Astrée and the static analysis of reactive control programs

Laure Gonnord   David Monniaux

2016

(general intro on synchronous systems, on board ?)

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

Architecture

Memory

Numerical domains

Iteration trickery

General architecture

C source

↓ C lexer and parser

C AST

↓ C typer

C typed/simplified AST

↓ iterator

(optional) printout of invariants
printout of possible errors

Lexing / parsing + typing

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

- 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

Architecture

Memory

Numerical domains

Iteration trickery

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,

Other

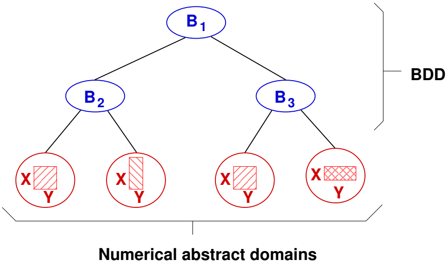
Octogons, ellipsoids, filters ...

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.

Plan

Architecture

Memory

Numerical domains

Iteration trickery

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 }` $\Longrightarrow$ `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
                ⇓
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

Already seen