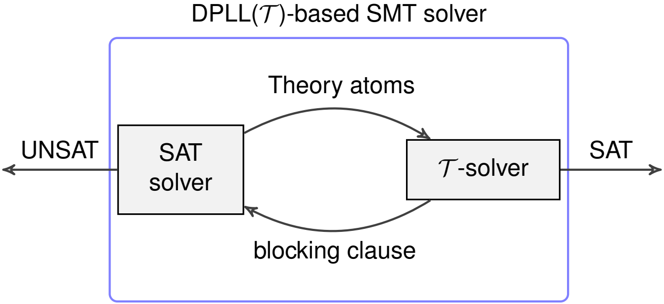
```
if (b1) { /*c1=2*/ } else { /*c1=3*/ } //cost c1
if (b1) { /*c1'=3*/ } else { /*c1'=2*/ } //cost c1'
...
if (bn) { /*cn=2*/ } else { /*cn=3*/ } //cost cn
if (bn) { /*cn'=3*/ } else { /*cn'=2*/ } //cost cn'
```

"Obviously" all traces take time $(3 + 2)n = 5n$.

SMT approach (using DPLL($\mathcal{T}$)) will find 5n, but in exponential time. . .

Why such high cost?

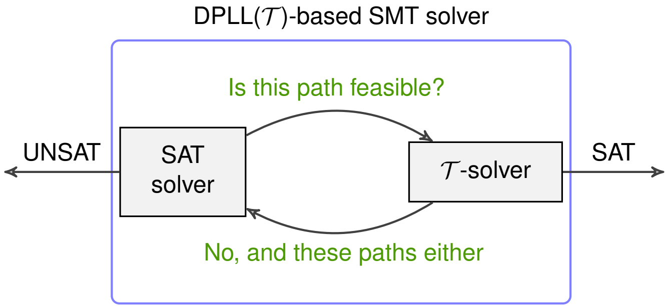
SMT solver relaxes the SMT formula into a Boolean abstraction



DPLL($\mathcal{T}$) : [Nieuwenhuis & Oliveras & Tinelli, JACM06]

Why such high cost?

SMT solver relaxes the SMT formula into a Boolean abstraction



DPLL($\mathcal{T}$) : [Nieuwenhuis & Oliveras & Tinelli, JACM06]

Blocking Clause: **Simple** reason why it is UNSAT

Theory Atoms

$c_1 \leq 2$ $\cdots$ $c_n \leq 2$

$c_1 \leq 3$ $\cdots$ $c_n \leq 3$

$\neg(c'_1 \leq 2)$ $\cdots$ $\neg(c'_n \leq 2)$

$c'_1 \leq 3$ $\cdots$ $c'_n \leq 3$

$c_1 + c'_1 + \cdots + c_n + c'_n > 5n$

Blocking Clause?

BLOCKING CLAUSE: **Simple** reason why it is UNSAT

THEORY ATOMS			BLOCKING CLAUSE		
$c_1 \leq 2$		$c_n \leq 2$	$c_1 \leq 2$		$c_n \leq 2$
$c_1 \leq 3$	$\cdots$	$c_n \leq 3$	~~$c_1 \leq 3$~~	$\cdots$	~~$c_n \leq 3$~~
$\neg(c'_1 \leq 2)$		$\neg(c'_n \leq 2)$	~~$\neg(c'_1 \leq 2)$~~		~~$\neg(c'_n \leq 2)$~~
$c'_1 \leq 3$		$c'_n \leq 3$	$c'_1 \leq 3$		$c'_n \leq 3$
$c_1 + c'_1 + \cdots + c_n + c'_n > 5n$			$c_1 + c'_1 + \cdots + c_n + c'_n > 5n$		

Only cuts one single program trace...

BLOCKING CLAUSE: **Simple** reason why it is UNSAT

THEORY ATOMS			BLOCKING CLAUSE		
$c_1 \leq 2$		$c_n \leq 2$	$c_1 \leq 2$		$c_n \leq 2$
$c_1 \leq 3$	$\cdots$	$c_n \leq 3$	~~$c_1 \leq 3$~~	$\cdots$	~~$c_n \leq 3$~~
$\neg(c'_1 \leq 2)$		$\neg(c'_n \leq 2)$	~~$\neg(c'_1 \leq 2)$~~		~~$\neg(c'_n \leq 2)$~~
$c'_1 \leq 3$		$c'_n \leq 3$	$c'_1 \leq 3$		$c'_n \leq 3$
$c_1 + c'_1 + \cdots + c_n + c'_n > 5n$			$c_1 + c'_1 + \cdots + c_n + c'_n > 5n$		

Only cuts one single program trace...
2^n of them. The solver has to prove them inconsistent **one by one**.

Untractability Issue

SMT solvers miss “obvious” properties

```
...
if (b_i) { /* ci=2 */ } else { /* ci=3*/ }
if (b_i) { /* ci'=3 */ } else { /* ci'=2*/ }
...
```

“**Obviously**, $c_i + c_i' \leq 5$” $\rightarrow$ easy for a **static analyzer** !

“Normal*” DPLL($\mathcal{T}$)-based SMT solvers
do not invent new atomic predicates

* : [Nieuwenhuis & Oliveras & Tinelli, JACM06],
[Decision Procedures, Kroening & Strichman]

Untractability Issue

SMT solvers miss "obvious" properties

```
...
if (b_i) { /* ci=2 */ } else { /* ci=3*/ }
if (b_i) { /* ci'=3 */ } else { /* ci'=2*/ }
...
```

"**Obviously**, $c_i + c'_i \leq 5$" $\rightarrow$ easy for a **static analyzer** !

"Normal*" DPLL($\mathcal{T}$)-based SMT solvers
do not invent new atomic predicates

What if we simply conjoin these predicates to the SMT formula ?

* : [Nieuwenhuis & Oliveras & Tinelli, JACM06],
[Decision Procedures, Kroening & Strichman]

Theory Atoms

$c_1 \leq 2$ $\quad\cdots\quad$ $c_n \leq 2$

$c_1 \leq 3$ $\quad\cdots\quad$ $c_n \leq 3$

$\neg(c'_1 \leq 2)$ $\quad\cdots\quad$ $\neg(c'_n \leq 2)$

$c'_1 \leq 3$ $\quad\cdots\quad$ $c'_n \leq 3$

$c_1 + c'_1 \leq 5$ $\quad\cdots\quad$ $c_n + c'_n \leq 5$

$c_1 + c'_1 + \cdots + c_n + c'_n > 5n$

Blocking Clause?

THEORY ATOMS

$c_1 \leq 2$ $\quad$ $c_n \leq 2$

$c_1 \leq 3$ $\cdots$ $c_n \leq 3$

$\neg(c'_1 \leq 2)$ $\quad$ $\neg(c'_n \leq 2)$

$c'_1 \leq 3$ $\quad$ $c'_n \leq 3$

$c_1 + c'_1 \leq 5$ $\quad$ $c_n + c'_n \leq 5$

$c_1 + c'_1 + \cdots + c_n + c'_n > 5n$

BLOCKING CLAUSE

~~$c_1 \leq 2$~~ $\quad$ ~~$c_n \leq 2$~~

~~$c_1 \leq 3$~~ $\cdots$ ~~$c_n \leq 3$~~

~~$\neg(c'_1 \leq 2)$~~ $\quad$ ~~$\neg(c'_n \leq 2)$~~

~~$c'_1 \leq 3$~~ $\quad$ ~~$c'_n \leq 3$~~

$c_1 + c'_1 \leq 5$ $\quad$ $c_n + c'_n \leq 5$

$c_1 + c'_1 + \cdots + c_n + c'_n > 5n$

Prunes all the 2^n traces at once.

Our Solution: a “Better” SMT Encoding

- Distinguish “portions” in the program.
- Compute upper bound B_i on WCET for each portion
- Conjoin these constraints to the previous SMT formula $c_1 + \cdots + c_5 \leq B_1$, $c_6 + \cdots + c_{10} \leq B_2$, etc.
- The obtained formula is **equivalent**
- Do the binary search as before

Solving time from “nonterminating after one night” to “a few seconds”.

Experiments with ARMv7

OTAWA for Basic Block timings
Z3 SMT solver, timeout 8h

Benchmark name	WCET bounds (#cycles) Otawa	SMT	gain	Analysis time (seconds) with cuts	no cuts	#cuts
statemate	3297	3211	2.6%	943.5	$+\infty$	143
nsichneu (1 iteration)	17242	13298	22.7%	≈22000	$+\infty$	378
cruise-control	881	873	0.9%	0.1	0.2	13
digital-stopwatch	1012	954	5.7%	0.6	2104.2	53
autopilot	12663	5734	54.7%	1808.8	$+\infty$	498
fly-by-wire	6361	5848	8.0%	10.8	$+\infty$	163
miniflight	17980	14752	18.0%	40.9	$+\infty$	251
tdf	5789	5727	1.0%	13.0	$+\infty$	254

- Mälardalen WCET Benchmarks
- SCADE designs
- Industrial Code

Conclusion (1/2)

THEORETICAL CONTRIBUTIONS:

SMT can be used in static analysis in many ways:

- **Improve precision** of abstract interpreters (least upper bounds + widening) (2 papers in SAS'12)
- **incremental analysis**
  - Modularity with **summaries** and **counter-examples**
- **Worst-Case Execution Time** estimation using optimization (LCTES'14)

FUTURE WORK:

- Improve SMT solving with Abstract Interpretation

Conclusion (2/2)

PRACTICAL CONTRIBUTIONS:

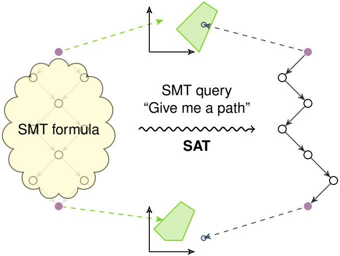
PAGAI static analyzer for LLVM, robust implementation (TAPAS'12)

- Extensive experiments
- Competitive

`http://pagai.forge.imag.fr`

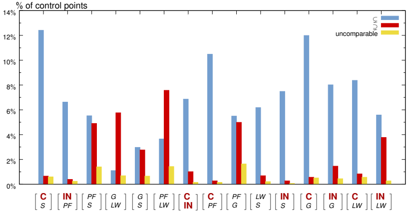
FUTURE WORK:

- Implement our modular static analysis
- Many improvements: data structures, floating points, etc.
- Combine with other program verifiers
- Tune for SV-COMP



Theory Atoms			Blocking Clause		
$c_1 \leq 2$		$c_n \leq 2$	~~$c_1 \leq 2$~~		~~$c_n \leq 2$~~
$c_1 \leq 3$	⋯	$c_n \leq 3$	~~$c_1 \leq 3$~~	⋯	~~$c_n \leq 3$~~
$\neg(c'_1 \leq 2)$		$\neg(c'_n \leq 2)$	~~$\neg(c'_1 \leq 2)$~~		~~$\neg(c'_n \leq 2)$~~
$c'_1 \leq 3$		$c'_n \leq 3$	~~$c'_1 \leq 3$~~		~~$c'_n \leq 3$~~
$c_1 + c'_1 \leq 5$		$c_n + c'_n \leq 5$	$c_1 + c'_1 \leq 5$		$c_n + c'_n \leq 5$
$c_1 + c'_1 + \cdots + c_n + c'_n > 5n$			$c_1 + c'_1 + \cdots + c_n + c'_n > 5n$		

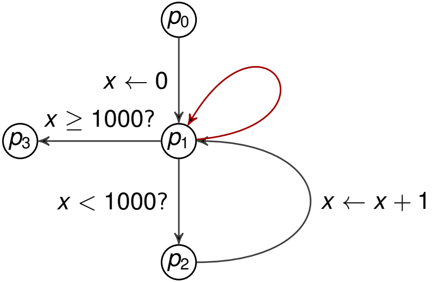
THANK YOU !



CONTRIBUTION: Improve the Descending Sequence

[Halbwachs & Henry, SAS12]

Descending sequence does not always work



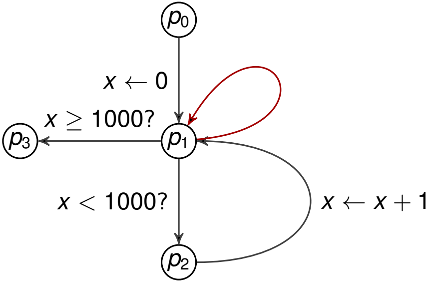
$$
\begin{aligned}
X_1 &= \{0\} \cup \{x \mid \exists x' \in X_2, x = x' + 1\} \cup X_1 && \subseteq [0, +\infty) \\
X_2 &= X_1 \cap \{x \mid x < 1000\} && \subseteq [0, 999]
\end{aligned}
$$

CONTRIBUTION: Improve the Descending Sequence

[Halbwachs & Henry, SAS12]

Descending sequence does not always work

Restart an analysis from a different, **well chosen**, initial value



$$
\begin{aligned}
X_1 &= \{0\} \cup \{x \mid \exists x' \in X_2, x = x' + 1\} \cup X_1 && \subseteq [0, +\infty) \\
X_2 &= X_1 \cap \{x \mid x < 1000\} && \subseteq [0, 999]
\end{aligned}
$$

CONTRIBUTION: Improve the Descending Sequence

[Halbwachs & Henry, SAS12]

Principle:

- Select some program location **supposedly already precise**
- Reset the other to $\bot$ $(= \emptyset)$
- Do ascending iterations

$$
\begin{aligned}
X_1 &= \{0\} \cup \{x \mid \exists x' \in X_2, x = x' + 1\} \cup X_1 && \subseteq [0, +\infty) \\
X_2 &= X_1 \cap \{x \mid x < 1000\} && \subseteq [0, 999]
\end{aligned}
$$

CONTRIBUTION: Improve the Descending Sequence

[Halbwachs & Henry, SAS12]

Principle:

- Select some program location **supposedly already precise**
- Reset the other to $\bot$ $(= \emptyset)$
- Do ascending iterations

$$\begin{aligned} X_1 &= \{0\} \cup \{x \mid \exists x' \in X_2, x = x' + 1\} \cup X_1 & \subseteq [0, +\infty) \\ X_2 &= X_1 \cap \{x \mid x < 1000\} & \subseteq [0, 999] \end{aligned}$$

CONTRIBUTION: Improve the Descending Sequence

[Halbwachs & Henry, SAS12]

Principle:

- Select some program location **supposedly already precise**
- Reset the other to $\bot$ $(= \emptyset)$
- Do ascending iterations

$$
\begin{aligned}
X_1 &= \{0\} \cup \{x \mid \exists x' \in X_2, x = x' + 1\} \cup X_1 &\overset{?}{\subseteq} \bot \\
X_2 &= X_1 \cap \{x \mid x < 1000\} &\overset{?}{\subseteq} [0, 999]
\end{aligned}
$$

CONTRIBUTION: Improve the Descending Sequence

[Halbwachs & Henry, SAS12]

Principle:

- Select some program location **supposedly already precise**
- Reset the other to $\perp$ $(= \emptyset)$
- Do ascending iterations

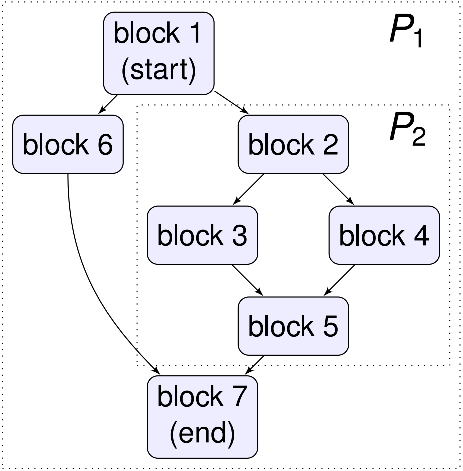
$$
\begin{aligned}
X_1 &= \{0\} \cup \{x \mid \exists x' \in X_2, x = x' + 1\} \cup X_1 && \overset{?}{\subseteq} [0, 1000] \\
X_2 &= X_1 \cap \{x \mid x < 1000\} && \overset{?}{\subseteq} [0, 999]
\end{aligned}
$$

How to choose the portions/cuts ?

Syntactic criterion

Between control-flow merges and their immediate dominators.

For P_2:
$c2 + c3 + c4 + c5 \leq$
$c2 + max(c3, c4) + c5$



How to choose the portions/cuts ?

Semantic criterion

```
...
if (b_i) { /* timing 2 */ } else { /* timing 3*/ }
if (b_i) { /* timing 3 */ } else { /* timing 2*/ }
...
```

- Slice the program w.r.t b_i.
- Recursively call the WCET procedure over the resulting graph
- The obtained WCET gives the upper bound for the portion

Note: Instead of recursive call, an Abstract Interpretation based technique would be possible. . .

WCET: Classical Approach