PLDI'03

A Static Analyzer for Large Safety-Critical Software

B. Blanchet, P. Cousot, R. Cousot, J. Feret
L. Mauborgne, A. Miné, D. Monniaux, X. Rival

CNRS
École normale supérieure
École polytechnique
Paris France

Automatic Program Verification by Abstract Interpretation

Result:

- Can produce **zero or very few false alarms** while checking **non-trivial properties** (absence of Run-Time Error);
- **Does scale up**.

How ?

- We **specialize** the abstract interpreter for a **family of programs** (which correctness proofs would be similar).
- The abstract domains are **generic** invariants **automatically** instantiated by the analyzer (to make these proofs).

Considered Programs and Semantics

Which Programs are Considered ?

- Embedded avionic programs;
- Automatically generated from a proprietary graphical system control language (à la Simulink);
- Synchronous real-time critical programs:

```
declare volatile input, state, and output variables;
initialize state variables;
loop forever
  read volatile input variables,
  compute output and state variables,
  write to volatile output variables;
  wait for next clock tick
end loop
```

Main Characteristics of the Programs

Difficulties:

- **Many global variables and arrays** ($>$ **10 000**);
- A **huge loop** ($>$ **75 000 lines** after simplification);
- Each iteration depends on the state of the previous iterations (state variables);
- **Floating-point** computations
  (**80% of the code** implements non-linear control with feed-back);
- Everything is **interdependent** (live variables analysis, slicing ineffective);
- Abstraction by elimination of any variable is too imprecise.

Simplicities:

- All data is **statically allocated**;
- Pointers are restricted to call-by-reference, **no pointer arithmetics**;
- Structured, **recursion-free** control flow.

Semantics

- The standard **ISO C99 semantics**:
  - arrays should not be accessed out of their bounds, ...

  restricted by:
- The **machine semantics:**
  - integer arithmetics is 2's complement,
  - floating point arithmetics is IEEE 754-1985,
  - `int` and `float` are 32-bit, `short` is 16-bit, ...

  restricted by:
- The **user's semantics:**
  - integer arithmetics should not wrap-around,
  - some IEEE exceptions (invalid operation, overflow, division by zero) should not occur, ...

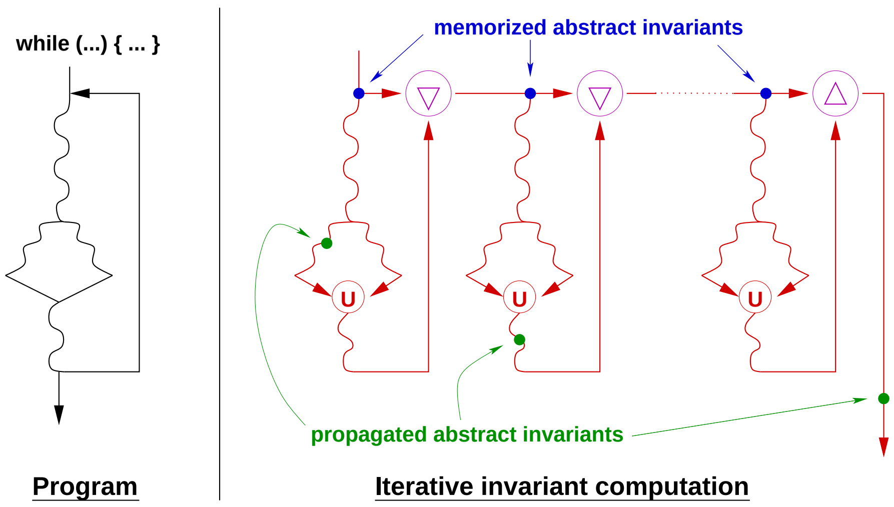
Goal of the Program Static Analyzer

- **Correctness verification**.
- Nothing can go wrong at execution:
  - no integer overflow or division by zero,
  - no exception, NaN, or $\pm\infty$ generated by IEEE floating-point arithmetics,
  - no out of bounds array access,
  - no erroneous type conversion.
- The execution semantics on the machine **never reaches an indetermination or an error case** in the standard / machine / user semantics.

Information about the Program Execution Automatically Inferred by the Analyzer

- The analyzer effectively computes a **finitely represented**, **compact** over-approximation of the **immense** reachable state space.
- The information is **valid for any execution** interacting with **any possible environment** (through undetermined volatiles).
- It is inferred **automatically** by abstract interpretation of the collecting semantics and convergence acceleration (∇, Δ).

Iterations to Over-Approximate the Reachable States



Abstract Domains

Choice of the Abstract Domains

Abstract Domain:

- Computer representation of a **class** of **program properties**;
- **Transformers** for propagation through expressions and commands;
- Primitives for **convergence acceleration**: ∇, Δ.

Composition of Abstract Domains:

- Essentially approximate **reduced product** (conjunction with simplification).

Design of Abstract Domains:

- Know-how;
- **Experimentation**.

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 the volatile input `I` has been true for the last `n` clock ticks.
- The clock ticks every `s` seconds for at most `h` hours, thus `R` **is bounded**.
- To prove that `R` **cannot overflow**, we must prove that `R` **cannot exceed the elapsed clock ticks** (impossible using only intervals).

Solution:

- We add a phantom variable `clock` in the concrete user semantics to track elapsed clock ticks.
- For each variable `X`, we abstract **three intervals**: `X`, `X+clock`, and `X-clock`.
- If `X+clock` or `X-clock` is bounded, so is `X`.

Octagon Abstract Domain

Code Sample:

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

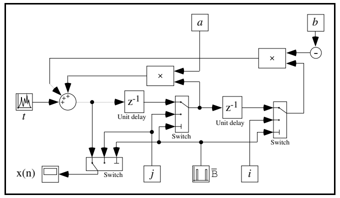
- At ★, the interval domain gives `L` ≤ max(max `A`, (max `Z`)+(max `V`)).
- In fact, we have `L` ≤ `A`.
- To discover this, we must know at ★ that `R` = `A-Z` and `R` > `V`.

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

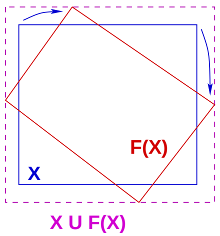
- The **octagon** abstract domain [Miné 03] is a good cost / precision trade-off.
- Invariants of the form $\pm x \pm y \le c$, with $\mathcal{O}(N^2)$ memory and $\mathcal{O}(N^3)$ time cost.
- Here, `R` = `A-Z` cannot be discovered, but we get `L-Z` ≤ max `R` which is sufficient.
- We use many octagons on **small packs** of variables instead of a large one using all variables to cut costs.

Ellipsoid Abstract Domain

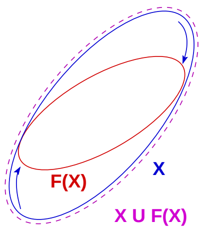
2^d Order Filter Sample:



- 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

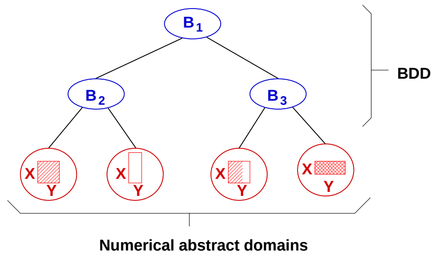
Decision Tree Abstract Domain

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

Code Sample:

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

Decision Tree:



There are 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.

Iteration Strategies for Fixpoint Approximation

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

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

Application:

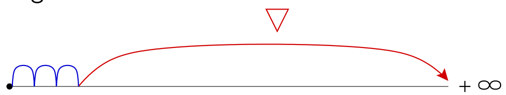
```
if (B)
  { X=0; Y=1; }
else
  { X=1; Y=0; }
R = 1 / (X-Y);
```

/ cannot result in a division by zero

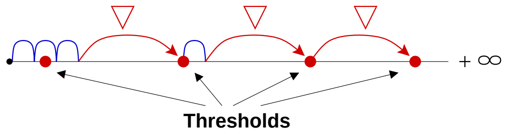
Convergence Accelerator: Widening

Principle:

- Brute-force widening:



- Widening **with thresholds**:



Examples:

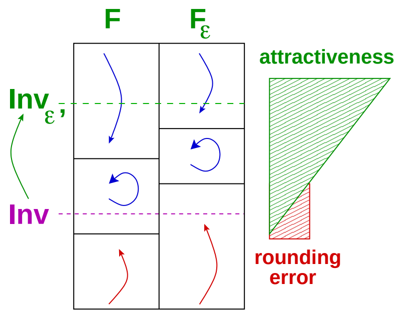
- 1., 10., 100., 1000., etc. for floating-point variables;
- maximal values of data types;
- syntactic program constants, etc.

Fixpoint Stabilization for Floating-point

Problem:

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

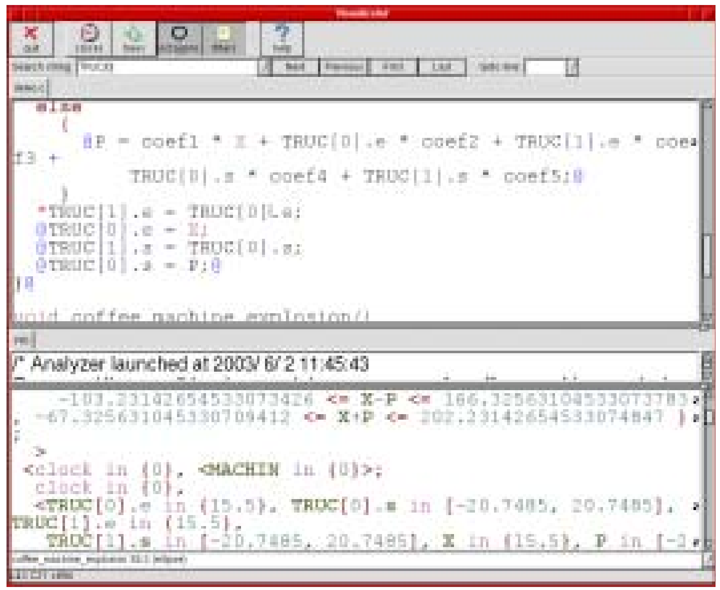
Solution:



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

Results

Example of Analysis Session



Results

- **Efficient:**
  - tested on two **75 000** lines programs,
  - **120 min** and **37 min** computation time on a 2.8GHz PC,
  - **200 Mb** memory usage.
- **Precise:**
  - **11** and **3** lines containing a warning.
- **Exhaustive:**
  - full control and data **coverage** (unlike checking, testing, simulation).

Conclusion

- **Success story**:
  - we succeed where **a commercial abstract interpretation-based static analysis tool failed**
    (because of prohibitive time and memory consumption and very large number of false alarms);
- **Usable** in practice for verification:
  - **directly applicable** to other similar programs
    by changing some analyzer parameters,
  - approach **generalizable** to other program families
    by including new abstract domains and specializing the iteration strategy.
    (Work in progress: power-on self-test for a family of embedded systems.)