Static Analysis and Verification of Aerospace Software by Abstract Interpretation

Patrick Cousot and Radhia Cousot
École normale supérieure, Paris École normale supérieure & CNRS, Paris

Motivation

Computer scientists have made great contributions to the failure of complex systems

Ariane 5.01 failure (overflow) Patriot failure (float rounding) Mars orbiter loss (unit error)

• Checking the presence of bugs is great but never ends
• Proving their absence is even better!

Content

• Brief motivation
• An informal introduction to abstract interpretation
• A short overview of a few applications and on-going work at ENS on aerospace software

A recent comprehensive overview paper (with all theoretical and practical details and references):
J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné and X. Rival
Static analysis and verification of aerospace software by abstract interpretation
AIAA Infotech@Aerospace 2010, Atlanta, Georgia, USA, April 20, 2010

Joint work with:

Julien Bertrane
École normale supérieure, Paris

Laurent Mauborgne
École normale supérieure & INRIA, Paris

Antoine Miné
École normale supérieure & CNRS, Paris

Xavier Rival
École normale supérieure & CNRS, Paris

Workshop on formal verification of avionics software products
Airbus France, Toulouse, France
June 24, 2010
Abstract interpretation

- Started in the 70’s and well-developed since then
- Originally for inferring program invariants (with first applications to compilation, optimization, program transformation, to help hand-made proofs, etc)
- Based on the idea that undecidability and complexity of automated program analysis can be fought by approximation
- Applications evolved from static analysis to verification
- Does scale up!

Fighting undecidability and complexity in program verification

- Any automatic program verification method will definitely fail on infinitely many programs (Gödel)
- Solutions:
  - Ask for human help (theorem prover based deductive methods)
  - Consider (small enough) finite systems (model-checking)
  - Do sound approximations or complete abstractions (abstract interpretation)

An informal introduction to abstract interpretation
I) Define the programming language semantics

Formalize the concrete execution of programs (e.g. transition system)

II) Define the program properties of interest

Formalize what you are interested to know about program behaviors

III) Define which specification must be checked

Formalize what you are interested to prove about program behaviors

IV) Choose the appropriate abstraction

Abstract away all information on program behaviors irrelevant to the proof
V) Mechanically verify in the abstract

The proof is fully *automatic*.

Forbidden zone

Abstraction of the trajectories

---

Soundness of the abstract verification

Never forget any possible case so the *abstract proof is correct in the concrete*.

Forbidden zone

Abstraction of the trajectories

---

Unsound validation: testing

Try a few cases

Forbidden zone

Test of a few trajectories

Error !!!

---

Unsound validation: bounded model-checking

Simulate the beginning of all executions

Forbidden zone

Bounded model-checking

Error !!!
Unsound validation: static analysis

Many static analysis tools are **unsound** (e.g. Coverity, etc.) so inconclusive

Incompleteness

When abstract proofs may fail while concrete proofs would succeed

True error

The abstract alarm may correspond to a concrete error

False alarm

The abstract alarm may correspond to no concrete error (false negative)
What to do about false alarms?

- **Automatic refinement**: inefficient and may not terminate (Gödel)
- **Domain-specific abstraction**:
  - Adapt the abstraction to the *programming paradigms* typically used in given *domain-specific applications*
  - e.g. *synchronous control/command*: no recursion, no dynamic memory allocation, maximum execution time, etc.

Target language and applications

- **C programming language**
  - Without recursion, long jump, dynamic memory allocation, conflicting side effects, backward jumps, system calls (stubs)
  - With all its horrors (union, pointer arithmetics, etc)
  - Reasonably extending the standard (e.g. size & endianess of integers, IEEE 754-1985 floats, etc)
- **Synchronous control/command**
  - e.g. generated from Scade

The semantics of C implementations is very hard to define

What is the effect of out-of-bounds array indexing?

```c
#include <stdio.h>
int main () { int n, T[1];
    n = 2147483647;
    printf("n = %i, T[n] = %i\n", n, T[n]);
}
```

Yields different results on different machines:

<table>
<thead>
<tr>
<th>Machine</th>
<th>Result 1</th>
<th>Result 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macintosh PPC</td>
<td>n = 2147483647, T[n] = 2147483647</td>
<td>n = 2147483647, T[n] = 2147483647</td>
</tr>
<tr>
<td>Macintosh Intel</td>
<td>n = 2147483647, T[n] = -1208492044</td>
<td>n = 2147483647, T[n] = -135294988</td>
</tr>
<tr>
<td>PC Intel 32 bits</td>
<td>Bus error</td>
<td></td>
</tr>
<tr>
<td>PC Intel 64 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implicit specification

- Absence of runtime errors: overflows, division by zero, buffer overflows, null & dangling pointers, alignment errors, ...
- Semantics of runtime errors:
  - Terminating execution: stop (e.g., floating-point exceptions when traps are activated)
  - Predictable outcome: go on with worst case (e.g., signed integer overflows result in some integer, some options: e.g., modulo arithmetics)
  - Unpredictable outcome: stop (e.g., memory corruption)

Example of general purpose abstraction: octagons

- Invariants of the form $\pm x \pm y \leq c$, with $O(N^2)$ memory and $O(N^3)$ time cost.
- Example:

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

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

- 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 instead of a large one using all variables to cut costs.

Abstractions

- Collecting semantics:
  - Partial traces
- Intervals:
  - $x \in [a, b]$
- Simple congruences:
  - $x \equiv a[b]$
- Octagons:
  - $\pm x \pm y \leq a$
- Ellipses:
  - $x^2 + by^2 - axy \leq d$
- Exponentials:
  - $-a^{bt} \leq y(t) \leq a^{bt}$

Example of general purpose abstraction: decision trees

```c
/* boolean.c */
typedef enum {F=0, T=1} BOOL;
BOOL B;
void main () {
  unsigned int X, Y;
  while (1) {
    ...
    B = (X == 0);
    ...
    if (!B) {
      Y = 1 / X;
    }
    ...
  }
}
```

The boolean relation abstract domain is parameterized by the height of the decision tree (an analyzer option) and the abstract domain at the leaves.
Example of domain-specific abstraction: ellipses

typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN INIT; float P, X;
void filter () {
    static float E[2], S[2];
    if (INIT) { S[0] = X; P = X; E[0] = X; } 
    else { P = (((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4))
                + (S[0] * 1.5)) - (S[1] * 0.7)); }
    E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
    /* S[0], S[1] in [-1327.02698354, 1327.02698354] */
}
void main() { X = 0.2 * X + 5; INIT = TRUE;
    while (1) {
        X = 0.9 * X + 35; /* simulated filter input */
        filter(); INIT = FALSE; }
}

Example of domain-specific abstraction: exponentials

% cat count.c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
volatile BOOLEAN I; int R; BOOLEAN T;
void main() {
    R = 0;
    while (TRUE) {
        __ASTREE_log_vars((R));
        if (I) { R = R + 1; }
        else { R = 0; }
        T = (R >= 100); 
        __ASTREE_wait_for_clock();
    }
}
% cat count.config
__ASTREE_log_vars((I [0,1]));
__ASTREE_wait_for_clock();
% astree -exec-fn main -config-sem count.config count.c|grep '|R' |R| <= 0. + clock *1. <= 3600001.

Example of domain-specific abstraction: exponentials

typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN INIT; float P, X;
void filter () {
    static float E[2], S[2];
    if (INIT) { S[0] = X; P = X; E[0] = X; } 
    else { P = (((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4))
                + (S[0] * 1.5)) - (S[1] * 0.7)); }
    E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
    /* S[0], S[1] in [-1327.02698354, 1327.02698354] */
}
void main() {
    X = 0.2 * X + 5; INIT = TRUE;
    while (1) {
        X = 0.9 * X + 35; /* simulated filter input */
        filter(); INIT = FALSE; }
}

Example of domain-specific abstraction: exponentials

void dev( ) {
    X=E;
    if (FIRST) { P = X; }
    else { P = ((P - (((2.0 * P) - A) - B)
                * 4.91048e-03)); };
    B = A;
    if (SWITCH) { A = P; }
    else (A = X; }
}

Example of domain-specific abstraction: exponentials

% cat retro.c
typedef enum {FALSE=0, TRUE=1} BOOL;
BOOL FIRST;
volatile BOOL SWITCH;
volatile float E;
float P, X, A, B;
void dev( ) {
    X=E;
    if (FIRST) { P = X; }
    else { P = ((P - (((2.0 * P) - A) - B)
                * 4.91048e-03)); };
    B = A;
    if (SWITCH) { A = P; }
    else (A = X; }

An erroneous common belief on static analyzers

“The properties that can be proved by static analyzers are often simple” [2]

Like in mathematics:

- May be simple to state (no overflow)
- But harder to discover (S[0], S[1] in [-1327.02698354, 1327.02698354])
- And difficult to prove (since it requires finding a non trivial non-linear invariant for second order filters with complex roots [Fer04], which can hardly be found by exhaustive enumeration)

Reference

Industrial applications

Examples of applications

- Verification of the absence of runtime-errors in
  - Fly-by-wire flight control systems
  - ATV docking system
  - Flight warning system (on-going work)

Industrialization

- 8 years of research (CNRS/ENS/INRIA):
  www.astree.ens.fr

- Industrialization by AbsInt (since Jan. 2010):
  www.absint.com/astree/

On-going work
Verification of compiled programs

- The valid source may be proved correct while the certified compiler is incorrect so the target program may go wrong.

- Possible approaches:
  - Verification at the target level
  - Source to target proof translation and proof check on the target
    - Translation validation (local verification of equivalence of run-time error free source and target)
  - Formally certified compilers

Imperfect synchrony

- Example of (buggy) communicating synchronous systems:
  - Synchronized and desynchronized executions:

- System 1 and System 2
  - negates previous input (on clocks C and C')
  - compare inputs
  - flawed alarms

Verification of imperfectly clocked synchronous systems

- Verification of target programs
Semantics and abstractions

- **Continuous semantics** (value $s(t)$ of signals $s$ at any time $t$)
- **Clock ticks and serial communications** do happen in known time intervals $[l, h], l \leq h$
- **Examples of abstractions:**
  - $\forall t \in [a; b] : s(t) = x$.
  - $\exists t \in [a; b] : s(t) = x$.
  - change counting ($\leq k, a \rightarrow \downarrow b$) and ($\geq k, a \rightarrow \uparrow b$)
    (signal changes less (more) than $k$ times in time interval $[a, b]$)

THÉSÉE: Verification of embedded real-time parallel C programs

Example of static analysis

For how long should the input be stabilized before deciding on disagreement?

Parallel programs

- Bounded number of processes with shared memory, events, semaphores, message queues, blackboards,…
- Processes created at initialization only
- Real time operating system (ARINC 653) with fixed priorities (highest priority runs first)
- Scheduled on a single processor

Verified properties

- Absence of runtime errors
- Absence of unprotected data races
Semantics

- No memory consistency model for C
- Optimizing compilers consider sequential processes out of their execution context

```c
init: flag1 = flag2 = 0
```

```c
process 1: process 2: write to flag1/2 and read of flag2/1 are independent so can be reordered \rightarrow error!
flag1 = 1; flag2 = 1;
if (!flag2) if (!flag1)
{
  /* critical section */
  /* critical section */
}
```

- We assume:
  - sequential consistency in absence of data race
  - for data races, values are limited by possible interleavings between synchronization points

Example of application: FWS

- Degraded mode (5 processes, 100 000 LOCS):
  - 1h40 on 64-bit 2.66 GHz Intel server
  - 98 alarms

- Full mode (15 processes, 1 600 000 LOCS):
  - 50 h
  - 12 000 alarms !!! more work is being done !!! (e.g. analysis of complex data structures, logs, etc)

Abstractions

- Based on Astrée for the sequential processes
- Takes scheduling into account
- OS entry points (semaphores, logbooks, sampling and queuing ports, buffers, blackboards, ...) are all stubbed (using Astrée stubbing directives)
- Interference between processes: flow-insensitive abstraction of the writes to shared memory and inter-process communications

Conclusion
Cost-effective verification

• The rumor has it that:
  • Manuel validation (testing) is costly, unsafe, not a verification!
  • Formal proofs by theorem provers are extremely laborious and not reusable hence costly
  • Model-checkers do not scale up

• Why not try abstract interpretation?
  • Domain-specific static analysis scales and can deliver no false alarm (but this requires developments of the analyzer by specialists)