Mining Specifications

(lots of) code → specifications

Verification: beyond engine-less cars

Recent successes.
- specifications languages
- checkers
- abstractors

What’s still missing?
- specifications

Drivers wanted.

So who formulates specifications?

Programmers? Probably not.

Why they won’t:
- too busy; yet another language to learn?
- specifications aren’t cool.

Why they shouldn’t:
- may misunderstand usage rules.
- may not know all usage rules.

Mining Specifications:
- Convenience.
- Like in data mining, discover surprise rules.
Advantages of mining

- Exploits the massive programmers’ effort reflected in the code.

- Programmers resolved many problems:
  - Incomplete system requirements.
  - Incomplete API documentation.
  - Implementation-dependent rules.

- Want redundancy? (without redundant programming)
  - Ask multiple programmers (and vote).

Our output: a specification

```
x = socket()
bind(x)
listen(x)
y = accept(x)
read(y)
write(y)
close(y)
close(x)
```

How do we mine?

Underlying premise:

Even bad software is debugged enough to show hints of correct behavior.

- Maxim: Common usage is the correct usage.
Mining = machine learning

Reduce the problem into the well-known problem of learning regular languages.

Obstacles:
1. bugs from source code may be learned into specification
2. what is "common" behavior?

Solutions:
1. learn from dynamic behavior
2. learn probabilistically
   - learn from traces into probabilistic FSMs

---

Input: trace(s)

7 = socket(2, 1, 0);
bind(7, 0x400110, 16);
listen(7, 5);
8 = accept(7, 0x400200, 0x400240);
read(8, 0x400320, 255);
write(8, 0x400320, 12);
read(8, 0x400320, 255);
write(8, 0x400320, 7);
close(8);
10 = accept(7, 0x400200, 0x400240);
read(10, 0x400320, 255);
write(10, 0x400320, 13);
close(10);
close(7);
...

The mining algorithm

dynamic execution (traces) -- trace abstraction -- usage scenarios (strings) -
(specification (NFA))
generalized scenarios (probabilistic NFA)
extract heavy core (and approve)
dynamic exe. to be checked (trace) -- dynamic checker
OK/bug
Trace abstraction: 4 challenges

- Traces interleave useful and useless events.
  - RegExp learner cannot separate them.

- Specifications must include both temporal and value-flow constraints.
  - RegExp learner only good with temporal constraints.

- Only some of API calls' arguments impose "true" dependences.
  - Infeasible to learn value-flow constraints on all arguments.

- Specifications may impose only partial order.
  - Encoding all legal partial orders would produce a huge FSM.

Preliminary experiments

Attempted to learn and verify two published X Windows rules

As of Friday:

1. A timestamp-passing rule
   - learned the rule! (compact: 6 states)
   - bugs in 2 out of 17 programs (ups, e93)

2. SetOwner(x) must be followed by GetSelection(x)
   - failed to learn the rule (small learning set) but
   - bugs in 2 out of 5 programs (xemacs, ups)
Related work

**Arithmetic pre/post conditions**
- Daikon, Houdini
- properties orthogonal from us
- eventually, we may need to include and learn some arithmetic relationships

**Temporal relationships over calls**
- intrusion detection: [Ghosh et al], [Wagner and Dean]
- software processes: [Cook and Wolf]
- error checking: [Engler et al’ SOSP 2001]
- lexical and syntactic pattern matching
- user must write templates (e.g., <a> always follows <b>)

Ongoing work

*Mechanize tool. Find more gold.*

Future work

*Give gold to jewelers.*
Summary

- Semi-automatically creating well-formed, non-trivial specifications is an important part of the verification tool chain.

- Contributions:
  - introduced specifications mining
  - phrased it as probabilistic learning from dynamic traces
  - decomposed it into a sequence of subproblems (using an off-the-shelf learner)
  - developed dynamic checker
  - found bugs