Verification for Legacy Programs

Michael Ernst
MIT Computer Science and Artificial Intelligence Lab

Verified Software: Theories, Tools, Experiments
10 October 2005

Verification: long- and short-term

Long-term goal:

- All programs are written with verification in mind
- All are accompanied by a formal specification

In the short term, this will not be the case

- Tool deficiencies
- Programmer skills and mindset
- Legacy code

What can we do?

- To address current needs
- To move toward the desired future

Legacy code and executions

Legacy code is neither specified nor well-understood
Rich information is available in test suites and executions

- Can be integrated into formal verification

Formal and informal techniques are compatible

- Each has unique value

Applications

1. Specification inference from executions
2. Guiding human proofs
3. Automating automated theorem-proving
4. Detecting incompatible software upgrades
5. Predicting test outcomes

For each of these, a theory, a tool, and an experiment

1. Specification inference from executions

Theory: Program executions can yield semi-formal properties

Observational abstractions state program properties [ICSE99]

- Syntactically identical to formal specification
- Reports what the program actually did
- Automatically generated from executions
    - No guarantee (as with any dynamic technique)
    - Typically much richer than static analysis output
- Relatively insensitive to test suite, overwhelmingly accurate, and useful

Example

```
// theArray != null;
// \typeof(theArray) == \type(java.lang.Object[]);
// -1 <= topOfStack <= theArray.length-1;
// theArray[0..topOfStack] elements != null
// theArray[topOfStack+1..] elements == null
public class Stack {
    private Object [ ] theArray;
    private int        topOfStack;

    ...

    // (\result == false)  ==  (topOfStack >= 0);
    // (\result == true)  ==  (topOfStack == -1);
    public boolean isEmpty( ) { ... }
}
```

Dynamic detection of likely invariants

Tool: Daikon invariant detector outputs operational abstractions

- Works on C, C++, Java, Perl, ...
- Rich output, customizable and extensible
- Relatively scalable [FSE04]
- Integrated with many other tools
- Publicly available (with source code):
  `http://pag.csail.mit.edu/daikon/`
- Well-documented and supported

Experiments: Used in over 60 published papers

- See `http://pag.csail.mit.edu/daikon/pubs/`

2. Guiding human proofs

Theory: executions can aid humans in formal verification

- Need not throw away results of testing

Proof assistants require lemmas

- Properties always true during testing

Proof assistants require tactics

- Adapt test suite generation approach

Providing assistance

Tool: Generates lemmas and tactics

- Integrated with both Isabelle and LP

Experiments: [STTT04]

- 3 distributed algorithms
- Eliminated 90% of human interaction with both theorem-provers

3. Automating automated theorem-proving

Theory: Possibly unsound data can be automatically verified

Problem with operational abstractions: unsound
Problem with ATP: requires annotations
Solution: Use run-time properties as annotations

- Ease static checking, and gain guarantees on results

Annotations for static checking

Tools: ESC/Java, integration code [RV01]

Experiments: [ISSTA02, FSE02]

- 90% of properties are verifiable by ESC/Java
- 90% of necessary annotations are present
- Humans are aided even by artificially bad results
    - It is easy to check, but hard to generate

4. Detecting incompatible software upgrades

Theory: Can automatically warn of bad component upgrades

Scenario: A vendor releases version 2.0 of a software package

- Will it break your system?

Use a novel logical test to compare

- tested behavior of the new component
- observed behavior of the old component [FSE03]

Results:

- Guarantee is as good as for current component
- Proof of relative soundness [SAVCBS04]

Preventing bad upgrades

Tools: Simplify theorem prover

Experiments: [ECOOP04]
Upgrades to Perl components and to the C standard library
The tool detected incompatible upgrades
The tool approved of safe upgrades

5. Predicting test outcomes

Theory: We can effectively predict test outcomes

It is easy to generate many test inputs
It is hard to determine a (legacy) program's desired behavior

- That is, to construct a test case from a test input

Automatically classifying test inputs

Tools: Eclat system for generating and classifying test inputs

- `http://pag.csail.mit.edu/eclat/`

Randomly generates many inputs
Comparing to previous behavior, classifies each as

- normal operation
- illegal input
- bug — show these to the user

Experiments: [ECOOP05]

- Outputs a small number (2–3) of suspicious test inputs
- Found bugs in real programs
    - including formally specified ones!

Conclusion

Legacy programs are here to stay
Sound and unsound techniques are complementary

- . . . and compatible, even for verification
- combining them leads to rich new ideas and useful tools

Automatically generated pseudo-specifications are

- quite accurate in practice
- an aid to formal verification
- a step toward a fully verified future
- useful for many other tasks