Static Analysis and Verification of Aerospace Software by Abstract Interpretation

Julien Bertrane
École normale supérieure, Paris

Patrick Cousot
Courant Institute of Mathematical Sciences, NYU, New York & École normale supérieure, Paris

Radhia Cousot
École normale supérieure & CNRS, Paris

Jérôme Feret
École normale supérieure & INRIA, Paris

Laurent Mauborgne
École normale supérieure, Paris & IMDEA Software, Madrid

Antoine Miné
École normale supérieure & CNRS, Paris

Xavier Rival
École normale supérieure & INRIA, Paris

We discuss the principles of static analysis by abstract interpretation and report on the automatic verification of the absence of runtime errors in large embedded aerospace software by static analysis based on abstract interpretation. The first industrial applications concerned synchronous control/command software in open loop. Recent advances consider imperfectly synchronous, parallel programs, and target code validation as well. Future research directions on abstract interpretation are also discussed in the context of aerospace software.

Nomenclature

- $S$: program states
- $C$: collecting semantics
- $F$: $P$: prefix trace transformer
- $F$: $ι$: interval transformer
- $V$: set of all program variables
- $t$: reflexive transitive closure of relation
- $lfp$: least fixpoint of
- $|x|$: absolute value of
- $q$: quaternion
- $N$: naturals
- $I$: initial states
- $T$: trace semantics
- $R$: reachability semantics
- $1$: $S$: identity on $S$ (also $t_0$)
- $x$: program variable
- $t$: powers of relation
- $α$: abstraction function
- $X$: abstract counterpart of $X$
- $q$: conjugate of quaternion
- $Z$: integers
- $R$: reachability transformer
- $x$: program variable
- $t$: powers of relation
- $ρ$: reduction
- $γ$: concretization function
- $℘$: parts of set $S$ (also $2^S$)
- $||q||$: norm of quaternion
- $R$: reals
Content

• Motivation

• An informal introduction to abstract interpretation

• A short overview of a few applications and on-going work on aerospace software

• All necessary theoretical and practical details and references are given in the paper
Motivation
All computer scientists have experienced bugs

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

• Checking the presence of bugs is great
• Proving their absence is even better!
Abstract interpretation
Abstract interpretation

- Started in the 70’s and well-developed since then
- Originally for program dataflow analysis in compilers (e.g. check for conditions under which optimizations are applicable such as elimination of useless runtime checks)
- 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
1) Define the programming language semantics

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

Trajectory in state space

Space/time trajectory
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

Forbiden zone
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*
Soundness of the abstract verification

Never forget any possible case so the **abstract proof is correct in the concrete**
Unsound validation: testing

Try a few cases
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

By soundness an alarm must be raised for this overapproximation!
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.
ASTRÉE
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:

- Macintosh PPC:  
  - n = 2147483647, T[n] = 2147483647
- Macintosh Intel:  
  - n = 2147483647, T[n] = -1208492044
- PC Intel 32 bits:  
  - n = 2147483647, T[n] = -135294988
- PC Intel 64 bits:  
  - Bus error
Implicit specification

• Absence of runtime errors: overflows, division by zero, buffer overflow, 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)
II. Combination of abstract domains

Abstract interpretation-based tools usually use several different abstract domains since the design of a complex one is best decomposed into a combination of simpler abstract domains. Here are a few abstract domain examples used in the Astrée static analyzer:

- **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:** \( -a^{bt} \leq y(t) \leq a^{bt} \)

However, some classic abstract domains are not used in Astrée because they are either too imprecise, not scalable, or difficult to implement correctly. For instance, soundness may be an issue in the event of floating point rounding, or out of scope: determining program properties which are usually of no interest to prove the specification.

Because abstract domains do not use a uniform machine representation of the information they manipulate, combining them is not completely trivial. The conjunction of abstract program properties has to be performed, ideally, by a reduced product for Galois connection abstractions. In absence of a Galois connection or for performance reasons, the conjunction is performed using an easily computable but not optimal overapproximation of this combination of abstract domains.

Assume that we have designed several abstract domains and compute \( lfp \subseteq F_1 \in D_1 \lor \lor \lor f p \subseteq F_n \in D_n \) in these abstract domains \( D_1 \lor \lor \lor D_n \) relative to a collecting semantics \( C \subseteq t \subseteq I \). The combination of these analyses is sound as \( C \subseteq t \subseteq I \subseteq \gamma_1 \lor \lor \lor \gamma_n \subseteq lfp \subseteq F_1 \lor \lor \lor F_n \). However, only combining the analysis results is not very precise, as it does not permit analyses to improve each other during the computation. Consider, for instance, that interval and parity analyses find respectively that \( x \in [t_u, u t_t] \) and \( x \) is odd at some iteration. Combining the results would enable the interval analysis to continue with the interval \( x \in [u, 99] \) and, ergo, avoid a useless widening. This is not possible with analyses carried out independently.
Example of general purpose abstraction: octagons

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

- Example:

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

- At ★, the interval domain gives $L \leq \max(\max A, (\max Z)+(\max V))$.

- In fact, we have $L \leq A$.

- To discover this, we must know at ★ that $R = A-Z$ and $R > V$.

- 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.
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

```c
% 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_volatile_input((I [0,1]));
__ASTREE_max_clock((36000000));
% astree -exec-fn main -config-sem count.config count.c |grep '|R|'
|R| <= 0. + clock *1. <= 36000001.
```

← potential overflow!
Example of domain-specific abstraction: exponentials

```c
% 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) 
     * 4.491048e-03)); }
   B = A;
   if (SWITCH) {A = P;} 
   else {A = X;}
}
```

```c
void main()
{ FIRST = TRUE;
   while (TRUE) {
      dev( );
      FIRST = FALSE;
      __ASTREE_wait_for_clock();
   }
}
% cat retro.config
__ASTREE_volatile_input((E [-15.0, 15.0]));
__ASTREE_volatile_input((SWITCH [0,1]));
__ASTREE_max_clock((3600000));

|P| <= (15. + 5.87747175411e-39
/ 1.19209290217e-07) * (1 + 1.19209290217e-07)^clock - 5.87747175411e-39
/ 1.19209290217e-07 <= 23.0393526881
```

Collecting semantics:

- Polyhedra:
- Octagons:
- Ellipses:
- Intervals:
- Signs:
- Linear congruences:
- Simple congruences:
- Complex one is best decomposed into a combination of simpler abstract domains.

Abstract interpretation-based tools usually use several different abstract domains. For example, assume that we have designed several abstract domains and compute the least fixed point (lfp) between them. Because abstract domains do not use a uniform machine representation of the information they manipulate, combining the analysis results is not a simple task. This is not possible with analyses carried out independently. This is why, in these abstract domains overapproximation of this combination of abstract domains is sound as in these abstract domains combining them is not completely trivial. The conjunction of abstract program properties has to be translated to prove the specification:

- too costly
too imprecise
determining program properties which are usually of no interest
- not scalable

In these abstract domains examples used in the static analyzer:

- Astrée: the reduced product
- AIAA Infotech@Aerospace 2010, Atlanta, Georgia, 04/20/2010

© P Cousot et al.
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 target programs
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
Verification of imperfectly clocked synchronous systems
Imperfect synchrony

• Example of (buggy) communicating synchronous systems:

- negate previous input (on clocks C and C’)
- compare inputs

• Synchronized and dysynchronous executions:

...flawed alarms
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 $(\leq k, a \triangleright \blacklozenge b)$ and $(\geq k, a \triangleright \blacklozenge b)$ (signal changes less (more) than $k$ times in time interval $[a, b]$)
Example of static analysis

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

**Specification**: no alarm raised with a normal input

- Input stability $< \Delta$:
- Between $\frac{2}{3} \Delta$ and $\Delta$:
- Input stability $> \Delta$:

**counter-example**

| $\Delta$ | the analyzer proves the specification |

0

Julien Bertrane, ENS Paris

Static analysis of imperfectly-clocked synchronous systems using continuous-time abstract domains
THÉSÉE: Verification of embedded real-time parallel C programs
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

\[
\text{init: flag1 = flag2 = 0}
\]

<table>
<thead>
<tr>
<th>process 1</th>
<th>process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag1 = 1;</td>
<td>flag2 = 1;</td>
</tr>
<tr>
<td>if (!flag2)</td>
<td>if (!flag1)</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>/* critical section */</td>
<td>/* critical section */</td>
</tr>
</tbody>
</table>

write to flag1/2 and read of flag2/1 are independent so can be reordered \(\rightarrow\) error!

• We assume:
  • sequential consistency in absence of data race
  • for data races, values are limited by possible interleavings between synchronization points
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
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 to be done !!! (e.g. analysis of complex data structures, logs, etc)
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 hence costly
  - Model-checkers do not scale up

- Why not try abstract interpretation?
  - Domain-specific static analysis scales and can deliver no false alarm
The End