Abstract

Since almost any complex software has bugs, researchers have developed program correctness proof methods. This consists in defining a semantics formally describing the executions of a program and then in proving a theorem stating that these executions have a given property (for example that an expected result is provided in a finite time). Fundamental mathematical results show that these proofs cannot be done automatically by computers.

Confronted with this fundamental difficulty, abstract interpretation proceeds by correct approximation of the semantics. If the approximation is coarse enough, it is computable. If it is precise enough, it yields a correctness proof. The goal is therefore to find cheap approximations which are precise enough.

We will introduce a few elements of abstract interpretation and explain how to formalize the abstraction of semantic properties so as to obtain computable approximations leading to effective algorithms for the static analysis of the possible behaviours of programs.

Finally, we will describe an example of application of the theory to the proof of absence of runtime errors on synchronous control/command and underly the difficulties (such as floating point computations). This approach was applied with success to the verification of the electric flight control of the A380.
Software is hidden everywhere

Origin of accidents (metro)

- Paris métro line 12 accident: the driver was going too fast
- Roma metro line A accident: the driver was given OK to ignore red light in tunnel
- New high-speed métro line 14 (Météor): fully automated, no operators

Origin of accidents (aviation)

Worldwide analysis of the primary cause of major commercial-jet accidents between 1995 and 2004 as determined by the investigating authority ³ [1]

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight crew</td>
<td>90%</td>
</tr>
<tr>
<td>Weather</td>
<td>13%</td>
</tr>
<tr>
<td>Other</td>
<td>0%</td>
</tr>
<tr>
<td>Repair and maintenance</td>
<td>3%</td>
</tr>
<tr>
<td>Airport/Weather control</td>
<td>6%</td>
</tr>
</tbody>
</table>

Reference


³ Includes only accidents with known causes.

Software replaces human operators

- Computer control is recognized as the safest and less expansive way to eliminate human mistakes
- Software is massively present in all mission-critical and safety-critical industrial infrastructures
Why software is bugged?

As computer hardware capacity grows...

ENIAC
5,000 flops\(^4\)

NEC Earth Simulator
\(35 \times 10^{12}\) flops\(^5\)

Software size grows...

(1) Software gets huge

Text editor
1,700,000 lines of C\(^6\)

Operating system
35,000,000 lines of C\(^7\)

---

\(^4\) Floating point operations per second

\(^5\) \(10^{12}\) = Thousand Billion

\(^6\) 3 months for full-time reading of the code

\(^7\) 5 years for full-time reading of the code

---
... and so does the number of bugs

<table>
<thead>
<tr>
<th>Text editor</th>
<th>Operating system</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,700,000 lines of C&lt;sup&gt;6&lt;/sup&gt;</td>
<td>35,000,000 lines of C&lt;sup&gt;7&lt;/sup&gt;</td>
</tr>
<tr>
<td>1,700 bugs (estimation)</td>
<td>30,000 known bugs</td>
</tr>
</tbody>
</table>

<sup>6</sup> 3 months for full-time reading of the code
<sup>7</sup> 5 years for full-time reading of the code

October 23rd, 2006

---

Computers are finite

- Scientists use mathematics to deal with continuous, infinite structures (e.g. \( \mathbb{R} \))
- Computers can only handle discrete, finite structures

October 23rd, 2006

---

Putting big things into small containers

- Numbers are encoded onto a limited number of bits (binary digits)
- Some operations may overflow (e.g. integers: 32 bits \( \times \) 32 bits = 64 bits)
- Using different number sizes (32, 64, ... bits) can also be the source of overflows

October 23rd, 2006
The Ariane 5.01 maiden flight
- June 4th, 1996 was the maiden flight of Ariane 5

The Ariane 5.01 maiden flight failure
- June 4th, 1996 was the maiden flight of Ariane 5
- The launcher was destroyed after 40 seconds of flight because of a software overflow.

A 16 bit piece of code of Ariane 4 had been reused within the new 32 bit code for Ariane 5. This caused an uncaught overflow, making the launcher uncontrollable.

(3) Computers go round

Modular arithmetic...
- Todays, computers avoid integer overflows thanks to modular arithmetic
- Example: integer 2's complement encoding on 8 bits
Modular arithmetic is not very intuitive

\[
\# -1073741823 \div -1; \\
- : \text{int} = 1073741823 \\
\# -1073741824 \div -1; \\
- : \text{int} = -1073741824
\]

Mapping many to few

- Reals are mapped to floats (floating-point arithmetic)
  \[ \pm d_0.d_1d_2\ldots d_{p-1}\beta^e \]
- For example on 6 bits (with \( p = 3, \beta = 2, e_{\text{min}} = -1, e_{\text{max}} = 2 \)), there are 32 normalized floating-point numbers. The 16 positive numbers are

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]

\( d_0 \neq 0 \)
- \( p \) is the number of significative digits,
- \( \beta \) is the basis (2), and
- \( e \) is the exponent (\( e_{\text{min}} \leq e \leq e_{\text{max}} \))

Rounding

- Computations returning reals that are not floats, must be rounded
- Most mathematical identities on \( \mathbb{R} \) are no longer valid with floats
- Rounding errors may either compensate or accumulate in long computations
- Computations converging in the reals may diverge with floats (and ultimately overflow)
Example of rounding error

```c
/* float-error.c */
int main () {
  float x, y, z, r;
  x = 1.000000019e+38;
  y = x + 1.0e21;
  z = x - 1.0e21;
  r = y - z;
  printf("%f\n", r);
}
percent gcc float-error.c
% ./a.out
0.000000
```

Example of accumulation of small rounding errors

```ocaml
% ocaml
Objective Caml version 3.08.1
# let x = ref 0.0;;
val x : float ref = {contents = 0.}
# for i = 1 to 1000000000 do
  x := !x +. 1.0/.10.0
done; x;;
done; x;;
- : float ref = {contents = 99999998.7454178184}
since (0.1)_{10} = (0.0001100110011001100...)_2
```

Explanation of the huge rounding error

```c
/* double-error.c */
int main () {
  double x; float y, z, r;
  x = ldexp(1.,50)+ldexp(1.,26);
  y = x + 1;
  z = x - 1;
  r = y - z;
  printf("%f\n", r);
}
percent gcc double-error.c
% ./a.out
134217728.0
```

Example of accumulation of small rounding errors

```ocaml
# let x = ref 0.0;
val x : float ref = {contents = 0.}
# for i = 1 to 1000000000 do
  x := !x +. 1.0/.10.0
done; x;
done; x;
- : float ref = {contents = 99999998.7454178184}
since (0.1)_{10} = (0.0001100110011001100...)_2```
The Patriot missile failure

- “On February 25th, 1991, a Patriot missile ... failed to track and intercept an incoming Scud."  
- The software failure was due to a cumulated rounding error.

---

Warranty

Excerpt from an GPL open software licence:

NO WARRANTY. ... BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

You get nothing for free!

---

Warranty

Excerpt from Microsoft software licence:

DISCLAIMER OF WARRANTIES. ... MICROSOFT AND ITS SUPPLIERS PROVIDE THE SOFTWARE, AND SUPPORT SERVICES (IF ANY) AS IS AND WITH ALL FAULTS, AND MICROSOFT AND ITS SUPPLIERS HEREBY DISCLAIM ALL OTHER WARRANTIES AND CONDITIONS, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, ANY (IF ANY) IMPLIED WARRANTIES OF MERCHANTABILITY, OF FITNESS FOR A PARTICULAR PURPOSE, OF RELIABILITY OR AVAILABILITY, OF ACCURACY OR COMPLETENESS OF RESPONSES, OF WORKMANLIKE EFFORT, OF LACK OF VIRUSES, AND OF LACK OF NEGLIGENCE, ALL WITH REGARD TO THE SOFTWARE, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT OR OTHER SERVICES, INFORMATION, SOFTWARE, AND RELATED CONTENT THROUGH THE SOFTWARE OR OTHERWISE ARISING OUT OF THE USE OF THE SOFTWARE. ... 

You get nothing for your money either!

---

What can be done about bugs?
**Traditional software validation methods**

- The law cannot enforce more than “best practice”
- Manual software validation methods (code reviews, simulations, tests, etc.) do not scale up
- The capacity of programmers/computer scientists remains essentially the same
- The size of software teams cannot grow significantly without severe efficiency losses

**Interprétation abstraite**

There are two fundamental concepts in computer science (and in sciences in general):

- Abstraction: to reason on complex systems
- Approximation: to make effective undecidable computations

These concepts are formalized by abstract interpretation

**References**


**Mathematics and computers can help**

- Software behavior can be mathematically formalized
  → semantics
- Computers can perform semantics-based program analyses to realize verification → static analysis
  - but computers are finite so there are intrinsic limitations → undecidability, complexity
  - which can only be handled by semantics approximations → abstract interpretation

**Abstract interpretation (1) a very informal introduction**
Operational semantics

Test/debugging is unsafe

Safety property

Abstract interpretation is safe
Soundness requirement: erroneous abstraction

This situation is always excluded in static analysis by abstract interpretation.

Global interval abstraction $\rightarrow$ false alarms

Imprecision $\Rightarrow$ false alarms

Local interval abstraction $\rightarrow$ false alarms
Refinement by partitioning

Abstract interpretation
(2) a few elements

Intervals with partitioning

(2.1) Program semantics
Description of a computation step

- Transition system \((\Sigma, \tau)\), states \(\Sigma = \{\bullet, \ldots, \bullet\ldots\}\), 
  transitions \(\tau = \{\bullet\rightarrow, \ldots, \bullet\rightarrow\ldots\}\)

- Example
  - States: \((p, v)\), \(p\) is a program point, \(v\) assigns values to variables
  - Transitions \((p, v) \rightarrow (p', v')\) for assignment:
    \[
    \begin{align*}
    p': 
    & x = x + 1; \\
    v'(x) &= v(x) + 1 \text{ if } v(x) < \text{maxint} \\
    v'(\gamma) &= v(\gamma) \text{ if } \gamma \neq x
    \end{align*}
    \]
    Blocking state (\(\bullet\)) if \(v(x) \geq \text{maxint}\).

Least Fixpoint Trace Semantics

\[
\text{Traces} = \{\bullet | \bullet \text{ is a final state}\} \\
\cup \{(p, v) | p \text{ is a transition step} \& v \in \text{Traces}^*\} \\
\cup \{(\bullet\ldots\bullet) | \bullet\ldots\bullet \in \text{Traces}^{\infty}\}
\]

- In general, the equation has multiple solutions;
- Choose the least one for the computational ordering:
  “more finite traces & less infinite traces”.

Description of a complete computation by a trace

Initial states

Intermediate states

Final states of the finite traces

Final states of the infinite traces

States \(\Sigma = \{\bullet, \ldots, \bullet\ldots\}\), transitions \(\tau = \{\bullet\rightarrow, \ldots, \bullet\rightarrow\ldots\}\)

Iterative computation of the trace semantics

Iteeates Finite traces Infinite traces

\[
\begin{align*}
F^0 &= 0 \\
F^1 &= \{\bullet\} \\
F^2 &= \{0, 1, 0, 1\} \\
F^3 &= \{0, 1, 1, 2\} \\
\vdots \\
F^n &= \{0, \ldots, 1, n\} \\
F^\omega &= \{0, 1, \ldots, n, n+1, n+2, \ldots | n \geq 0\}
\end{align*}
\]
Trace Semantics, Formally

Trace semantics of a transition system \(\langle \Sigma, \tau \rangle\):

- \(\Sigma^+ \overset{\text{def}}{=} \bigcup \{[0, n] \mapsto \Sigma \mid n > 0\}\) finite traces
- \(\Sigma^\omega \overset{\text{def}}{=} [0, \omega] \mapsto \Sigma\) infinite traces
- \(S = \text{lfp} \supset F \in \Sigma^+ \cup \Sigma^\omega\) trace semantics
- \(F(X) = \{s \in \Sigma^+ \mid s \in \Sigma \wedge \forall s' \in \Sigma : (s, s') \notin \tau\} \cup \{ss'\sigma \mid (s, s') \in \tau \wedge s'\sigma \in X\}\) trace transformer
- \(X \subseteq Y \overset{\text{def}}{=} (X \cap \Sigma^+) \subseteq (Y \cap \Sigma^+) \wedge (X \cap \Sigma^\omega) \supseteq (Y \cap \Sigma^\omega)\) computational ordering

Program properties & Static analysis

- A program property \(\mathcal{P} \in \wp(D)\) is a set of semantics for that program (and so a subset of the semantic domain \(D\))
- The strongest program property\(^\text{13}\) is \(\{S[\mathcal{P}]\} \in \wp(D)\)
- A Static analysis consists ineffectively approximating the strongest program property:
  \[\text{Compute } \mathcal{P} \in \wp(D) : \{S[\mathcal{P}]\} \subseteq \mathcal{P}\]

Example of program property

- Correct implementations: print 0, print 1, [print 1|loop], ...
- Incorrect implementations: [print 0|print 1]
- Note for specialists: neither a safety nor a liveness property.
(2.3) Abstraction of program properties

Common requirements for abstraction

- [In this talk,] we consider overapproximations: \( \mathcal{P} \subseteq \alpha(\mathcal{P}) \)
  - If the abstract property \( \alpha(\mathcal{P}) \) is true then the concrete property \( \mathcal{P} \) is also true
  - If the abstract property \( \alpha(\mathcal{P}) \) is false then the concrete property \( \mathcal{P} \) may be true or false!
- All information is lost at once: \( \alpha(\alpha(\mathcal{P})) = \alpha(\mathcal{P}) \)
- The abstraction of more precise properties is more precise:
  \[ \text{si } \mathcal{P} \subseteq \mathcal{Q} \text{ alors } \alpha(\mathcal{P}) \subseteq \alpha(\mathcal{Q}) \]

Abstraction

- Replace a concrete property \( \mathcal{P} \in \wp(\mathcal{D}) \) by an abstract property \( \alpha(\mathcal{P}) \)
- Example:
  - \( \mathcal{D} = \wp(\Sigma^+ \cup \Sigma^\omega) \) semantic domain
  - \( \mathcal{P} \in \wp(\mathcal{D}) \) concrete property
  - \( \alpha(\mathcal{P}) \overset{\text{def}}{=} \wp(\bigcup \mathcal{P}) \) abstract property

Galois Connections

- One obtain a Galois connection:
  \[ \langle \wp(\mathcal{D}), \subseteq \rangle \overset{\alpha}{\leftarrow} \langle \wp(\mathcal{D}), \subseteq \rangle \]
  Concrete properties Abstract properties  

- With an isomorphic mathematical/computer representation
  \[ \langle \wp(\mathcal{D}), \subseteq \rangle \overset{\gamma}{\leftarrow} \langle \mathcal{D}^\|, \subseteq \rangle \]
  Concrete properties Abstract domain
  \[ \forall \mathcal{P} \in \wp(\mathcal{D}) : \forall \mathcal{Q} \in \mathcal{D}^\| : \alpha(\mathcal{P}) \subseteq \mathcal{Q} \iff \mathcal{P} \subseteq \gamma(\mathcal{Q}) \]
Example 1 of Galois connection

Example 2 of Galois connection

Example 3 of Galois connection

Traces: set of finite or infinite maximal sequences of states for the operational transition semantics

Set of reachable states: set of states appearing at least once along one of these traces (global invariant)

\[ \alpha_1(X) = \{ \sigma_i \mid \sigma \in X \land 0 \leq i < |\sigma| \} \]

Partitionned set of reachable states: project along each control point (local invariant)

\[ \alpha_2(\{(c_i, \rho_i) \mid i \in \Delta\}) = \lambda c \cdot \{ \rho_i \mid i \in \Delta \land c = c_i \} \]

Partitionned cartesian set of reachable states: project along each program variable (relationships between variables are now lost)

\[ \alpha_3(\{c \cdot \{ \rho_i \mid i \in \Delta_c \} \}) = \lambda c \cdot \lambda x \cdot \{ \rho_i(x) \mid i \in \Delta_c \} \]

Partitionned cartesian interval of reachable states: take min and max of the values of the variables

\[ \alpha_4(c \cdot \lambda x \cdot \{ v_i \mid i \in \Delta_{c,x} \}) = \lambda c \cdot \lambda x \cdot \langle \min \{v_i \mid i \in \Delta_{c,x}\}, \max \{v_i \mid i \in \Delta_{c,x}\} \rangle \]

\[ \alpha_1, \alpha_2, \alpha_3 \text{ and } \alpha_4 \text{, whence } \alpha_4 \circ \alpha_3 \circ \alpha_2 \circ \alpha_1 \text{ are lower-adjoints of Galois connections} \]

15 assuming these values to be totally ordered.
Example 4: Reduced Product of Abstract Domains

To combine abstractions $\langle \mathcal{D}, \subseteq \rangle \xrightarrow{\gamma_1} \langle \mathcal{D}_1, \subseteq \rangle$ and $\langle \mathcal{D}, \subseteq \rangle \xrightarrow{\gamma_2} \langle \mathcal{D}_2, \subseteq \rangle$

the reduced product is

$\alpha(X) \overset{\text{def}}{=} \{x, y \mid X \subseteq \gamma_1(x) \land X \subseteq \gamma_2(y)\}$

such that $\subseteq \overset{\text{def}}{=} \subseteq_1 \times \subseteq_2$ and

$\langle \mathcal{D}, \subseteq \rangle \xrightarrow{\gamma_1 \times \gamma_2} \langle \alpha(\mathcal{D}), \subseteq \rangle$

Example: $x \in [1, 9] \land x \text{ mod } 2 = 0$ reduces to $x \in [2, 8] \land x \text{ mod } 2 = 0$

Abstraction of fixpoints

- Let $\langle \wp(\mathcal{D}), \subseteq \rangle \xrightarrow{\gamma} \langle \mathcal{D}^\parallel, \subseteq \rangle$

- How can we abstract a fixpoint property $\llpq F$ where $F \in \wp(\mathcal{D}) \xrightarrow{m} \wp(\mathcal{D})$?

- Approximate correct abstraction:

  $\llpq F \subseteq \gamma(\llpq F \circ \gamma)$

- Complete abstraction: if $\alpha \circ F = F^\parallel \circ \alpha$ then

  $F^\parallel = \alpha \circ F \circ \gamma$, and

  $\alpha(\llpq F^\parallel) = \llpq F^\parallel$

Abstraction of functions

- Let $\langle \wp(\mathcal{D}), \subseteq \rangle \xrightarrow{\gamma} \langle \mathcal{D}^\parallel, \subseteq \rangle$

- How can we abstract an operator $F \in \wp(\mathcal{D}) \xrightarrow{m} \wp(\mathcal{D})$?

- The most precise overapproximation is

  $F^\parallel \in \mathcal{D}^\parallel \xrightarrow{m} \mathcal{D}^\parallel$

  $F^\parallel = \alpha \circ F \circ \gamma$

- This is a Galois connection

  $\langle \wp(\mathcal{D}) \xrightarrow{m} \wp(\mathcal{D}), \subseteq \rangle \xrightarrow{\lambda F \cdot \gamma \circ \gamma} \langle \mathcal{D}^\parallel \xrightarrow{m} \mathcal{D}^\parallel, \subseteq \rangle$

Example 5: reachable states

- Transition system: $\langle \Sigma, \tau \rangle$

- Initial states: $I \subseteq \Sigma$

- Abstraction:

  $I \subseteq \Sigma \xrightarrow{\alpha} \{s \mid \exists s \in \Sigma: \langle s, s \rangle \in \tau\}$

- Reachable states: $\llpq F^\parallel$

  $F^\parallel(X) = I \cup \{s' \mid \exists s \in X : \langle s, s' \rangle \in \tau\}$
Accelerating the convergence of iterative fixpoint computations

- The fixpoint $\mathbb{F} \subseteq \mathcal{D}$, $\mathbb{F} \in \mathcal{D} \mapsto \mathcal{D}$ is computed iteratively\(^{16}\):

\[
X^0 = \bot, \quad X^{n+1} = \mathbb{F}(X^n), \quad X^\omega = \bigsqcup_{n \geq 0} X^n
\]

- For systems of equations $\mathcal{D} = \prod_{i=1}^n \mathcal{D}_n$, one can use asynchronous iterations
- Convergence acceleration techniques have been developed to overapproximated the limit.

\(^{16}\) $(\mathcal{P}, \sqsubseteq)$ is a partially ordered set, $\mathbb{F}$ is monotone, $\sqsubseteq$ is the infimum, the least upper bound $\sqcup$ must exist for all iterates (in general transfinite).

Abstract-interpretation-based static analysis

1. Define the semantics of the language $S \in \mathcal{L} \mapsto \mathcal{D}$ and the concrete properties $\rho(\mathcal{D})$;
2. Let $Q \in \rho(\mathcal{D})$ be a property to be proved for program $P$: $S[P] \in Q$
3. Choose an abstraction $(\rho(\mathcal{D}), \subseteq) \models_{\gamma} (\mathcal{D}, \subseteq)$
4. Use abstract interpretation theory to formally design an abstract semantics $S\llbracket P \rrbracket \triangleq \alpha(S[P])$
5. The static analysis algorithm is the computation of this abstract semantics (whence is correct by construction)

6. The result of the computation is either
   - $S\llbracket P \rrbracket \in \gamma(S\llbracket P \rrbracket) \subseteq Q$ (correctness proof), or
   - $\gamma(S\llbracket P \rrbracket) \not\subseteq Q$ (the property is not satisfied or the approximation is too coarse)
7. The abstraction must be chosen depending on the property $Q$ to be proved
   - coarse enough to be automatically computable,
   - precise enough to obtain a formal correctness proof: $\gamma(S\llbracket P \rrbracket) \subseteq Q$;
Abstract interpretation
(3) a simple example of application

October 23rd, 2006 — 71 —

Syntax of programs

- **variables** $X \in X$
- **types** $T \in T$
- **arithmetic expressions** $E \in E$
- **boolean expressions** $B \in B$

$D ::= T \times \times;$

$C ::= X = E;$

$P ::= D C$

Traces to postcondition abstraction

- **Traces**: set of finite or infinite maximal sequences of states for the operational transition semantics

- **Strongest liberal postcondition**: final states $s$ reachable from a given precondition $P$

  $\alpha(X) = \lambda R. \{ s \mid \exists \sigma_0 \sigma_1 \ldots \sigma_n \in X : \sigma_0 \in R \land s = \sigma_n \}$

  We have ($\Sigma'$: set of states, $\subseteq$ pointwise):

  $\langle p(\Sigma^\infty), \subseteq \rangle \xleftarrow{\gamma} \langle p(\Sigma), \subseteq \rangle \xrightarrow{\cup} p(\Sigma), \subseteq \rangle$
States

Values of given type:

\( \nu[T] \) : values of type \( T \in T \)

\( \nu[\text{int}] \equiv \{z \in \mathbb{Z} | \min_{\text{int}} \leq z \leq \max_{\text{int}} \} \)

Program states \( \Sigma[P] \) \(^\dagger\):\(^\dagger\)

\[ \begin{align*}
\Sigma[D \ C] & \equiv \Sigma[D] \\
\Sigma[T \ X] & \equiv \{X\} \mapsto \nu[T] \\
\Sigma[T \ X; D] & \equiv \{X\} \mapsto \nu[T] \cup \Sigma[D]
\end{align*} \]

\(^\dagger\) States \( \rho \in \Sigma[P] \) of a program \( P \) map program variables \( X \) to their values \( \rho(X) \)

Concrete Reachability Semantics of Programs

\[ \begin{align*}
S[X = E][R] & \equiv \{\rho[X \leftarrow E][R] | \rho \in R \cap \text{dom}(E)\} \\
\rho[X \leftarrow v](X) & \equiv v, \quad \rho[X \leftarrow v](Y) = \rho(Y) \\
S[\text{if } B \ C'][R] & \equiv S[C'][B][R] \cup B[-B][R] \\
B[B][R] & \equiv \{\rho \in R \cap \text{dom}(B) | B \text{ holds in } \rho\} \\
S[\text{if } B \ C' \text{ else } C''][R] & \equiv S[C''][B][B][R] \cup S[C''][-B][R] \\
S[\text{while } B \ C'][R] & \equiv \text{let } \nu = \text{wfl}(\lambda X. R \cup S[C'][B][X] \text{ in } (B[-B]\nu) \\
S[\{\}][R] & \equiv R \\
S[C_1 \ldots C_n][R] & \equiv S[C_n] \circ \ldots \circ S[C_1] \quad n > 0 \\
S[D \ C][R] & \equiv S[C][R]
\end{align*} \]

Not computable (undecidability).

Concrete Semantic Domain of Programs

Concrete semantic domain for reachability properties:

\[ \mathcal{D}[P] \equiv \rho(\Sigma[P]) \quad \text{sets of states} \]

i.e. program properties where \( \sqsubseteq \) is implication, \( \emptyset \) is false, \( \cup \) is disjunction.

Abstract Semantic Domain of Programs

\( \langle \mathcal{D}^\#, P \rangle, \sqsubseteq, \perp, \sqcup \rangle \)

such that:

\[ \langle \mathcal{D}[P], \sqsubseteq \rangle \xrightarrow{\gamma} \langle \mathcal{D}^\#, P \rangle, \sqsubseteq \]

i.e.

\[ \forall X \in \mathcal{D}[P], Y \in \mathcal{D}[P] : \alpha(X) \sqsubseteq Y \iff X \subseteq \gamma(Y) \]

hence \( \langle \mathcal{D}^\#, P \rangle, \sqsubseteq, \perp, \sqcup \rangle \) is a complete lattice such that \( \perp = \alpha(\emptyset) \) and \( \sqcup X = \alpha(\gamma(X)) \)
Abstract Reachability Semantics of Programs

\[
S^d[X = E; R] \triangleq \alpha(\{p | X \leftarrow E[p] \mid p \in \gamma(R) \cap \text{dom}(E)\})
\]

\[
S^d[\text{if } B C' R] \triangleq S^d[C'](B^d[R] \cup B^d[\neg B] R)
\]

\[
B^d[R] \triangleq \alpha(\{p \in \gamma(R) \cap \text{dom}(B) \mid B \text{ holds in } p\})
\]

\[
S^d[\text{if } B C' \text{ else } C'' R] \triangleq S^d[C'](B^d[R] \cup S^d[C'](B^d[\neg B] R))
\]

\[
S^d[\text{while } B C' R] \triangleq \text{let } W = \text{fix } \lambda X. R \cup S^d[C'](B^d[B] X)
\]

\[
\text{in } (B^d[\neg B] W)
\]

\[
S^d[\{\}] R \triangleq R
\]

\[
S^d[\{C_1 \ldots C_n\}] R \triangleq S^d[C_n] \circ \ldots \circ S^d[C_1] \quad n > 0
\]

\[
S^d[D C] R \triangleq S^d[C](R)
\]

Abstract Semantics with Convergence Acceleration

\[
S^d[X = E; R] \triangleq \alpha(\{p | X \leftarrow E[p] \mid p \in \gamma(R) \cap \text{dom}(E)\})
\]

\[
S^d[\text{if } B C' R] \triangleq S^d[C'](B^d[R] \cup B^d[\neg B] R)
\]

\[
B^d[R] \triangleq \alpha(\{p \in \gamma(R) \cap \text{dom}(B) \mid B \text{ holds in } p\})
\]

\[
S^d[\text{if } B C' \text{ else } C'' R] \triangleq S^d[C'](B^d[R] \cup S^d[C'](B^d[\neg B] R))
\]

\[
S^d[\text{while } B C' R] \triangleq \text{let } W = \text{fix } \lambda X. R \cup S^d[C'](B^d[B] X)
\]

\[
\text{in } (B^d[\neg B] W)
\]

\[
S^d[\{\}] R \triangleq R
\]

\[
S^d[\{C_1 \ldots C_n\}] R \triangleq S^d[C_n] \circ \ldots \circ S^d[C_1] \quad n > 0
\]

\[
S^d[D C] R \triangleq S^d[C](R)
\]

Applications of abstract interpretation

Any reasoning on complex computer systems must consider a correct approximation of its behaviors formalized by Abstract interpretation [5, 20, 21, 34]

- Syntax of programming languages [30]
- Semantics of programming languages [13, 27]
- Proofs of program correctness [11, 12]
- Typing and type inference [18]
Abstract interpretation (5) application to the A380 flight control software

ASTRÉE is a specialized static analyzer

- Embedded real-time synchronous control/command C programs:

Declare and initialize state variables;
loop forever
read volatile input variables,
compute output and state variables,
write state variables;
wait for next clock tick
end loop
Objective of ASTRÉE

- Prove automatically the absence of runtime errors:
  - No division by 0, NaN, out of range array access
  - No signed integer/float overflows
  - Verification of user-defined properties (for example machine dependent properties)

- Requirements:
  - efficiency (must operate on a workstation)
  - precision (few false alarms)

- No alarm → full certification

General purpose numerical abstract domains

Intervals: \[ \bigwedge_{i=1}^{n} a_i \leq x_i \leq b_i \]

Octogons: \[ \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \pm x_i \pm y_j \leq a_{ij} \]

Polyhedra: \[ \bigwedge_{j=1}^{n} \left( \sum_{i=1}^{m} a_{ji} x_i \right) \leq b_j \]

(5.2) Examples of abstractions

Ellipsoid Abstract Domain for Filters

2nd Order Digital Filter:

- 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.
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;  
Filter Example [36]  
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; }  
}

Slow divergences by rounding accumulation  
X = 1.0;  
while (TRUE) {  
    X = X / 3.0;  
    X = X * 3.0;  
}

Solution [35]: bound the cumulated rounding error as a function of the number of iterations by arithmetico-geometric progressions:  
- Relation $|x| \leq a \cdot b^n + c$, where $a, b, c$ are constants determined by the analysis, $n$ is the iterate number  
- Number of iterates bounded by $N$: $|x| \leq a \cdot b^N + c$

Application to the A 340/A 380  
- Primary flight control software of the electric flight control system of the Airbus A340 family and the A380  
- C program, automatically generated from of high-level specification (à la Simulink/SCADAB)  
- A340: 100.000 to 250.000 LOCs  
- A380: 400.000 to 1.000.000 LOCs
A world première
Analysis of 400,000 lines of C code

<table>
<thead>
<tr>
<th>time</th>
<th>memory</th>
<th>false alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>13h 52mn</td>
<td>2.2 Gb</td>
<td>0</td>
</tr>
</tbody>
</table>

19 on an AMD Opteron 248, 64 bits, a single processor

The Current Situation

(1) Model design
(2) Simulation
(3) Implementation
(4) Program analysis

Perspectives

The Project

(1) Model design
(2) Model analysis
(3) Program analysis
(4) Program analysis

20 greatly simplified, system dependability is simply ignored!

21 greatly simplified, system dependability is simply ignored!
References on the web: www.di.ens.fr/~cousot

Bibliographic References


