Building a specialized static analyzer
The Astrée experience

Antoine Miné

CNRS, École Normale Supérieure

Security and Reliability of Software Systems
12 December 2008
Analyse Statique Temps RÉEl

- check statically for the absence of run-time errors (RTE)
- on synchronous reactive codes, in a subset of C
- fast ($\approx 50$ KLoc/h), precise (aims at 0 alarms) and sound

Development team

CNRS / ENS
(B. Blanchet), P. Cousot, R. Cousot, L. Mauborgne,
(D. Monniaux), J. Feret, A. Miné, X. Rival

http://www.astree.ens.fr
The Astéree analyzer

- academic tool, with industrial applications in mind
- 110 000 lines of OCaml

**Time-line**

- **Nov. 2001** Astrée project starts
- **Nov. 2003** primary control software of the Airbus A340 analyzed
  - proof of the absence of RTE
- **Jan. 2004** start working on Airbus A380
  - (while A380 development in progress)
- **Apr. 2005** maiden flight of the A380
  - proof of the absence of RTE (for current version)
- **Sep. 2008** study on applicability to space software
Overview

- Introduction
  - Static analysis
  - First analyses
  - Language and semantics

- Design of Astrée
  - Architecture
  - Iterator
  - Abstract domains

- Results
Introduction
Existing verification methods

Testing

- well-established method
- but no formal warranty, high cost
Existing verification methods

Testing
- well-established method
- but no formal warranty, high cost

Formal methods:

Theorem proving
- proof essentially manual, but checked automatically
- powerful, but very steep learning curve

Model checking
- check a model of the program (usually user-specified, finite)
- automatic and complete (wrt. model), but costly
(Semantic-based) static analysis

- works directly on the source code (not a model)
- automatic, always terminating
- sound (full control and data coverage)
- incomplete (false alarms)
- parameterized by one/several abstraction(s)
- mostly used to check simple properties, with low precision requirements (e.g., for optimisation)
### Existing verification methods (cont.)

(Semantic-based) static analysis

- works directly on the source code  *(not a model)*
- automatic, always terminating
- sound  *(full control and data coverage)*
- incomplete  *(false alarms)*
- parameterized by one/several abstraction(s)
- mostly used to check simple properties, with low precision requirements  *(e.g., for optimisation)*

Specialized static analyzer

- checks for *run-time errors*  *(overflow, etc.)*
- is *very precise* at least on a chosen class of programs  *(no false alarm)*
- gives *sound* results on all programs
Abstract interpretation

Abstract Interpretation

General theory of semantic approximations [Cousot Cousot 77,91]

\[ \mathcal{D} \xrightarrow{\gamma} \mathcal{D}^\# \]

Concrete domain

\( \mathcal{D} \)

(e.g. \( \mathcal{P}(\mathbb{Z}^n) \))

Abstract domain

\( \mathcal{D}^\# \)

(e.g. \( n \)-dim boxes)

Elements in \( \mathcal{D}^\# \)

- represent properties of interest (semantical)
- are computer-representable (algorithmic)

Choosing \( \mathcal{D}^\# \) is a trade-off between cost and expressiveness
Abstract interpretation (cont.)

For each concrete semantic operator $F : D^n \rightarrow D$:
- assignment,
- test,
- control-flow join, etc.

Define $F^\# : D^{\# n} \rightarrow D^\#$ that:
- can be implemented (algorithm)
- is sound: $F(\gamma(X_1^\#), \ldots, \gamma(X_n^\#)) \subseteq \gamma(F^\#(X_1^\#, \ldots, X_n^\#))$
- various precision / cost trade-offs

$\implies$ computable over-approximation
Construction by refinement

**Approach**

- define a concrete semantics
- build a simple and fast analyzer (intervals)
- refine the analyzer until 0 false alarm:
  - determine which necessary properties are missed
  - add / refine an abstract domain to infer it

**Benefits**

- sound by construction
- efficient (adapted cost / precision trade-off)
- encourages modular, reusable abstractions
Analysis input and output

C sources (preprocessed) → Astrée → alarms, invariants
(behavior configuration)
Example analysis

```c
1 void main() {
2    unsigned i;
3    for (i=10; i>=0; i--) {
4        /* */
5    }
6 }
```

Starting the analysis

```
% astree loop.c --exec-fn main | egrep WARN
```
Example analysis

```
void main() {
    unsigned i;
    for (i=10; i>=0; i--) {
        /* */
    }
}
```

Analysis result

```
% astree loop.c --exec-fn main | egrep WARN
loop.c:3.19-22::[call#main@1:loop@4>=4::]: WARN:
    unsigned int arithmetic range [-1, 4294967294]
    not included in [0, 4294967295]
%
```
Example analysis (corrected)

```c
loop2.c
1 void main() {
2    unsigned i;
3    for (i=10; i> 0; i--) {
4        /* */
5        }
6    } 
```

Analysis result

```
% astree loop2.c --exec-fn main | egrep WARN
%
```
False alarm example

```c
void main() {
    int x, y;
    if ((-4681 < y) && (y < 4681) && (x < 32767) &&
        (-32767 < x) && ((7*y*y-1) == x*x))
        y = 1 / x;
}
```
Introduction

First analyses

False alarm example

```c
void main() {
    int x, y;
    if ((-4681 < y) && (y < 4681) && (x < 32767) &&
        (-32767 < x) && ((7*y*y-1) == x*x))
        y = 1 / x;
}
```

Analysis result

```
% astree false-alarm.c --exec-fn main | egrep WARN
false-alarm.c:5.8-13::[call#main@1::]: WARN: integer division by zero [-32766, 32766]
%
```
### False alarm example

```c
void main() {
  int x, y;
  if ((-4681 < y) && (y < 4681) && (x < 32767) &&
      (-32767 < x) && ((7*y*y-1) == x*x))
    y = 1 / x;
}
```

Actually, $7y^2 - 1 = x^2$ has no integer solution in $[-32766; 32766] \times [-4680; 4680]$

$\implies$ the alarm is spurious

Astrée is not knowledgeable of Diophantine equations
(difficult theory, Matiyasevich’s Theorem)
Astrée shows an over-approximation of the range of $x$
(a loop invariant valid starting at the fourth iteration)
Class of analyzed codes

- synchronous reactive codes
- compiled to C from a graphical language *a la Scade / Simulink*
- avionics codes

**Structure**

```plaintext
initialize state variables
while ( clock \( \leq \) 3 600 000 ) {
  read input from sensors (*volatile*)
  compute output and new state
  write output to actuators
  wait for clock tick
}
```
Example

- each box is a built-in C function
- boxes have state variables (static)
- input (volatile) are bounded by physical constraints
Impact on the analysis

**Difficult side**
- unstructured code
- large code (50K–1M loc)
- floating-point computation
- many variables (10K–)
- some complex (non-linear) numerical invariants

**Easy side**
- homogeneous code
- simple algorithms
- no recursion
- simple data-structures
- no dynamic allocation
### C subset

**Handled**
- machine integers, enum
- IEEE floats
- structures, arrays
- bitfields
- pointers, aliases
- pointer arithmetic
- unions
- tests (if)
- loops (for, while)
- break, return, forward goto
- switch

**Unhandled**
- dynamic allocation
- recursivity
- backward goto
- longjmp
- libraries (*libc*, etc.)
  \[\Rightarrow\] stubs needed
- concurrency (*threads*)
Considered semantics

Definition of the semantics

- **C99 norm** *(portable programs)*
- **IEEE 754-1985 norm** *(floating-point arithmetics)*

**platform-dependent choices:**
- range of types
- bit-representation (*sizeof*, endianess, *struct* padding, etc.)

**compiler- and linker-dependent choices:**
- automatic variable initialization (optional)
- symbol redefinition (forbidden)

Some choices are configurable through command-line options
Kinds of run-time errors

- overflows in float, integer, enum arithmetic and cast
- division, modulo by 0 on integers and floats
- invalid right argument of bit-shift
- out-of-bound array access
- invalid pointer arithmetic or dereference
- violation of user-specified assertions (___ASTREE_assert)
Run-time errors (cont.)

Several semantics are possible after an error:

- **halt** the program
  - division, modulo by zero
  - floating-point overflow
  - assertion failure
- **return all possible values** in the type range
  - invalid bit-shift
- **well-defined** result
  - modulo on integer arithmetics overflow
- **unpredictable** behavior
  - invalid dereference
  
  (Astreé treats this as halting the program)

Some alarm reporting and semantics is configurable through command-line options
Semantic configuration example

```c
enum.c
1 enum { FALSE=0, TRUE=1 } B;
2 void main() {
3    __ASTREE_log_vars((B;interv));
4    B = B+1;
5    __ASTREE_log_vars((B;interv));
6 }
```

Default semantic parameters

```bash
% astree enum.c --exec-fn main | egrep "WARN|B in"
enum.c:3.2-31:  log:  B in {0}
enum.c:5.2-31:  log:  B in {1}
%```
No zero-initialization of globals

% astree enum.c --exec-fn main --no-global-initialization
 | egrep "WARN\|B in"
enum.c:3.2-31: log: B in [-2147483648, 2147483647]
enum.c:4.6-9::[call#main@2::]: WARN: signed int arithmetic range [-2147483647, 2147483648]
 not included in [-2147483648, 2147483647]
enum.c:4.2-9::[call#main@2::]: WARN: signed int->unnamed enum conversion range [-2147483648, 2147483647]
 not included in {0,1}
enum.c:5.2-31: log: B in [-2147483648, 2147483647]
%
Enum clamping

% astree enum.c --exec-fn main --no-global-initialization
   --clamp-enum | egrep "WARN\|B in"
enum.c:3.2-31:  log:  B in [-2147483648, 2147483647]
enum.c:4.6-9::[call#main@2::]:  WARN: signed int
   arithmetic range [-2147483647, 2147483648]
   not included in [-2147483648, 2147483647]
enum.c:4.2-9::[call#main@2::]:  WARN: signed int->unnamed
   enum conversion range [-2147483648, 2147483647]
   not included in {0,1}
enum.c:5.2-31:  log:  B in [0,1]
%
Global view

preprocessor (cpp)
↓
C99 parser
↓
source-level linker
↓
intermediate code generation and typing
↓
constant propagation and code simplification
↓
global dependency analysis
↓
abstract interpreter
Abstract interpreter

iterator

trace partitioning

memory model and alias analysis

(reduced product of) numerical abstract domains

intervals  octagons  decision trees  ...

intervals
Iterator
Syntax-directed interpreter

Astrée works as an interpreter:

- start from a main function
- follow the control-flow of the program
- the current state $X^\# \in \mathcal{D}^\#$ is an abstraction of an environment set in $\mathcal{D} \simeq \mathcal{P}(\text{Var} \rightarrow \text{Val})$
  \[\implies \text{collecting semantics, compute reachable states}\]
- low memory cost: one environment per loop and if level

The interpretation is by induction on the syntax

Atomic instructions:

- assignments $[V \leftarrow expr]^\# : \mathcal{D}^\# \rightarrow \mathcal{D}^\#$ update environments
- tests $[expr == 0?]^\# : \mathcal{D}^\# \rightarrow \mathcal{D}^\#$ filter environments
- add / remove variables
Instruction blocks:

- sequential evaluation $[[i_1; \ldots; i_n]] = [[i_n]] \circ \cdots \circ [[i_1]]$

Conditionals: if (expr) $i_T$ else $i_F$

- evaluate both branches, then join:

$$[[\text{if}]](X) = [[i_T]]([[\text{expr}! = 0?]](X)) \cup [[i_F]]([[\text{expr} == 0?]](X))$$

where $\gamma(X \cup Y) \supseteq \gamma(X) \cup \gamma(Y)$

Function calls:

- inline all function calls

  $\implies$ high precision (full stack context sensitivity)

  costly, no recursivity
Analysis of conditionals example

```c
void main() {
    int b;
    float x; __ASTREE_log_vars((x;interv));
    if (b) { x = 0; __ASTREE_log_vars((x;interv)); }
    else { x = 10; __ASTREE_log_vars((x;interv)); }
    __ASTREE_log_vars((x;interv));
}
```

Analysis result

```
% astree cond.c --exec-fn main | egrep "up |x in"
cond.c:3.11-40: log: x in [-3.4028235e+38, 3.4028235e+38]
cond.c:4.18-47: log: x in {0.}
cond.c:5.17-46: log: x in {10.}
cond.c:6.2-31: log: x in [0., 10.]
%
```
Context-sensitive analysis example

```
fun.c
1 int f(int b) { return 1/b; }
2
3 void main() {
4   f(2);
5   f(0);
6   f(0);
7 }
```

Analysis result

```
% astree fun.c --exec-fn main | egrep WARN
fun.c:1.22-25::[call#main@3:call#f@5::]: WARN: integer division by zero {0}
% 
```
Function stub example

```
extern double acos(double d);
void main() {
  double x;
  double y = acos(x);
  __ASTREE_log_vars((x,y;interv));
}
```

Analysis result

```
% astree nostub.c --exec-fn main | egrep "WARN| in "
nostub.c:1.14-17:  WARN: stub log called
x in [-1.7976932e+308, 1.7976932e+308]
y in [-1.7976932e+308, 1.7976932e+308]
%```

Antoine Miné

Building a specialized static analyzer

p. 36 / 112
Function stub example (corrected)

```c
stub.c

1  double acos(double d) {
2      double r;
3      __ASTREE_assert((d>=-1 && d<=1));
4      __ASTREE_known_fact((r>=0 && r<=3.2));
5      return r;
6  }
```

Analysis result

```
% astree nostub.c stub.c --exec-fn main | egrep "WARN| in "
stub.c:3.19-32::[call#main@2:call#acos@4::):
   WARN: assert failure
  x in [-1., 1.]
  y in [0., 3.2000001]
%
```
Loop analysis

**Loops:** \( \text{while (expr) } i \)

**Concrete semantics:** fixpoint

\[
\llbracket \text{while} \rrbracket(X) = \llbracket \text{expr} == 0? \rrbracket(lfp Y \mapsto X \cup ([i] \circ \llbracket \text{expr!} = 0? \rrbracket)(Y))
\]

**Abstract semantics:** iterations with widening

\[
\llbracket \text{while} \rrbracket^\#(X^\#) = \llbracket \text{expr} == 0? \rrbracket(X^\#_n)
\]

where

\[
\begin{align*}
X^\#_0 &= X^\# \\
X^\#_{i+1} &= X^\#_i \nabla ([i]^\# \circ \llbracket \text{expr!} = 0? \rrbracket^\#)(X^\#_i) \\
X^\#_{n+1} &= X^\#_n
\end{align*}
\]

\( \nabla \) is an extrapolation operator

- \( \gamma(X^\#) \cup \gamma(Y^\#) \subseteq \gamma(X^\# \nabla Y^\#) \)
- \( \nabla \) enforces termination

(e.g., start with \( \cup^\# \), enlarge unstable bounds to threshold, then max-type)
Design of Astrée Iterator

Loop analysis example

fltloop.c

1 void main() {
2 float x = 0.1;
3 while (1) {
4 int r;
5 if (r) x = 0.2; else x = 0.9*x + 0.1;
6 __ASTREE_log_vars((x;interv));
7 }
8 }

Analysis result

% astree fltloop.c --exec-fn main --unroll 0
  | egrep "up |x in"
x in [0.18999993, 0.20000001]
...
Loop analysis example

fltloop.c

```c
1 void main() {
2     float x = 0.1;
3     while (1) {
4         int r;
5         if (r) x = 0.2; else x = 0.9*x + 0.1;
6         __ASTREE_log_vars((x;interv));
7     }
8 }
```

Analysis result

```
fltloop.c:3.2-9.3: up iteration #0
x in [0.18999993, 0.28000004]
fltloop.c:3.2-9.3: up iteration #1
x in [0.18999993, 0.35200006]
...```

Antoine Miné
Building a specialized static analyzer
### Loop analysis example

```c
void main() {
    float x = 0.1;
    while (1) {
        int r;
        if (r) x = 0.2; else x = 0.9*x + 0.1;
        __ASTREE_log_vars((x;interv));
    }
}
```

### Analysis result

```
fltloop.c:3.2-9.3: up iteration #2
x in [0.18999993, 0.41680009]
fltloop.c:3.2-9.3: up iteration #3
x in [0.18999993, 1.00000002]
...```

Antoine Miné
Building a specialized static analyzer
p. 40 / 112
Design of Astrée

Loop analysis example

```
fltloop.c
1  void main() {
2    float x = 0.1;
3    while (1) {
4        int r;
5        if (r) x = 0.2; else x = 0.9*x + 0.1;
6        __ASTREE_log_vars((x;interv));
7    }
8 }
```

Analysis result

```
fltloop.c:3.2-9.3: up iteration #4
x in [0.18999993, 1.0000002]
fltloop.c:3.2-9.3: up iteration #5
x in [0.18999993, 1.0000002]
%```
Improved loop analysis

Actually, Astére performs more complex iterations:

- **unrolls** the first iterations
  (separate analysis of initialization)
- performs **increasing** iterations with **widening**
- performs **decreasing** iterations after stabilisation
  (improves the fixpoint)
- alarms are printed in a final **checking iteration**

Loop analysis can be configured by command-line options
Loop unrolling example

initloop.c

1 void main() {
2   int I = 1, x;
3   while (1) {
4       __ASTREE_log_vars((I, x; interv));
5       if (I) { x = 0; I = 0; }
6       else { x++; if (x > 100) x = 0; }
7   }
8 }

Antoine Miné
Building a specialized static analyzer
p. 43 / 112
# Loop unrolling example

```c
#include __ASTREE_log_vars((I, x; interv));

int main() {
    int I = 1, x;
    while (1) {
        if (I) { x = 0; I = 0; }
        else { x++; if (x > 100) x = 0; }
    }
}
```

## Analysis result without unrolling

```bash
% astree initloop.c --exec-fn main --unroll 0 | egrep "iteration| in|WARN"
```

- I in {1}, x in [-2147483648, 2147483647]
- ...
Loop unrolling example

```c
1 void main() {
2    int I = 1, x;
3    while (1) {
4        __ASTREE_log_vars((I, x; interv));
5        if (I) { x = 0; I = 0; }
6        else { x++; if (x > 100) x = 0; }
7    }
8 }
```

Analysis result **without** unrolling

```
initloop.c:3.2-7.3: up iteration #0
I in [0, 1], x in [-2147483648, 2147483647]
initloop.c:3.2-7.3: up iteration #1
I in [0, 1], x in [-2147483648, 2147483647]
...```

Antoine Miné Building a specialized static analyzer p. 43 / 112
Loop unrolling example

```c
void main() {
    int I = 1, x;
    while (1) {
        __ASTREE_log_vars((I, x; interv));
        if (I) { x = 0; I = 0; }
        else { x++; if (x > 100) x = 0; }
    }
}
```

Analysis result without unrolling

```
initloop.c:3.2-7.3:down iteration #0
I in [0, 1], x in [-2147483648, 2147483647]
initloop.c:6.11-14::[call#main@1::]: WARN: signed int arithmetic range [-2147483647, 2147483648]
not included in [-2147483648, 2147483647]
%```
Loop unrolling example

initloop.c

```c
1 void main() {
2    int I = 1, x;
3    while (1) {
4        __ASTREE_log_vars((I, x; interv));
5        if (I) { x = 0; I = 0; }
6        else { x++; if (x > 100) x = 0; }
7    }
8 }
```
Loop unrolling example

```
initloop.c

1 void main() {
2    int I = 1, x;
3   while (1) {
4       __ASTREE_log_vars((I, x, interv));
5       if (I) { x = 0; I = 0; }
6       else { x++; if (x > 100) x = 0; }
7   }
8 }
```

Analysis result with unrolling

```
% astree initloop.c --exec-fn main --unroll 1
     | egrep "iteration| in|WARN" -B 1
loop@3=1:  I in {1}, x in [-2147483648, 2147483647]
...
Loop unrolling example

```c
void main() {
    int I = 1, x;
    while (1) {
        __ASTREE_log_vars((I, x; interv));
        if (I) { x = 0; I = 0; }
        else { x++; if (x > 100) x = 0; }
    }
}
```

Analysis result with unrolling

```
loop@3>=2:  I in {0}, x in {0}
initloop.c:3.2-7.3: up iteration #0
loop@3>=2:  I in {0}, x in [0, 1]
initloop.c:3.2-7.3: up iteration #1
...
```
**Loop unrolling example**

```c
1 void main() {
2     int I = 1, x;
3     while (1) {
4         __ASTREE_log_vars((I, x; interv));
5         if (I) { x = 0; I = 0; }
6         else { x++; if (x > 100) x = 0; }
7     }
8 }
```

**Analysis result with unrolling**

- loop@3>=2: I in {0}, x in [0, 2]
- initloop.c:3.2-7.3: up iteration #2
- ...
- loop@3>=2: I in {0}, x in [0, 19]
- initloop.c:3.2-7.3: up iteration #9
- ...

Antoine Miné Building a specialized static analyzer p. 43 / 112
Loop unrolling example

```c
1 void main() {
2    int I = 1, x;
3    while (1) {
4       __ASTREE_log_vars((I, x; interv));
5       if (I) { x = 0; I = 0; }
6       else { x++; if (x > 100) x = 0; }
7    }
8 }
```

Analysis result with unrolling

<table>
<thead>
<tr>
<th>Loop</th>
<th>Iteration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop@3&gt;=2</td>
<td>I in {0}, x in [0, 41]</td>
<td>initloop.c:3.2-7.3: up iteration #10</td>
</tr>
<tr>
<td>loop@3&gt;=2</td>
<td>I in {0}, x in [0, 32767]</td>
<td>initloop.c:3.2-7.3: up iteration #11</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Loop unrolling example

### initloop.c

```c
1 void main() {
2    int I = 1, x;
3    while (1) {
4      __ASTREE_log_vars((I, x; interv));
5      if (I) { x = 0; I = 0; }
6      else { x++; if (x > 100) x = 0; }
7    }
8 }
```

### Analysis result with unrolling

- loop@3>=2: I in {0}, x in [0, 32767]
- initloop.c:3.2-7.3:down iteration #0
- loop@3>=2: I in {0}, x in [0, 100]
- initloop.c:3.2-7.3:down iteration #1
- loop@3>=2: I in {0}, x in [0, 100]
  ```
Nested loops

```c
void main() {
    int i, j, x[10][20];
    for (i=0; i<10; i++)
        for (j=0; j<20; j++) {
            x[i][j] = 1;
            __ASTREE_log_vars((i, j; interv));
        }
}
```

Analysis result

```
% astree nested.c --exec-fn main --unroll 0
   | egrep "i in|WARN"
i in {0}, j in {0}
i in {0}, j in [0, 1]
   ...
i in {0}, j in [0, 19]
```
Nested loops

```c
1 void main() {
2   int i, j, x[10][20];
3   for (i=0; i<10; i++)
4       for (j=0; j<20; j++) {
5           x[i][j] = 1;
6           __ASTREE_log_vars((i,j;interv));
7       }
8 }
```

Analysis result

```
i in [0, 1], j in {0}
i in [0, 1], j in [0, 1]
...
i in [0, 9], j in [0, 18]
i in [0, 9], j in [0, 19]
```
Numerical domains
Integer interval domain

Integer interval definition
\[ D^\# = \text{Var} \rightarrow (\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{+\infty\}) \cup \{\perp\} \]
maps variables to interval bounds

Benefits
- can express the absence of (most) RTE
  (overflow, out-of-bound access)
- easy to implement
  (e.g. assignments by induction on the syntax of expressions
  \([a; b] +^\# [c; d] = [a + c; b + d]\))
- low memory and time cost (linear?)
Data-structures

**Naïve idea: arrays**

- fetch, update in $O(1)$
- copy in $O(|\text{Var}|)$ (tests and loops)
- $\cup$ in $O(|\text{Var}|)$

$\implies$ cost of one iteration: $\simeq O(|\text{Var}| \times |P|) \simeq O(|P|^2)$
**Naïve idea:** arrays

- fetch, update in $\mathcal{O}(1)$
- copy in $\mathcal{O}(|\text{Var}|)$  
  (tests and loops)
- $\cup$ in $\mathcal{O}(|\text{Var}|)$

$\implies$ cost of one iteration: $\simeq \mathcal{O}(|\text{Var}| \times |P|) \simeq \mathcal{O}(|P|^2)$

**Better idea:** functional maps (balanced binary trees)

- fetch, update in $\mathcal{O}(\log|\text{Var}|)$
- copy in $\mathcal{O}(1)$
- $\cup$ in $\mathcal{O}(|P_i| \log|\text{Var}|)$  
  (for $i$–th block)

$\implies$ cost of one iteration: $\simeq \mathcal{O}(|P| \log|\text{Var}|) \simeq \mathcal{O}(|P| \log|P|)$
Float interval domain

**Concrete semantics:** IEEE 754-1985 norm

- compute over a finite set $\mathbb{F} \subseteq \mathbb{Q}$
- two-step evaluation
  - evaluate exactly in $\mathbb{Q}$
  - round the result in $\mathbb{F}$, in direction $r \in \{0, +\infty, -\infty, n\}$
    
    $$\llbracket x \oplus_r y \rrbracket = R_r(\llbracket x \rrbracket + \llbracket y \rrbracket)$$
    $$R_{+\infty}(x) = \min \{z \in \mathbb{F} \mid z \geq x\}$$

- possibility of run-time errors: overflow and division by zero
  (semantics halts with a RTE instead of constructing $\pm\infty$ or $NaN$)

**Abstract semantics:**

- intervals with **float bounds** in $\mathbb{F}$
- round upper bounds towards $+\infty$, lower bounds towards $-\infty$
  $$[a, b] \oplus^\# [c, d] = [a \oplus_{-\infty} c, b \oplus_{+\infty} d] \cap [\min \mathbb{F}, \max \mathbb{F}]$$
Interval widening

Widening with threshold, parameterized by a finite set $T$:

$[-; b] \uplus [-; d] = \begin{cases} b & \text{if } b \geq d \\ \min\{ t \in T \cup \{+\infty\} \mid t \geq d \} & \text{otherwise} \end{cases}$

Example

X=0;
while (1) {
    if (?) X=[0;10];
    X=0.1*X+[0;75]; /* X ∈ [0; 83.33333587646484375] */
}

Range of $X$:

- in rationals: $X \leq 83.333 \cdots$
- in floats (concrete semantics): $X \leq 83.33333587646484375$
- abstract float semantics with widening:

$X \leq \min\{ t \in T \mid t \geq 83.33333587646484375 \}$
Choice of widening thresholds

On floats
- the precise bound is generally useless for RTE-detection
- computations can be stable if the bound is overshot
  \((\simeq \text{attractive fixpoint})\)

\(\implies \) use a sufficiently dense exponential ramp

On integers
The exact bound is important for array bound checking

Solutions:
- some bounds can be discovered by decreasing iterations
- static thresholds \((\text{e.g., array size})\)
- dynamic thresholds: enrich \(T_X\) when encountering
  \([\text{if } (X \leq c) \ldots]^{\#}\)
Design of Astrée

Numerical domains

Interval widening delay

```
void main() {
    int X=0, Y=0;
    while (1) {
        __ASTREE_log_vars((X,Y;interv));
        if (X<100) X++;
        if (X>=60+Y) Y=20;
    }
}
```

Analysis result

```
% astree delay.c --exec-fn main | grep "up |WARN|X in"
up iter #0:  X in [3, 4], Y in {0}
up iter #1:  X in [3, 5], Y in {0}
up iter #2:  X in [3, 6], Y in {0}
up iter #3:  X in [3, 17], Y in {0}
```
Interval widening delay

```c
1 void main() {
2    int X=0, Y=0;
3    while (1) {
4        □□ASTREE_log_vars((X,Y;interv));
5        if (X<100) X++;
6        if (X>=60+Y) Y=20;
7    }
8 }
```

Analysis result

...  
up iter #8:  X in [3, 43], Y in {0}  
up iter #9:  X in [3, 99], Y in {0}  
up iter #10: X in [3, 100], Y in [0, 20] 
%
Interval widening delay

```c
void main() {
  int X=0, Y=0;
  while (1) {
    __ASTREE_log_vars((X,Y;interv));
    if (X<100) X++;
    if (X>=60+Y) Y=20;
  }
}
```

▽ may be replaced with ∪ to increase precision:

- use a per-variable, per-domain freshness counter
  incremented when unstable, unchanged when stable
- ▽ only for certain counter values
  and always after some some value
Other non-relational domains

\[ D\# \simeq \text{Var} \rightarrow D_b\]  

Congruences

\[ D_b\# = \{a\mathbb{Z} + b\} \]  
useful to:
- check (multi-dimensional) array traversal
- check pointer alignment constraints

Bit-fields

\[ D_b\# = [0; 31] \rightarrow \mathcal{P}([0, 1]) \]  
useful to abstract bit-operations precisely &amp;, |, &lt;&lt;, &gt;&gt;  
also generate congruence information
### Congruence analysis example

**cong.c**

```c
1 void main() {
2    int i,x[100];
3    for (i=1;i<=100;i+=2) {
4        __ASTREE_log_vars((i;inter,cong));
5        x[i] = 1;
6    }
7 }
```

**Analysis result**

```
% astree cong.c --exec-fn main | egrep "i in|WARN"
 i in {1}
 i in {7}
 i in [7, 9] \ (2Z+1)
 ...
 i in [7, 99] \ (2Z+1)
%
```
Octagon domain

Definition

\[ D^\# = \bigwedge_{X, Y \in \text{Var}} \text{constraints } \pm X \pm Y \leq c \]
Octagon domain

Algorithms
- generalize DBMs  \((X - Y \leq c)\)
- representation: square matrix of constraints
- constraint-propagation: based on shortest-path closure
- exact abstraction for \([X \leftarrow \pm Y + c], [\pm X \pm Y \leq c]\)

Cost
- memory: \(O(|\text{Var}|^2)\) (full matrices)
- time \(O(|\text{Var}|^3)\) (closure)
The need for relational domains

```
rel.c

1 void main() {
2     int I, V=0;
3     for (I=10;I>=0;I--) {
4         int B;
5         if (B) V=V+1;
6     }
7     __ASTREE_log_vars((V));
8 }
```

Analysis result with standard domains

```
% astree rel.c --exec-fn main | egrep "WARN|V in"
V in [0, 11]
%
```
The need for relational domains

rel.c

```c
void main() {
    int I, V=0;
    for (I=10; I>=0; I--) {
        int B;
        if (B) V = V + 1;
    }
    __ASTREE_log_vars((V));
}
```

Analysis result without relational domains

```
% astree rel.c --exec-fn main --no-relational
  | egrep "WARN|V in"
rel.c:5.13-16::[call#main@1:loop@4>=4]: WARN: signed int arithmetic range [-2147483647, 2147483648] not included in [-2147483648, 2147483647]
V in [-2147483648, 2147483647]
%```
The need for relational domains

V is not stable in the loop, and not bounded by any test

To bound V we must:

- infer the loop invariant $V + I \leq 10$
- combine it with $I = -1$ at loop exit
The need for relational domains

```c
void main() {
  int I, V=0;
  for (I=10;I>=0;I--) {
    int B;
    if (B) V=V+1;
  }
  __ASTREE_log_vars((V));
}
```

Octagon invariant

% astree rel.c --exec-fn main | egrep "V <="
0 <= V <= 11, I = -1, -12 <= I-V <= -1, -1 <= I+V <= 10 %
The need for relational domains

Even when looking for a non-relational invariant at loop exit, a **relational loop invariant** is often needed.
Rate limiter example

```c
float x, d, R, S, Y;
__ASTREE_volatile_input((x [-128, 128]));
__ASTREE_volatile_input((d [0, 16]));
void main() {
  while (1) {
    float X = x, D = d;
    S = Y; R = X - S; Y = X;
    if (R <= -D) Y = S - D; else if (R >= D) Y = S + D;
    __ASTREE_log_vars((Y));
  }
}
```

|Y| is bounded by max(128, |S|) because either:
- \(Y = X \in [-128, 128]\)
- \(Y = S - D \leq S \leq X - S \leq -D\), so \(Y = S - D \geq X \geq -128\)
- \(Y = S + D \geq S \geq X - S \geq D\), so \(Y = S + D \leq X \leq 128\)
\(\Rightarrow Y \in [-128, 128]\)
Rate limiter example (interval analysis)

Analysis without octagons

% astree rlim.c --exec-fn main --no-octagon
  | egrep "iter|WARN|Y in"
unroll:  Y in [-128., 128.]
unroll:  Y in [-144., 144.]
...
up iter #0:  Y in [-192., 192.]
up iter #1:  Y in [-208., 208.]
up iter #2:  Y in [-224., 224.]
up iter #3:  Y in [-1016., 1016.]
...
up iter #18:  Y in [-3.4028235e+38, 3.4028235e+38]
rlim.c:7.11-14::[call#main@4:loop@5>=4::]:  WARN:
  float arithmetic range [-inf., +inf.]
  not included in [-3.4028235e+38, 3.4028235e+38]

- \( Y = S - D \implies \min Y = \min Y - \max D \)
- \( Y = S + D \implies \max Y = \max Y - \min D \)
- after control-flow join: \( Y^\# = (S^\# + [16, 16]) \cup [128, 128] \)
Rate limiter example (octagon analysis)

Analysis result with octagons

% astree rlim.c --exec-fn main | egrep "iter|WARN|Y in"
unroll:  Y in [-16.000016, 16.000016]
Y in [-32.000033, 32.000033]
...
up iter #0:  Y in [-80.000096, 80.000096]
up iter #1:  Y in [-96.000121, 96.000121]
up iter #2:  Y in [-112.00015, 112.00015]
up iter #3:  Y in [-1000., 1000.]

• we have an approximate octagon abstraction for
  \[ V \leftarrow [a_0, b_0] + \sum [a_i, b_i] V_i \]
  \[ \Rightarrow \quad Y = S - D \Rightarrow -16 \leq Y - S \leq 0 \]
  \[ \Rightarrow \quad Y = S + D \Rightarrow 0 \leq Y - S \leq 16 \]

• any \(|Y| \leq M\) with \(M \geq 144\) can be proved to be a loop invariant
  \[ \Rightarrow \text{iterations stop at the first widening threshold } \geq 144 \]
Octagon packing

Cost in $O(|\text{Var}|^n)$ with $n > 1$ is too expensive!
Octagon packing

Cost in $\mathcal{O}(|\text{Var}|^n)$ with $n > 1$ is too expensive!

Solution

Do not put all \text{Var} is a single large octagon, but make many very small packs:

- local dependency pre-analysis
- link only variables manipulated together
- cut dependencies at syntactic block boundaries

Result: on the kind of code we analyze

- linear number of packs in $|\text{Var}|, |P|$  
- constant size of packs
## Octagon packing statistics

<table>
<thead>
<tr>
<th># lines</th>
<th># variables</th>
<th># packs</th>
<th>size</th>
<th>$\sqrt{\sum \text{size}^2}$</th>
<th>$\sqrt[3]{\sum \text{size}^3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>370</td>
<td>100</td>
<td>20</td>
<td>3.6</td>
<td>4.8</td>
<td>6.2</td>
</tr>
<tr>
<td>9 500</td>
<td>1 400</td>
<td>200</td>
<td>3.1</td>
<td>4.6</td>
<td>6.6</td>
</tr>
<tr>
<td>70 000</td>
<td>14 000</td>
<td>2 470</td>
<td>3.5</td>
<td>5.2</td>
<td>7.8</td>
</tr>
<tr>
<td>226 000</td>
<td>47 500</td>
<td>7 429</td>
<td>3.5</td>
<td>4.5</td>
<td>5.8</td>
</tr>
<tr>
<td>400 000</td>
<td>82 000</td>
<td>12 964</td>
<td>3.3</td>
<td>4.1</td>
<td>5.3</td>
</tr>
</tbody>
</table>
Manual octagon packing

```c
1 int X=10, Y=100;
2 void f() { Y--; }
3 void main() {
4     while (X >= 0) {
5         X--; f();
6     }
7 }
```

Analysis result with automatic packing

```
% astree octpack.c --exec-fn main --print-packs
| egrep "WARN\|(X Y)"
octpack.c:2.11-14::[call#main@3:loop@4>=4:call#f@5::]: WARN:
  signed int arithmetic range [-2147483649, 2147483646]
  not included in [-2147483648, 2147483647]
%
```
Manual octagon packing

octpack2.c

```
1 int X=10, Y=100;
2 void f() { Y--; }
3 void main() {
4     __ASTREE_octagon_pack((X,Y));
5     while (X >= 0) {
6         X--; f();
7     }
8 }
```

Analysis result with manual packing

```
% astree octpack2.c --exec-fn main --print-packs
   | egrep "WARN|(X Y)"
octpack2.c@4@1 X Y
%
```
Linearization

Issue

Relational domains are generally bad at non-linear expressions (multiplications, logical, bit-wise operations, etc.)
Linearization

Issue
Relational domains are generally bad at non-linear expressions (multiplications, logical, bit-wise operations, etc.)

Solution
Linearize expressions: put into the form

\[ [a_0; b_0] + \sum_i [a_i; b_i] V_i \]

- linear \(\implies\) easy to manipulate
- intervals \(\implies\) express non-determinism
  a powerful way to abstract
Linearization $l$: defined by induction on the syntax of expressions

- $l(e_1 + e_2) = l(e_1) + l(e_2)$
- $l(e_1 - e_2) = l(e_1) - l(e_2)$
- $l(e_1 \times e_2) = \begin{cases} 
  \text{either } & i(e_1) \times l(e_2) \\
  \text{or } & i(e_2) \times l(e_1)
\end{cases}$
- $l(e_1/e_2) = l(e_1)/i(e_2)$
- otherwise $l(e) = i(e)$

where

- $+,-$ are extended to linear expressions
- $\times,/$ are extended to an expression and an interval
- $i(e)$ evaluates $e$ using interval arithmetics

For $\times$, we need a strategy to choose between two linearizations (e.g. minimize the interval factor)
Using linearization

Application:

- **octagons** can handle interval linear expressions
  
  e.g.: if $Y \in [0; 1]$, $Z \geq 0$, then $l(Y \times Z) = [0; 1] \times Z$

  $[[X \leftarrow Y \times Z]]^\#$ is abstracted as
  
  $[[X \leftarrow [0; 1] \times Z]]^\#

  \implies$ octagons can infer that $X \leq Z$

- linearization provides **symbolic simplification** for free
  
  e.g.: $l(2X - X) = (X)$

  if $X \in [0; 1]$ plain intervals evaluate $2X - X$ to $[-1; 2]$

  after linearization, we have $[0; 1]$

Linearization is an abstraction $\Rightarrow$ it can also lose precision
Symbolic constant propagation

Example

1  Y=[-100;100];
2  X=Y+10;
3  if (X<0) X=-X;
4  if (X<60) {  ...  Y  ...  }

How can we find a bound on $Y$... without octagons?
Symbolic constant propagation

Example

1. \( Y = [-100; 100] \);
2. \( X = Y + 10 \);
3. if \((X < 0)\) \(X = -X;\)
4. if \((X < 60)\) \{ \(\ldots Y \ldots\) \}

How can we find a bound on \( Y \) ... without octagons?

Alternate solution

Remember that \( X = Y + 10 \)
and substitute \( X \) with \( Y + 10 \) at line 3 and on
The interval domain will find \( Y \in [-70; 50] \)
Symbolic constant propagation

Comparison with a relational domain

Benefits
- $\simeq$linear cost ($\simeq$ vanilla constant propagation)
- simple to implement
- interacts well with linearization
  (more opportunities for simplification)

Issues
- no inference of non-syntactic properties (joins, loops)
- not optimal, or even monotonic scheme
- substitution strategies are fragile (wrt. code transformations)
Floating-point semantics and symbolic computations

Problem

Most algebraic rules valid in $\mathbb{Z}$ or $\mathbb{Q}$ are no longer valid in floating-point

How can we:

- perform sound floating-point linearization?
- use relational domains on floating-point expressions?
Floating-point simplification example (float)

```c
float-c.c
1  void main() {
2      float a = 1.0;
3      float x = 1125899973951488.0;
4      float y = (x + a);
5      float z = (x - a);
6      float r1 = y - z;
7      float r2 = 2*a;
8      assert(r1==a2);
9  }
```

Concrete execution (rounding to nearest)

```
y = 1125899906842624
z = 1125899906842624
r1 = 0
r2 = 2
```
Floating-point simplification example (float)

```c
void main() {
    float a = 1.0;
    float x = 1125899973951488.0;
    float y = (x + a);
    float z = (x - a);
    float r1 = y - z;
    float r2 = 2*a;
    __ASTREE_log_vars((y,z,r1,r2;interv));
    __ASTREE_assert((r1==r2));
}
```

Analysis (all roundings)

```bash
% astree float-a.c --exec-fn main | egrep " in | WARN"
y in [1.1258999e+15, 1.1259002e+15]
z in [1.1258996e+15, 1.1259001e+15]
r1 in [-134217730., 335544320.]
r2 in {2.}
float-a.c:9.19-25::[call#main@1]: WARN: assert failure
```
Floating-point simplification example (double)

double-c.c

1  void main() {
2   double a = 1.0;
3   double x = 1125899973951488.0;
4   double y = (x + a);
5   double z = (x - a);
6   double r1 = y - z;
7   double r2 = 2*a;
8   assert(r1==r2);
9  }

Concrete execution (rounding to nearest)

y = 1125899973951489
z = 1125899973951487
r1 = 2
r2 = 2
Floating-point simplification example (double)

```c
void main() {
    double a = 1.0;
    double x = 1125899973951488.0;
    double y = (x + a);
    double z = (x - a);
    double r1 = y - z;
    double r2 = 2*a;
    __ASTREE_log_vars((y,z,r1,r2;inter));
    __ASTREE_assert((r1==r2));
}
```

Analysis (all roundings)

```
% astree double-a.c --exec-fn main | egrep " in | WARN"
y in {≈1.1258999e+15}
z in {≈1.1258999e+15}
r1 in {2.}
r2 in {2.}
%
```
Floating-point simplification example (mixed)

```c
void main() {
    float a = 1.0;
    double x = 1125899973951488.0;
    float y = (x + a);
    float z = (x - a);
    float r1 = y - z;
    float r2 = 2*a;
    assert(r1==a2);
}
```

Concrete execution (rounding to nearest)

```
y = 11259000041060352
z = 1125899906842624
r1 = 134217728
r2 = 2
```

```
```
Floating-point simplification example (mixed)

```c
#include <stdio.h>

void main() {
  float a = 1.0;
  double x = 1125899973951488.0;
  float y = (x + a);
  float z = (x - a);
  float r1 = y - z;
  float r2 = 2*a;
  __ASTREE_log_vars((y,z,r1,r2;interv));
  __ASTREE_assert((r1==r2));
}
```

Analysis (all roundings)

```
% astree fltdbl-a.c --exec-fn main | egrep " in | WARN"
y in [1.1258999e+15, 1.1259001e+15]
z in [1.1258999e+15, 1.1259001e+15]
r1 in [-134217730., 134217730.]
r2 in {2.}
fltdbl-a.c:9.19-25::[call#main@1::]: WARN: assert failure
```
Floating-point relational domains

Solution

- keep reasoning in rationals: $\mathcal{D}^\#$ still abstracts $\mathcal{P}(\text{Var} \to \mathbb{Q})$
- translate float expressions into real expressions, making rounding errors explicit
- we use linearization, abstracting rounding using non-deterministic intervals

Example

On 32-bit single precision floats:

- relative error of magnitude $2^{-23}$ (normalized)
- absolute error of magnitude $2^{-159}$ (denormalized)

$Z \leftarrow X \oplus (0.25 \otimes X)$ is linearized into:

$Z \leftarrow [0.749 \cdots ; 0.750 \cdots] \times X + 2.35 \cdots 10^{-159}[-1; 1]$
Floating-point relational domains

We can implement $[[\cdot]]^\#$ fully in float:

- round upper bounds towards $+\infty$, lower bounds towards $-\infty$
- huge time gain (wrt. exact rationals)
- (small) precision loss

Tower of abstractions

deterministic IEEE semantics

$\Downarrow$

interval linear form

$\Downarrow$

domain-specific abstractions $[[\cdot]]^\#$

$\Downarrow$

float implementation of $[[\cdot]]^\#$
Numerical filters

2d order filter

\[ I_n : \text{input at time } n \]
\[ O_n : \text{output at time } n \]
\[ O_n = \alpha O_{n-1} + \beta O_{n-2} + aI_n + bI_{n-1} + cI_{n-2} \]

Ellipsoid domain

\[ D^\# \simeq \bigwedge_{X,Y \in \text{Var}} (Y^2 - aYX - bX^2 \leq c) \]

- discovers the variables \( X, Y \in \text{Var} \)
- discovers stable values \( a, b, c \in \mathbb{R} \)
Numerical filter example

```c
1  int INIT = 1;
2  float P, X, E1, E2, S1, S2, INPUT;
3  __ASTREE_volatile_input((INPUT [-10,10]));
4
5  void filtre2 () {
6     if (INIT) P = S1 = E1 = X;
7     else P = (0.4677826 * X) -
8          (E1 * 0.7700725) + (E2 * 0.4344376) +
9          (S1 * 1.5419) - (S2 * 0.6740477);
10    E2 = E1; E1 = X; S2 = S1; S1 = P;
11 }
12
13 void main () {
14    while (1) {
15        X = INPUT; filtre2(); INIT = INPUT;
16    }
17 }
```
Numerical filter example analysis

Analysis with filter domain

% astree filter.c --exec-fn main --dump-invariants
   | egrep "WARN|P in"
P in [-13.388927, 13.388927]
%

Analysis without filter domain

% astree filter.c --exec-fn main --no-filters --dump-invariants
   | egrep "WARN|P in"
filter.c:7.8-9.44::[call#main@13:loop@14>=4:call#filtre2@15::):
   WARN: double->float conversion range [-inf., +inf.]
   not included in [-3.4028235e+38, 3.4028235e+38]
P in [-3.4028235e+38, 3.4028235e+38]
%
Synchronous hypothesis

Synchronous programs
- the program has an implicit clock
- the clock ticks for a (configurable) maximal count

```c
#include <astree.h>

int X, B;

__ASTREE_volatile_input((B));
__ASTREE_max_clock((3600));

void main() {
    while (1) {
        if (B) X+=2; else X=0;
        __ASTREE_wait_for_clock();
    }
}
```
Clock relationship

Analysis with clock domain

% astree clock-ok.c --exec-fn main --dump-invariants
  | egrep "WARN|X in|\|X\|"
X in [0, 7202]
|X| <= 0. + clock *2. <= 7202.
%

clock is an implicit variable incremented at clock tick

The clock domain infers relations of the form: |V| ≤ αclock + β
with a linear cost

which in turns gives bounds for |V|
Arithmetic-geometric progressions

```c
void main() {
    float X;
    __ASTREE_known_fact((X>=0 && X <=100.));
    while (1) {
        X=X/101.;
        X=X*101.;
    }
}
```

Analysis result

```
% astree arigeo-bad.c --exec-fn main --dump-invariants | egrep "WARN\|X in"
arigeo-bad.c:6.4-12::[call#main@1:loop@4>=4::]: WARN: double->float conversion range [0., +inf.] not included in [-3.4028235e+38, 3.4028235e+38] X in [0., 3.4028235e+38]
%```
Arithmetic-geometric progressions

In float, $X/101. \times 101 \neq X$ due to rounding-errors

$\implies X$ is enlarged at each iteration step
Arithmetic-geometric progressions (corrected)

```c
void main() {
  float X;
  __ASTREE_known_fact((X>=0 && X <=100.));
  while (1) {
    X=X/101.;
    X=X*101.;
    __ASTREE_wait_for_clock();
  }
}
```

Analysis result

```
% astree arigeo-ok.c --exec-fn main --dump-invariants
  | egrep "WARN\|X in\|\X\"
X in [0., 235.91632]
|X| <= (99.999988 + 5.9950219e-37/(1.0000002-1))*(1.0000002)^clock - 5.9950219e-37/(1.0000002-1)
  <= 235.91632
%
```
The arithmetic-geometric deviation domain infers constraints:

$$|V| \leq \alpha (1 + a)^{\text{clock}} + \beta$$

which can provide bounds for $V$
Reduced product

**Formalization**

- Domain product $\mathcal{D}^\#$ = $\mathcal{D}_1^\# \times \mathcal{D}_2^\#$
- $\gamma_1 \times_2 (x_1, x_2) = \gamma_1(x_1) \cap \gamma_2(x_2)$
- Compute independently $\llbracket \cdot \rrbracket_1^\#$ and $\llbracket \cdot \rrbracket_2^\#$, and
- Reduce $\rho : \mathcal{D}^\# \rightarrow \mathcal{D}^\#$
  
  $\gamma_i(\rho(x)_i) \subseteq \gamma_i(x_i)$ et $\gamma_1 \times_2 (\rho(x)) = \gamma_1 \times_2 (x)$

**Applications**

- Interval $\leftrightarrow$ congruence reduction $[1; 7] \cap 2\mathbb{Z} = [2; 6] \cap 2\mathbb{Z}$
- Interval $\leftrightarrow$ octagon **partial** reduction
Widening and reduced product

In $D_1^\# \times D_2^\#$:

- $X_{i+1}^\# = X_i^\# \nabla_{1 \times 2} F^\#(X_i^\#)$ converges
- $X_{i+1}^\# = \rho(X_i^\# \nabla_{1 \times 2} F^\#(X_i^\#))$ may not converge

Solutions

- do not reduce successive iterates
  ($\Rightarrow$ precision loss)
- define a new widening $\nabla$ for the product $D_1^\# \times D_2^\#$
  ($\Rightarrow$ more work to do, and an extra proof of convergence)
Communications between domains

Abstract domain modules communicate invariants through channels

**Principe**

\[ \llbracket \cdot \rrbracket_1^{\#} \] can use an information \( X_2^{\#} \) computed by \( D_2^{\#} \)

- \( X_2^{\#} \) not necessarily representable in \( D_1^{\#} \)
- less systematic than the reduced product \( D_1^{\#} \times D_2^{\#} \)

**Application:** numerical filtering domain

Intervals used as pivot information:

- input: filter initialization from an interval (not found by the filter domain)
- output: interval bounding the filter output (usable by other domains)
Partitioning domains
Boolean decision trees

Issue

In our programs, control-flow is often encoded in booleans

Example

\[ B = (X > 0); \]
\[ \quad \vdots \quad \text{long code} \]
\[ \quad \vdots \]
\[ \text{if } (B) \; Y = 1/X; \]

\( (B = 1 \land X > 0) \lor (B = 0 \land X \leq 0) \) not convex

We need to partition wrt. the value of \( B \)
Boolean decision trees

- booleans in nodes
- numerical domains at leaves (intervals, octagons)
- opportunistic sharing (e.g. $B=?$), not maximal
- packing, using local dependency pre-analysis
Trace partitioning: test partitioning

Idea

A program transformation to improve the analysis precision

Example

\[
\begin{align*}
\text{if (\ldots) } & \ i=0; \\
\text{else } & \ i=1; \quad \Rightarrow \quad \text{if (\ldots) } & \{ i=0; \ X=a[i]+b[i]; \} \\
& \text{else } & \{ i=1; \ X=a[i]+b[i]; \}
\end{align*}
\]

Unlike boolean partitioning: control, not data criterion
Trace partitioning: loop partitioning

We can also partition finite loops

**Before transformation**

```c
for (i = 0; i < 10 && x <= X[i]; i++)
  y = A[i] + B[i](x - X[i]);
```

**After transformation**

```c
if (x > X[0])
  y = A[0] + B[0](x - X[0]);
else if (x > X[1])
  y = A[1] + B[1](x - X[1]);
...  
else
```

The control after the loop exit is partitioning wrt. the number of loop iterations
We do not actually use a program transformation

**Semantical formalization**

- control points $L$
- concrete trace semantics: $D_t = (L \times (\text{Var} \rightarrow \mathbb{Z}))^*$
- abstraction: $D_t^\# \simeq (L^\#)^* \rightarrow D^\#$
- abstract control points $L^\#$:
  $$L^\# = \{ \text{if}_l = \text{true}, \text{if}_l = \text{false}, \text{loop}_l = i, \text{loop}_l \geq i \mid l \in L \}$$

**Intuition:** program states are enriched with an (abstract) history of the control flow

A local dependency pre-analysis is used to determine the abstract points $(L^\#)^*$ of interest
(cost / precision trade-off)
Memory domain
Basic memory model

**Memory abstraction**: in extension

Abstracted as $\mathcal{P}(\text{Var} \rightarrow T)$, Var fixed:

- one cell per scalar variable
- recursively split arrays and structures
- we can also smash big arrays on a single ’summary’ cell

Made possible as there is no dynamic memory allocation and the stack is fully known

**Pointers**

Concrete pointer $= \text{base} \in \text{Var} + \text{offset} \in \mathbb{Z}$

abstracted separately:

- base abstracted as a set $\subseteq \text{Var}$
- offset abstracted using a synthetic integer variable
**Pointer analysis example**

```c
ptr.c
1  struct { int a; int b; } s;
2  void main() {
3      int b, *p = (int*)&s;
4      if (b) p++;
5      *p = 12;
6      __ASTREE_log_vars((p,s;inter,cong,ptr));
7  }
```

**Analysis result**

```bash
% astree ptr.c --exec-fn main | egrep "WARN| in |="
base(p) = { s }
off(p) in [0, 4] ∧ (4Z)
s.b in [0, 12] ∧ (12Z)
s.a in [0, 12] ∧ (12Z)
%```
Advanced memory model

Dealing with union types, pointer arithmetic and pointer casts

There may be aliasing at the byte level

Example

union {
  struct { uint8 al,ah,bl,bh; } b;
  struct { uint16 ax,bx; } w;
} r;

r.w.ax = 0; r.b.ah = 2;

Solution

⇒ as before, we abstract the memory as $\mathcal{P}(V \rightarrow T)$, $V$ fixed
- allocate cells of arbitrary scalar type at arbitrary offset
- when cells overlap, use an intersection semantics
- create only as needed, using reduction
Memory copy example

memcpy.c

```c
void memcpy(char* dst, const char* src, unsigned size) {
    int i;
    __ASTREE_unroll((10))
        for (i=0;i<size;i++) dst[i] = src[i];
}

void main() {
    float x = 10, y;
    memcpy(&y,&x,sizeof(y));
    __ASTREE_log_vars((x,y;inter,mem));
}
```

Generic (untyped) byte-per-byte memory copy function
At the end of the loop, the memory equality predicate gives $y = x$. 

i-th unrolled iteration:
- materializes $((\text{char}*)\&x)[i]$ as $[0, 255]$
- creates $((\text{char}*)\&y)[i]$ and store $[0, 255]$
- updates a predicate: $i$-th first bytes of $x$ and $y$ are equal
Results
analyses of industrial avionic codes
performed on a AMD Opteron 248, 64-bit, mono-processor

<table>
<thead>
<tr>
<th># lines</th>
<th>times</th>
<th>memory</th>
<th>alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>370</td>
<td>3.1s</td>
<td>16 MB</td>
<td>0</td>
</tr>
<tr>
<td>9 500</td>
<td>160s</td>
<td>80 MB</td>
<td>8</td>
</tr>
<tr>
<td>70 000</td>
<td>1h 16mn</td>
<td>582 MB</td>
<td>0</td>
</tr>
<tr>
<td>226 000</td>
<td>8h 5mn</td>
<td>1.3 GB</td>
<td>1</td>
</tr>
<tr>
<td>400 000</td>
<td>13h 52mn</td>
<td>2.2 GB</td>
<td>0</td>
</tr>
</tbody>
</table>

On the last versions: 0 alarm
⇒ the absence of run-time errors is proved
### Analysis parametrization

#### Parameters
- which domains to activate (22 available)
- iteration parameters (unrolling, widening, etc.)
- partitioning parameters
- packing parameters

130 analysis options, 26 code directives

#### Automated parametrization
- octagon and boolean partition packing
- trace partitioning
Analysis parallelization

Idea

When the control-flow is not statically deterministic, the analysis must explore all paths.

$\Longrightarrow$ we can distribute the cases on several CPUs.

Example: sequencer

```c
while (1) {
    (f[i])();
    i = (i+1) mod 12;
    __ASTREE_wait_for_clock();
}
```

After some iterations, $i \in [0; 11]$

all $f[i]$ are analyzed at each iteration.
Results

- huge cost for communication invariants (despite compression) ⇒ only worth for large computations between split and join (e.g., sequencers, not local if)
- CPU load is difficult to predict ⇒ we randomize tasks
- time cost is about $0.75/n + 0.25$ for $n$ CPUs ⇒ most useful for $n = 3$ or $4$
Interactive invariant visualisation

Web-based interface

% astree filter.c --exec-fn main --export-invariant stat
% astree --reload invariant.data --web-visu &
% firefox http://localhost:8080 &

invariant.data can grow large

⇒ does not scale-up well to large programs
It is possible to build a static analyzer that is:

- efficient in time, memory, and development cost
- very precise on a given (infinite) class of programs

Recipe

- start from a simple analyzer
- while there are false alarms
  - find their cause, and either
    - tune analysis parameters, or
    - improve automatic parametrization, or
    - improve some existing domain, or
    - add some reduction between existing domains, or
    - add a new domain
Future work

Still much research to do:

- improve invariant visualisation
- automate discovery of the cause of alarms
- take into account or prove user-given properties (functional or temporal properties, etc.)
- consider other programming languages (ADA, Simulink, assembly)
- consider different / more complex program classes
  - other kinds of embedded software (space, automotive, etc.)
  - parallel programs
  - programs with complex data-structures