Automatic Generation of Program Specifications

Jeremy Nimmer

MIT Lab for Computer Science

http://pag.lcs.mit.edu/

Joint work with Michael Ernst

Synopsis

Specifications are useful for many tasks

- Use of specifications has practical difficulties

Dynamic analysis can capture specifications

- Recover from existing code
  - Infer from traces
- Results are accurate (90%+)
  - Specification matches implementation

Outline

- Motivation
- Approach: Generate and check specifications
- Evaluation: Accuracy experiment
- Conclusion

Advantages of specifications

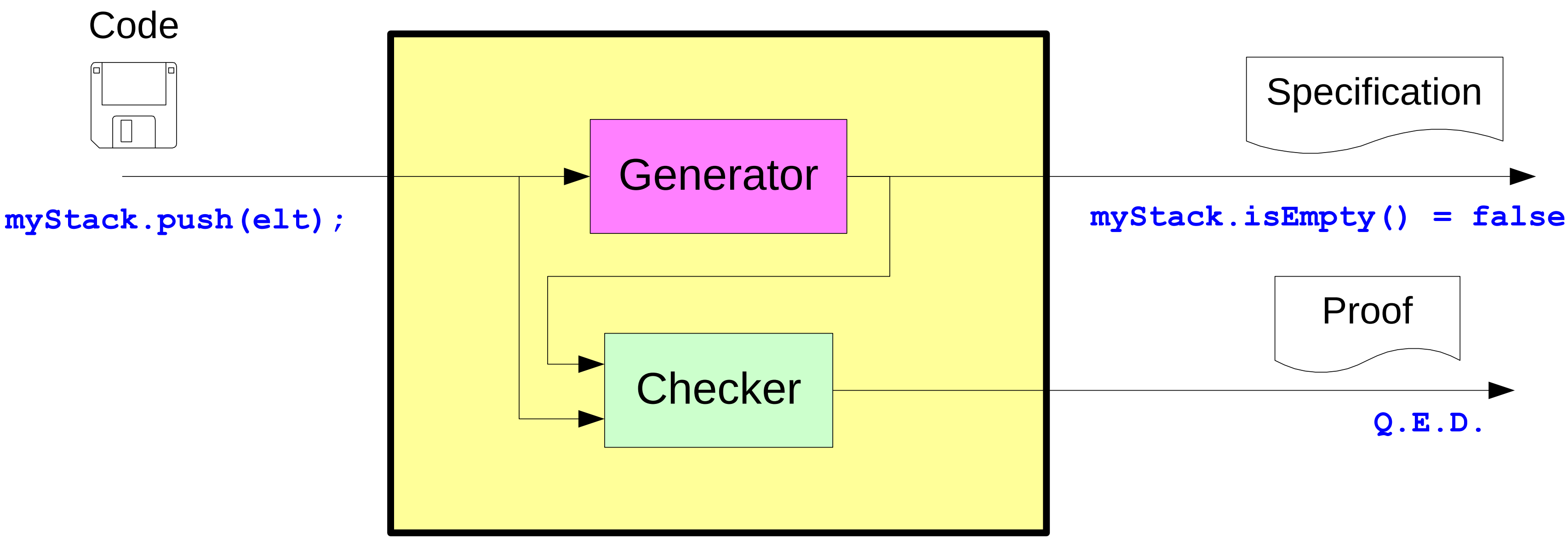
- Describe behavior precisely
- Permit reasoning using summaries
- Can be verified automatically

Problems with specifications

- Describe behavior precisely
  - Tedious and difficult to write and maintain
- Permit reasoning using summaries
  - Must be accurate if used in lieu of code
- Can be verified automatically
  - Verification may require uninteresting annotations

Solution

Automatically generate and check specifications from the code



Solution scope

- Generate and check "complete" specifications
  - Very difficult
- Generate and check partial specifications
  - Nullness, types, bounds, modification targets, ...
- Need not operate in isolation
  - User might have some interaction
  - Goal: decrease overall effort

Outline

- Motivation
- Approach: Generate and check specifications
- Evaluation: Accuracy experiment
- Conclusion

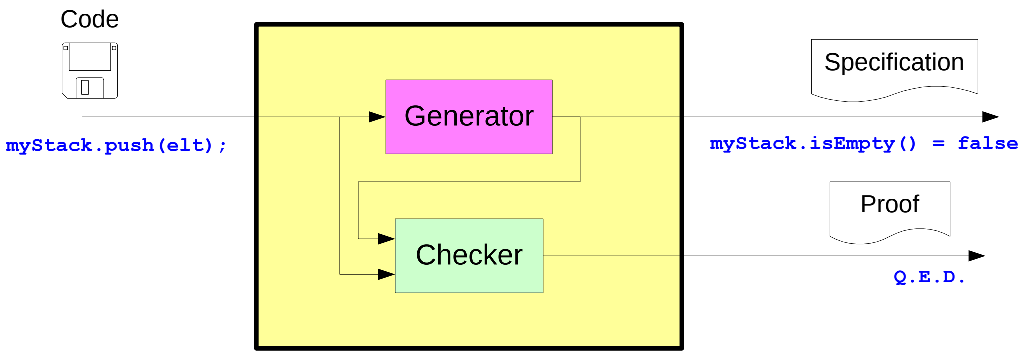
Previous approaches

Generation:

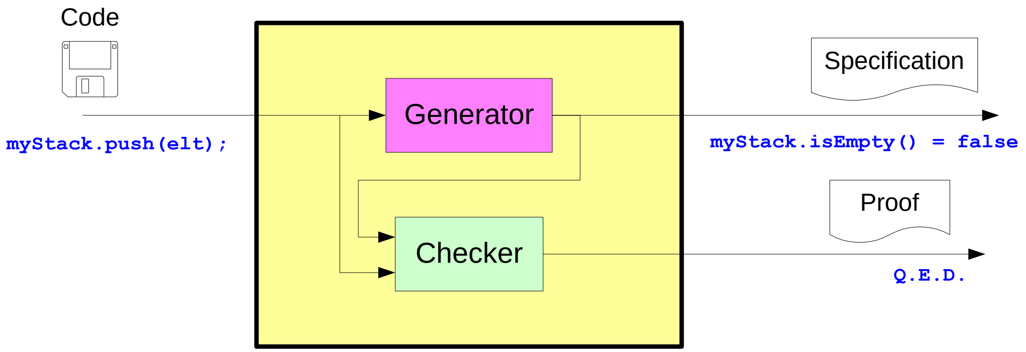
- By hand
- Static analysis

Checking

- By hand
- Non-executable models

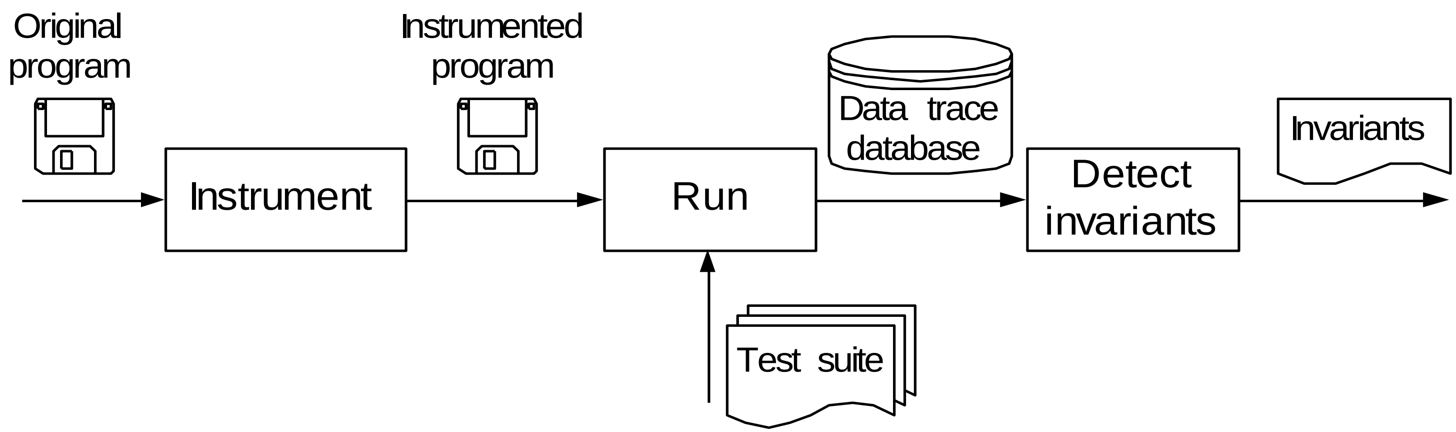


Our approach



- Dynamic detection proposes *likely* properties
- Static checking verifies properties
- Combining the techniques overcomes the weaknesses of each
  - Ease annotation
  - Guarantee soundness

Daikon: Dynamic invariant detection



Look for patterns in values the program computes:

- Instrument the program to write data trace files
- Run the program on a test suite
- Invariant detector reads data traces, generates potential invariants, and checks them

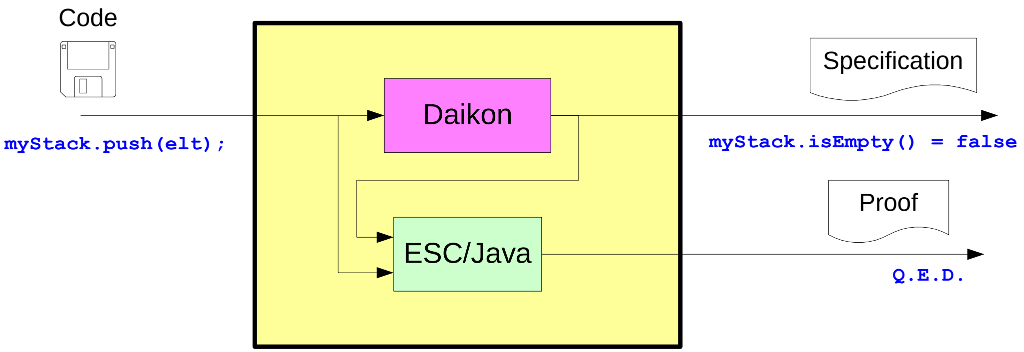
ESC/Java: Invariant checking

- ESC/Java: Extended Static Checker for Java
- Lightweight technology: intermediate between type-checker and theorem-prover; unsound
- Intended to detect array bounds and null dereference errors, and annotation violations

```
/*@ requires x != null */
/*@ ensures this.a[this.top] == x */
void push(Object x);
```

- Modular: checks, and relies on, specifications

Integration approach



Run Daikon over target program

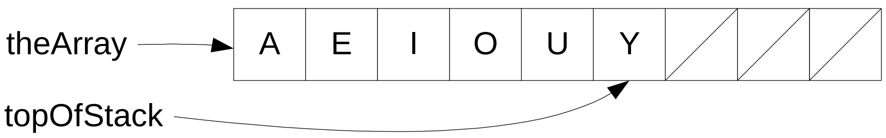
Insert results into program as annotations

Run ESC/Java on the annotated program

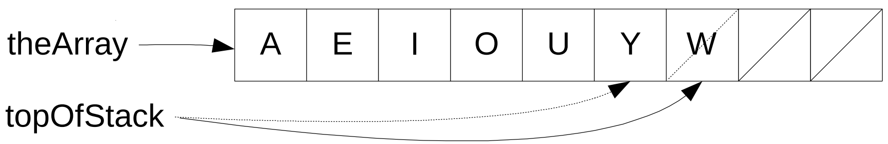
All steps are automatic.

Stack object invariants

```
public class StackAr {
  Object[] theArray;
  int topOfStack;
 /*@
   invariant theArray != null;
   invariant \typeof(theArray) == \type(Object[]);
   invariant topOfStack >= -1;
   invariant topOfStack < theArray.length;
   invariant theArray[0..topOfStack] != null;
   invariant theArray[topOfStack+1..] == null;
 */
   ...
```



Stack push method




```
/*@ requires x != null;
    requires topOfStack < theArray.length - 1;
    modifies topOfStack, theArray[*];
    ensures topOfStack == \old(topOfStack) + 1;
    ensures x == theArray[topOfStack];
    ensures theArray[0..\old(topOfStack)];
         == \old(theArray[0..topOfStack]); */
public void push( Object x ) {
  ...
}
```

Stack summary

- ESC/Java verified all 25 Daikon invariants
- Reveal properties of the implementation
  (e.g., garbage collection of popped elements)
- No runtime errors if callers satisfy preconditions
- Implementation meets generated specification

Outline

- Motivation
- Approach: Generate and check specifications
- Evaluation: Accuracy experiment
- Conclusion

Accuracy experiment

- Dynamic generation is potentially unsound
  - How accurate are its results in practice?
- Combining static and dynamic analyses should produce benefits
  - But perhaps their domains are too dissimilar?

Programs studied

- 11 programs from libraries, assignments, texts
  - Total 2449 NCNB LOC in 273 methods
- Test suites
  - Used program’s test suite if provided (9 did)
    - If just example calls, spent <30 min. enhancing
  - ~70% statement coverage

Accuracy measurement

- Compare generated specification to a verifiable specification

```
invariant theArray != null;
invariant topOfStack >= -1;
invariant topOfStack < theArray.length;
~~invariant theArray[0..length-1] == null;~~
invariant theArray[0..topOfStack] != null;
invariant theArray[topOfStack+1..] == null;
```

- Standard measures from info ret [Sal68, vR79]
  - Precision (correctness) : 3 / 4 = 75%
  - Recall (completeness) : 3 / 5 = 60%

Experiment results

- Daikon reported 554 invariants
  - Precision: **96**% of reported invariants verified
  - Recall: **91**% of necessary invariants were reported

Causes of inaccuracy

- Limits on tool grammars
  - Daikon: May not propose relevant property
  - ESC: May not allow statement of relevant property
- Incompleteness in ESC/Java
- Always need programmer judgment
- Insufficient test suite
  - Shows up as overly-strong specification
    - Verification failure highlights problem; helpful in fixing
  - System tests fared better than unit tests

Experiment conclusions

- Our dynamic analysis is accurate
  - Recovered partial specification
    - Even with limited test suites
  - Enabled verifying lack of runtime exceptions
  - Specification matches the code
- Results should scale
  - Larger programs dominate results
  - Approach is class- and method-centric

Value to programmers

Generated specifications are accurate

- Are the specifications useful?
- How much does accuracy matter?
- How does Daikon compare with other annotation assistants?

Answers at FSE'02

Outline

- Motivation
- Approach: Generate and check specifications
- Evaluation: Accuracy experiment
- Conclusion

Conclusion

- Specifications via dynamic analysis
  - Accurately produced from limited test suites
  - Automatically verifiable (minor edits)
  - Specification characterizes the code
- Unsound techniques useful in program development

Questions?

Formal specifications

- Precise, mathematical desc. of behavior [LG01]
  - (Another type of spec: requirements documents)
- Standard definition; novel use
  - Generated after implementation
  - Still useful to produce [PC86]
- Many specifications for a program
  - Depends on task
  - e.g. runtime performance

Effect of bugs

- Case 1: Bug is exercised by test suite
  - Falsifies one or more invariants
    - Weaker specification
  - May cause verification to fail
- Case 2: Bug is not exercised by test suite
  - Not reflected in specification
    - Code and specification disagree
  - Verifier points out inconsistency