# Abstract Interpretation Based Program Testing

**Patrick COUSOT**
École Normale Supérieure
45 rue d'Ulm
75230 Paris cedex 05, France
mailto:Patrick.Cousot@ens.fr
http://www.di.ens.fr/~cousot

**Radhia COUSOT**
École Polytechnique
91128 Palaiseau cedex, France
mailto:Radhia.Cousot@polytechnique.fr
http://lix.polytechnique.fr/~radhia

SSGRR'2000, L'Aquila, Italy July 31st – August 6th, 2000

# Introductive Motivations

# Bugs

- **Software bugs**
  - whether anticipated (Y2K bug)
  - or unforeseen (failure of the 5.01 flight of Ariane V launcher)

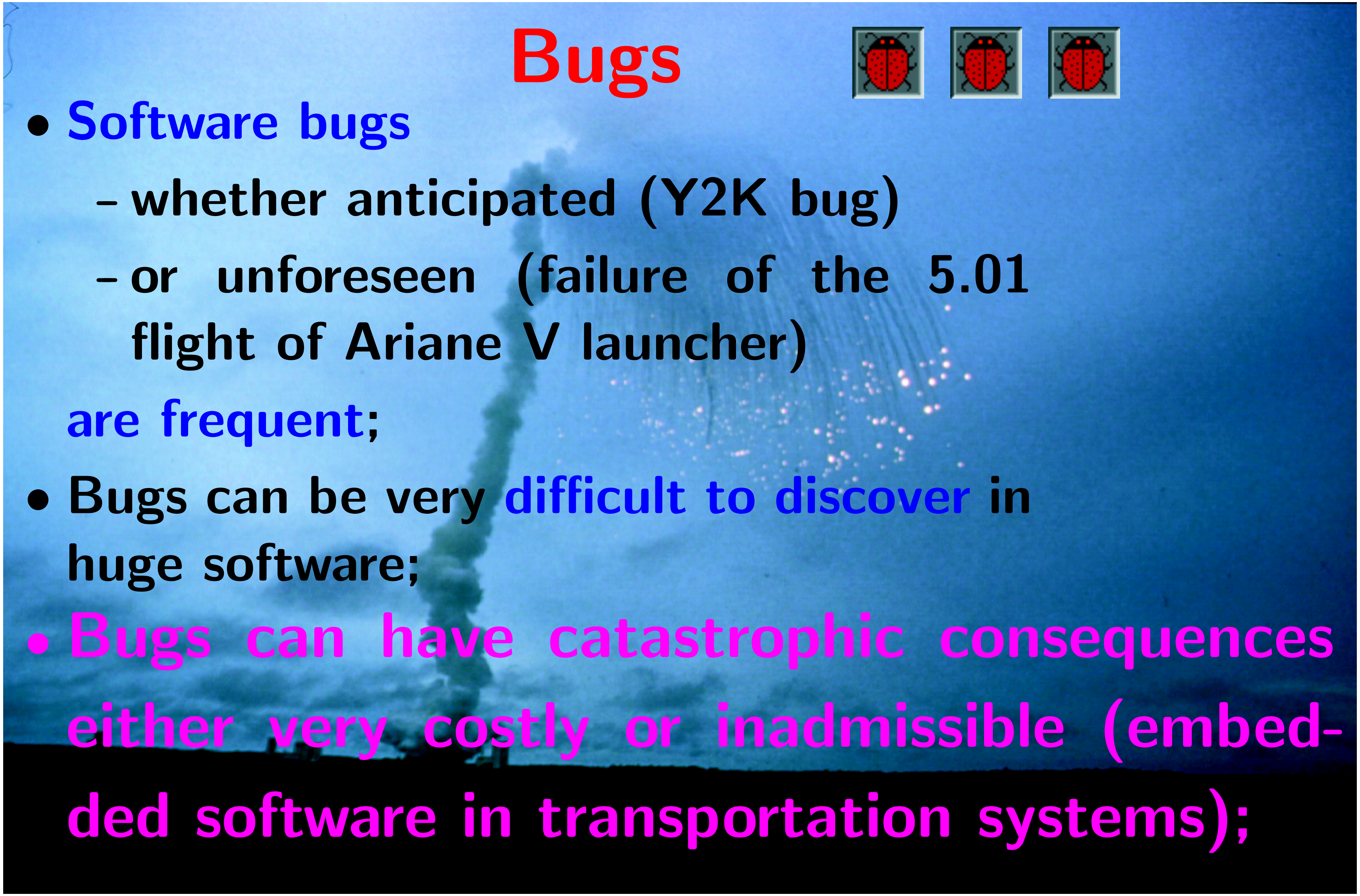
  **are quite frequent**;

# Bugs

- Software bugs
  - whether anticipated (Y2K bug)
  - or unforeseen (failure of the 5.01 flight of Ariane V launcher)

  are quite frequent;

- Bugs can be very difficult to discover in huge software;



# Bugs

- Software bugs
  - whether anticipated (Y2K bug)
  - or unforeseen (failure of the 5.01 flight of Ariane V launcher)

  are frequent;
- Bugs can be very difficult to discover in huge software;
- Bugs can have catastrophic consequences either very costly or inadmissible (embedded software in transportation systems);

# The estimated cost of an overflow

- **$ 500 000 000**
- **Including indirect costs (delays, lost markets, etc):**

**$ 2 000 000 000**

# Overview



Please feal free to ask questions during the talk.

# Present Day Empirical Debugging and Formal Verification Methods

# Present day responses to bugs

**Use the computer to find/prevent programming errors.**

- **Empirical methods:** try to **execute/simulate the program** in enough representative possible environments;
- **Formal methods:** try to **mecanically prove** that program execution is correct in all specified environments.

# Formal method based program verification

**Deductive methods**: The proof size is exponential in the program size!

**Model-checking**: Restricted to finite models. Gained only a factor of 100 in 10 years. The limit seems to be reached!

**Program static analysis**: Can analyze large programs (220 000 lines of C) but specifications are simple and the abstraction hence the design of the analyzer is manual!

**No single formal method can ultimately solve the verification problem.**

# Current trend: combine formal methods

- **User designed abstraction: derive a program finite abtract model by abstract interpretation, prove the correctness of the abstraction by deductive methods, later verify the abstract model by model-checking;**

# Current trend: combine formal methods

- **User designed abstraction:** derive a program finite abtract model by **abstract interpretation**, prove the correctness of the abstraction by **deductive methods**, later verify the abstract model by **model-checking**;
- **Fundamental limitation [1]: finding the appropriate abstraction and deriving the abstract semantics is as difficult as doing the proof!**

## Reference

[1] P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B.Y. Choueiry and T. Walsh, editors, *Proc. 4th Int. Symp. on Abstraction, Reformulations and Approximation, SARA '2000*, Horseshoe Bay, TX, USA, Lecture Notes in Artificial Intelligence 1864, pages 1–25. Springer-Verlag, 26–29 July 2000.

**No combination of formal methods can ultimately solve the verification problem either.**

# Proposed Alternative: Abstract Interpretation Based Program Testing

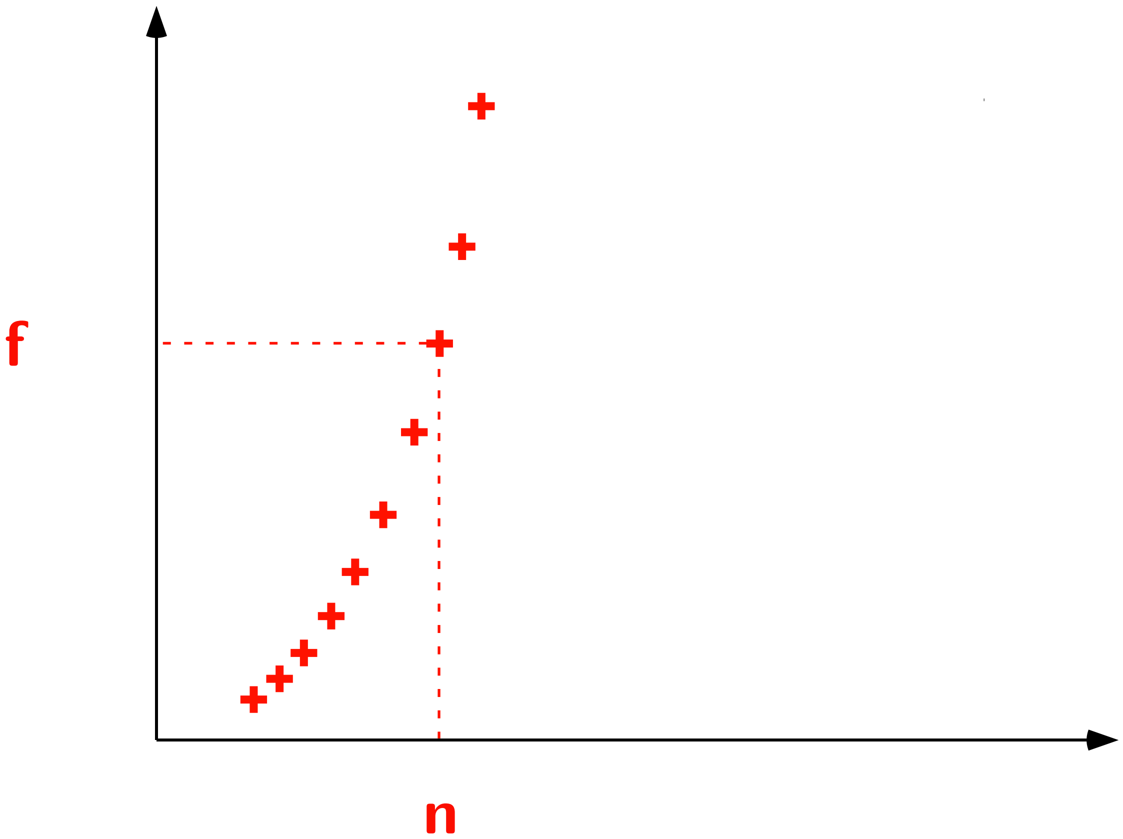
# Combine empirical and formal methods

- The user provides **local formal abstractions of the program specifications** using predefined abstractions[1];
- The program is evaluated by **abstract interpretation of the formal semantics** of the program[2];
- If the local abstract specification **cannot be proved** correct, a **more precise abstract domain** must be considered[3];
- **The process is repeated until appropriate coverage of the specification.**

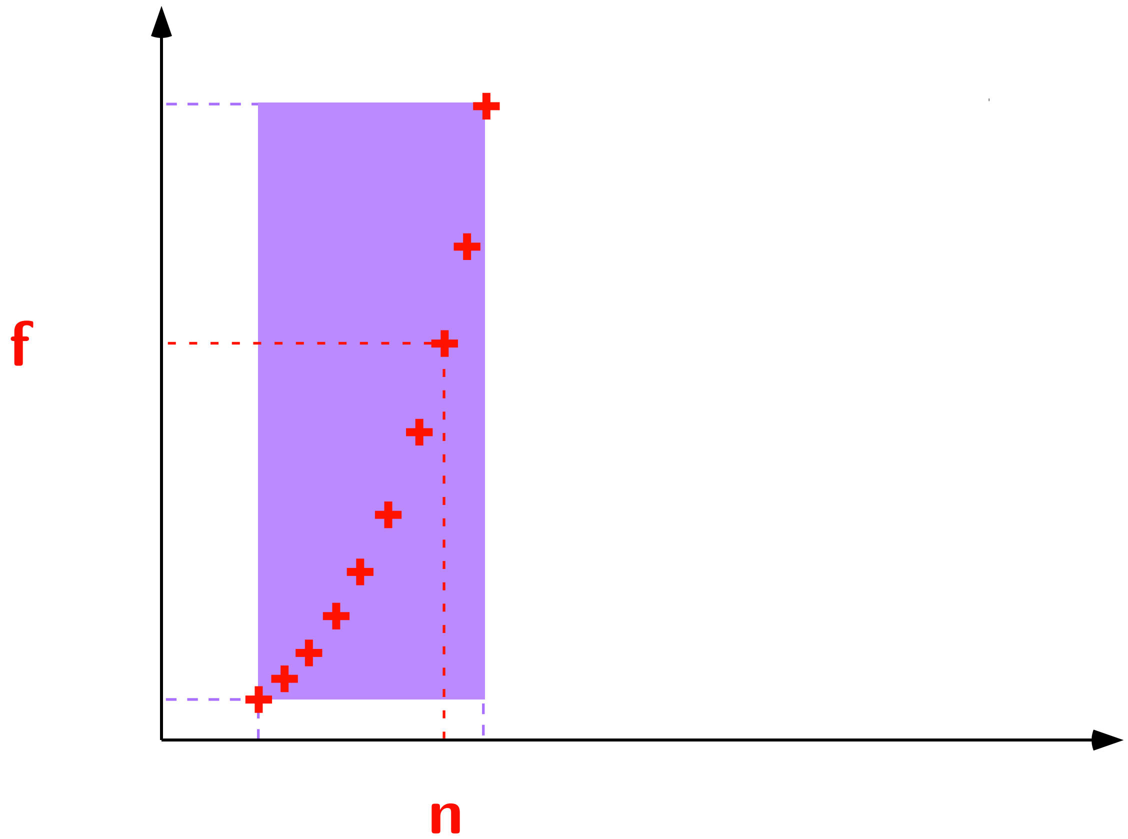
---

1 thus replacing infinitely many test data.
2 thus replacing program execution on the test data.
3 similarly to different test data.

# Example of predefined abstraction



# Example of predefined abstraction: intervals



# A tiny example

```
read(n);
f := 1;
while (n <> 0) do
    f := (f * n);
    n := (n - 1)
od;
```

■ user program

# A tiny example

```
read(n);
f := 1;
while (n <> 0) do
    f := (f * n);
    n := (n - 1)
od;
sometime true;;
```

- user program
- user specification

# A tiny example

```
0: { n:[−∞,+∞]?; f:[−∞,+∞]? }
  read(n);
1: { n:[0,+∞]; f:[−∞,+∞]? }
  f := 1;
2: { n:[0,+∞]; f:[−∞,+∞] }
  while (n <> 0) do
    3: { n:[1,+∞]; f:[−∞,+∞] }
      f := (f * n);
    4: { n:[1,+∞]; f:[−∞,+∞] }
      n := (n - 1)
    5: { n:[0,+∞]; f:[−∞,+∞] }
  od;
6: { n:[−∞,+∞]?; f:[−∞,+∞] }
sometime true;;
```

- static analyzer inference
- user program
- user specification

# A tiny example

```
0: { n:[−∞,+∞]?; f:[−∞,+∞]? }
  read(n);
1: { n:[0,+∞]; f:[−∞,+∞]? }
  f := 1;
2: { n:[0,+∞]; f:[−∞,+∞] }
  while (n <> 0) do
    3: { n:[1,+∞]; f:[−∞,+∞] }
      f := (f * n);
    4: { n:[1,+∞]; f:[−∞,+∞] }
      n := (n - 1)
    5: { n:[0,+∞]; f:[−∞,+∞] }
  od;
6: { n:[−∞,+∞]?; f:[−∞,+∞] }
sometime true;;
```

- static analyzer inference
- definite error
- no error
- potential error
- user program
- user specification

# A tiny example (cont'd)

```
initial (n < 0);
f := 1;
while (n <> 0) do
    f := (f * n);
    n := (n - 1)
od
```

- user specification
- user program

# A tiny example (cont'd)

```
0: { n:⊥; f:⊥ }
  initial (n < 0);
1: { n:[−∞,-1]; f:Ω }
  f := 1;
2: { n:[−∞,-1]; f:[−∞,1] }
  while (n <> 0) do
    3: { n:[−∞,-1]; f:[−∞,1] }
      f := (f * n);
    4: { n:[−∞,-1]; f:[−∞,0] }
      n := (n - 1)
    5: { n:[−∞,-2]; f:[−∞,0] }
  od
6: { n:⊥; f:⊥ }
```

- static analyzer inference
- user specification
- user program

# A tiny example (cont'd)

```
0: { n:⊥; f:⊥ }[4]
  initial (n < 0);
1: { n:[−∞,-1]; f:Ω }
  f := 1;
2: { n:[−∞,-1]; f:[−∞,1] }
  while (n <> 0) do
    3: { n:[−∞,-1]; f:[−∞,1] }
      f := (f * n);
    4: { n:[−∞,-1]; f:[−∞,0] }
      n := (n - 1)
    5: { n:[−∞,-2]; f:[−∞,0] }
  od
6: { n:⊥; f:⊥ }
```

- static analyzer inference
- user specification
- user program
- no error
- potential error

⊥ unreachable code

4 If execution is ever started under the initial conditions, an error (∗ or – overflow) is inevitable.

# Comparing with program debugging

- **Similarity:** user interaction;
- **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).**

# Comparing with abstract model-checking

- **Similarities:**
  - use of specifications instead of test data sets;
  - hability to automatically produce counter-examples [5];

[5] or specifications of infinitely many such counter-examples in the case of abstract program testing.

# Comparing with abstract model-checking (cont'd)

- Essential differences:
  - reasoning on the concrete program (not on a program model);
  - no attempt to make a one-shot complete formal proof of the specification;
  - interaction with user repeatedly providing partial specifications in a form close to conventional debugging;
  - predefined abstractions (not user defined);
  - finite and infinite abstract domains are allowed.

# A Few Technical Issues

# Paper content

- The paper discusses a few technical issues showing that:

  (abstract) model-checking based techniques are not adequate

  for program abstract testing and that

  program analysis based techniques are more precise

  because they take approximation into account.

# Needless limitations of model-checking

- The basic state to state abstraction of model checking ($\alpha(S) = \{h(s) \mid s \in S\}$) is not general enough;
- Finite abstract properties are not expressive enough;
- Abstract predicate transformers are imprecise [6], because no local iteration is performed;
- Fixpoint checking algorithms are imprecise [6], because they don't incorporate all available information;
- **Fixpoint combinations approximations are suboptimal [6], since fixpoint computations are not exact [7].**

---

6 Although they are optimal in the case of finite abstract property spaces.

7 which is impossible with infinite abstract domains (but is anyway more precise than with any finite domain).

# A single simple illustration

- The basic state to state abstraction of model checking ($\alpha(S) = \{h(s) \mid s \in S\}$) is not general enough;
- Finite abstract properties are not expressive enough;
- Abstract predicate transformers are imprecise [6], because no local iteration is performed;
- **Fixpoint checking algorithms are imprecise [6], because they don't incorporate all available information;**
- Fixpoint combinations approximations are suboptimal [6], since fixpoint computations are not exact [7].

---

6 Although they are optimal in the case of finite abstract property spaces.

7 which is impossible with infinite abstract domains (but is anyway more precise than with any finite domain).

# Na ve fixpoint checking

- In order to check that [6]:

$$\mathsf{lfp}^{\sqsubseteq} F\ ^{7} \sqsubseteq I\ ^{8}$$

  - Compute $J$ such that $F(J) \sqsubseteq J$ by fixpoint approximation methods;
  - It follows that $\mathsf{lfp}^{\sqsubseteq} F \sqsubseteq J$ [9];
  - Check that $J \sqsubseteq I$.

---

6 $F$ is a monotonic operator on a complete lattice ordered by $\sqsubseteq$; $\mathit{lfp}^{\sqsubseteq} F$ is the $\sqsubseteq$-least fixpoint of $F$.

7 $\mathit{lfp}^{\sqsubseteq} F$ is the program abstract semantics.

8 $I$ is a user-provided (so-called "safety") specification.

9 by Tarski's fixpoint theorem. In general the problem is undecidable so equality is impossible.

# Precise fixpoint checking

- In order to check that [6]:

$$\mathsf{lfp}^{\sqsubseteq} F \sqsubseteq I$$

  - Compute $J$ such that $F(J) \sqcap I \sqsubseteq J$ by fixpoint approximation methods;
  - It follows that $\mathsf{lfp}^{\sqsubseteq} \boldsymbol{\lambda} X \cdot F(X) \sqcap I \sqsubseteq J$;
  - Check that $F(J) \sqsubseteq I$.
- It follows that $\mathsf{lfp}^{\sqsubseteq} F \sqsubseteq I$;

# Precise fixpoint checking

- In order to check that [6]:

$$\mathsf{lfp}^{\sqsubseteq} F \sqsubseteq I$$

  - Compute $J$ such that $F(J) \sqcap I \sqsubseteq J$ by fixpoint approximation methods;
  - It follows that $\mathsf{lfp}^{\sqsubseteq} \boldsymbol{\lambda} X \cdot F(X) \sqcap I \sqsubseteq J$;
  - Check that $F(J) \sqsubseteq I$.
- It follows that $\mathsf{lfp}^{\sqsubseteq} F \sqsubseteq I$;
- Correct even if the user specification is erroneous (i.e. $\mathsf{lfp}^{\sqsubseteq} F \not\sqsubseteq I$).

# Conclusions

# Conclusions

- As an alternative to program debugging, formal methods have been developed to prove that a semantics or model of the program satisfies a specification;
- Because of theoretical and practical limitations, these formal methods have had more successes for finding bugs than for proving their absence;
- **For complex programs, the basic idea of complete program verification underlying the deductive and model checking methods must be abandoned in favor of debugging.**

# Conclusion (cont'd)

- In the context of debugging, we have shown that abstract interpretation based program static analysis can be extended to **abstract program testing**;

- **Abstract interpretation** methods offer powerful techniques which, in the presence of **approximation**, can be viable **alternatives to** both the exhaustive search of **model-checking** and the partial exploration methods of classical **debugging**.

# THE END, THANK YOU.