Astrée and the static analysis of reactive control programs

Laure Gonnord David Monniaux

November 3, 2015

Plan

Astrée

- Architecture
- Memory
- Numerical domains
- Iteration trickery

Filters

Astrée

Static analyzer for proving

- absence of runtime errors
- absence of assertion violations (`assert()`)

Takes C (subset of C) code as input

Output an exhaustive list of **possible violations**

Plan

General architecture

C source

$\downarrow$ C lexer and parser

C AST

$\downarrow$ C typer

C typed/simplified AST

$\downarrow$ iterator

(optional) printout of invariants
printout of possible errors

Lexing / parsing + typing

- C parsing is almost context-free
  Almost: handling of `typedef`

```c
typedef int foo;

extern int in(void);

int main() {
  int bar = in();
  return (foo)(bar);
}
```

```c
extern int foo(int);

extern int in(void);

int main() {
  int bar = in();
  return (foo)(bar);
}
```

- C typing (integer operations and promotions) is surprisingly tricky

Iterator architecture

syntax-directed iterator

↓

domain of forward jumps (`break`, `continue`, `goto`)

↓

memory domain

↓

numerical domain “interchange”

↓

numerical domains

Forward jumps

Carry on:

- a "normal flow" abstract element
- `break`, `continue`: a stack (one level per loop nesting)
- one abstract element per label to which a `goto` is made

For backward `goto`, possibility to add a fixed point around (a bit painful and not needed by most software).

Plan

Memory model

C memory = “separate” memory blocks

Base pointer (incomparable) + offset

Memory abstraction mk.1 “Java-like”

memory domain = array of cells

each cell = pointer to set of other cells (or invalid), or index of variable into numerical domain

arrays:

- either “smashed” (one single may-alias cell: all writes are may-writes)
- either expanded

kludges when programs to analyze use type aliasing or pointer arithmetic

Memory model, mk.2

(Antoine Miné, LCTES'06)

Pointer = block identifier + offset (numeric variable)

View each block as an array of bytes

View numeric data as superimposed on this byte array

A practical note on implementation

Several layers of indexed maps (variable $\rightarrow$ memory domain cell, memory domain cell $\rightarrow$ numeric variable)

When control flow splits, two maps that may get altered differently

In an if-then-else, maps exiting both branches are almost the same

The cost of merge ($\sqcup$) should be counted wrt the number of updated variables, not the total number of variables.

In large-scale control code (l = number of lines):

- total # of variables $= \Theta(l)$
- total # of tests $= \Theta(l)$

If "linear cost" of $\sqcup$: total $\Theta(l^2)$, intolerable.

Data structures

Important: identical sub-parts of partials maps $X \to Y$ should not be traversed (e.g. $\sqcup$ on intervals when most intervals are identical)

- **Patricia trees**: trees indexed by the binary decomposition of the index (opportunistic sharing of sub-trees)
- **Balanced binary trees** (opportunistic sharing of sub-trees)
- **Hash-consing**?

Plan

Astrée

- Architecture
- Memory
- Numerical domains
- Iteration trickery

Filters

Interval Abstract Domain

- Classical domain [Cousot & Cousot '76]
- Minimum information needed to check the correctness conditions;
- **Not precise enough** to express a useful inductive invariant
  (thousands of false alarms);
- $\Longrightarrow$ must be refined by:
  - combining with existing domains through reduced product,
  - designing **new domains**, until all false alarms are eliminated.

Clock Abstract Domain

Code Sample:

```
R = 0;
while (1) {
  if (I)
    { R = R+1; }
  else
    { R = 0; }
  T = (R>=n);
  wait_for_clock ();
}
```

- Output T is true iff volatile input I true for last **n** clock ticks.
- Dlock ticks every s seconds for at most h hours, thus **R bounded**.
- To prove that **R cannot overflow**, prove that **R cannot exceed the elapsed clock ticks** (impossible using only intervals).

Solution:

- We add a phantom variable clock
- *For each variable* X*, we abstract three intervals:* X*,*

Octagon Abstract Domain

Code Sample:

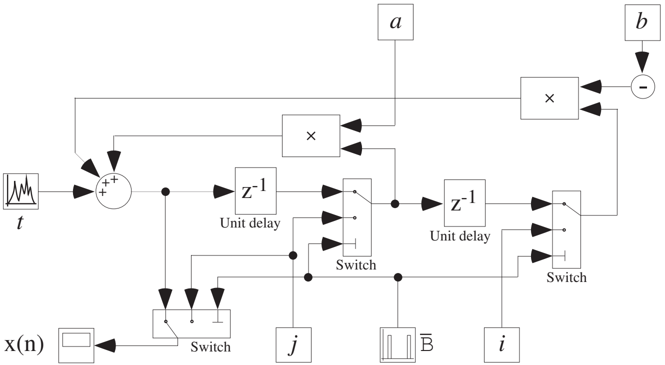
```
while (1) {
  R = A-Z;
  L = A;
  if (R>V)
    { ★ L = Z+V; }
  ★
}
```

- At ★, intervals give L $\leq$ max(max A, (max Z)+(max V)).
- In fact, we have L $\leq$ A.
- To discover this, we must know at ★ that R = A-Z and R $>$ V.

Solution: we need a numerical *relational abstract domain.*

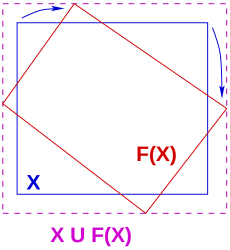
- *Octagons a good cost / precision trade-off.*
- *Invariants of the form* $\pm x \pm y \leq c$, with $\mathcal{O}(\mathbf{N^2})$ memory and $\mathcal{O}(\mathbf{N^3})$ time cost.
- Here, R = A-Z cannot be discovered, but we get L-Z $\leq$ max R which is sufficient.
- We use many octagons on small packs of variables

Block diagram

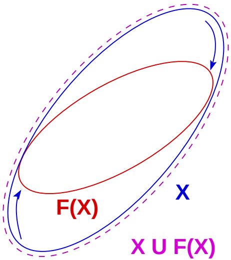


Ellipsoids

- Computes $X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \\ I_n \end{cases}$
- The concrete computation is **bounded**, which must be proved in the abstract.
- There is no stable interval or octagon.
- The simplest stable surface is an *ellipsoid.*



unstable interval



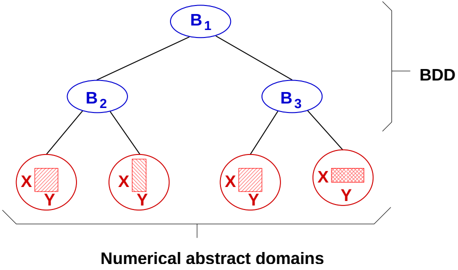
stable ellipsoid

Decision Tree Abstract Domain

Synchronous reactive programs **encode control flow in boolean variables**.

```
bool B1,B2,B3;
float N,X,Y;
N = f(B1);
if (B1)
  { X = g(N); }
else
  { Y = h(N); }
```

Decision Tree:



Too many booleans (**4 000**) to build one big tree so we:

- limit the **BDD height** to 3 (analysis parameter);
- use a **syntactic criterion** to select variables in the BDD and the numerical parts.

Relational Domains on Floating-Point

Problems:

- Relational numerical abstract domains rely on a **perfect mathematical concrete semantics** (in $\mathbb{R}$ or $\mathbb{Q}$).
- Perfect arithmetics in $\mathbb{R}$ or $\mathbb{Q}$ is costly.
- IEEE 754-1985 floating-point concrete semantics **incurs rounding**.

Solution:

- Build an **abstract mathematical semantics** *in* $\mathbb{R}$ *that over-approximates the concrete floating-point semantics, including rounding.*
- *Implement the abstract domains on* $\mathbb{R}$ *using floating-point numbers rounded in a sound way.*

Plan

Astrée

- Architecture
- Memory
- Numerical domains
- Iteration trickery

Filters

Basic iterator: recursive descent

On the syntactic structure of programs (not CFG).

- **Assignment**: forward abstract propagation
- **Procedure call**: recurse into procedure (virtual inlining)
- **Tests / switches**: go into each branch after filtering by guard, $\sqcup$ at the end
- **Loops**: fixed point

Iteration Refinement: Loop Unrolling

Principle:

- Semantically equivalent to:

```
while (B) { C }  ⟹  if (B) { C }; while
                   (B) { C }
```

- More precise in the abstract:
  - **less** concrete execution paths are **merged** in the abstract.

Application:

- Isolate the **initialization phase** in a loop (e.g. first iteration).

Iteration Refinement: Trace Partitioning

Principle:

- Semantically equivalent to:

```
if (B) { C1 } else { C2 }; C3
```

$$\Downarrow$$

```
if (B) { C1; C3 } else { C2; C3 };
```

- More precise in the abstract:
  - concrete execution paths are **merged later**.

Application:

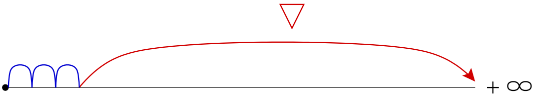
```
if (B)
  { X=0; Y=1; }
else
  { X=1; Y=0; }
R = 1 / (X-Y);
```

/ cannot result in a division by zero

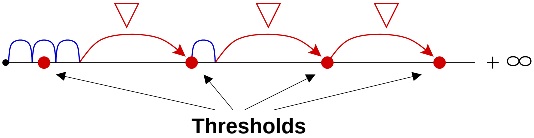
Convergence Accelerator: Widening

Principle:

- Brute-force widening:



- Widening **with thresholds**:



Examples:

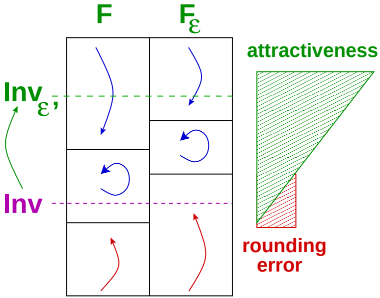
- 1., 10., 100., 1000., etc. for floating-point variables;
- maximal values of data types;
- syntactic program constants, etc.

Fixpoint Stabilization for Floating-point

Problem:

- Mathematically, we look for an abstract invariant **inv** such that $F(\textbf{inv}) \subseteq \textbf{inv}$.
- Unfortunately, abstract computation uses floating-point and incurs rounding: maybe $F_\varepsilon(\textbf{inv}) \not\subseteq \textbf{inv}$!

Solution:



- Widen **inv** to $\textbf{inv}_{\varepsilon'}$ with the hope to jump into a **stable zone of F_ε**.
- Works if F has some **attractiveness** property (otherwise iteration goes on).
- ε' is an analysis parameter.

Plan

Astrée
- Architecture
- Memory
- Numerical domains
- Iteration trickery

Filters

The problem

Discrete-time digital filters implemented in software (general-purpose CPUs, DSPs) or in hardware.
A lot of the filtering linear: what we'll deal with
Implemented in fixed- or **floating-point**.
Need to provide sound assurance of absence of **runtime errors** in the program, **including overflows**.
Thus **need to bound all filter outputs** (and all intermediate values).

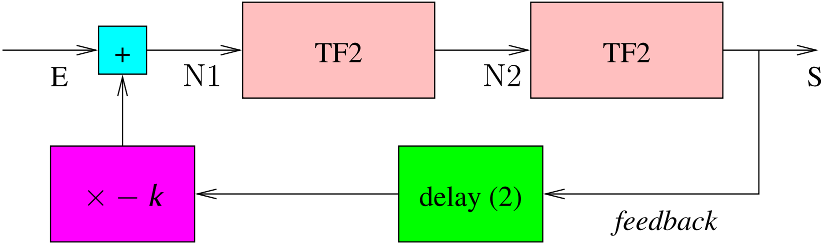
Discrete-time causal linear filters

Inputs and outputs **streams** of data on “wires”
Complex filters made of elementary blocks connected by wires:

- **delays** (buffer for 1 or more clock tick(s))
- multiplication by a scalar
- addition of 2 streams

Network topology may contain **feedback loops** going through a delay

Example of complex filter



Each TF2 element itself a complex filter with internal filter feedback loop.

Causal, time-invariant linearity

Causal: values at clock tick $n \geq 0$ depend on those at clock ticks $\geq n$ only
(Non causal filters typically need entire buffering of the data — we do not cover them here.)

Linearity: outputs a **linear** function of the inputs (**over the reals**)
$\Rightarrow$ each output at time n a linear function of the inputs at times $\geq n$

Time invariance: this function is always the same **convolution**, i.e.

$$o^{(n)} = \sum_k t^{(k)} i^{(n-k)}$$

Transfer function

Several inputs and initial values in the “delay” operators:

$$o^{(n)} = \sum_x \sum_k r_x^{(k)} i_x^{(n-k)} + \sum_y k_y d_y^{(n)}$$

Define $O = \sum_{n=0}^{\infty} o^{(n)} z^n$ a formal power series. The equation becomes

$$O = \sum_x T_x . I_x + \sum_y k_y . D_y$$

k_y initial value of delay labeled y;
I_x series for input stream labeled x; T_x **unit response** for input x (**Z-transform**)

Bounding the transfer

We know some bound $[-B_x, B_x]$ on input I_x: $\|I_x\|_\infty \leq B_x$.

What is the $\|I \mapsto T.I\|$ **operator norm** w.r.t $\|\cdot\|_\infty$?
I.e. the least M such that $\|T.I\|_\infty \leq M.\|I\|_\infty$.

Answer: $\|I \mapsto T.I\| = \|T\|_1$ with $\|T\| = \sum_n |t_n|$.

Then

$$\|O\|_\infty \leq \sum_x \|T_x\|_1.\|I_x\|_\infty + \left\|\sum_y k_y.D.y\right\|_\infty$$

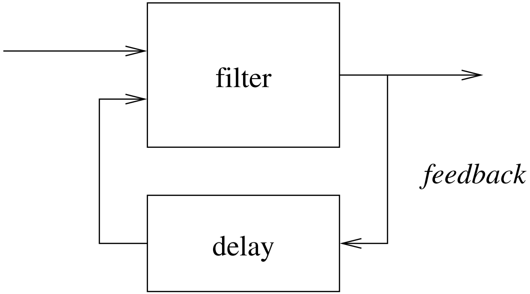
Examples

Multiplication by a scalar: $T = \alpha$,

Addition: $T_1 = 1$, $T_2 = 1$

Delay: by n clock ticks, $T = z^n$

Feedback loop



Filter F with two inputs I and L, output L fed back into L through unit delay (we leave out the initialization):
$O = T_I.I + T_L.L = T_I.I + T_L.zO$ and thus $O = T.I$ with

$$T = (1 - zT_L)^{-1}.T_I.I$$

Rational functions

All the T power series that we construct are the developments around 0 of **rational functions** $P(z)/Q(z)$ (P, Q polynomials, $Q(0) = 1$) – ring $\mathbb{R}[z]_{(z)}$.

If a filter has m inputs I_1, ..., I_n, r initialization values k_1, ..., k_r, and n outputs O_1, ..., O_m, then

T is a $n \times m$ matrix over $\mathbb{R}[z]_{(z)}$; D is a $n \times r$ matrix over $\mathbb{R}[z]_{(z)}$;

K is a m-vector of $\mathbb{R}$;

I is a m-vector of $\mathbb{R}[z]_{(z)}[[z]]$ (series); O is a n-vector of $\mathbb{R}[z]_{(z)}[[z]]$ (series) and

$$O = T.I + D.K$$

Feedback loops

Take a filter $F(T^F, D^F...)$, feedback its n outputs into the last of its m inputs.
$O = T^F_1.I + T^F_2.zO + D.K$ and thus

$$O = (\mathrm{Id}_n - z.T^F_2)^{-1}.(T^F_1.I + D.K)$$

(this matrix is necessarily **invertible**)
Computations doable over $\mathbb{Q}[z]_{(z)}$!

Summary

Any of the filters can be summarized by **matrices** of **rational functions** over the rationals.
These matrices can be computed simply from the coefficients of the various elementary blocks **or from the matrices of whole sub-filters** (compositional design).
Example: filter $O^{(n)} = \sum_{k=0}^{d} \alpha_k I^{(n-k)} + \sum_{k=1}^{e} \beta_k O^{(n-k)}$ has transfer function

$$\frac{\alpha_0 + \alpha_1 z + \cdots + \alpha_d z^d}{1 - \beta_1 z - \cdots - \beta_e z^e}$$

$d = e = 2$: TF2 filter in our example

Bounding the output

Let N_x apply $\|\cdot\|_x$ to all coordinates in a matrix or vector.
Then $N_\infty(O) \leq N_1(T).N_\infty(I) + N_\infty(D.K)$
The main problem: given a rational function $P(z)/Q(z)$, $Q(0) \neq 0$, **give an upper bound on $\|P/Q\|_1$** (of its development around 0).
Idea:

- compute explicitly the first N terms of this development, compute a bound for $\|P/Q\|_1^{<N}$
- bound the tail: $\|P/Q\|_1^{\geq N}$

Explicit development

Development of P/Q: division by ascending powers of $P(z)$ by $Q(z)$ (eqv. to running a filter $O^{(n)} = \sum_{k=0}^{d} p_k I^{(n-k)} - \sum_{k=1}^{e} q_k O^{(n-k)}$).
Problem: **numerical instability** using **interval arithmetics on floating-point numbers**.
After a while, error on the same order as the coefficients, **sign of the coefficients unknown**, then quick amplification.
Solution: develop until sign of the coefficients unknown. (Can go further with extended-precision arithmetic, see GNU MP's MPFR).

Tail bounding

Poles: the zeroes of $Q(z)$ are called **poles** of $P(z)/Q(z)$
The poles determine the behavior of the system.
Theorem: system stable ($\|P/Q\|_1 < \infty$) iff all poles have absolute value > 1
Intuition: (distinct poles) $[P(z)/Q(z)]^{(n)} = \sum_i \alpha_i \xi^{-n}$, ξ_i poles.
Theorem: let R be the remainder of the division by ascending powers of P/Q up to order N, then

$$\|P/Q\|_1^{\geq N} \leq \frac{\|R\|_1}{(|\xi_1| - 1) \dots (|\xi_n| - 1)}$$

Tail bounding

Implementation: Good algorithms and libraries (GSL...) for finding approximate roots $\tilde{\xi}_i$ of polynomial Q
Methods for sound bounds on the error on a root ($|\tilde{\xi}_i - \xi| \leq e(Q, \tilde{\xi}_i)$)

Decomposition

Let $\tilde{O}$ be the output of the filter implemented in fixed- or floating- point, O the ideal real output, then we obtain for single input, output, no initialization:

$$\|\tilde{O} - O\|_\infty \leq \varepsilon_{\mathrm{rel}}.\|I\|_\infty + \varepsilon_{\mathrm{abs}}$$

$\varepsilon_{\mathrm{rel}}$ **relative error**, $\varepsilon_{\mathrm{abs}}$ **absolute error**.
In general: $N_\infty(\tilde{O} - O) \leq \varepsilon_{\mathrm{rel},T}.N_\infty(I) + \varepsilon_{\mathrm{rel},D}.N_\infty(K) + \varepsilon_{\mathrm{abs}}$
with $\varepsilon_{\mathrm{rel},T}$, $\varepsilon_{\mathrm{rel},D}$ matrices in $\mathbb{R}_+$, $\varepsilon_{\mathrm{abs}}$ vector in $\mathbb{R}_+$

Error in elementary blocks

$x \oplus y$ in fixed- or floating-point = $r(x+y)$, $x \otimes y$ in fixed- or floating-point = $r(x.y)$; r **rounding function** such that $|r(x) - x| \leq \varepsilon_{\mathrm{rel}}.|x| + \varepsilon_{\mathrm{abs}}$

Fixed-point: $\varepsilon_{\mathrm{rel}} = 0$, $\varepsilon_{\mathrm{abs}} = u/2$ (round-to-nearest), $\varepsilon_a bs = u$ (other modes) with u the least representable positive number

Floating-point: $\varepsilon_{\mathrm{rel}}$ error at the n-th binary position after the point

$\varepsilon_{\mathrm{abs}}$ is the **denormalization** error: ex, if u is the least representable positive number, then $0.25 \otimes u = 0$ ($\varepsilon_{\mathrm{abs}}$ very small, but included for soundness)

$|r(x) - x| \leq \max(\varepsilon_{\mathrm{rel}}.|x|, \varepsilon_{\mathrm{abs}})$ overapproximated by affine form

Error propagation

Simple blocks: With the above: easy for addition and multiplication by scalar, no error on delays

Feedback loop: around filter F: somewhat complex computation ending up with $\varepsilon_{\text{rel}} = A.\varepsilon_{\text{rel}} + B$ with A with very small coefficients $\Rightarrow$ resolution by fixpoint iteration

Simplification: with a P/Q filter (P, Q with rational coefficients, can be large), replace by $\tilde{P}/\tilde{Q}$ (approximations for P and Q) with some larger ε_{rel}

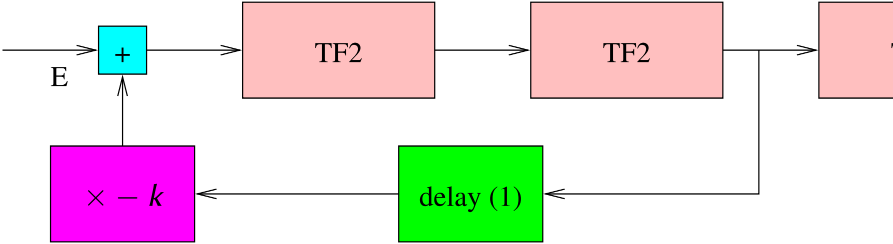
To summarize

Any causal linear filter F with finite memory implemented over fixed-point or floating-point (or a mix thereof) can be summarized into:

- matrices T^F and D^F over $\mathbb{Q}[z]_{(z)}$ expressing the ideal, real input-output relationship by **Z-transform**: T wrt input streams, D wrt values initially in the delay memories
- matrices $\varepsilon^F_{\mathrm{rel},T}$ and $\varepsilon^F_{\mathrm{abs},D}$ over $\mathbb{R}_+$ expressing the **relative errors** wrt input streams and delay memories
- vector $\varepsilon^F_{\mathrm{abs}}$ of **absolute error**

This is **compositional**: computation for complex filters from the analysis results of sub-filters.

Implementation results



Defined compositionally in the analyzer
Computation in 0.1s (could be optimized), P/Q P of 6th degree, Q of 7th degree
$\varepsilon_{\mathrm{rel}} \leq 4.781.10^{-13}$, $\|O\|_\infty \leq 2.0576\|I\|_\infty$.

Reconstruction of filters

From C or similar program (SSA form)

```
while (1) {
  ...
  filter
}
```

Read all lines in filter, obtain $v = e$ equations from $v := e$ assignment; variables v in e not already initialized in loop become $z.v$

In case of nonlinearity: use approximation to remove the linear part and get large $\varepsilon_{\mathrm{rel}}$.

Solve the resulting system.

Summary

Compositional abstract semantics for fixed- and floating-point digital linear filters with fixed memory of **arbitrary complexity**.

Sound bounds on the output obtained as affine function of bounds on the inputs.

Simple implementation already gives good results.

Good for analysis of data-flow languages for automatic systems (graphical Scade etc.). Extension for imperative languages.