Introduction to a Discussion on Mechanical Formal Methods for Software Verification

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

IFIP WG2.3 Santa Cruz Meeting, January 7–12, 2001

Find the **last error** in a software system [1];

Can **mechanical formal methods** solve the ultimate verification problem?

[1] Obviously the *ultimate verification problem* is restricted to the unhappy many who are completely unable to derive their programs correctly from sound and complete formal specifications.

Mechanical Formal Methods

Considered mechanical formal methods for **software verification**:

- Program **typing**;
- Static **program analysis**;
- Abstract **model checking**;
- **Deductive methods**;
- **Combinations** of mechanical formal methods.

Consider decidable analyses only, by restricting both on specifications (allowed types) and on programs;

Clean presentation of the type analysis (inference algorithm) through an equivalent logical formal system (type verification);

Extended to complex data structures, polymorphism, exceptions and separate modules in a way that scales up for large

Can a single mechanical formal method ultimately solve the verification problem?

NO!

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

[3] even 1 400 000 lines for control-independent very weak properties.

Fundamental limitation [1]: 1^{o}) abstraction discovery and 2^{o}) abstract semantics derivation is as difficult as doing the proof! (resp. 1^{o}) invariant discovery & 2^{o}) invariant verification).

Reference

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

Can some combination of formal methods ultimately solve the verification problem?

NO!

Possible Alternative: Combine Empirical and Formal Methods

Example: Abstract Program Testing

Debugging
Run the program
On test data
Checking if all right
Providing more tests
Until coverage

Example: Abstract Program Testing

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

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

More details needed? Yes No

The user provides the program specifications using predefined abstractions [4];

The program is evaluated by abstract interpretation of the formal semantics of the program [5];

Example: Abstract Program Testing

A Tiny Example

4 thus replacing infinitely many test data.
5 thus replacing program execution on the test data.
6 similarly to different test data.

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

■ user program

A Tiny Example

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

■ static analyzer inference
■ user program
■ user specification

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:[1,+∞] }
  while (n <> 0) do
    3: { n:[1,+∞]; f:[1,+∞] }
      f := (f * n);
    4: { n:[1,+∞]; f:[1,+∞] }
      n := (n - 1)
    5: { n:[0,+∞]; f:[1,+∞] }
  od;
6: { n:[0,0]; f:[1,+∞] }
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

Comparing with Program Debugging

- **Similarity:** user interaction;

A Tiny Example (Cont'd)

```
0: { n:⊥; f:⊥ }
  initial (n < 0);
1: { n:[−∞,-1]; f:[−∞,+∞]? }
  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

Comparing with Program Debugging

- **Similarity:** user interaction;
- **Essential differences:**
  - user provided test data are replaced by abstract specifications;

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;

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;i
  - 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 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;

Comparing with Abstract Model Checking

- Similarities:
  - use of specifications instead of test data sets;

Comparing with Abstract Model Checking

- **Similarities:**
  - use of specifications instead of test data sets;
  - ability to automatically produce counter-examples[7];

[7] 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;

Comparing with Abstract Model Checking (Cont'd)

- **Essential differences:**
  - reasoning on the concrete program (not on a program model);

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;

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);

Points of Discussion

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.

Questions

- Program debugging is still the prominent industrial program "verification" method, why?

Questions

- Program debugging is still the prominent industrial program "verification" method, why?
- Full program verification by formal methods is either impossible (e.g. typing/program analysis) or costly since it ultimately requires user interaction (e.g. abstract model checking/deductive methods) so are mechanized formal methods widely applicable?

Questions (Cont'd)

If for large and complex programs, low cost complete verification by mechanized formal methods is not viable then:

- Universal and reusable hence commercializable abstractions lead to cost-effective[8] and automatic program analyzers so can approximate program analysis be enhanced to partial program verification?

[8] Less than 0.25$ per program line costing 50 to 80$.

Questions

- Program debugging is still the prominent industrial program "verification" method, why?
- Full program verification by formal methods is either impossible (e.g. typing/program analysis) or costly since it ultimately requires user interaction (e.g. abstract model checking/deductive methods) so are mechanized formal methods widely applicable?
- For program verification, semantic abstraction is mandatory but difficult whence hardly automatizable so can abstractions be designed by programmers?

Questions (Cont'd)

If for large and complex programs, low cost complete verification by mechanized formal methods is not viable then:

- Universal and reusable hence commercializable abstractions lead to cost-effective[8] and automatic program analyzers so can approximate program analysis be enhanced to partial program verification?
- Otherwise, if user interaction is definitely needed, can abstract program testing be viable alternative to both the exhaustive search of model-checking and the partial exploration methods of classical debugging?

[8] Less than 0.25$ per program line costing 50 to 80$.

FURTHER CONTROVERSIAL POINTS TO DISCUSS?

THE END