Semantics and Validation
Lecture 1. Informal Introduction

Laboratoire Modélisation et Analyse de Systèmes en Interaction,
CEA-LIST
and Ecole Polytechnique

Eric Goubault and Sylvie Putot
November 26, 2013
Overall organisation and objectives

How to give a formal meaning to programming languages, and hence to verification?

- Formalism (the case of Abstract Interpretation)
- Application: exercises, and use of “PAG/WWW” (Program Analyzer Generator, from ABSINT, web version) to conceive simple analyzers, to understand difficulties and what can be expected

How to use existing off-the-shelf analyzers?

- Numerical software analysis; Hoare logic and program proof; Model-checking
- FLUCTUAT, Frama-C/Why, SPIN

(in this course we will use only freely available tools.)

Page of the course: www.enseignement.polytechnique.fr/profs/informatique/Sylvie.Putot/Enseignement/SemantiqueValidation.html
Course/TP outline and evaluation

1. Informal introduction
2. Semantic equations, fixpoint computations
3. Abstract interpretation, interval analysis
4. Relational abstract domains
5. Numerical program analysis
6. Introduction to program proof using Hoare logics
7. Introduction to model-checking

Evaluation:
- 1/3 on the project (interval or zones static analyzer with PAG)
  - send me code and report on January 26 - before the last course
- 2/3 on 2h written exam on January 28 (last course)
Program verification: what can we do?

- Code reading, quality management, code generation by certified generators, UML etc.
  - development with strict norms and procedures
  - model driven development
  - code reading, but the size of programs is growing far more than programmer or reader’s abilities...

- Test: manual, and/or automatic test case generation with coverage objectives. We can expect to
  - find and correct bugs
  - have reasonable confidence in the absence of bugs that will often appear...

Useful, but not enough, especially for critical systems (ex. avionics, automobile industry, banks, medical etc.)
Formal methods (static analysis)

- "Formal specifications (B, Z etc.): not in this course"
- Proof: inputs and properties are logical formula, and the program is interpreted as deduction rules between these formula. Not automatic because of undecidability problems (interaction with the user is necessary) - Hoare Logics in this course
- Model-Checking: the main idea is to verify refined “temporal” properties (written in temporal logics) on bounded memory programs by enumerating all accessible states. No property synthesis. We will see SPIN, for properties of parallel programs.
- Abstract-interpretation based static analysis: (almost) fully automatic method to prove partial correctness. The main idea is that we can still prove some properties while not computing exactly the program states: we will abstract them in a conservative way. Potential problem of “false alarms”.
The goals of static analysis: prove properties on the program analyzed

- fully automatically
- without executing the program, for (possibly infinite) sets of inputs and parameters

Some properties

- invariant properties (true on all trajectories - for all possible inputs or parameters).
  Example: bounds on values of variables, absence of run-time errors
- liveness properties (that become true at some moment on one trajectory).
  Examples: state reachability, termination
Abstract interpretation

Two main influences

- Formal meaning to simple data-flow analyses in compiler optimizations (constant propagation, use-def analysis, parallelisation, type checking - Kildall’s lattice 73, Karr’s lattice 76):

Replace:

```c
int f(int j)
{
    int i, j;
    i = 42;
    while (j < 100)
    {
        j = 2*i + 1 + 2*j;
    }
    return j;
}
```

by...

```c
int f(int j)
{
    int i;
    i = 42;
    while (j < 100)
    {
        j = 85 + 2*j;
    }
    return j;
}
```

and even...

```c
int f(int j)
{
    while (j < 100)
    {
        j = 85 + 2*j;
    }
    return j;
}
```

The code transformation is justified by semantic properties of programs (invariant and liveness properties), which are synthesized statically, without executing the program.
Compilation and verification

```c
int f(int i) {
    int j, k;
    k = i + j;
    j = 2;
    return k;
}
```

```bash
[]> gcc -c f.c -O -Wall
f.c: In function ‘f’:
f.c:3: warning: ‘j’ might be used uninitialized in this function
```

```c
int g() {
    int j[10];
    return j[12];
}
```

```bash
[]> gcc -c g.c -O -Wall
[7]>
```

The verifications done by gcc are very basic... ➔ complementary tools for validation!
Abstract interpretation

Two main influences

- Formal meaning to compiler optimizations
- Program proof using Hoare logics

Seminal papers

- Patrick Cousot, Radhia Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”, POPL 1977
- Patrick Cousot, Nicolas Halbwachs, “Automatic Discovery of Linear Restraints Among Variables of a Program”, POPL 1978
- Patrick Cousot, Radhia Cousot, “Systematic Design of Program Analysis Frameworks”, POPL 1979
Proofs and abstract interpretation

Example: can we prove that there is no runtime error in this program?

```c
fft(a, n)
{
    cplx b[n/2], c[n/2];
    if (n > 2)
    {
        for (i = 0; i < n; i = i + 2)
        {
            b[i/2] = a[i];
            c[i/2] = a[i + 1];
        }
        fft(b, n/2);
        fft(c, n/2);
        for (i = 0; i < n; i = i + 1)
        {
            a[i] = F1(n) * b[...] + F2(n) * c[...];
        }
    }
    else
    {
        a[0] = g*a[0] + d*a[1];
        a[1] = a[0] - 2*d*a[1];
    }
}
```
Example: can we prove that there is no runtime error in this program?

```c
fft (a, n)
{  complex b[n/2], c[n/2];
   if (n > 2)
      {  for (i=0; i<n; i=i+2)
         {  b[i/2]=a[i];
             c[i/2]=a[i+1];  }
          fft (b, n/2);
          fft (c, n/2);  
          for (i=0; i<n; i=i+1)
            a[i]=F1(n)*b[...] + F2(n)*c[...]; }
   else
      a[0]=g*a[0]+d*a[1];
      a[1]=a[0]-2*d*a[1];  }
}
```

No!

- Erroneous executions when starting with $n$ not a power of 2
- Example: $n=3$:
  - $i=2$, interpret $c[i/2]=a[i+1]$
  - but $i+1$ is 3, out of arrays bounds! ($a[0..2]$)
Proofs and abstract interpretation

Example: can we prove that there is no runtime error in this program?

```c
fft (a, n)
{   cplx b[n/2], c[n/2];
    if (n > 2)
    {       for (i=0; i<n; i=i+2)
           {         b[i/2]=a[i];
                     c[i/2]=a[i+1]; }
               fft (b, n/2);
               fft (c, n/2);
               for (i=0; i<n; i=i+1)
                    a[i]=F1(n)*b[...]+F2(n)*c[...];
    }
    else
         a[0]=g*a[0]+d*a[1];
         a[1]=a[0]-2*d*a[1];  }
}
```

No!
- Erroneous executions when starting with \( n \) not a power of 2

But yes when \( n \) is a power of 2
Proofs and abstract interpretation

Example: looking at proof statements

```plaintext
fft (a, n)
// a.length=n ∧ ∃k > 0 n=2^k
{ complex b[n/2], c[n/2];
  // a.length=n ∧ ∃k > 0 n=2^k ∧ b.length=n/2 ∧ c.length=n/2
  if (n > 2)
    { for (i=0; i<n; i=i+2)
      {
        // a.length=n ∧ ∃k > 0 n=2^k ∧ b.length=c.length=n/2 ∧ i≥0 ∧ i<n ∧ ∃j ≥ 0 i=2j
        b[i/2]=a[i];
        // i+1<n
        c[i/2]=a[i+1];
      }
    }
  fft (b, n/2);
  fft (c, n/2);
  for (i=0; i<n; i=i+1)
    a[i]=F1(n)*b[...]+F2(n)*c[...];
  }
else
  // a.length=2
  a[0]=g*a[0]+d*a[1];
  a[1]=a[0]−2*d*a[1];
```

Example: can we automatize a proof of partial correctness?

- When trying to prove “simple” statements about programs, can we automatize the synthesis of (strong enough) loop invariants?
- Here we want to be able to bound the indexes for array dereferences to prove absence of array out of bounds accesses
- We see we only need linear inequalities between variables (i and n in particular) and parity of i and of n at all recursive calls
Example: can we automatize a proof of partial correctness?

- When trying to prove “simple” statements about programs, can we automatize the synthesis of (strong enough) loop invariants?

Main idea: carefully choose some particular predicates

- Abstract all first-order predicates into these chosen predicates:
- So that proofs on these abstract predicates become tractable
- And abstract predicates are strong enough to prove some properties of interest (e.g. absence of runtime errors)
Example: can we automatize a proof of partial correctness?
- When trying to prove “simple” statements about programs, can we automatize the synthesis of (strong enough) loop invariants?

Main idea: carefully choose some particular predicates
- Abstract all first-order predicates into these chosen predicates:
  Here: necessary predicates are of the form \((v_i - v_j \leq c_{ij})\) (octagons) and \((v_i \text{ is even})\) and \((v_i \text{ is a power of 2})\)
- So that proofs on these abstract predicates become tractable
- And abstract predicates are strong enough to prove some properties of interest (e.g. absence of runtime errors)
Example: can we automatize a proof of partial correctness?

- When trying to prove “simple” statements about programs, can we automatize the synthesis of (strong enough) loop invariants?

Main idea: carefully choose some particular predicates

- Abstract all first-order predicates into these chosen predicates:
  Here: necessary predicates are of the form \((v_i - v_j \leq c_{ij})\) (octagons) and \((v_i \text{ is even})\) and \((v_i \text{ is a power of 2})\)

- So that proofs on these abstract predicates become tractable

- And abstract predicates are strong enough to prove some properties of interest (e.g. absence of runtime errors)
  Here:
  - at \(c[i/2]=a[i+1]\): \((i \text{ is even})\) and \((i-n \leq -1)\) and \((n \text{ is a power of 2})\) imply \((i+1 \text{ is odd})\) and \((i+1 \leq n)\) and \((n \text{ is even})\) imply \((i+1 < n)\)
  - at else: \(n=2\) since \((n \text{ is a power of 2})\) and \((n \leq 2)\) (else condition) and \((a.length=n)\) (part of the loop invariant) imply \(a.length=n=2\)
Numerical abstract domains

These inferences can be made into an algorithm (interpreting program statements by transfer functions in so called abstract domains)

We will focus in the sequel on numerical abstract domains for finding invariants on values of variables

Some numerical domains

- Intervals (Cousot, Bourdoncle)
- Linear equalities (Karr)
- Polyhedra i.e. linear inequalities (Cousot, Halbwachs)
- Congruences and linear congruences (Granger)
- Octagons (Mine)
- Linear templates (Sankaranarayanan, Sipma, Manna)
- Zonotopes (Goubault, Putot)
- Non-linear templates (Adjé, Gaubert, Goubault, Seidl, Gawlitza)
Numerical abstract domains: example

Concrete set of values $(x_i, y_i)$: program executions
Numerical abstract domains: example

Concrete set of values \((x_i, y_i)\): program executions

Forbidden zone \(E_1\): is the program safe with respect to \(E_1\)?
Numerical abstract domains: example

- Concrete set of values \((x_i, y_i)\): program executions
- Forbidden zone \(E_1\): is the program safe with respect to \(E_1\)?
- Polyhedra abstraction:
Numerical abstract domains: example

- Concrete set of values \((x_i, y_i)\): program executions
- Forbidden zone \(E_1\): is the program safe with respect to \(E_1\)?
- Polyhedra abstraction: proves the program is safe wrt \(E_1\): \(P \cap E_1 = \emptyset\)
Numerical abstract domains: example

- Concrete set of values \((x_i, y_i)\): program executions
- Forbidden zone \(E_1\): is the program safe with respect to \(E_1\)?
- Polyhedra abstraction: proves the program is safe wrt \(E_1\): \(P \cap E_1 = \emptyset\)
- Interval abstraction:
Numerical abstract domains: example

Concrete set of values \((x_i, y_i)\): program executions

Forbidden zone \(E_1\): is the program safe with respect to \(E_1\)?

Polyhedra abstraction: proves the program is safe wrt \(E_1\): \(P \cap E_1 = \emptyset\)

Interval abstraction: cannot prove the program is safe wrt \(E_1\) (false alarm): \(I \cap E_1 \neq \emptyset\)
Numerical abstract domains: example

- Concrete set of values \((x_i, y_i)\): program executions
- **Forbidden zone** \(E_1\): is the program safe with respect to \(E_1\)?
- **Polyhedra abstraction**: proves the program is safe wrt \(E_1\): \(P \cap E_1 = \emptyset\)
- **Interval abstraction**: cannot prove the program is safe wrt \(E_1\) (false alarm): \(I \cap E_1 \neq \emptyset\)
• Concrete set of values \((x_i, y_i)\): program executions
• **Forbidden zone** \(E_1\): is the program safe with respect to \(E_1\)?
• **Polyhedra abstraction**: proves the program is safe wrt \(E_1\): \(P \cap E_1 = \emptyset\)
• **Interval abstraction**: cannot prove the program is safe wrt \(E_1\) (false alarm): \(I \cap E_1 \neq \emptyset\)
• Both abstractions prove the program is unsafe wrt to \(E_2\): \(P \subseteq I \subseteq E_2\)
Expected properties on the FFT example

- **No Run Time Error** (RTE: division by zero, out of bounds array dereference, null dereference, overflows etc.)
  - Out of bounds array dereference only for arrays whose size is not a power of 2! [should crash, or give an incoherent result]
  - Under conditions on the “input signal”, no overflow (even better, under what conditions?)

- **Bounds on the drift due to finite precision** (difference between a real number calculation and the floating-point computation)

- **Functional proof**
  - in general not achievable
  - here Parseval?

\[
\sum_i |a'[i]|^2 = \sum_i |a[i]|^2
\]

\[\rightarrow\] characterisation of reachable values for all variables (invariants) at each control point.
Expected properties on the FFT example

- **No Run Time Error** (RTE: division by zero, out of bounds array dereference, null dereference, overflows etc.)
  - Out of bounds array dereference only for arrays whose size is not a power of 2! [should crash, or give an incoherent result]
  - Under conditions on the “input signal”, no overflow (even better, under what conditions?)

- **Bounds on the drift due to finite precision** (difference between a real number calculation and the floating-point computation)

- **Functional proof**
  - in general not achievable
  - here Parseval?

\[ \sum_i |a'[i]|^2 = \sum_i |a[i]|^2 \]

- equality only approximately true because of floating-point numbers (≠ real numbers).

\[ \rightarrow \] characterisation of reachable values for all variables (invariants) at each control point.
int f(int i)
{
    int i, j;
    i = 42;
    j = 0;
    while (j < 100)
        j = 2 * i + 1 + 2 * j;
    return j;
}

int f(int i)
{
    int i, j;
    i = 42;
    j = 0;
    while (j < 100)
        j = 85 + 2 * j;
    return j;
}
Intuitively: constant propagation synthesis phase

Associate to each variable:

- \( C(n) \) if we know for sure that the variable is constant, and its value is \( n \)

We add for some technical reasons to be explained later on:

- \( BOT \) (also noted \( \bot \)) for “I have not examined this control point (yet)”
- \( TOP \) (also noted \( \top \)) for “I do not know”

We “propagate” along the control flow (as a first approximation) these “abstract values”.
int f(int i) {
    int i, j;
    i = 42;
    j = 0;
    while (j < 100) {
        j = 2*i + 1 + 2*j;
    }
    return j;
}
int f(int i) {
    int i,j;
i = 42;
j = 0; [1]
    while (j<100) [2]
        j = 2*i+1+2*j; [3]
    return j;
}

$k$ (being $i$ or $j$) is ⊥ in [2]
⇔ \[ k \text{ is } ⊥ \text{ in } [1] \text{ and } k \text{ is } ⊥ \text{ in } [3] \]
⇔ \[ k \text{ is } C(n) \text{ or } ⊥ \text{ in } [1] \text{ and } k \text{ is } ⊥ \text{ or } C(n) \text{ in } [3] \text{ (but not both } ⊥) \]
else $k$ is ⊤
We will make an initial guess of the “abstract values” for $i$ and $j$, with least information (i.e. BOT),

- and iterate the loop
- until we find a “fixed” set of abstract values

We will implement this “method” during the first exercise session. We will formalize this in next courses.
int f(int i)
{
    int i,j;
    i = 42;
    j = 0;  [1]
    while (j<100)  [2]
        j = 2*i+1+2*j;  [3]
    return j;
}
A glimpse at the formalization in that case, in PAG/WWW

The TYPE section defines the “abstract” elements used for the analysis:

- here the \( C(-1), C(0), C(1), \ldots \) plus \( BOT \) and \( TOP \)
- \( BOT \) is less than every element (smaller set of potential values: empty!), \( TOP \) is bigger than any element (bigger set of potential values: everything!)
- all \( C(i), C(j) \) for \( i \neq j \) are incomparable (distinct set of potential values)

We associate in State an abstract value to each variable:

\[
\begin{align*}
\text{TYPE} \\
\text{ConstFlattened} &= \text{flat(snum)} \\
\text{State} &= \text{Var} \rightarrow \text{ConstFlattened}
\end{align*}
\]
Declare in the “problem” section,

- that the analysis is a forward analysis (you will understand later...could only be backward otherwise),
- that the abstract elements it will manipulate are elements of State,
- that we start “Kleene iteration” with the bottom element, and that, at each iteration, we combine with the previous iterate using the lub (“least upper bound” or sup) operator
- that we initialize all non user-initialized variables with the top value of our abstract lattice

PROBLEM Constant_Propagation

direction : forward
carrier : State
init : bot
init_start : [-> top]
combine : lub
In the TRANSFER section, define how each statement acts on State; here, for assignments:

TRANSFER
// in assignments calculate the new value of the variable
// and add it to the state
    ASSIGN(variable, expression) =
        @[variable -> evalAExp(expression, @)]
// in procedure calls pass the value of the actual
// argument to the formal parameter
CALL(_, param, exp), call_edge =
  @\[param -> evalAEExp(exp, @)]

CALL(_, _, _), local_edge = bot

// at the end of procedures reset the formal parameter
END(_, param) =
  @\[param -> top]
// in case of tests, propagation of reduced states
// in true/false branch
IF(expression), true_edge = branch(expression,@,true)

IF(expression), false_edge = branch(expression,@,false)

WHILE(expression), true_edge = branch(expression,@,true)

WHILE(expression), false_edge = branch(expression,@,false)
Some “support” functions

Evaluation of expressions:

evalAExp :: Expression * State -> ConstFlattened
evalAExp(expression, state) =
case expType(expression) of
  "ARITH_BINARY" => case expOp(expression) of
    "+"=>let valLeft<=evalAExp(expSubLeft(expression),state),
          valRight<=evalAExp(expSubRight(expression),state) in
          lift(valLeft + valRight);
    (...)  
  endcase;

  (be careful of the specific semantics of <=!)


"ARITH_UNARY" =>
case expOp(expression) of
    "-"=>let value<=evalAExp(expSub(expression),state) in
        lift(-(value));
endcase;
"VAR"=>state(expVar(expression));
"CONST"=>lift(expVal(expression));
_=>error("Runtime Error: evalAExp applied to nonarithmetic Expression");
endcase
Function branch used in the TRANSFER section has to be defined (here tests are basically uninterpreted, the function returns the state without modification):

\[
\text{branch} :: \text{Expression} \times \text{State} \times \text{bool} \rightarrow \text{State} \\
\text{branch}(\text{expression}, \text{state}, \text{is\_true}) = \text{state}
\]
### Programming/debugging under PAG/WWW
- Please always test every new function or change of your program under PAG/WWW; otherwise, it is very difficult to debug!
- Implement all transfer functions asked! (not only the arithmetics)
- Look at the memento, and at the indications on the PAG/WWW web page [http://www.program-analysis.com](http://www.program-analysis.com)

### Full PAG version available on the Polytechnique server
- More possibilities: analyzes C code, more data types, more flexibility for fixpoint iteration, aliases analyses available
- But....:
  - slightly different language for abstract domain definition
  - command-line instructions to compile analysis, launch analysis, visualize results, etc
- Start with PAG/WWW and ask me if you want to try full PAG
This is within the V cycle

(see real-world applications in aeronautics/automotive industry in later courses)
Ex.: aeronautics

(extract of ex-COMASIC course by J. Souyris, Airbus)

- Specification and code generation: SCADE (Lustre - M. Pouzet’s course)
- Validation by tests (DO178B standard) and static analysis (CAVEAT, ASTREE, FLUCTUAT etc)
- Evolution in recent standards (DO-178C standard) towards replacing (part of) tests by static analysis
- Heavy certification process (DPAC)
Ex.: the automotive industry

(source: Numatec Automotive)

- up to 25% of the overall cost lies in the embedded systems
- 30 ECUs, dedicated network etc. (synchronous)
  - security functions: ABS, airbags (opening policy of doors)
  - comfort functions, with interactions with the security functions (speed regulator)
  - complex mapping of functions onto the ECUs
- complex interactions between suppliers, OEMs etc.

Typical tools
- Requirements analysis
- Specifications, simulation and sometimes even some code generation (generally by Matlab/Simulink)
- A lot of third-party code
- Validation by tests, and also by POLYSPACE and some others
In real world: a catastrophic example

The facts
- 25/02/91: a Patriot anti-missile misses a Scud in Dharan and crashes on an American building: 28 deads.

Analysis
- The missile program had been running for 100 hours, incrementing an integer every 0.1 second.
- But 0.1 not representable in a finite number of digits in base 2.
  \[
  \frac{1}{10} = 0.00011001100110011001100\cdots
  \]
  
  
<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Truncation error</td>
<td>(0.000000095)</td>
</tr>
<tr>
<td>Drift, on 100 hours</td>
<td>(0.34s)</td>
</tr>
<tr>
<td>Location error on the scud</td>
<td>(500m)</td>
</tr>
</tbody>
</table>
Another famous failure: The Ariane 5 Maiden flight, June 1996

The facts

- Satellites launcher Ariane 5
- Successor of Ariane 4, but much more powerful
- Desintegration after 40 seconds in Maiden flight in June 96

Failure analysis

- Re-use in Ariane 5 of some code conceived and tested for Ariane 4
- But Ariane 4 is much more powerful than Ariane 4 and has a different flight profile
- Capacity overflow in a conversion from a 64 bit double precision float to a 16 bits integer: exception not caught
- Material redundancy of no use when the conception is erroneous
What we know more or less how to do

(in “formal methods”)

- Generate **synchronous** control-command code (ex. SCADE/LUSTRE)
- Find a precise over-approximation of RTEs “Run-Time Errors” (division by zero, nil dereference, out of bounds array dereference, overflows etc.) for programs of several hundred thousand lines of code (ex. ASTREE, Polyspace)
- Find a good over-approximation of WCET (“Worst-Case Execution Time”, ABSINT)
- Find a good over-approximation of uncertainties: computation, model, finite precision (FLUCTUAT)
- Generate “interesting” test cases, for a DO178B type of test covering (ESTEREL/GaTeL etc.)
Some abstract interpreters used in an industrial context

- Polyspace (now at Mathworks)
  http://www.mathworks.fr/products/polyspace/

- WCET, Stack Analyzer, Astrée (ABSINT) http://www.absint.com

- Clousot: code contracts checking with abstract interpretation, PREfix and PREfast: drivers checking (Microsoft)
  http://research.microsoft.com/apps/pubs/?id=138696

- C global surveyor (NASA) http://ti.arc.nasa.gov/tech/rse/vandv/cgs/ (used on flight software of Mars Path-Finder, Deep Space One, and Mars Exploration Rover)

- Frama-C’s value analysis http://frama-c.com/value.html

- Fluctuat (proprietary tool but academic version upon request)
Even a NASA conference
Some static analysis tools

Some you will see during the courses/TPs

- PAG: platform for defining simple static analyses
- FLUCTUAT: static analyzer by abstract interpretation of numerical computations (C/Ada code)
- Frama-C, Why3: platform for program analysis; Hoare proof (C)
- SPIN: model-checker, temporal logics (PROMELA models)

You can also have a look at the static analysis tools list on Wikipedia

- Oh yes, there are plenty of them ... : an abstract domain generally aims at one kind of property
- But beware, not all tools claiming to do static analysis are sound (there are some bug finders among them)