The Software Reliability Problem

- The evolution of hardware by a factor of $10^6$ over the past 25 years has lead to the explosion of the program sizes;
- The scope of application of very large software is likely to widen rapidly in the next decade;
- These big programs will have to be modified and maintained during their lifetime (often over 20 years);
- The size and efficiency of the programming and maintenance teams in charge of their design and follow-up cannot grow up in similar proportions;

The Software Reliability Problem (Cont’d)

- At a not so uncommon (and often optimistic) rate of one bug per thousand lines such huge programs might rapidly become hardly manageable in particular for safety critical systems;
- Therefore in the next 10 years, the software reliability problem is likely to become a major concern and challenge to modern highly computer-dependent societies.
What Can We Do About It?

- Use our intelligence (thinking/intellectual tools: abstract interpretation);
- Use our computer (mechanical tools: static program analysis/checking/testing, the early idea of using computers to reason about computer).

The Verification/Validation Problem

![Diagram of the verification/validation problem]

- **Computer program**
- **Formal specification**

**Saturation**

- Program semantics = model of actual program executions in all environments
- Specification semantics = model of required program executions in allowed environments

Example: Model Checking

![Diagram of model checking]

- **Computer program**
- **Formal specification**

**Model checking**

- Program model
- Temporal specification

**Saturation**

- Program semantics = model of actual program executions in all environments
- Specification semantics = model of required program executions in allowed environments
Other Examples of Software Verification/Validation Techniques

- Software testing;
- Simulation and prototyping;
- Technical reviews;
- Requirements tracing;
- Formal correctness proofs;
- Etc.

Fundamental Theoretical Limitations

- Undecidability: full automation of software verification/validation is impossible;
- Examples of undecidable questions:
  - Is my program bug-free? (i.e. correct with respect to a given specification);
  - Can a program variable take two different values during execution?

Practical Limitations

- Testing:
  - Testing all data on all paths is impossible;
- Formal methods:
  - No formal specification perfectly reflects informal human expectations;
  - Proofs grow exponentially in the size of programs/specifications which is incompatible with friendly user interaction and full automation;
- etc.

Undecidability and Approximation

- Since program verification is undecidable, computer aided program verification methods are all partial/incomplete;
- They all involve some form of approximation:
  - restricted specifications or programs (e.g. finiteness),
  - decidable questions or semi-algorithms,
  - practical time/memory complexity limitations,
  - require user interaction;
- Most of these approximations are formalized by Abstract Interpretation.
Examples of approximations

- **Testing**: coverage is partial (so errors are frequently found until the end of the software lifetime);
- **Proofs**: specifications are often partial, debugging proofs is often harder than testing programs (so only parts of very large software can be formally proved correct);
- **Model checking**: the model must fit machine limitations (so some facets of program execution must be left out) and be redesigned after program modifications;
- **Typing**: types are weak program properties (so type verification cannot be generalized to complex specifications).
**Example: trace semantics**

- **Initial states**
- **Intermediate states**
- **Final states of the finite traces**
- **Infinite traces**

**Examples of computation traces**

- **Finite** (C+1=)

- **Erroneous** (C+1+1+1...)

- **Infinite** (C+0+0+0...)

**Least Fixpoints: intuition**

\[
\text{Behaviors} = \{ \bullet | \bullet \text{ is a final state} \} \\
\cup \{ \quad \quad \quad \quad \quad | \quad \bullet \text{ is an elementary step } \& \quad \quad \in \text{Behaviors}^+ \} \\
\cup \{ \quad \quad \quad \quad \quad | \quad \bullet \text{ is an elementary step } \& \quad \quad \in \text{Behaviors}^\infty \}
\]

- In general, the equation has multiple solutions.
- Choose the least one for the partial ordering:

\[\text{more finite traces } \& \text{ less infinite traces}\]

**Abstract Interpretation**
Abstract Interpretation [1]

- Formalizes the idea of approximation of sets and set operations as considered in set (or category) theory;
- A theory of approximation of the semantics of programming languages;
- Main application: formal method for inferring general runtime properties of programs.

Reference

The Theory of Abstract Interpretation

- Abstract interpretation is a theory of conservative approximation of the semantics of computer systems.
  
Approximation: observation of the behavior of a computer system at some level of abstraction, ignoring irrelevant details;
Conservative: the approximation cannot lead to any erroneous conclusion.

Usefulness of Abstract Interpretation

- Thinking tools: the idea of abstraction is central to reasoning (in particular on computer systems);
- Mechanical tools: the idea of effective approximation leads to automatic semantics-based program manipulation tools.

Abstraction
**Abstraction: intuition**

- **Abstract interpretation** formalizes the intuitive idea that a semantics is more or less precise according to the considered observation level of the program executions;

- **Abstract interpretation theory** formalizes this notion of approximation/abstraction in a mathematical setting which is independent of particular applications.

**Intuition behind abstraction**

**Approximations of an [in]finite set of points:**

From Below

\[
\{(19, 78), \ldots, (20, 01)\}
\]

\[
\{(\ldots, (19, 78), \ldots, (20, 01))\}
\]
Approximations of an [in]finite set of points: From Above

\{ \ldots, (19, 78), \ldots, \\
(20, 01), (?, ?), \ldots \} 

Effective computable approximations of an [in]finite set of points; Signs [2]

\{ x \geq 0 \\
y \geq 0 \}

Reference

Intuition Behind Effective Computable Abstraction

Effective computable approximations of an [in]finite set of points; Intervals [3]

\{ x \in [19, 78] \\
y \in [20, 01] \}

Reference
Effective computable approximations of an \([\text{in}]\)finite set of points; Octagons [4]

\[
\begin{align*}
1 \leq x & \leq 9 \\
x + y & \leq 78 \\
1 \leq y & \leq 9 \\
x - y & \leq 99
\end{align*}
\]

Reference


Effective computable approximations of an \([\text{in}]\)finite set of points; Polyhedra [5]

\[
\begin{align*}
19x + 78y & \leq 2000 \\
20x + 01y & \geq 0
\end{align*}
\]

Reference


Effective computable approximations of an \([\text{in}]\)finite set of points; Simple congruences [6]

\[
\begin{align*}
x & = 19 \text{ mod } 78 \\
y & = 20 \text{ mod } 99
\end{align*}
\]

Reference


Effective computable approximations of an \([\text{in}]\)finite set of points; Linear congruences [7]

\[
\begin{align*}
1x + 9y & = 7 \text{ mod } 8 \\
2x - 1y & = 9 \text{ mod } 9
\end{align*}
\]

Reference

Effective computable approximations of an [in]finite set of points; Trapezoidal linear congruences [8]

\[
\begin{aligned}
1x + 9y &\in [0, 78] \pmod{10} \\
2x - 1y &\in [0, 99] \pmod{11}
\end{aligned}
\]

Reference

Conservative Approximation and Information Loss

Conservative Approximation
- Is the operation \(1/(x+1-y)\) well defined at run-time?
- Concrete semantics: yes
Conservative Approximation

- Is the operation $1/(x+1-y)$ well defined at run-time?
- Testing: You *never* know!

Conservative Approximation

- Is the operation $1/(x+1-y)$ well defined at run-time?
- Abstract semantics 1: I don’t know

Intuition Behind Information Loss
Information Loss

- All answers given by the abstract semantics are always correct with respect to the concrete semantics;
- Because of the information loss, not all questions can be definitely answered with the abstract semantics;
- The more concrete semantics can answer more questions;
- The more abstract semantics are more simple.

Basic Elements of Abstract Interpretation Theory
The Abstraction $\alpha$ is Monotone

\[ X \subseteq Y \Rightarrow \alpha(X) \subseteq \alpha(Y) \]

The Concretization $\gamma$ is Monotone

\[ X \subseteq Y \Rightarrow \gamma(X) \subseteq \gamma(Y) \]

The $\gamma \circ \alpha$ Composition

\[ \alpha \circ \gamma(Y) = Y \]

The $\alpha \circ \gamma$ Composition

\[ X \subseteq \gamma \circ \alpha(X) \]
Galois Connection

\[ \langle P, \subseteq \rangle \overset{\gamma}{\underset{\alpha}{\rightleftharpoons}} \langle Q, \sqsubseteq \rangle \]

iff

- \( \alpha \) is monotone
- \( \gamma \) is monotone
- \( X \subseteq \gamma \circ \alpha(X) \)
- \( \alpha \circ \gamma(Y) \subseteq Y \)

\[ ^1 \text{formalizations using closure operators, ideals, etc. are equivalent.} \]

Fixpoint Abstraction

\[ F^\sharp = \alpha \circ F \circ \gamma \]

\[ \text{lfp } F \sqsubseteq \gamma(\text{lfp } F^\sharp) \]

Function Abstraction

\[ \langle P, \subseteq \rangle \overset{\gamma}{\underset{\alpha}{\rightleftharpoons}} \langle Q, \sqsubseteq \rangle \]

\[ \langle P, \mon \rangle \overset{\gamma}{\underset{\alpha}{\rightleftharpoons}} \langle Q, \mon \rangle \]

\[ \lambda F^\sharp \cdot \gamma \circ F^\sharp \circ \alpha \overset{\lambda F \cdot \alpha \circ \gamma}{\rightleftharpoons} \lambda F \cdot \alpha \circ F \circ \gamma \]

\[ \text{lfp } F \sqsubseteq \gamma(\text{lfp } F^\sharp) \]
Exact/Approximate Fixpoint Abstraction

Exact Abstraction:
\[ \alpha(lfp F) = lfp F^{\#} \]

Approximate Abstraction:
\[ \alpha(lfp F) \sqsubseteq^\# lfp F^{\#} \]

A Few References on Foundations


(1) Exact Abstractions

Abstractions of Semantics [12]

Reference

Example 2 of Semantics Abstraction

(Small-Step) Operational Semantics

Example 3 of Semantics Abstraction

Partial Correctness / Invariance Semantics

(2) Effective Approximate Abstractions

Reference

Effective Abstractions of Semantics

- If the approximation is rough enough, the abstraction of a semantics can lead to a version which is less precise but is effectively computable by a computer;
- The computation of this abstract semantics amounts to the effective iterative resolution of fixpoint equations;
- By effective computation of the abstract semantics, the computer is able to analyze the behavior of programs and of software before and without executing them.

Objective of Static Program Analysis

- Program analysis is the automatic static determination of dynamic run-time properties of programs;
- The principle is to compute an approximate semantics of the program to check a given specification;
- Abstract interpretation is used to derive, from a standard semantics, the approximate and computable abstract semantics;
- This derivation is itself not (fully) mechanizable.

Static Program Analysis

- Basic idea: use effective computable approximations of the program semantics;
  - Advantage: fully automatic, no need for error-prone user designed model or costly user interaction;
  - Drawback: can only handle properties captured by the approximation;
  - Remedy: ask the user to choose among a variety of possible approximations (abstract algebras) at various cost/precision ratio.
Principle of a Static Program Analyzer

- Program
- Specification
- Generator
- System of fixpoint equations/constraints
- Solver
  (Approximate) solution
- Diagnoser
- Diagnosis

Design of a Static Program Analyzer by Abstract Interpretation

- Computer program
- Abstract program semantics
- Information about actual program executions in all environments
- Programming language semantics
- Program semantics = model of actual program executions in all environments

Generic Static Program Analyzer

- Abstract algebra
- Generator
- System of fixpoint equations/constraints
- Solver
  (Approximate) solution
- Interface
- Information on program executions

Effective Symbolic Abstractions
Effective Abstractions of Symbolic Structures

- Most structures manipulated by programs are *symbolic structures* such as control structures (call graphs), data structures (search trees), communication structures (distributed & mobile programs), etc;
- It is very difficult to find compact and expressive abstractions of such sets of objects (languages, automata, trees, graphs, etc.).

Example of Abstractions of Infinite Sets of Infinite Trees

**Binary Decision Graphs:** [15]

Example of Abstractions of Infinite Sets of Infinite Trees (Cont’d)

**Tree Schemata:** [16, 17]

Reference


Example: interval analysis (1975)

Equations (abstract interpretation of the semantics):

\[
\begin{align*}
X_1 &= [1, 1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1, 1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]

Constraints (abstract interpretation of the semantics):

\[
\begin{align*}
X_1 &\supseteq [1, 1] \\
X_2 &\supseteq (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &\supseteq X_2 \oplus [1, 1] \\
X_4 &\supseteq (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]
Example: interval analysis (1975)

Increasing chaotic iteration, initialization:

\[
\begin{cases}
  X_1 = [1, 1] \\
  X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\
  X_3 = X_2 \oplus [1, 1] \\
  X_4 = (X_1 \cup X_3) \cap [10000, +\infty]
\end{cases}
\]

\[
\begin{align*}
X_1 & = 1; \\
1: & \text{ while } x < 10000 \text{ do } \\
2: & \quad x := x + 1 \\
3: & \quad \text{ od; } \\
4: & \end{align*}
\]

\[\begin{cases}
  X_1 = [1, 1] \\
  X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\
  X_3 = X_2 \oplus [1, 1] \\
  X_4 = (X_1 \cup X_3) \cap [10000, +\infty]
\end{cases}\]
Example: interval analysis (1975)

Increasing chaotic iteration:

\[
\begin{align*}
\text{x} &:= 1; \\
1: &\quad \text{while } \text{x} < 10000 \text{ do} \\
2: &\quad \text{x} := \text{x} + 1 \\
3: &\quad \text{od}; \\
4: &
\end{align*}
\]

\[
\begin{align*}
X_1 &= [1, 1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1, 1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]

Example: interval analysis (1975)

Increasing chaotic iteration: convergence?

\[
\begin{align*}
\text{x} &:= 1; \\
1: &\quad \text{while } \text{x} < 10000 \text{ do} \\
2: &\quad \text{x} := \text{x} + 1 \\
3: &\quad \text{od}; \\
4: &
\end{align*}
\]

\[
\begin{align*}
X_1 &= [1, 1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1, 1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]

Example: interval analysis (1975)

Increasing chaotic iteration: convergence??

\[
\begin{align*}
\text{x} &:= 1; \\
1: &\quad \text{while } \text{x} < 10000 \text{ do} \\
2: &\quad \text{x} := \text{x} + 1 \\
3: &\quad \text{od}; \\
4: &
\end{align*}
\]

\[
\begin{align*}
X_1 &= [1, 1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1, 1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]
Example: interval analysis (1975)

Increasing chaotic iteration: convergence?????

\[
\begin{align*}
X_1 &= [1, 1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1, 1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]

\[
x := 1; \\
\text{while } x < 10000 \text{ do} \\
x := x + 1 \\
\text{od;}
\]

\[
\begin{align*}
X_1 &= [1, 1] \\
X_2 &= [1, 4] \\
X_3 &= [2, 4] \\
X_4 &= \emptyset
\end{align*}
\]

\[\text{Example: interval analysis (1975)} \ 2\]

Increasing chaotic iteration: convergence?????

\[
\begin{align*}
X_1 &= [1, 1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1, 1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]

\[
x := 1; \\
\text{while } x < 10000 \text{ do} \\
x := x + 1 \\
\text{od;}
\]

\[
\begin{align*}
X_1 &= [1, 1] \\
X_2 &= [1, 5] \\
X_3 &= [2, 5] \\
X_4 &= \emptyset
\end{align*}
\]
Example: interval analysis (1975)

Convergence speed-up by extrapolation:

\[ x := 1; \]
\[ \begin{align*}
1: & \quad \text{while } x < 10000 \text{ do} \\
2: & \quad x := x + 1 \\
3: & \quad \text{od;}
\end{align*} \]
\[ \begin{align*}
X_1 &= [1, 1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1, 1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*} \]

\[ \begin{align*}
X_1 &= [1, 1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1, 1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*} \]

Example: interval analysis (1975)

Decreasing chaotic iteration:

\[ x := 1; \]
\[ \begin{align*}
1: & \quad \text{while } x < 10000 \text{ do} \\
2: & \quad x := x + 1 \\
3: & \quad \text{od;}
\end{align*} \]
\[ \begin{align*}
X_1 &= [1, 1] \\
X_2 &= [1, +\infty] \quad \Leftarrow \text{widening} \\
X_3 &= [2, 6] \\
X_4 &= \emptyset
\end{align*} \]

Example: interval analysis (1975)

Decreasing chaotic iteration:

\[ x := 1; \]
\[ \begin{align*}
1: & \quad \text{while } x < 10000 \text{ do} \\
2: & \quad x := x + 1 \\
3: & \quad \text{od;}
\end{align*} \]
\[ \begin{align*}
X_1 &= [1, 1] \\
X_2 &= [1, +\infty] \\
X_3 &= [2, +\infty] \\
X_4 &= \emptyset
\end{align*} \]
Example: interval analysis (1975) ²

Decreasing chaotic iteration:

\[
\begin{align*}
X_1 &= [1,1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1,1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]

\[
\begin{align*}
x &:= 1; \\
1: & \quad \text{while } x < 10000 \text{ do} \\
2: & \quad x := x + 1 \\
3: & \quad \text{od;} \\
4: & \quad \text{end}
\end{align*}
\]

² P. Cousot & R. Cousot, ISOP’1976, POPL’77.

Example: interval analysis (1975) ²

Result of the interval analysis:

\[
\begin{align*}
X_1 &= [1,1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1,1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]

\[
\begin{align*}
x &:= 1; \\
1: & \quad \{ x = 1 \} \\
2: & \quad \{ x \in [1,9999] \} \\
3: & \quad \{ x \in [2,10000] \} \\
4: & \quad \{ x = 10000 \}
\end{align*}
\]

² P. Cousot & R. Cousot, ISOP’1976, POPL’77.

Example: interval analysis (1975) ²

Final solution:

\[
\begin{align*}
X_1 &= [1,1] \\
X_2 &= (X_1 \cup X_3) \cap [-\infty, 9999] \\
X_3 &= X_2 \oplus [1,1] \\
X_4 &= (X_1 \cup X_3) \cap [10000, +\infty]
\end{align*}
\]

\[
\begin{align*}
x &:= 1; \\
1: & \quad \text{while } x < 10000 \text{ do} \\
2: & \quad x := x + 1 \\
3: & \quad \text{od;} \\
4: & \quad \text{end}
\end{align*}
\]

² P. Cousot & R. Cousot, ISOP’1976, POPL’77.

A More Intriguing Example

program Variant_of_McCarthy_91_function;
var X, Y : integer;
function F(X : integer) : integer;
begin
if X > 100 then F := X - 10
else F := F(F(F(F(F(F(F(X + 90))))))));
end;
begin
readln(X);
Y := F(X);
\( \{ Y \in [91, +\infty] \} \)
end.

Reference

Probabilistic Program Analysis

```c
double x, i;
assume (-1.0 < x < 0.0);
i = 0.0;
while (i < 3.0) {
    x += uniform();
i += 1.0;
}
assert (x < 1.0);
```

With 99% safety:
- the probability of the outcome \( x < 1 \) is less than 0.859,
- assuming:
  - worst-case nondeterministic choices of the precondition \((-1.0 < x < 0.0)\),
  - random choices \( \text{uniform()} \) chosen in \([0,1]\) with the Lebesgue uniform distribution.

---

Communication Topology of Mobile Processes

```
A
  
  A
  
  S
  
  Request
  
  B
  
  Q1
  
  Data exchange
  
  B
  
  Q2
  
  Data exchange
  
  A
  
  Q2

```

---

Objective of Static Program Checking

```
Program

Specification

Program checker

Diagnosis
```

---

2. D. Monniaux, SAS'00, POPL'01

3. J. Feret, SAS'00, ENTCS Vol. 39
**Principle of a Static Program Checker**

![Diagram of a Static Program Checker]

- **Program**
- **Specification**
- **Generator**
- **Solver**
- **Diagnosis**

**System of fixpoint equations/constraints**

**Approximate) solution**

**Design of a Static Program Checker by Abstract Interpretation**

**Abstract Static Program Checking**

**Computer program**

**Programming language semantics**

**Program semantics**

**Satisfaction**

**Specification semantics**

**Specification language semantics**

**Formal specification**

**Abstract program semantics**

**Abstract semantics specification**

**ABSTRACTION**

- Program semantics = model of actual program executions in all environments
- Specification semantics = model of required program executions in allowed environments

**Example: interval analysis (1975)**

Exploitation of the result of the interval analysis:

```
x := 1;
1: {x = 1}
while x < 10000 do
2: {x ∈ [1, 9999]}
   x := x + 1
3: {x ∈ [2, 10000]}
od;
4: {x = 10000}
```

\[ X_1 = [1, 1] \]

\[ X_2 = (X_1 \cup X_3) \cap [−\infty, 9999] \]

\[ X_3 = X_2 \oplus [1, 1] \]

\[ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \]

2 P. Cousot & R. Cousot, ISOP ’76, POPL ’77.

**Other Examples of Faultless Execution Checks**

- Absence of runtime errors (array bounds violations, arithmetic overflow, erroneous data accesses, etc.).
- Absence of memory leaks (dangling pointers, uninitialized variables, etc.).
- Handling of all possible runtime exceptions (failures of I/O and system calls, etc.).
- No resource contention and race conditions in concurrent programs (deadlocks & livelocks).
- Termination / non termination conditions,
- Etc.

---

© P. Cousot
Abstract checking versus Abstract Testing

- **Abstract checking**: specification derived automatically from the program (e.g. using the language specification for run-time errors);
- **Abstract testing**: specification provided by the programmer.

Abstract Program Testing

<table>
<thead>
<tr>
<th>Debugging</th>
<th>Abstract testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run the program</td>
<td>Compute the abstract semantics</td>
</tr>
<tr>
<td>On test data</td>
<td>Choosing a predefined abstraction</td>
</tr>
<tr>
<td>Checking if all right</td>
<td>Checking user-provided abstract assertions</td>
</tr>
<tr>
<td>Providing more tests</td>
<td>With more refined abstractions</td>
</tr>
<tr>
<td>Until coverage</td>
<td>Until enough assertions proved or no predefined abstraction can do.</td>
</tr>
</tbody>
</table>

Combining Empirical and Formal Methods

- The user provides local formal abstractions of the program specifications using predefined abstractions;
- The program is evaluated by abstract interpretation of the formal semantics of the program;
- If the local abstract specification cannot be proved correct, a more precise abstract domain must be considered;
- The process is repeated until appropriate coverage of the specification.
Example of predefined abstraction

Example of predefined abstraction: intervals

A Tiny Example

0: \{ n:[-\infty, +\infty] ?; f:[-\infty, +\infty] ? \}  
  \text{read}(n);
1: \{ n:[0, +\infty]; f:[-\infty, +\infty] ? \}  
  f := 1;
2: \{ n:[0, +\infty]; f:[1, +\infty] \}  
  \text{while} (n \neq 0) \text{do}
  3: \{ n:[1, +\infty]; f:[1, +\infty] \}  
     f := (f \times n);
  4: \{ n:[1, +\infty]; f:[1, +\infty] \}  
     n := (n - 1)
  5: \{ n:[0, +\infty]; f:[1, +\infty] \}  
  od;
6: \{ n:[0,0]; f:[1, +\infty] \}  

A Tiny Example (Cont’d)

0: \{ n:\bot; f:\bot \}  
  \text{initial} (n < 0);
1: \{ n:[-\infty,-1]; f:[-\infty, +\infty] ? \}  
  f := 1;
2: \{ n:[-\infty,-1]; f:[-\infty, +\infty] \}  
  \text{while} (n \neq 0) \text{do}
  3: \{ n:[-\infty,-1]; f:[-\infty, +\infty] \}  
     f := (f \times n);
  4: \{ n:[-\infty,-1]; f:[-\infty, +\infty] \}  
     n := (n - 1)
  5: \{ n:[-\infty,-2]; f:[-\infty, +\infty] \}  
  od;
6: \{ n:\bot; f:\bot \}  

A More Intriguing Example

program Variant_of_McCarthy_91_function;
var X, Y : integer;
function F(X : integer) : integer;
begin
  if X > 100 then F := X − 1091
  else F := F(F(F(F(F(F(F(F(X + 90))))))));
end;
begin
  readln(X);
  if (% X > 100 %)
    Y := F(X);
  {% sometime true %}
end.

Example of cycle: F(100) → F(190) → F(180) → F(170) → F(160) → F(150) → F(140) → F(130) → F(120) → F(110) → F(100) → ...

Examples of Functional Specifications for Abstract Testing

• Worst-case execution/response time in real-time systems running on a computer with pipelines and caches;
• Periodicity of some action over time/with respect to some clock;
• Possible reactions to real-time event/message sequences;
• Compatibility with state/transition/sequence diagrams/charts;
• Absence of deadlock/livelock with different scheduling policies;

Comparing with program debugging

• Similarity: user interaction, on the source code;
• Essential differences:
  • user provided test data are replaced by abstract specifications;
  • evaluation of an abstract semantics instead of program execution/simulation;
  • one can prove the absence of (some categories of) bugs, not only their presence;
  • abstract evaluation can be forward and/or backward (reverse execution).

Conclusion
Concluding Remarks

- Program debugging is still the prominent industrial program “verification” method. Complementary program verification methods are needed;
- Fully mechanized program verification by formal methods is either impossible (e.g. typing/program analysis) or extremely costly since it ultimately requires user interaction (e.g. abstract model checking/deductive methods for large programs);
- For program verification, semantic abstraction is mandatory but difficult whence hardly automatizable, even with the help of programmers;

Concluding Suggestions

- Abstract interpretation introduces the idea of safe approximation within formal methods;
- So you might think to use it for partial verification of the source specification/program code:
  - Abstract checking (fully automatic and exhaustive diagnosis on run-time safety properties),
  - Abstract testing (interactive/planned diagnosis on functional, behavioural and resources-usage requirements),
  using tools providing predefined abstractions.

Industrialization of Static Analysis/Checking by Abstract Interpretation

- Does apply to any computer-related language with a well-specified semantics describing computations (e.g. specification languages, data base languages, sequential, concurrent, distributed, mobile, logical, functional, object oriented, ... programming languages, etc.);
- Does apply to any property and combinations of properties (such as safety, liveness, timing, event preconditions, ...);
- Can follow up program modifications over time;
- Very cost effective, especially in early phases of program development.

Internal use for compiler design.
**DAEDALUS** European project on the verification of critical real-time avionic software (oct. 2000 — sep. 2002):

- P. Cousot (ENS, France), scientific coordinator;
- R. Cousot (École polytechnique, France);
- A. Deutsch & D. Pilaud (Polyspace Technologies, France);
- C. Ferdinand (AbsInt, Germany);
- É. Goubault (CEA, France);
- N. Jones (DIKU, Denmark);
- F. Randimbivololona & J. Souyris (EADS Airbus, France), coord.;
- M. Sagiv (Univ. Tel Aviv, Israel);
- H. Seidel (Univ. Trier, Germany);
- R. Wilhelm (Univ. Sarrebrücken, Germany);

A reference (with a large bibliography)

P. Cousot.

Abstract interpretation based formal methods and future challenges.

In R. Wilhelm (editor), « Informatics — 10 Years Back, 10 Years Ahead ».

An extended electroning version is also available on Springer-Verlag website together with a very long electroning version with a complete bibliography.