An Empirical Comparison of Automated Generation and Classification Techniques for Object-Oriented Unit Testing

Marcelo d'Amorim (UIUC)
Carlos Pacheco (MIT)
Tao Xie (NCSU)
Darko Marinov (UIUC)
Michael D. Ernst (MIT)

Automated Software Engineering 2006

Motivation

- Unit testing validates individual program units
  - Hard to build correct systems from broken units
- Unit testing is used in practice
  - 79% of Microsoft developers use unit testing [Venolia et al., MSR TR 2005]
  - Code for testing often larger than project code
    - Microsoft [Tillmann and Schulte, FSE 2005]
    - Eclipse [Danny Dig, Eclipse project contributor]

Focus: Object-oriented unit testing

- Unit is one class or a set of classes
- Example [Stotts et al. 2002, Csallner and Smaragdakis 2004, …]

```
// class under test
public class UBStack {
  public UBStack(){...}
  public void push(int k){...}
  public void pop(){...}
  public int top(){...}
  public boolean
    equals(UBStack s){
     ...
  }
}
```

```
// example unit test case
void test_push_equals() {
  UBStack s1 = new UBStack();
  s1.push(1);
  UBStack s2 = new UBStack();
  s2.push(1);
  assert(s1.equals(s2));
}
```

Unit test case = Test input + Oracle

- Test Input
  - Sequence of method calls on the unit
  - Example: sequence of push, pop
- Oracle
  - Procedure to compare actual and expected results
  - Example: assert

```
void test_push_equals() {
  UBStack s1 = new UBStack();
  s1.push(1);
  UBStack s2 = new UBStack();
  s2.push(1);
  assert(s1.equals(s2));
}
```

Creating test cases

- Automation requires addressing both:
  - Test input generation
  - Test classification
    - Oracle from user: rarely provided in practice
    - No oracle from user: users manually inspect generated test inputs
      - Tool uses an approximate oracle to reduce manual inspection
- Manual creation is tedious and error-prone
  - Delivers incomplete test suites

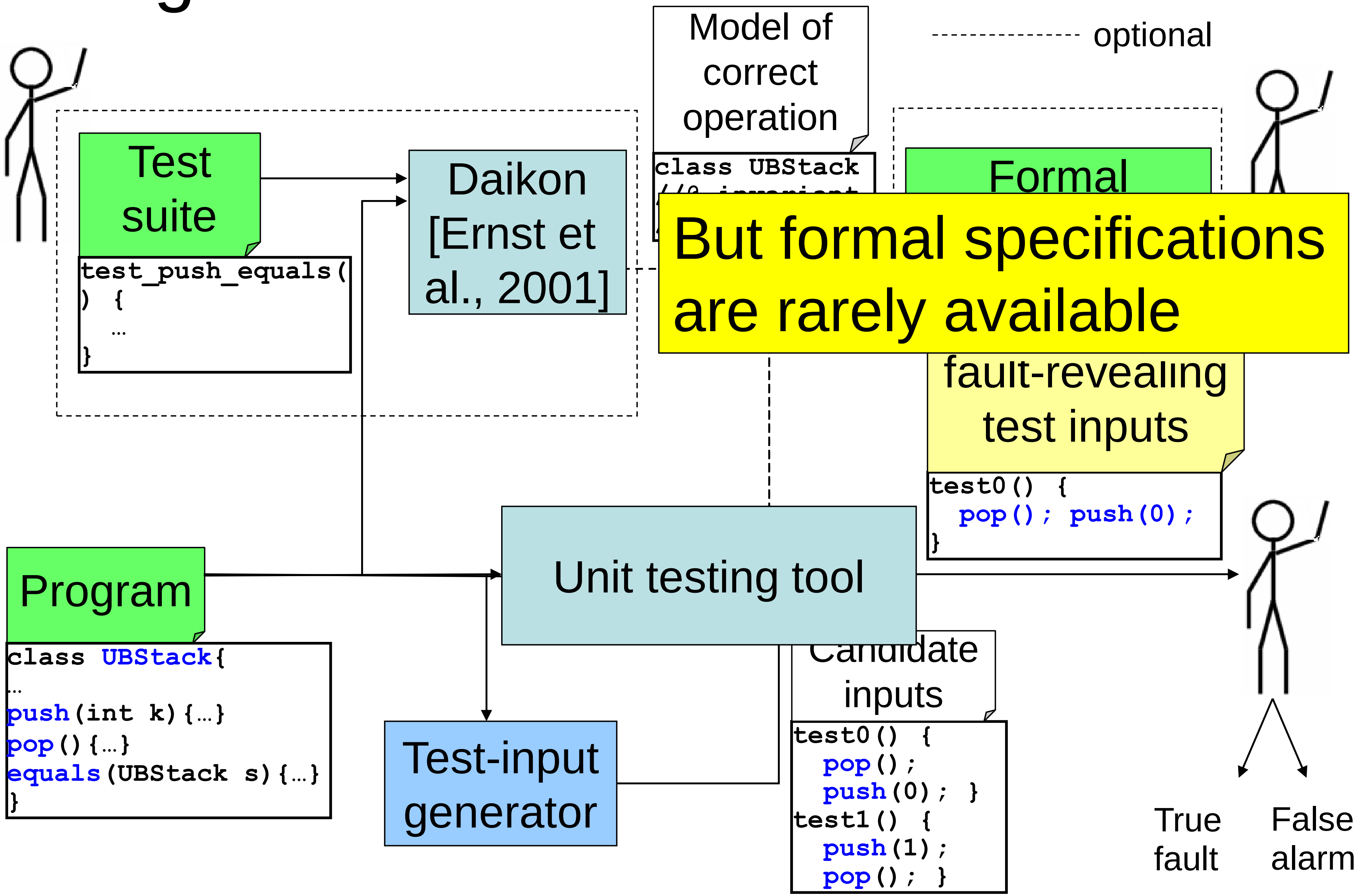
Problem statement

- Compare automated unit testing techniques by effectiveness in finding faults

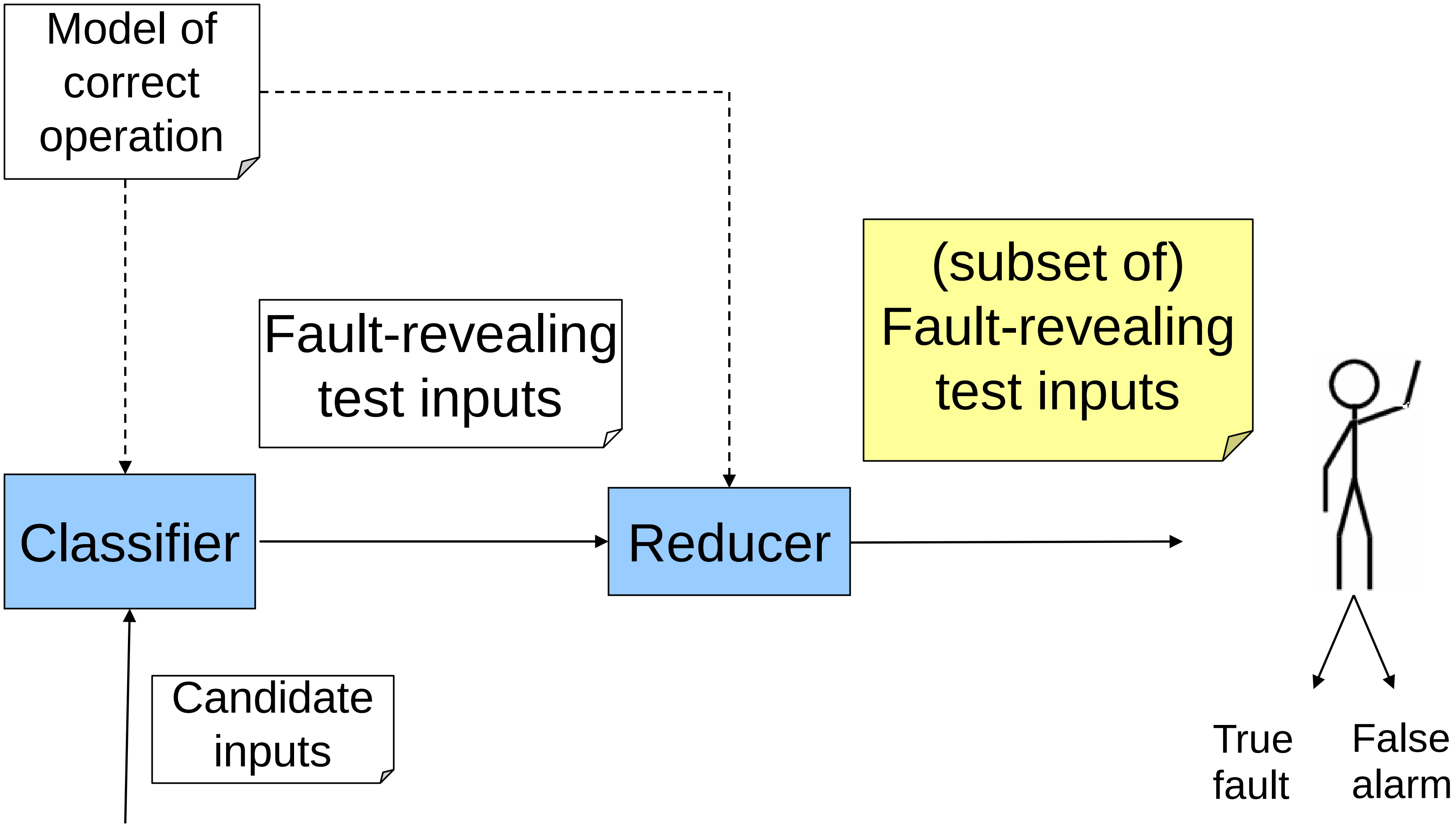
Outline

- Motivation, background and problem
- Framework and existing techniques
- New technique
- Evaluation
- Conclusions

A general framework for automation



Reduction to improve quality of output



Combining generation and classification

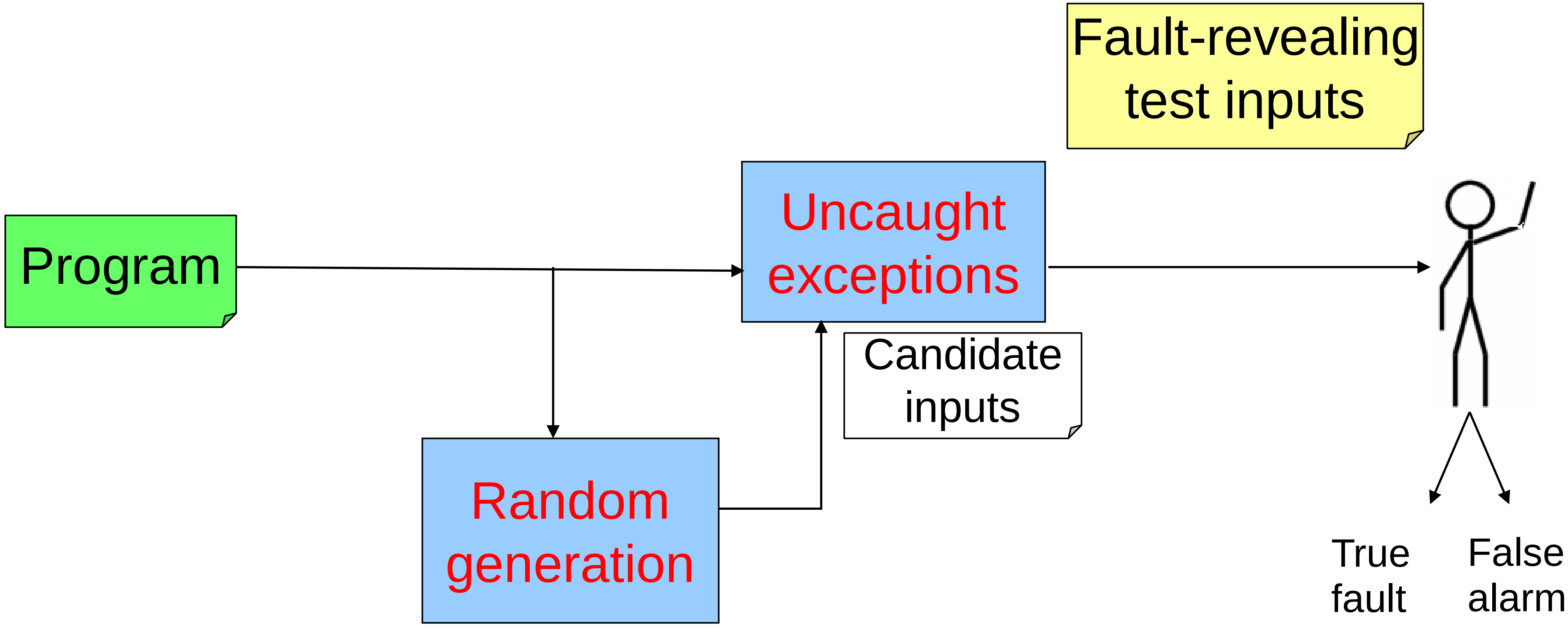
		classification	
		Uncaught exceptions (UncEx)	Operational models (OpMod)
generation	Random (RanGen)	[Csallner and Smaragdakis, SPE 2004], …	[Pacheco and Ernst, ECOOP 2005]
	Symbolic (SymGen)	[Xie et al., TACAS 2005]	?

⋮ ⋮ ⋮

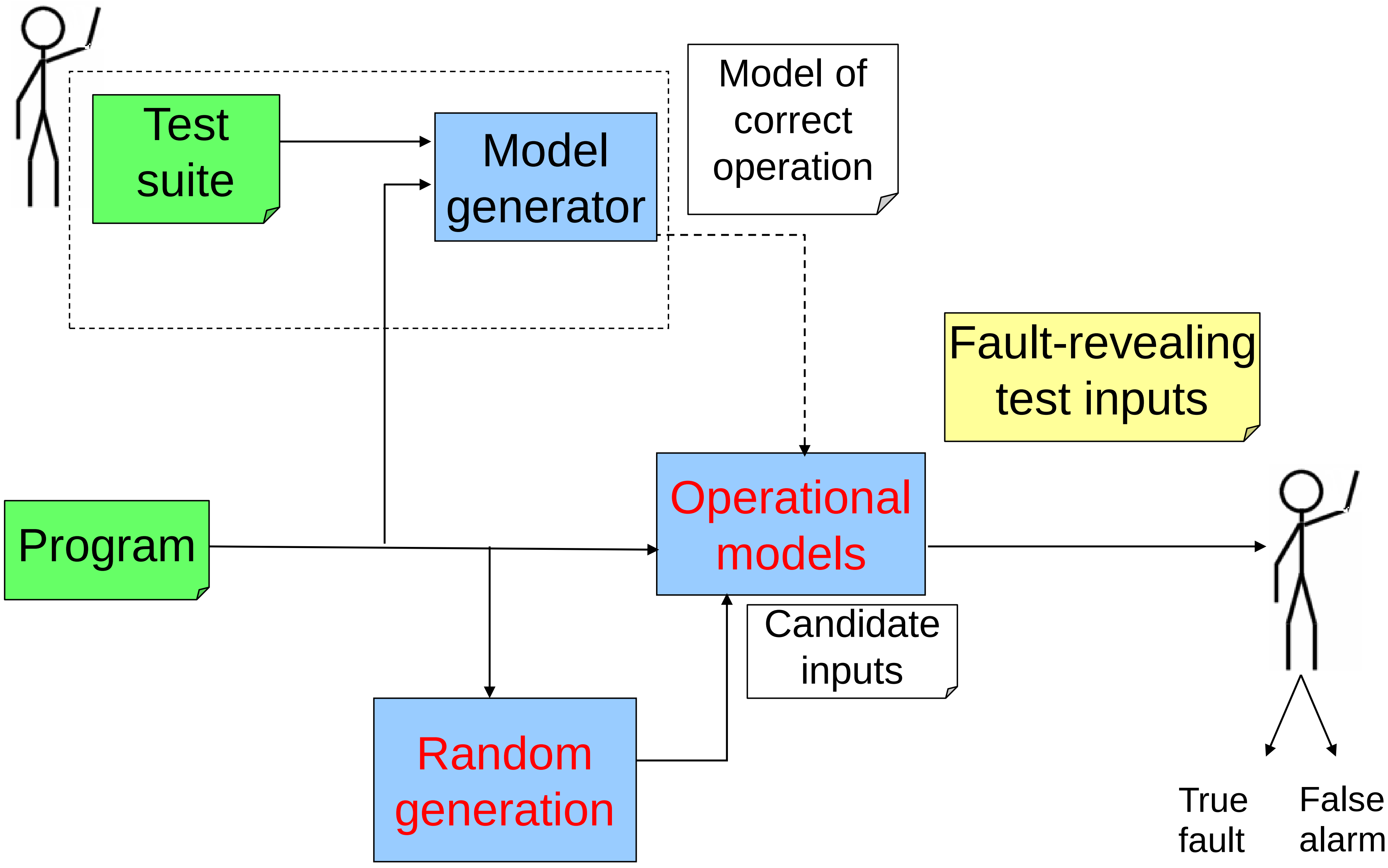
Random Generation

- Chooses sequence of methods at random
- Chooses arguments for methods at random

Instantiation 1: RanGen + UncEx



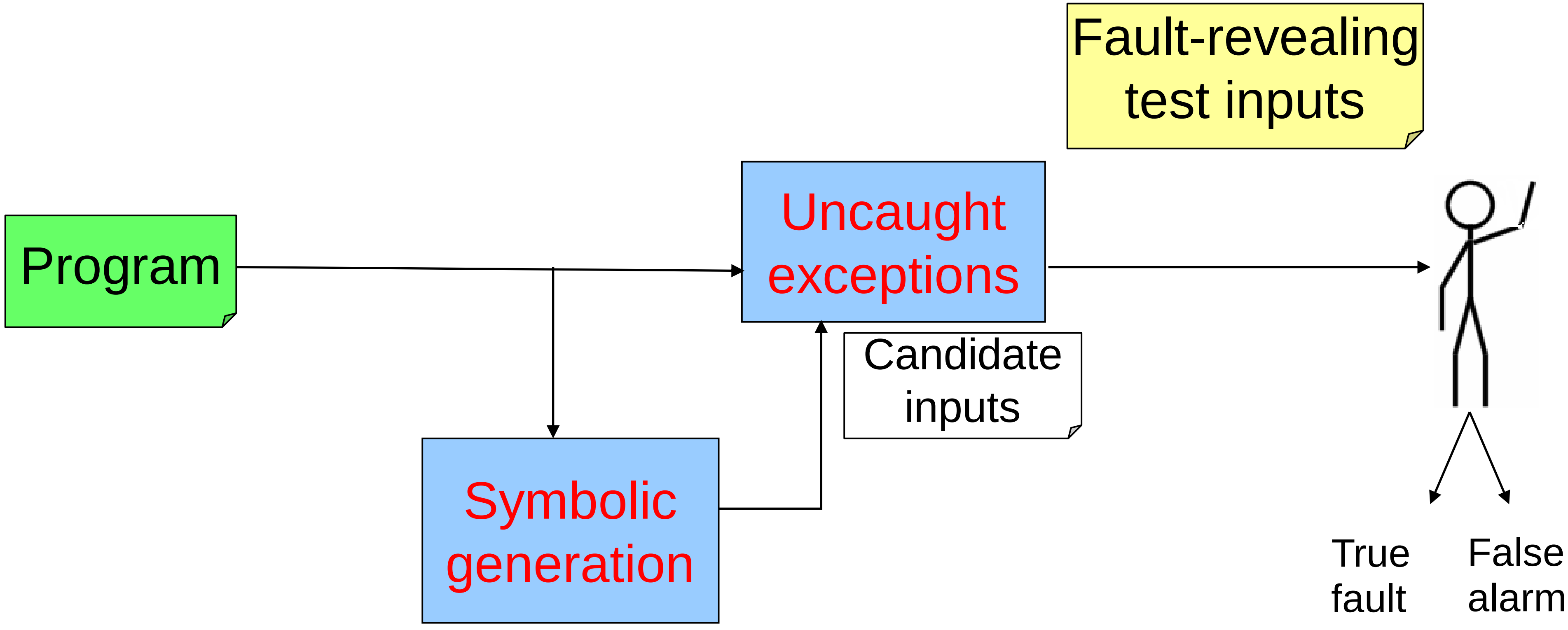
Instantiation 2: RanGen + OpMod



Symbolic Generation

- Symbolic execution
  - Executes methods with symbolic arguments
  - Collects constraints on these arguments
  - Solves constraints to produce concrete test inputs
- Previous work for OO unit testing
  [Xie et al., TACAS 2005]
  - Basics of symbolic execution for OO programs
  - Exploration of method sequences

Instantiation 3: SymGen + UncEx



Outline

- Motivation, background and problem
- Framework and existing techniques
- New technique
- Evaluation
- Conclusions

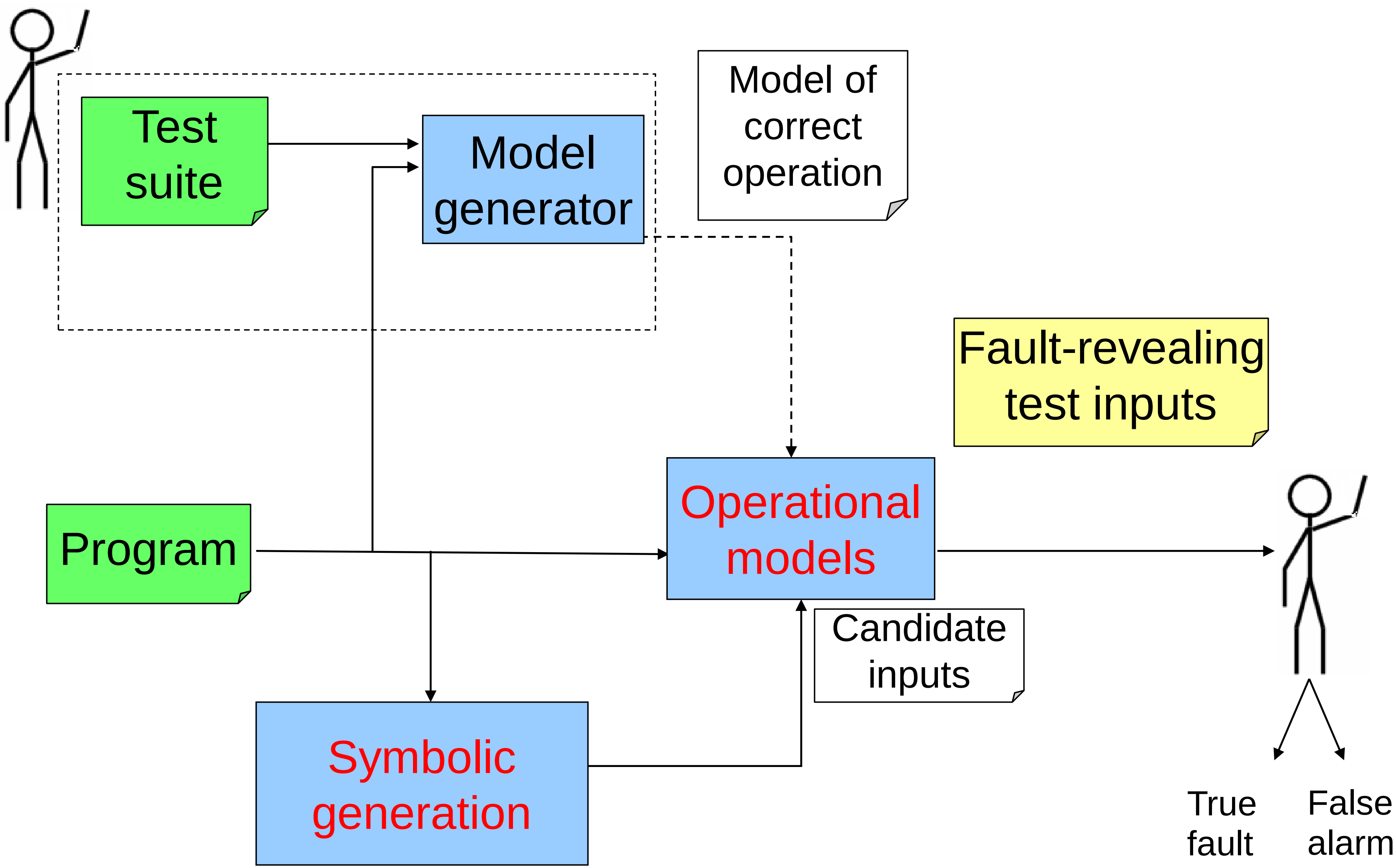
Proposed new technique

- Model-based Symbolic Testing (SymGen+OpMod)
  - Symbolic generation
  - Operational model classification
- Brief comparison with existing techniques
  - May explore failing method sequences that RanGen+OpMod misses
  - May find semantic faults that SymGen+UncEx misses

Contributions

- Extended symbolic execution
  - Operational models
  - Non-primitive arguments
- Implementation (Symclat)
  - Modified explicit-state model-checker Java Pathfinder [Visser et al., ASE 2000]

Instantiation 4: SymGen + OpMod



Outline

- Motivation, background and problem
- Framework and existing techniques
- New technique
- Evaluation
- Conclusions

Evaluation

- Comparison of four techniques

<table>
<tr><td colspan="2" rowspan="2"></td><th colspan="2">classification</th><th rowspan="2">Implementation tool</th></tr>
<tr><th>Uncaught exceptions</th><th>Operational models</th></tr>
<tr><th rowspan="2">generation</th><td>Random</td><td>RanGen+ UncEx</td><td>RanGen+ OpMod</td><td>Eclat [Pacheco and Ernst, 2005]</td></tr>
<tr><td>Symbolic</td><td>SymGen+ UncEx</td><td>SymGen+ OpMod</td><td>Symclat</td></tr>
</table>

Subjects

Source	Subject	NCNB LOC	#methods
UBStack [Csallner and Smaragdakis 2004, Xie and Notkin 2003, Stotts et al. 2002]	UBStack 8	88	11
	UBStack 12	88	11
Daikon [Ernst et al. 2001]	UtilMDE	1832	69
DataStructures [Weiss 99]	BinarySearchTree	186	9
	StackAr	90	8
	StackLi	88	9
JML samples [Cheon et al.2002]	IntegerSetAsHashSet	28	4
	Meter	21	3
	DLList	286	12
	E_OneWayList	171	10
	E_SLList	175	11
	OneWayList	88	12
	OneWayNode	65	10
	SLList	92	12
	TwoWayList	175	9
MIT 6.170 problem set [Pacheco and Ernst, 2005]	RatPoly (46 versions)	582.51	17.20

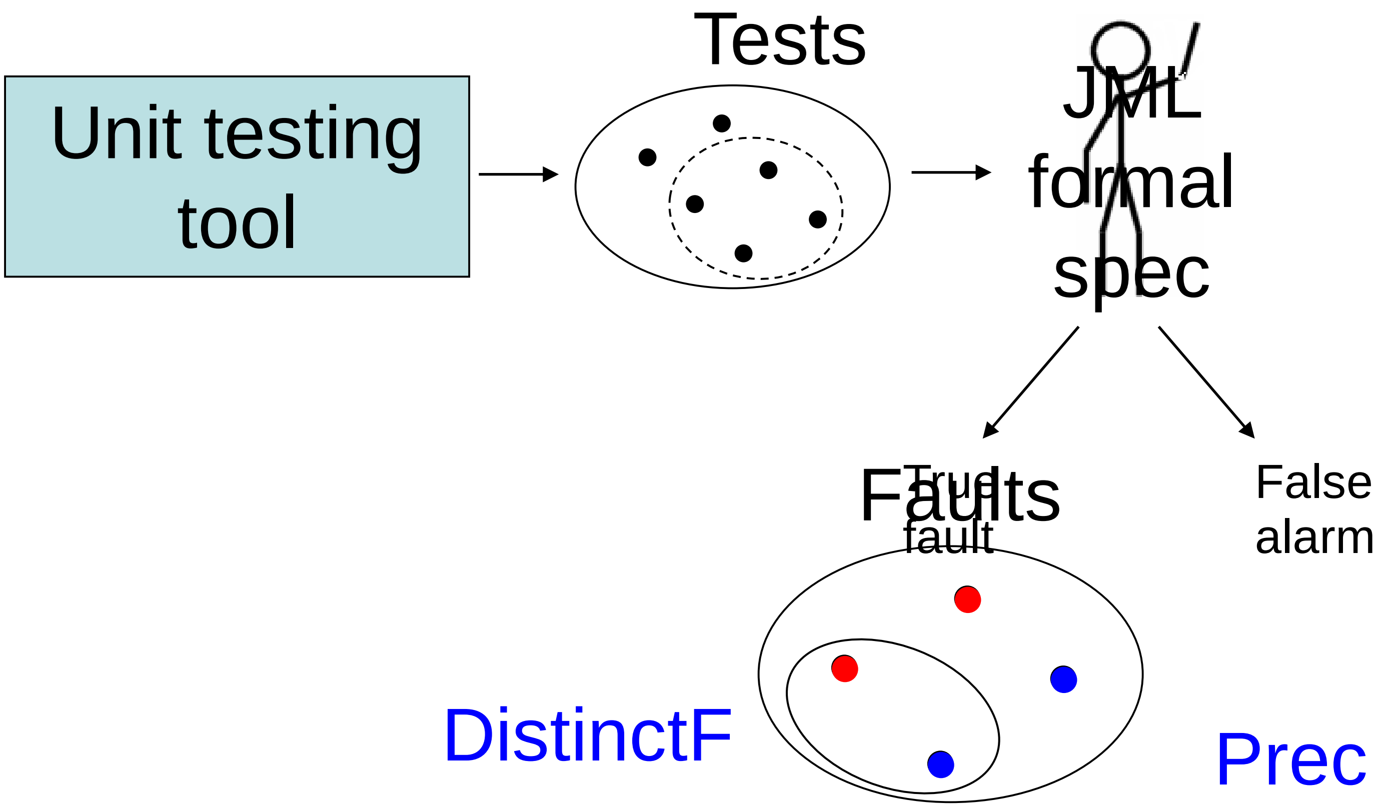
Experimental setup

- Eclat (RanGen) and Symclat (SymGen) tools
  - With UncEx and OpMod classifications
  - With and without reduction
- Each tool run for about the same time (2 min. on Intel Xeon 2.8GHz, 2GB RAM)
- For RanGen, Eclat runs each experiment with 10 different seeds

Comparison metrics

- Compare effectiveness of various techniques in finding faults
- Each run gives to user a set of test inputs
  - Tests: Number of test inputs given to user
- Metrics
  - Faults: Number of actually fault-revealing test inputs
  - DistinctF: Number of distinct faults found
  - Prec = Faults/Tests: Precision, ratio of generated test inputs revealing actual faults

Evaluation procedure



Prec = Faults/Tests

Summary of results

- All techniques miss faults and report false positives
- Techniques are complementary
- RanGen is sensitive to seeds
- Reduction can increase precision but decreases number of distinct faults

False positives and negatives

- Generation techniques can miss faults
  - RanGen can miss important sequences or input values
  - SymGen can miss important sequences or be unable to solve constraints
- Classification techniques can miss faults and report false alarms due to imprecise models
  - Misclassify test inputs (normal as fault-revealing or fault-revealing as normal)

Results without reduction

of test inputs given to the user

# of actual fault-revealing tests generated

	RanGen+ UncEx	RanGen+ OpMod	SymGen+ UncEx	SymGen+ OpMod
Tests	4,367.5	1,666.6	6,676	4,828
Faults	256.0	181.2	515	164
DistinctF	17.7	13.1	14	9
Prec	0.20	0.42	0.15	0.14

distinct actual faults

precision = Faults / Tests

Results with reduction

	RanGen+ UncEx	RanGen+ OpMod	SymGen+ UncEx	SymGen+ OpMod
Tests	124.4	56.2	106	46
Faults	22.8	13.4	11	7
DistinctF	15.3	11.6	11	7
Prec	0.31	0.51	0.17	0.20

- DistinctF ↓ and Prec ↑
  - Reduction misses faults: may remove a true fault and keep false alarm
  - Redundancy of tests decreases precision

Sensitivity to random seeds

- For one RatPoly implementation

	RanGen+ UncEx	RanGen+ OpMod
Tests	17.1	20
Faults	0.2	0.8
DistinctF	0.2	0.5
Prec	0.01	0.04

- RanGen+OpMod (with reduction)
  - 200 tests for 10 seeds 8 revealing faults
  - For only 5 seeds there is (at least) one test that reveals fault

Outline

- Motivation, background and problem
- Framework and existing techniques
- New technique
- Evaluation
- Conclusions

Key: Complementary techniques

- Each technique finds some fault that other techniques miss
- Suggestions
  - Try several techniques on the same subject
    - Evaluate how merging independently generated sets of test inputs affects Faults, DistinctF, and Prec
    - Evaluate other techniques (e.g., RanGen+SymGen [Godefroid et al. 2005, Cadar and Engler 2005, Sen et al. 2005])
  - Improve RanGen
    - Bias selection (What methods and values to favor?)
    - Run with multiple seeds (Merging of test inputs?)

Conclusions

- Proposed a new technique: Model-based Symbolic Testing
- Compared four techniques that combine
  - Random vs. symbolic generation
  - Uncaught exception vs. operational models classification
- Techniques are complementary
- Proposed improvements for techniques