Metamorphic Testing of Developer Tools

Michael Pradel

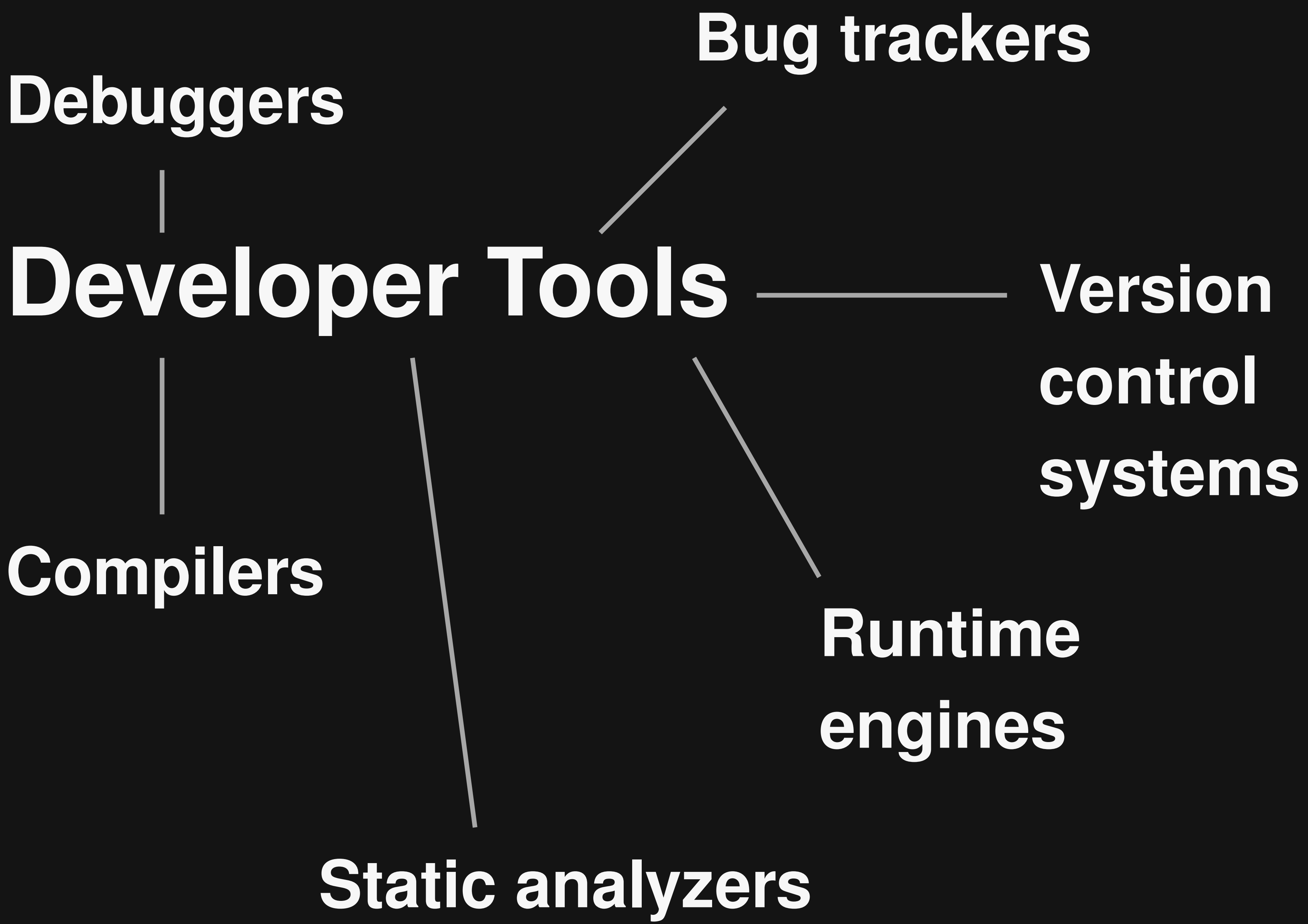
Software Lab – University of Stuttgart

Joint work with Daniel Lehmann, Matteo Paltenghi, and Sandro Tolksdorf

SOLA SoftwareLab University of Stuttgart Germany

European Research Council

Developer Tools



Developer Tools

Debuggers

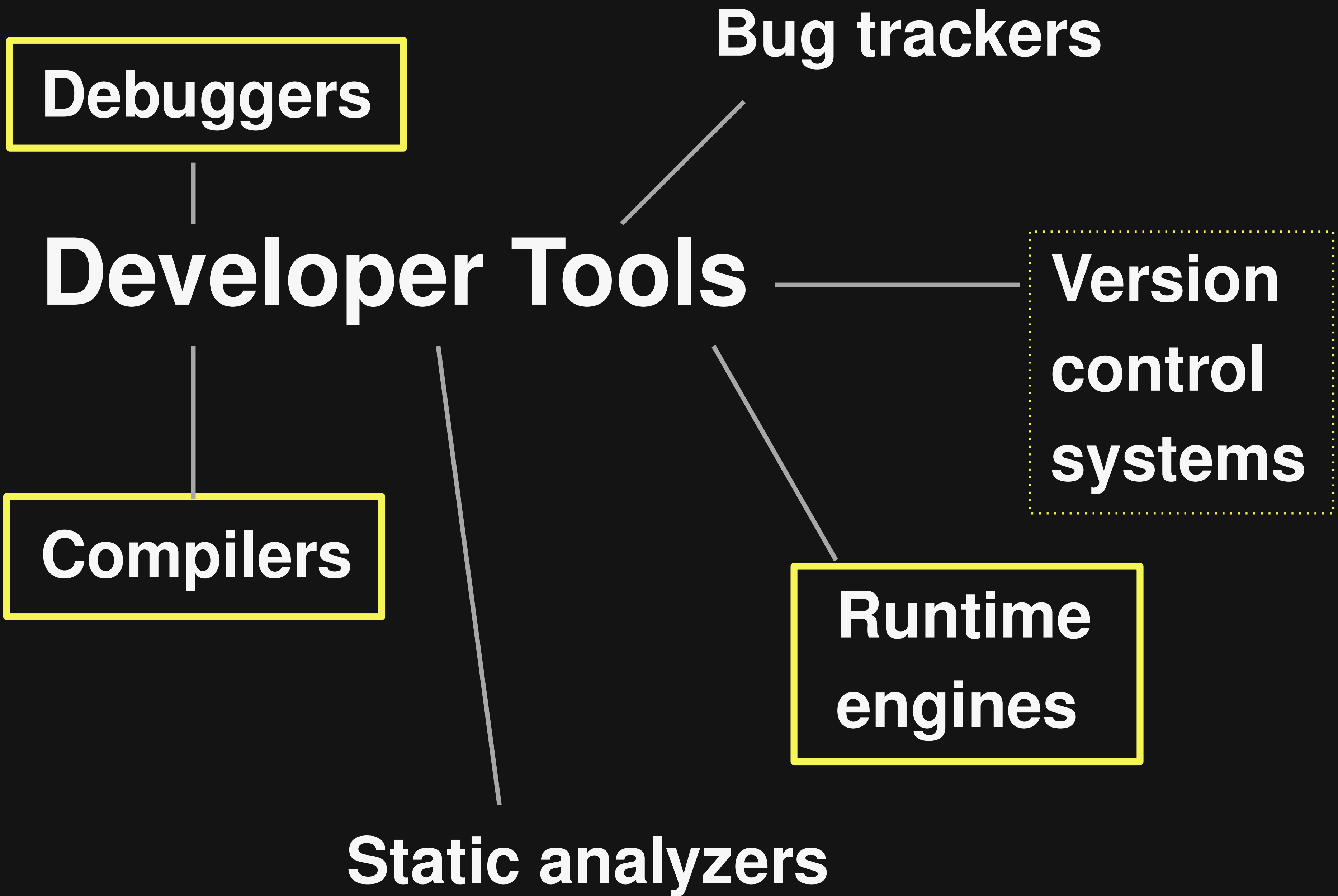
Bug trackers

Version control systems

Runtime engines

Static analyzers

Compilers



Developer Tools

Debuggers

Bug trackers

Version control systems

Compilers

Runtime engines

Static analyzers

Metamorphic Testing of Developer Tools

Michael Pradel

Software Lab – University of Stuttgart

Joint work with Daniel Lehmann, Matteo Paltenghi, and Sandro Tolksdorf

Why Testing of Developer Tools ?

Why Testing of Developer Tools ?

Foundation of successful software engineering

Buggy tools cause

- Misbehaving programs
- Confused developers

Metamorphic Testing of Developer Tools

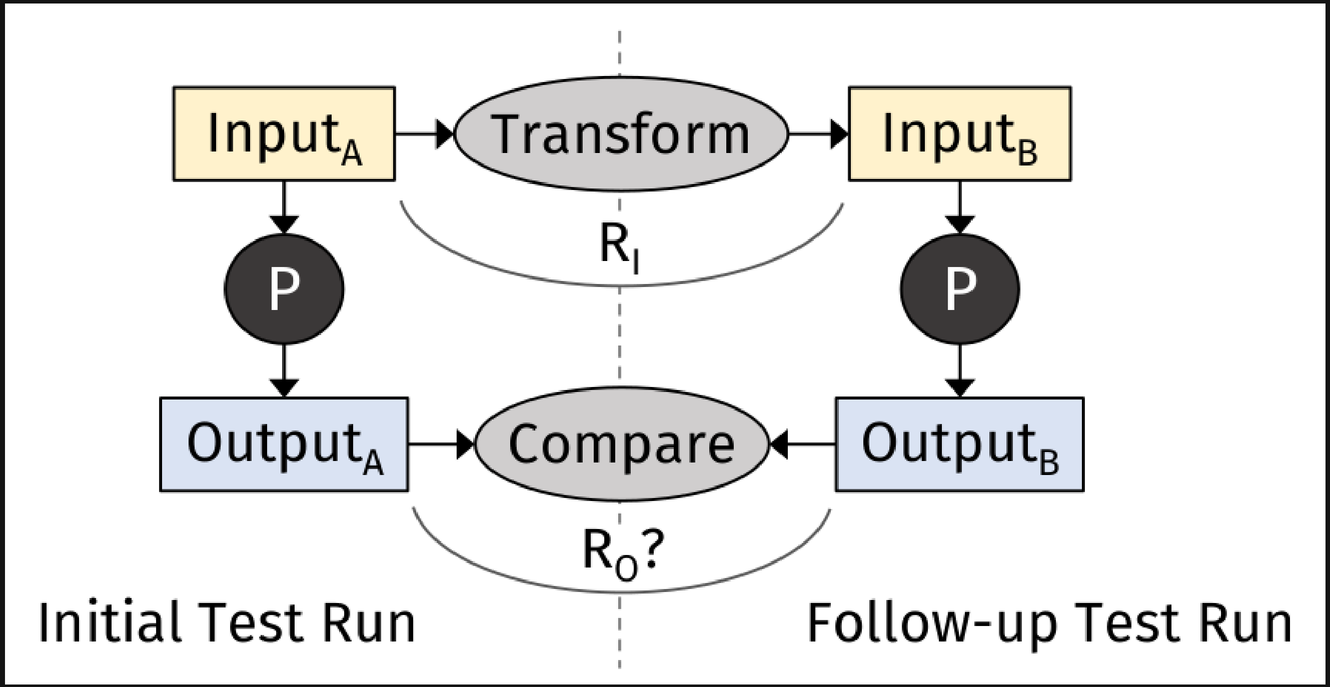
Michael Pradel

Software Lab – University of Stuttgart

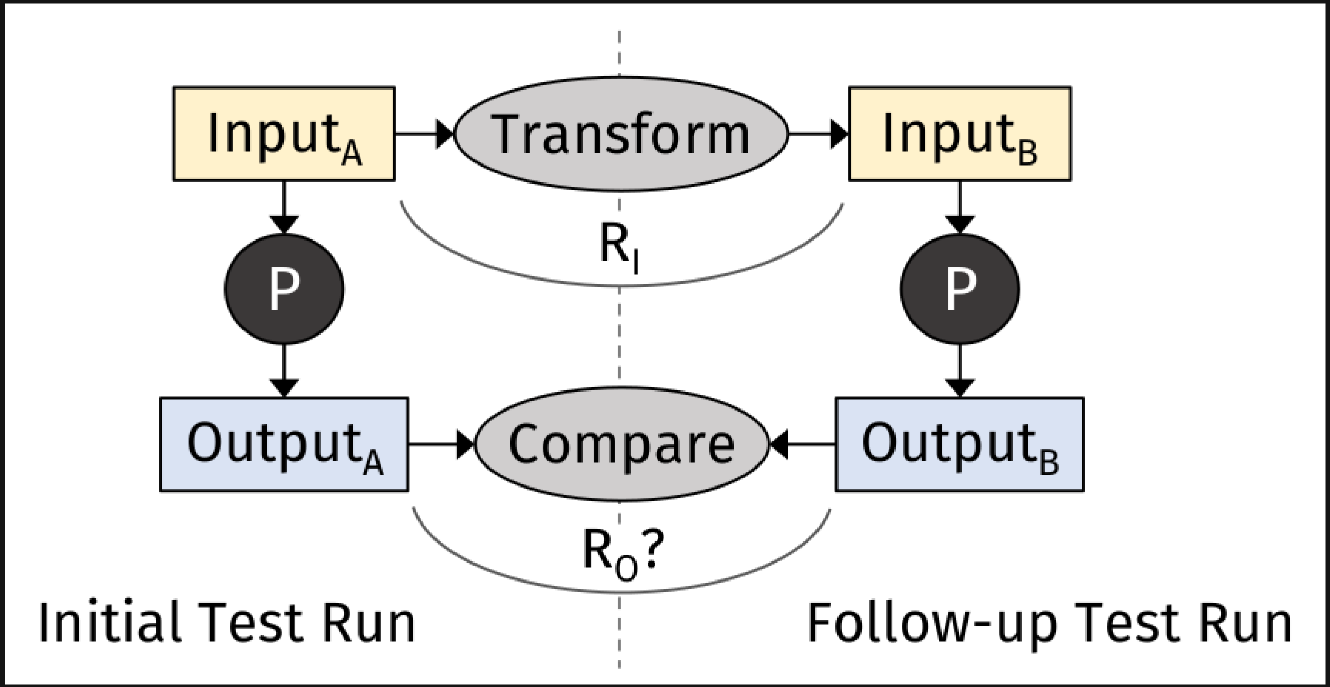
Joint work with Daniel Lehmann, Matteo Paltenghi, and Sandro Tolksdorf

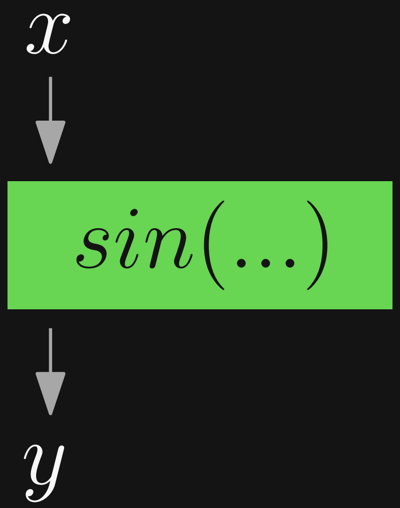
Metamorphic Testing

Metamorphic Testing



Metamorphic Testing

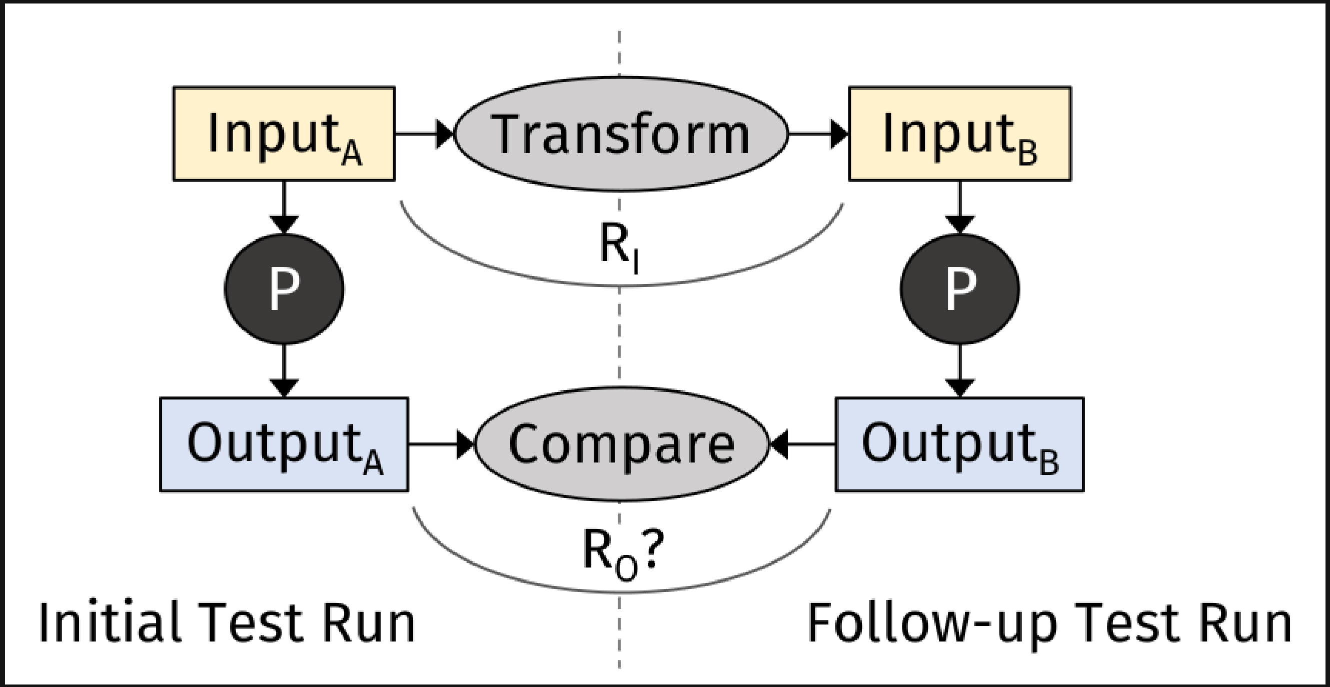


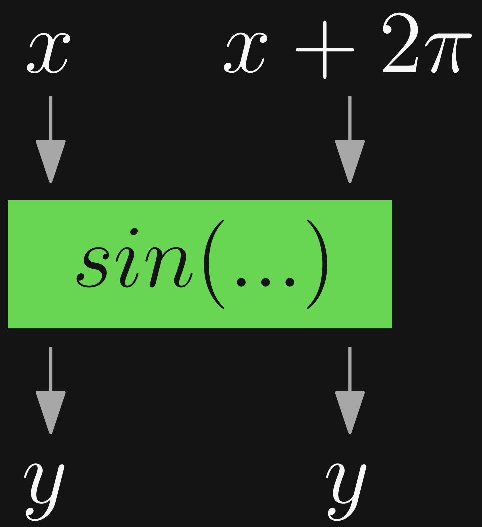

x

$sin(...)$

y

Metamorphic Testing





Why use

Metamorphic Testing ?

Why use Metamorphic Testing ?

General answer:

Addresses oracle problem

Specific to developer tools:

- Inputs (e.g., programs) have well-defined semantics
- Can design metamorphic transformations on top

This Talk

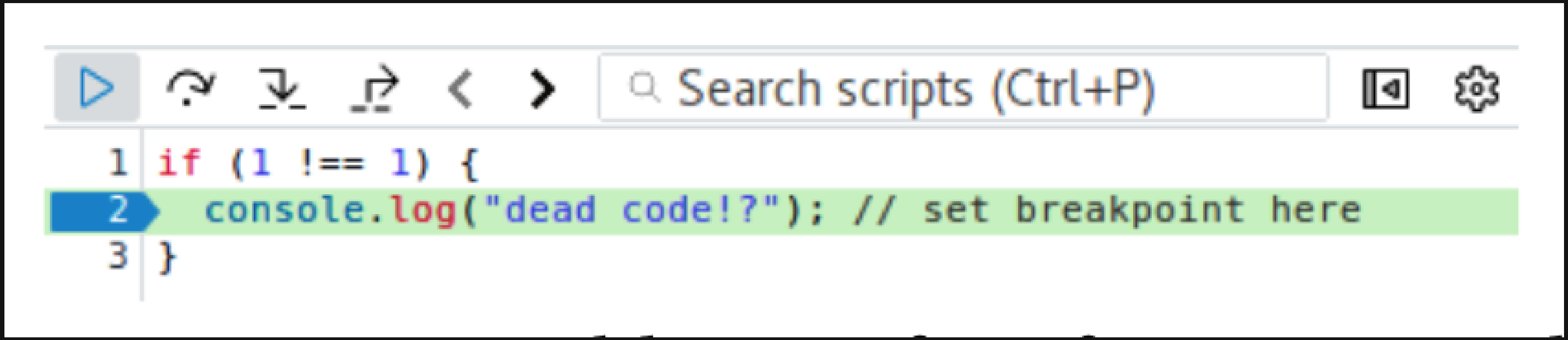
- Interactive Metamorphic Testing of Debuggers [ISSTA’19] ←
- MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform

  [ICSE’23]
- Lessons learned and open challenges

  [ICSE’24, ’25, etc. ?]

Motivating Example

Debugger pauses at a breakpoint in dead code:



```
if (1 !== 1) {
  console.log("dead code!?"); // set breakpoint here
}
```

Firefox bug # 1370648

Testing of Debuggers

- **Inputs**
  - Program-to-debug
  - Sequence of actions (e.g., set breakpoint)
- **Output**
  - Debugging trace (e.g., pausing, program state)

Goal & Challenges

Goal:

Automatically test interactive debuggers

Challenges:

- Complex input
- No well-defined oracle
- Interactive nature of debuggers

Goal & Challenges

Goal:

Automatically test interactive debuggers

Challenges:

- Complex input

Debugging actions depend on program

- No well-defined oracle
- Interactive nature of debuggers

Goal & Challenges

Goal:

Automatically test interactive debuggers

Challenges:

- Complex input
- No well-defined oracle — **Pause at a breakpoint on a comment line?**
- Interactive nature of debuggers

Goal & Challenges

Goal:

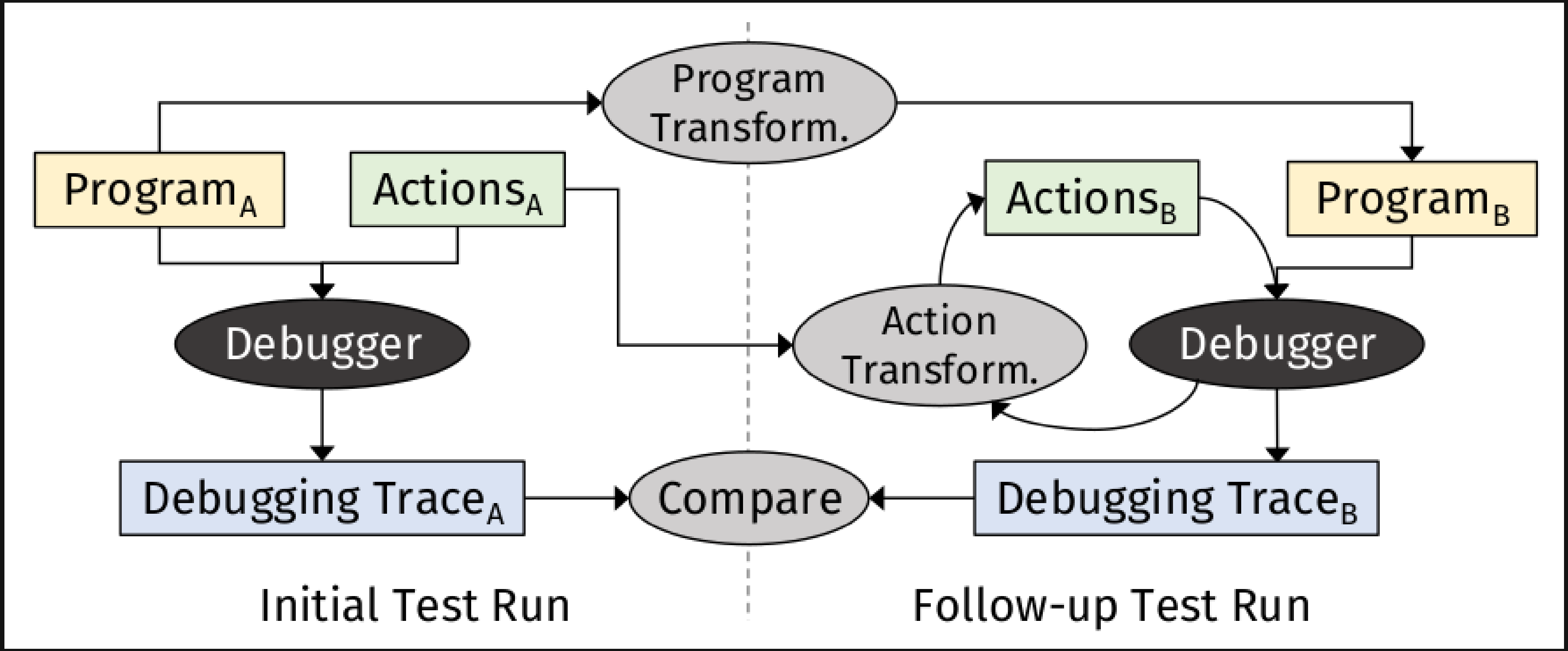
Automatically test interactive debuggers

Challenges:

- Complex input
- No well-defined oracle
- Interactive nature of debuggers

└── **Expected semantics of debugging actions become clear only when program executes**

Overview


Program Transform.
Program$_A$
Actions$_A$
Actions$_B$
Program$_B$
Debugger
Action Transform.
Debugger
Debugging Trace$_A$
Compare
Debugging Trace$_B$
Initial Test Run
Follow-up Test Run

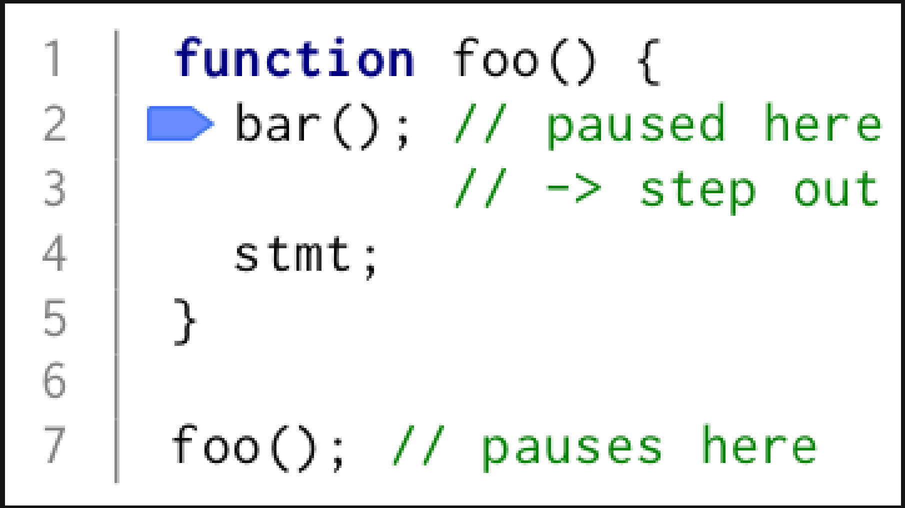
Action Transformations

- Add breakpoint and continue
- Replace continue by step
- Breakpoint sliding

Action Transformations

- Add breakpoint and continue
- Replace continue by step
- Breakpoint sliding

Adding a breakpoint at line l should cause only additional pauses at l

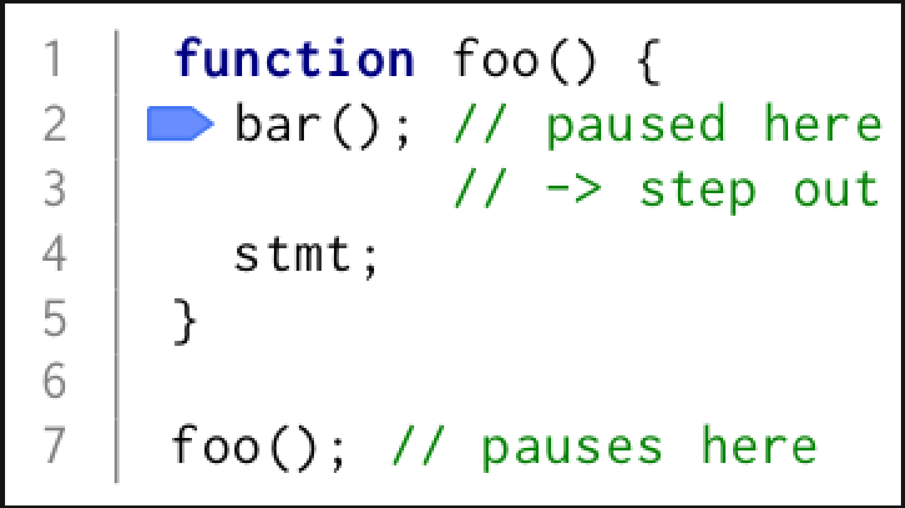


```
function foo() {
  bar(); // paused here
         // -> step out
  stmt;
}

foo(); // pauses here
```

Action Transformations

- Add breakpoint and continue
- Replace continue by step
- Breakpoint sliding

Adding a breakpoint at line l should cause only additional pauses at l



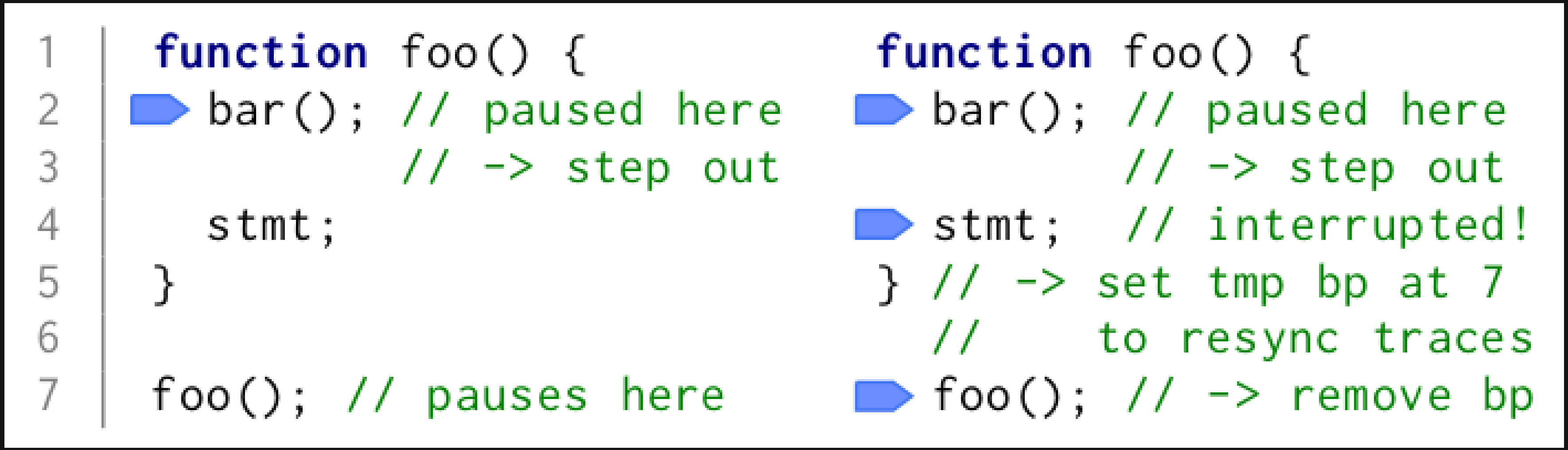
```
function foo() {
  bar(); // paused here
         // -> step out
  stmt;
}

foo(); // pauses here
```

New breakpoint
$\Rightarrow$ Should pause

Action Transformations

- Add breakpoint and continue
- Replace continue by step
- Breakpoint sliding

Adding a breakpoint at line l should cause only additional pauses at l

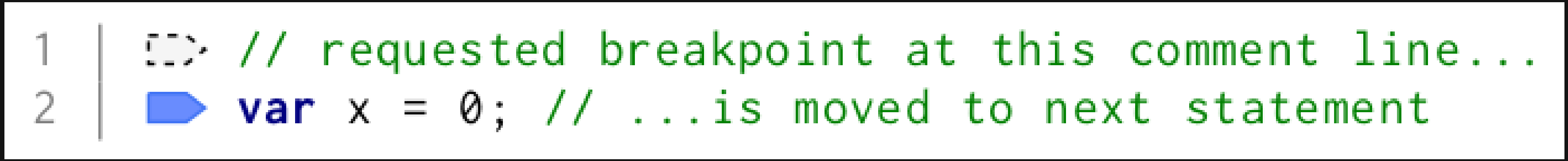


```
1   function foo() {                  function foo() {
2  ▶ bar(); // paused here           ▶ bar(); // paused here
3           // -> step out                    // -> step out
4    stmt;                           ▶ stmt;  // interrupted!
5   }                                 } // -> set tmp bp at 7
6                                       //    to resync traces
7   foo(); // pauses here            ▶ foo(); // -> remove bp
```

Action Transformations

- Add breakpoint and continue
- Replace continue by step
- Breakpoint sliding

Setting breakpoint at l, which slides to l', should be equal to directly setting it at l'



```
// requested breakpoint at this comment line...
var x = 0; // ...is moved to next statement
```

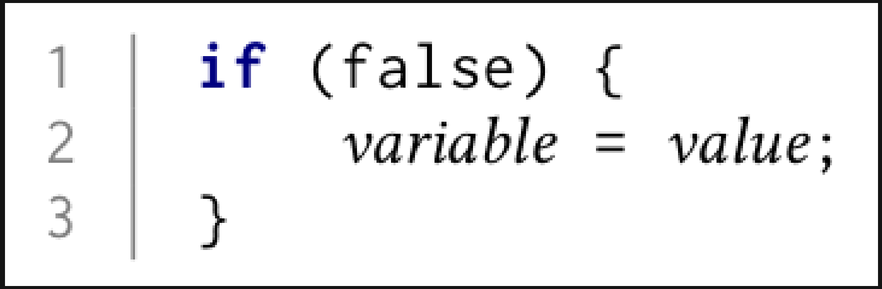
Program Transformations

- Insert or remove dead code
- Add parameter
- Add no-op
- Replace literal with expression

Program Transformations

- Insert or remove dead code
- Add parameter
- Add no-op
- Replace literal with expression

Should have no influence except changed line numbers

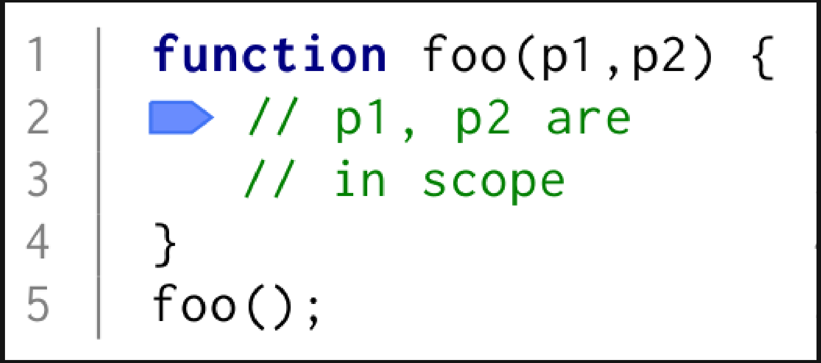


```
if (false) {
    variable = value;
}
```

Program Transformations

- Insert or remove dead code
- Add parameter
- Add no-op
- Replace literal with expression

Should show additional variable in program state

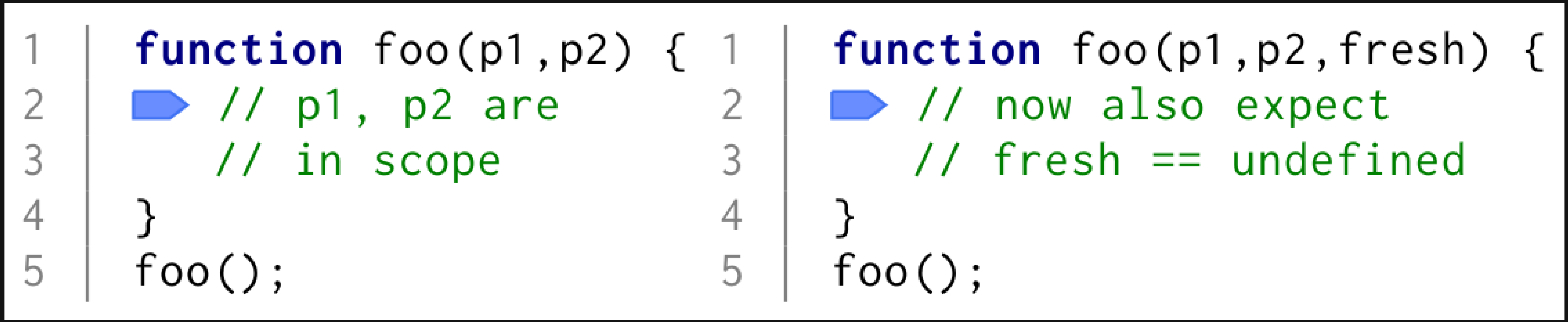


```
function foo(p1,p2) {
    // p1, p2 are
    // in scope
}
foo();
```

Program Transformations

- Insert or remove dead code
- Add parameter
- Add no-op
- Replace literal with expression

Should show additional variable in program state



```
function foo(p1,p2) {
   // p1, p2 are
   // in scope
}
foo();
```

```
function foo(p1,p2,fresh) {
   // now also expect
   // fresh == undefined
}
foo();
```

Interactive Metamorphic Testing

Traditional metamorphic testing:

- Apply transformations without executing the program

Here:

- Need to execute to know which transformations are applicable

Interactive Metamorphic Testing

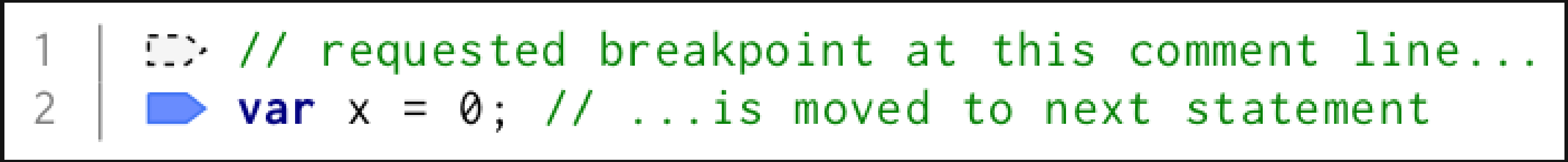
Traditional metamorphic testing:

- Apply transformations without executing the program

Here:

- Need to execute to know which transformations are applicable

E.g., knowing what line a breakpoint slides to

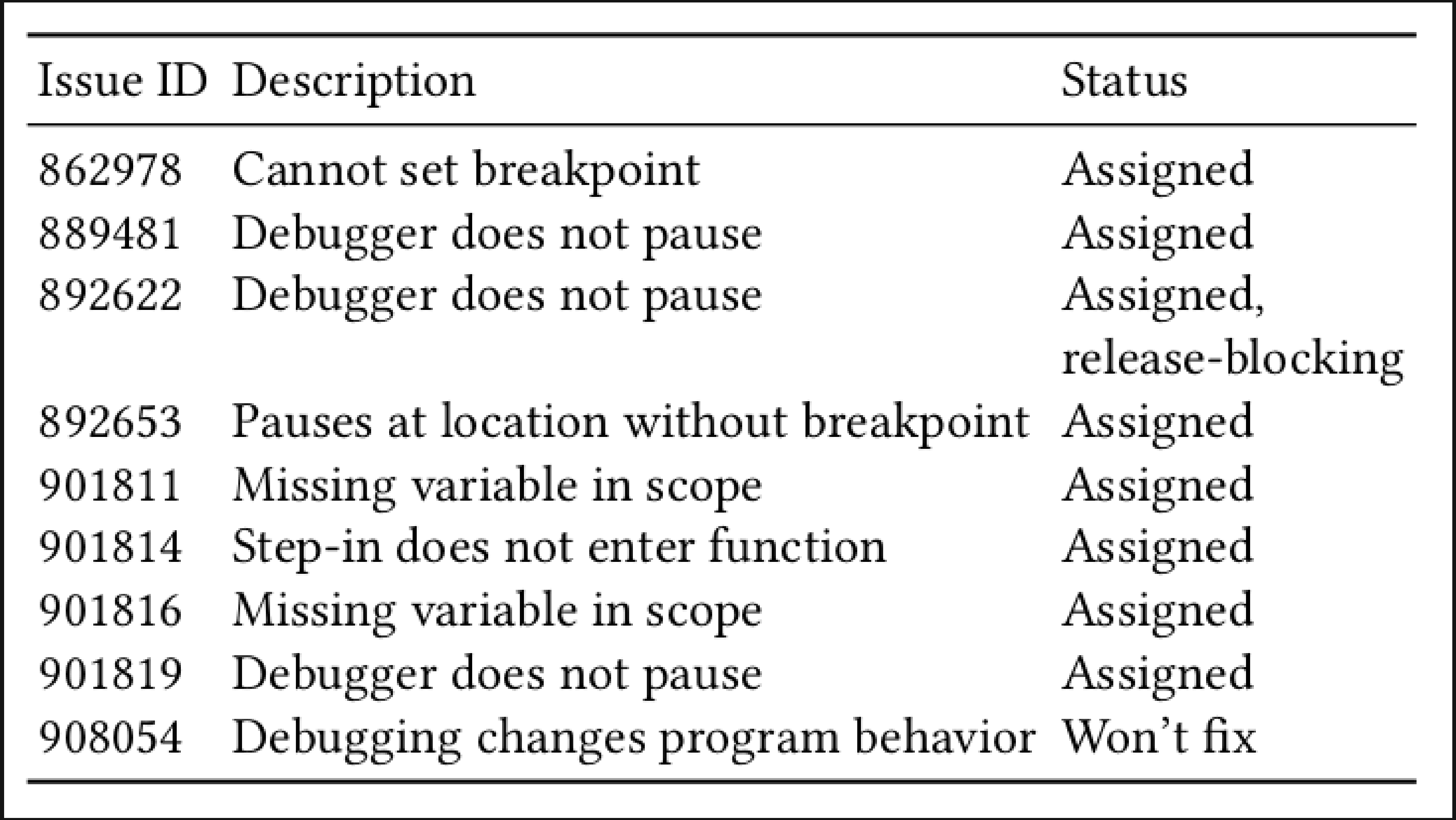


```
// requested breakpoint at this comment line...
var x = 0; // ...is moved to next statement
```

Evaluation

- **Target: JavaScript debugger of Chromium**
- **47k JavaScript programs**
  - Initial debugging actions: Randomly created by DBDB [FSE'18]
  - One follow-up input for each program

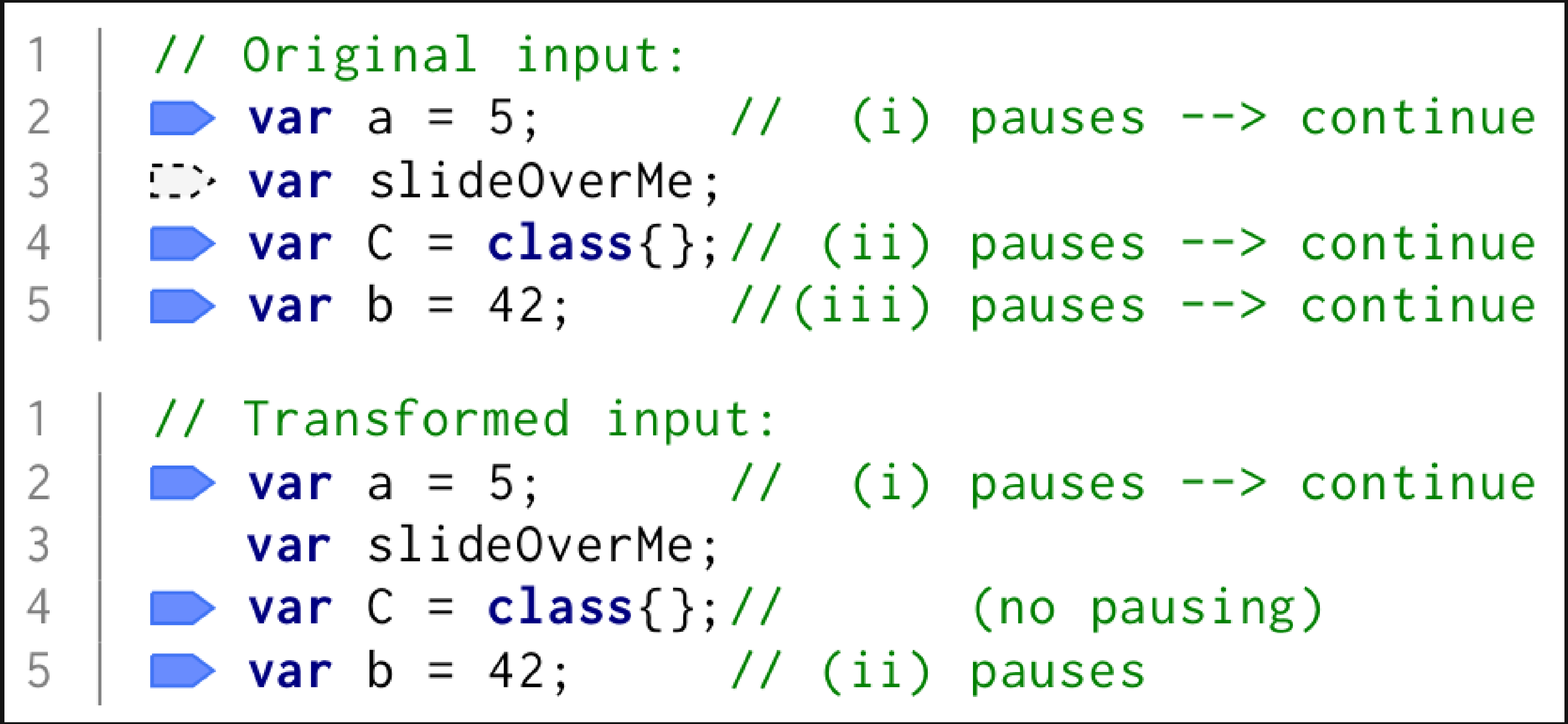
Effectiveness



Issue ID	Description	Status
862978	Cannot set breakpoint	Assigned
889481	Debugger does not pause	Assigned
892622	Debugger does not pause	Assigned, release-blocking
892653	Pauses at location without breakpoint	Assigned
901811	Missing variable in scope	Assigned
901814	Step-in does not enter function	Assigned
901816	Missing variable in scope	Assigned
901819	Debugger does not pause	Assigned
908054	Debugging changes program behavior	Won’t fix

Examples

Fails to stop at breakpoint:



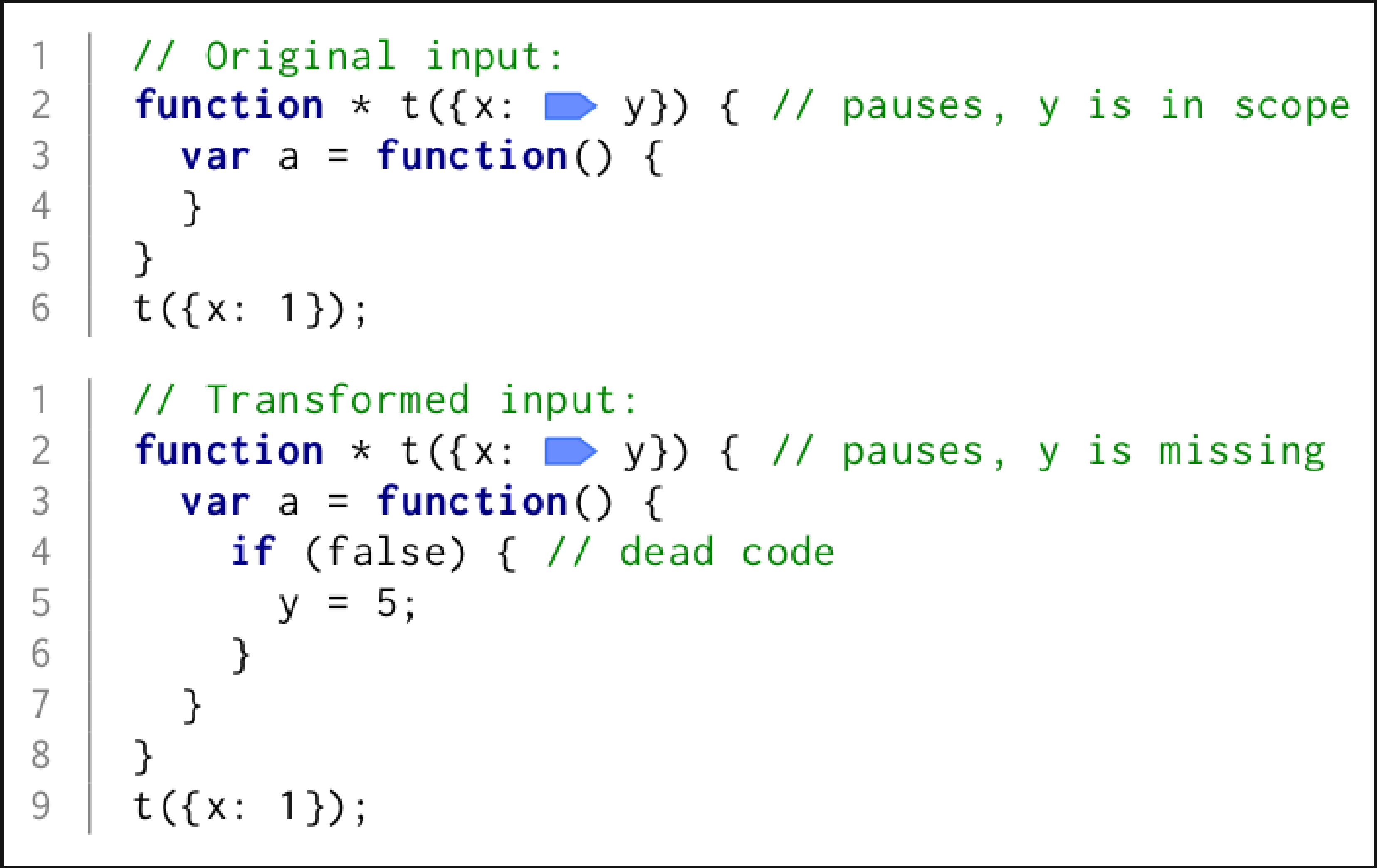
```
// Original input:
var a = 5;      //  (i) pauses --> continue
var slideOverMe;
var C = class{};// (ii) pauses --> continue
var b = 42;     //(iii) pauses --> continue
```

```
// Transformed input:
var a = 5;      //  (i) pauses --> continue
var slideOverMe;
var C = class{};//      (no pausing)
var b = 42;     // (ii) pauses
```

Chromium bug #889481

Examples

Incorrect program state:



```
// Original input:
function * t({x: ▶ y}) { // pauses, y is in scope
  var a = function() {
  }
}
t({x: 1});
```

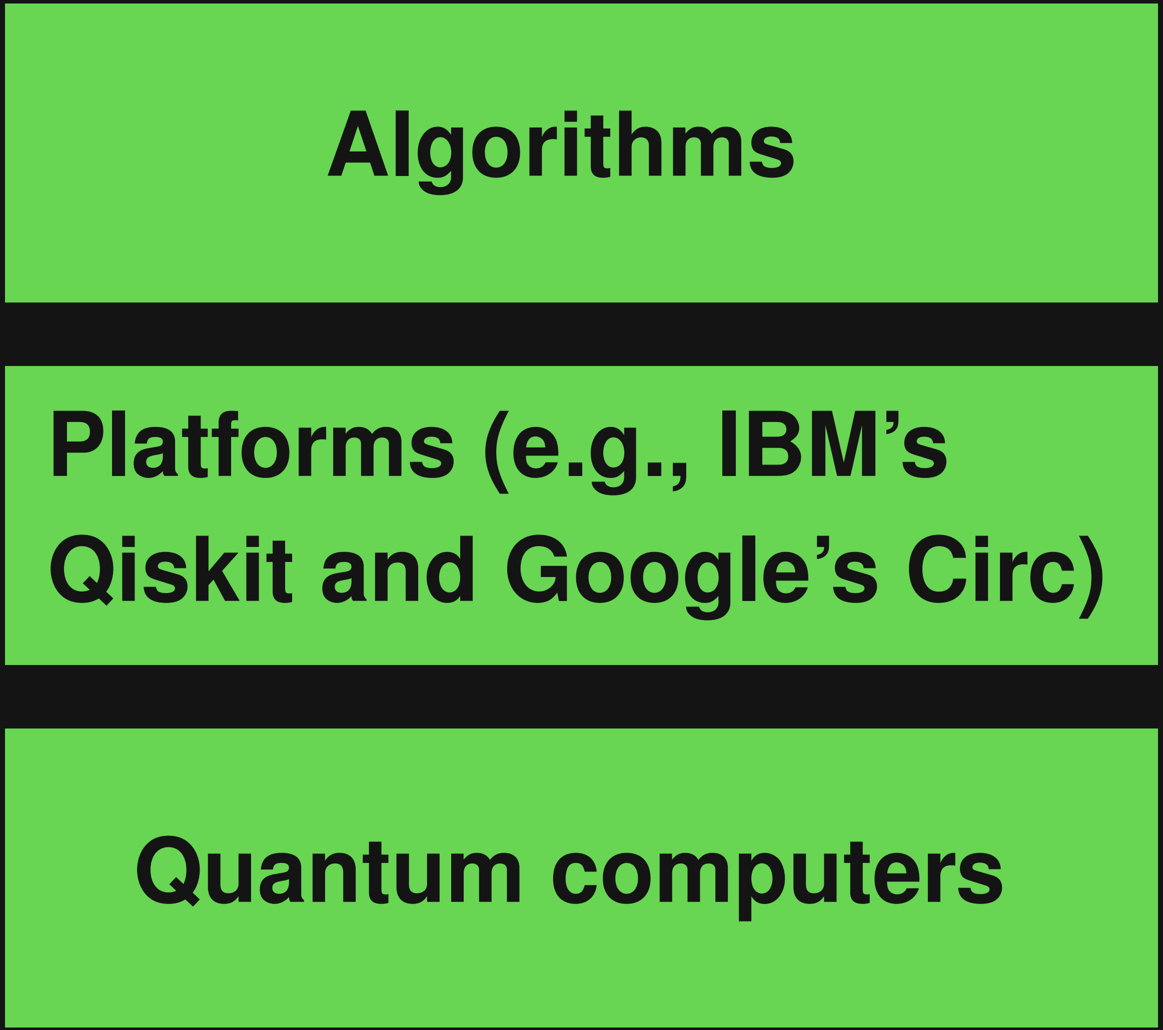
```
// Transformed input:
function * t({x: ▶ y}) { // pauses, y is missing
  var a = function() {
    if (false) { // dead code
      y = 5;
    }
  }
}
t({x: 1});
```

Chromium bug #901811

This Talk

- Interactive Metamorphic Testing of Debuggers [ISSTA’19]
- MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform [ICSE’23] ←
- Lessons learned and open challenges [ICSE’24, ’25, etc. ?]

Quantum Computing Stack

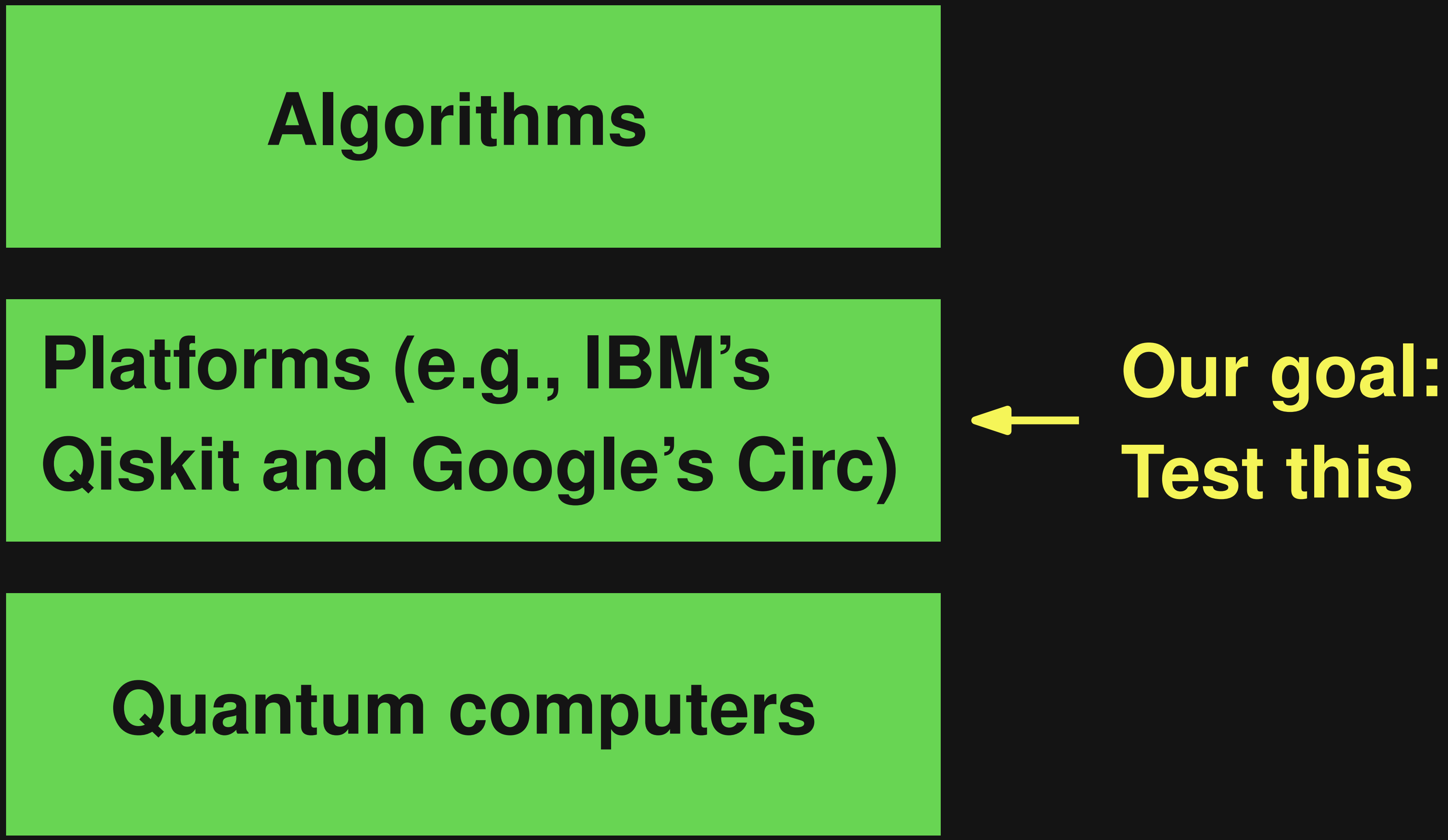


Algorithms

Platforms (e.g., IBM's Qiskit and Google's Circ)

Quantum computers

Quantum Computing Stack



Algorithms

Platforms (e.g., IBM's Qiskit and Google's Circ)

Quantum computers

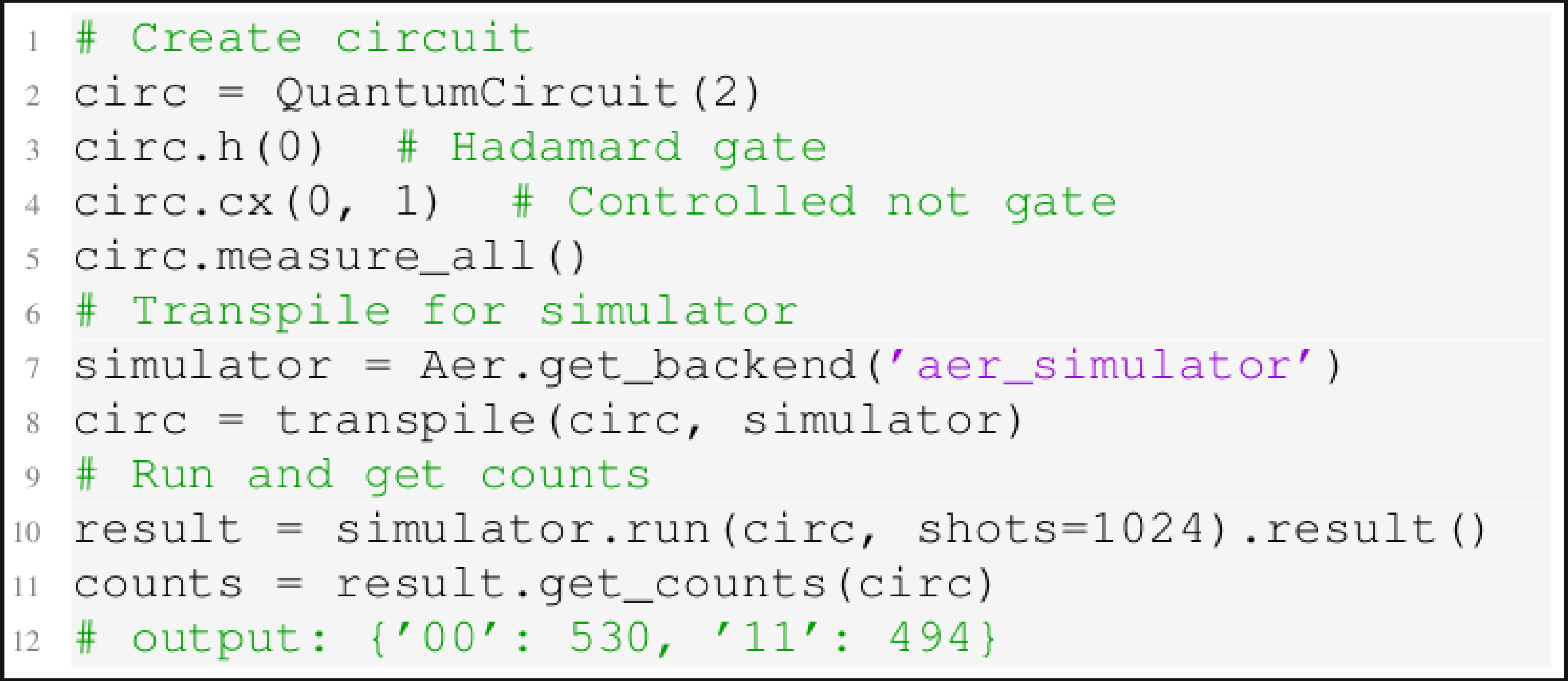
Our goal: Test this

Why Relevant?

- Quantum computing: Emerging field with huge investments
- Reliable platforms are crucial
- Novel, quantum-specific bug patterns

[OOPSLA’22]

Background: Quantum Software

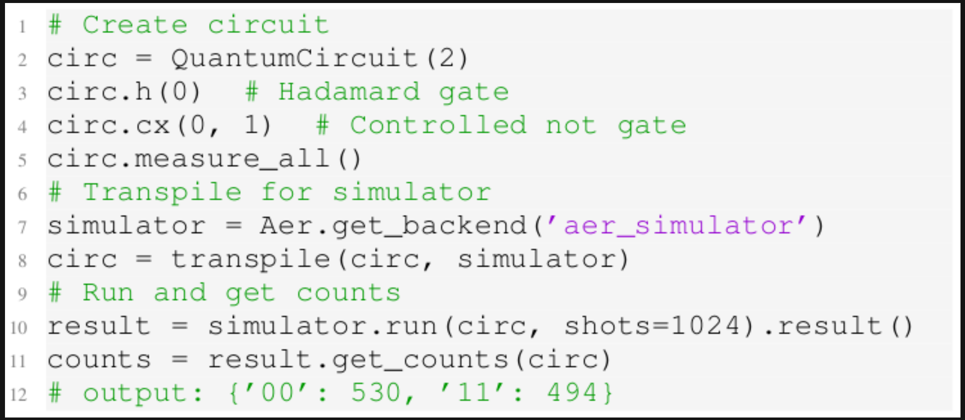


```
# Create circuit
circ = QuantumCircuit(2)
circ.h(0)   # Hadamard gate
circ.cx(0, 1)   # Controlled not gate
circ.measure_all()
# Transpile for simulator
simulator = Aer.get_backend(’aer_simulator’)
circ = transpile(circ, simulator)
# Run and get counts
result = simulator.run(circ, shots=1024).result()
counts = result.get_counts(circ)
# output: {’00’: 530, ’11’: 494}
```

Quantum algorithm (in Qiskit):

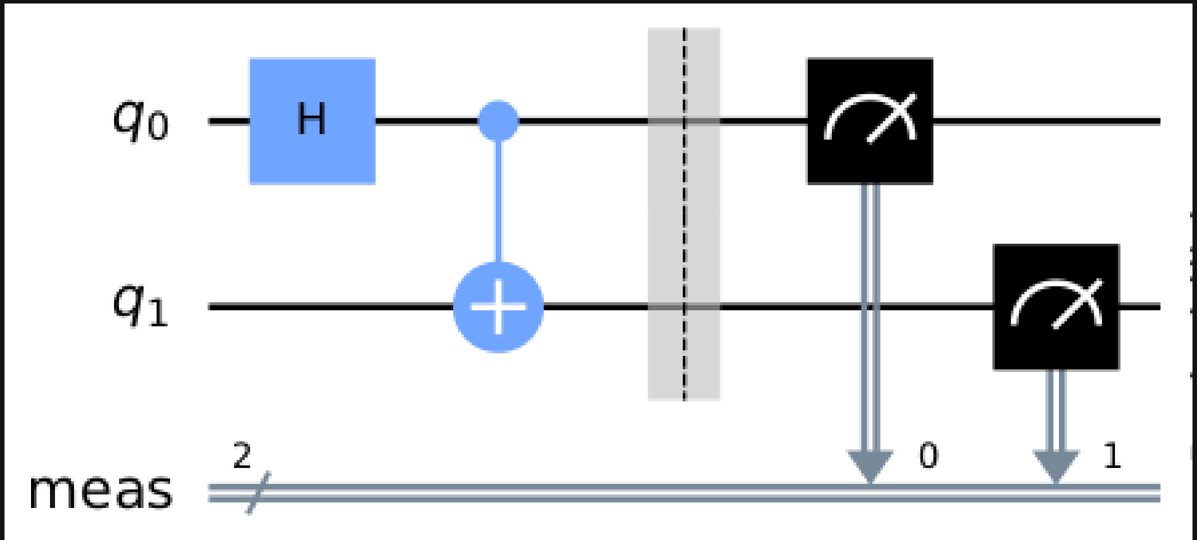
Python program

Background: Quantum Software

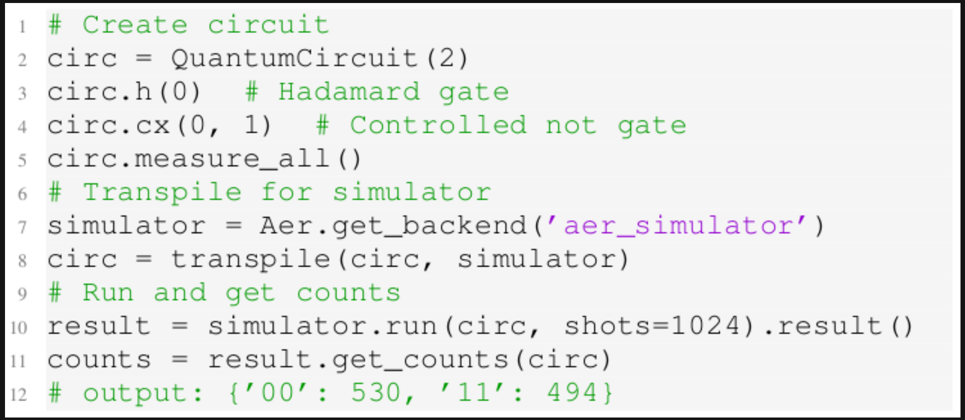


```
# Create circuit
circ = QuantumCircuit(2)
circ.h(0)  # Hadamard gate
circ.cx(0, 1)  # Controlled not gate
circ.measure_all()
# Transpile for simulator
simulator = Aer.get_backend(’aer_simulator’)
circ = transpile(circ, simulator)
# Run and get counts
result = simulator.run(circ, shots=1024).result()
counts = result.get_counts(circ)
# output: {’00’: 530, ’11’: 494}
```

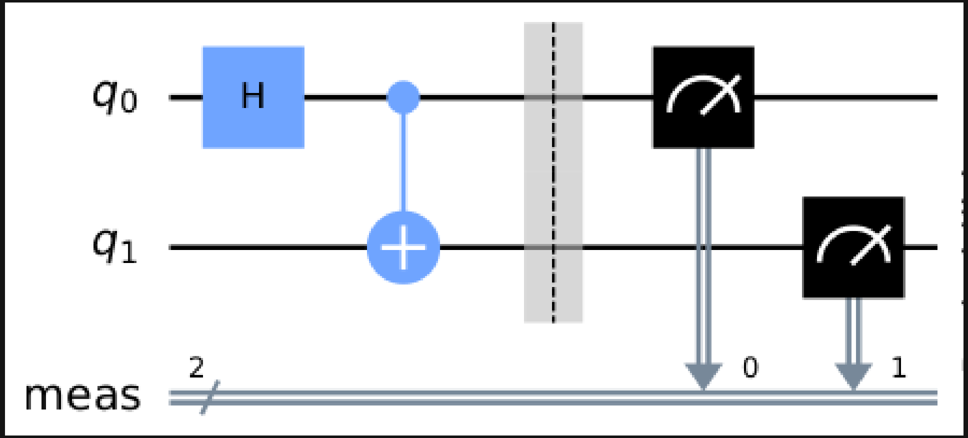
Visual representation



Background: Quantum Software

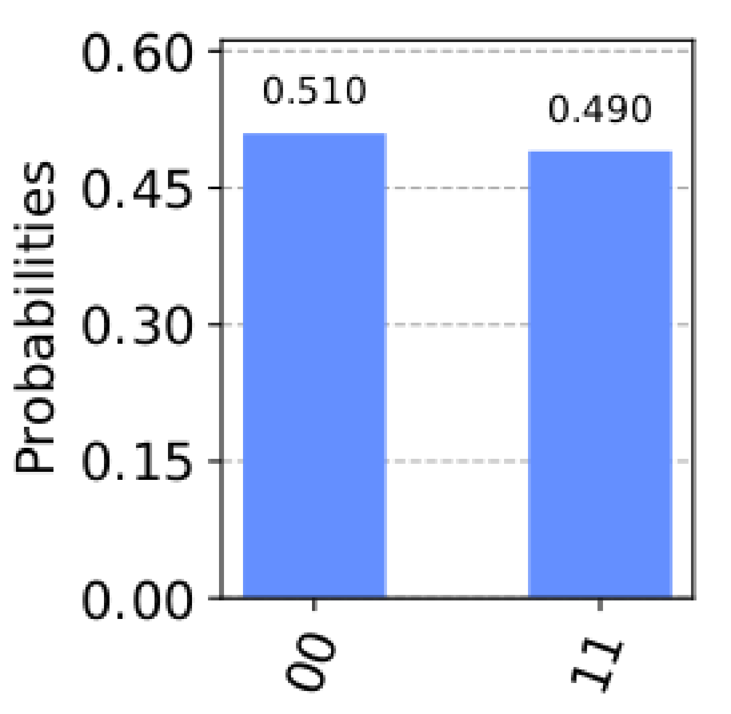


```
# Create circuit
circ = QuantumCircuit(2)
circ.h(0)   # Hadamard gate
circ.cx(0, 1)   # Controlled not gate
circ.measure_all()
# Transpile for simulator
simulator = Aer.get_backend('aer_simulator')
circ = transpile(circ, simulator)
# Run and get counts
result = simulator.run(circ, shots=1024).result()
counts = result.get_counts(circ)
# output: {'00': 530, '11': 494}
```



Output:

Probability distribution



Goal & Challenges

Goal: **Automatically test quantum computing platforms**

Challenges:

- Relatively few quantum programs
- No well-defined oracle
- Unreliable and difficult-to-access hardware

Goal & Challenges

Goal: **Automatically test quantum computing platforms**

Challenges:

New and emerging domain

- Relatively few quantum programs
- No well-defined oracle
- Unreliable and difficult-to-access hardware

Goal & Challenges

Goal: **Automatically test quantum computing platforms**

Challenges:

- Relatively few quantum programs
- No well-defined oracle
- Unreliable and difficult-to-access hardware

Low-level operations with sometimes counterintuitive semantics

Goal & Challenges

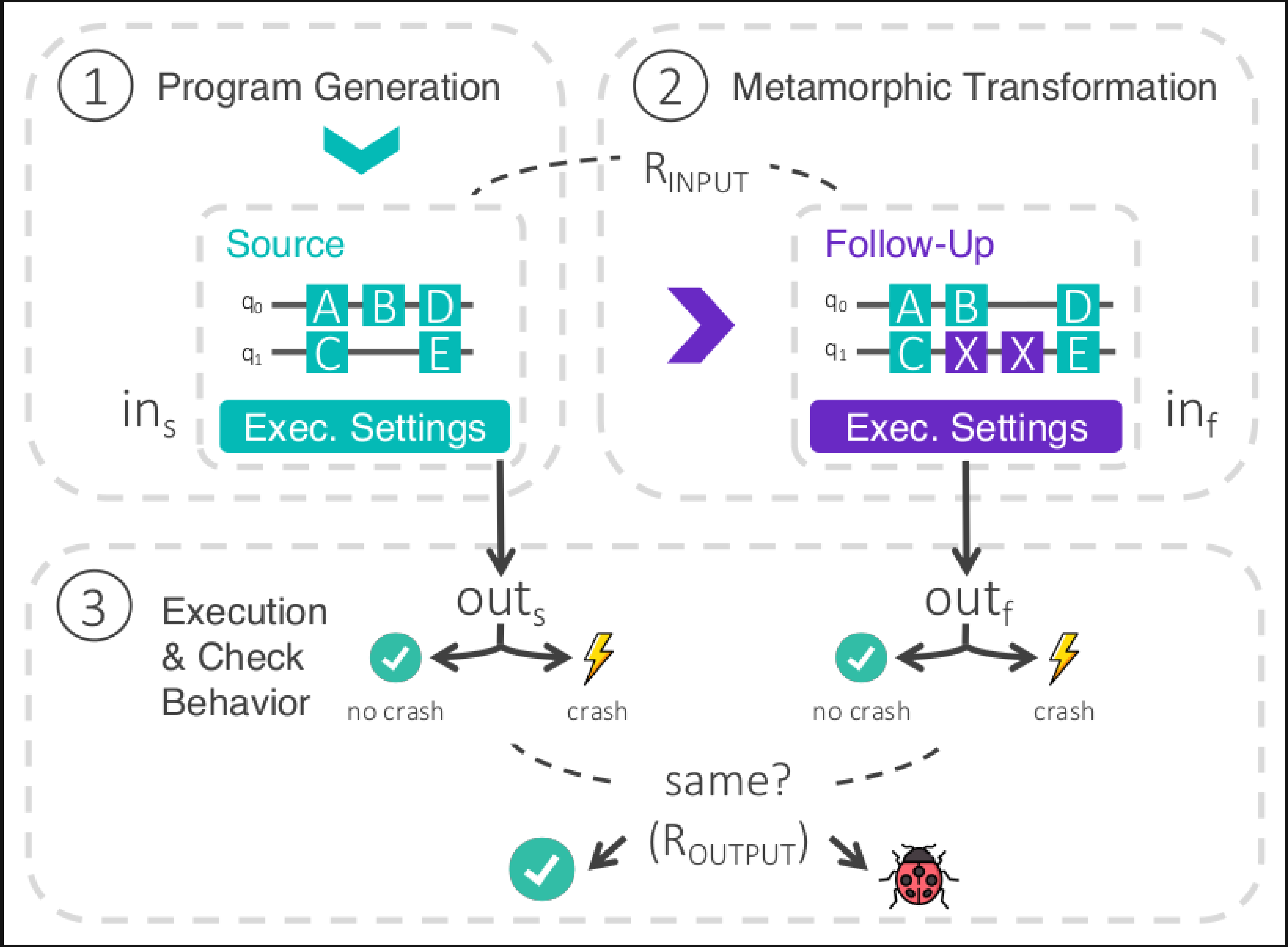
Goal: **Automatically test quantum computing platforms**

Challenges:

- Relatively few quantum programs
- No well-defined oracle
- Unreliable and difficult-to-access hardware

Quantum noise induced by stray electromagnetic fields or material defects

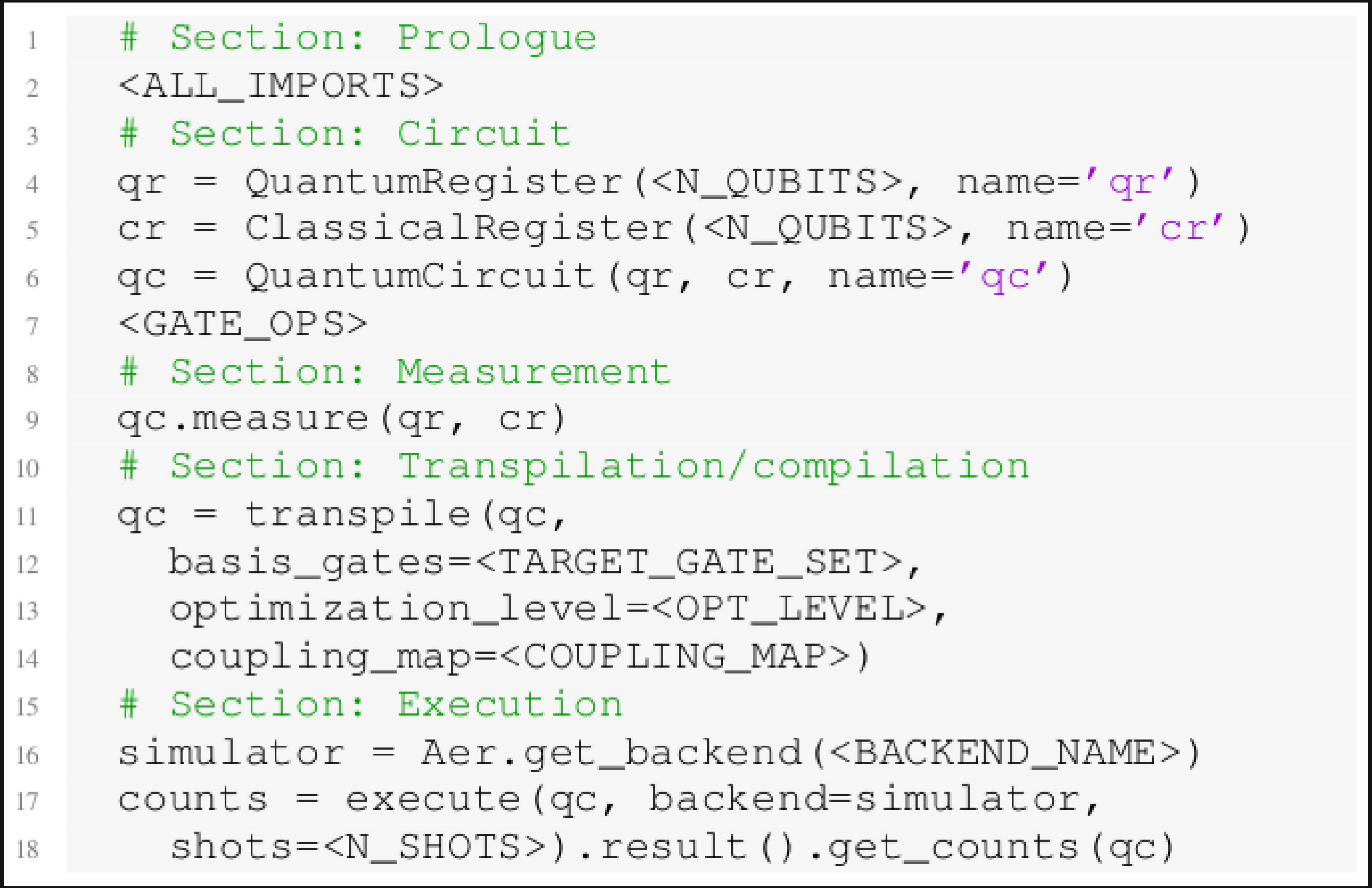
Overview of MorphQ



Generating Programs

- Template- and grammar-based, randomized algorithm
- Guarantee: Produces non-crashing program

Generating Programs



```
# Section: Prologue
<ALL_IMPORTS>
# Section: Circuit
qr = QuantumRegister(<N_QUBITS>, name='qr')
cr = ClassicalRegister(<N_QUBITS>, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
<GATE_OPS>
# Section: Measurement
qc.measure(qr, cr)
# Section: Transpilation/compilation
qc = transpile(qc,
  basis_gates=<TARGET_GATE_SET>,
  optimization_level=<OPT_LEVEL>,
  coupling_map=<COUPLING_MAP>)
# Section: Execution
simulator = Aer.get_backend(<BACKEND_NAME>)
counts = execute(qc, backend=simulator,
  shots=<N_SHOTS>).result().get_counts(qc)
```

Metamorphic Transformations

1) Circuit transformations

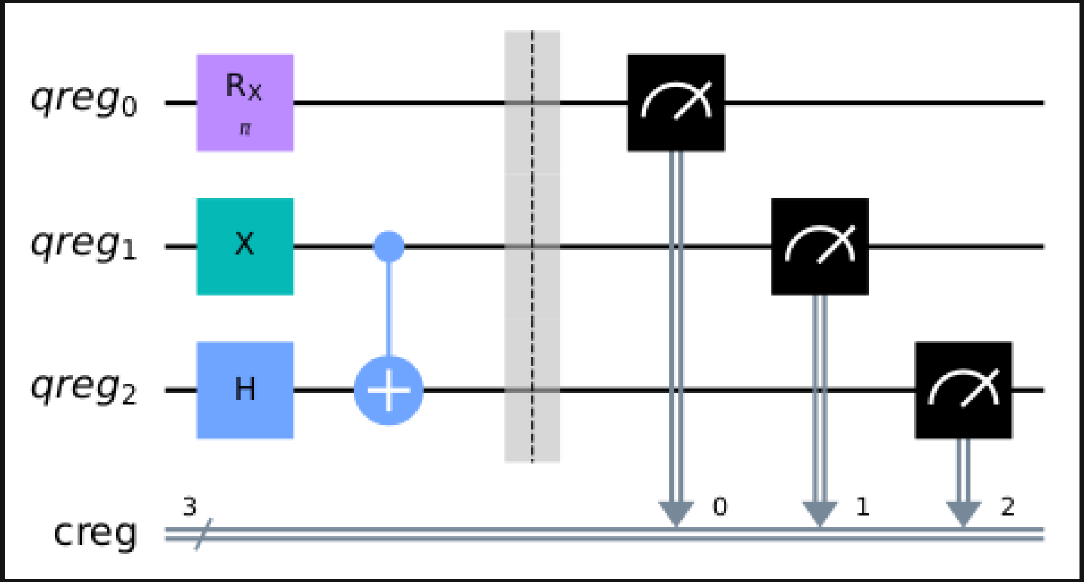
- Change qubit order
- Inject null-effect operation
- Add quantum register
- Inject parameters
- Partitioned execution

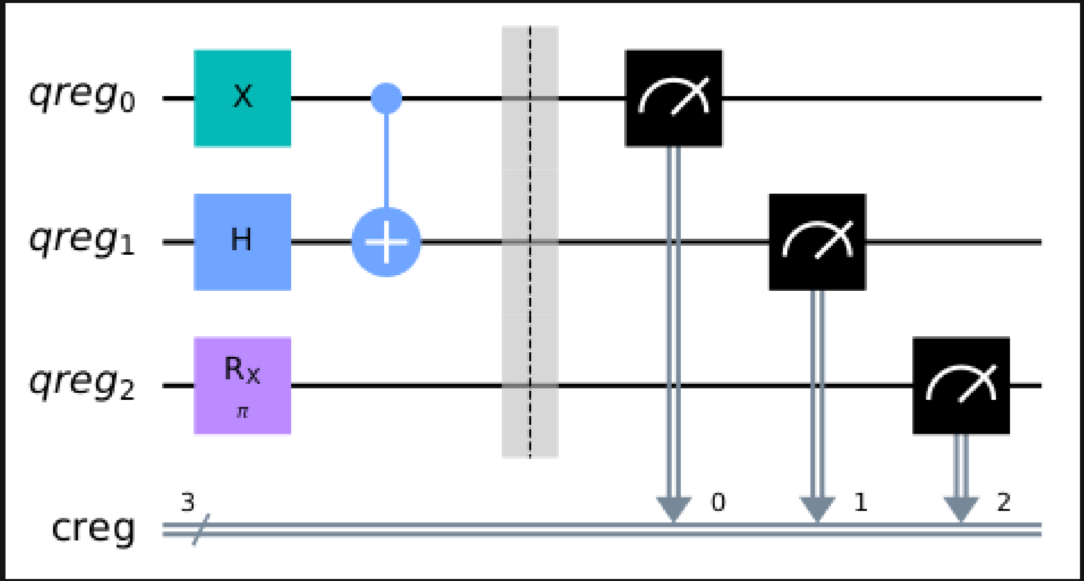
Metamorphic Transformations

1) Circuit transformations

- Change qubit order
- Inject null-effect operation
- Add quantum register
- Inject parameters
- Partitioned execution

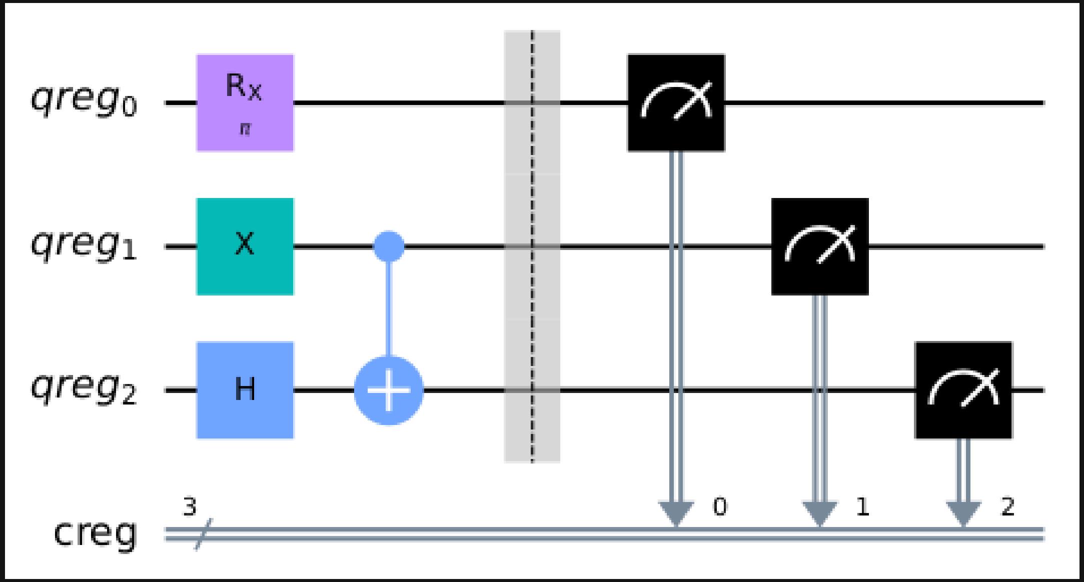
Metamorphic Transformations



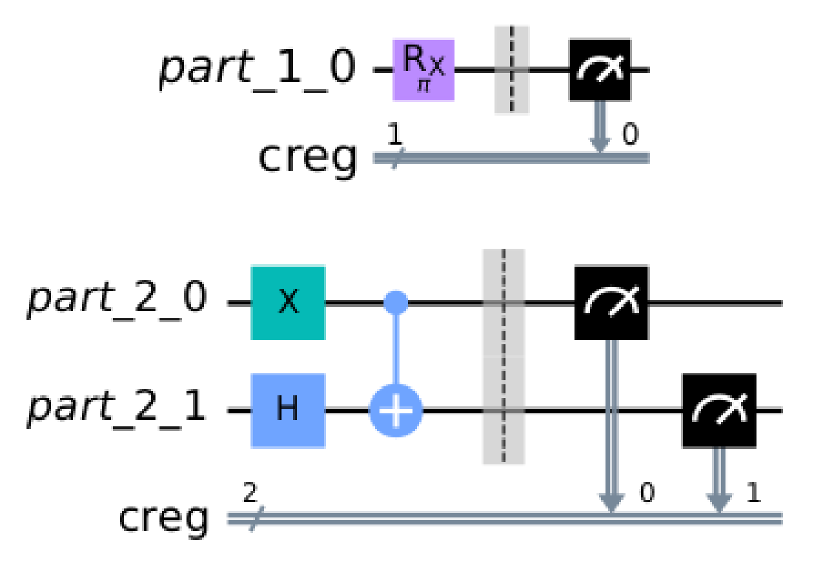


Change qubit order

Metamorphic Transformations



Partitioned execution



Metamorphic Transformations

2) Representation transformations

- Roundtrip conversion via QASM

3) Execution transformations

- Change of coupling map
- Change of gate size
- Change of optimization level
- Change of backend

Metamorphic Transformations

2) Representation transformations

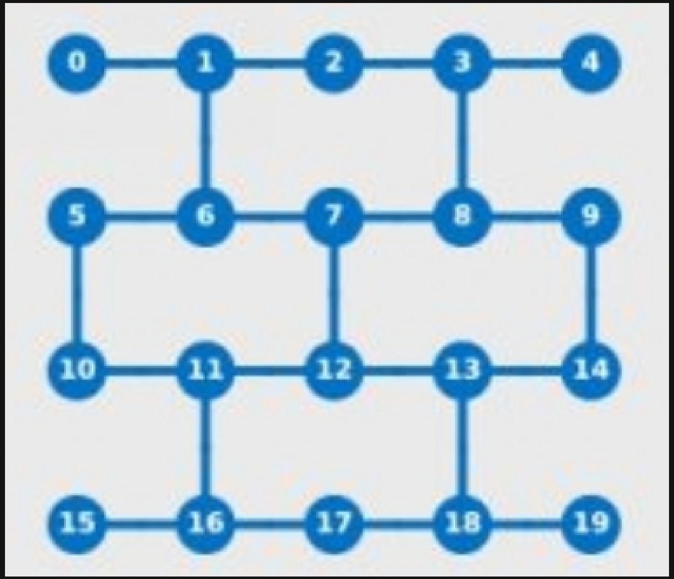
- Roundtrip conversion via QASM

3) Execution transformations

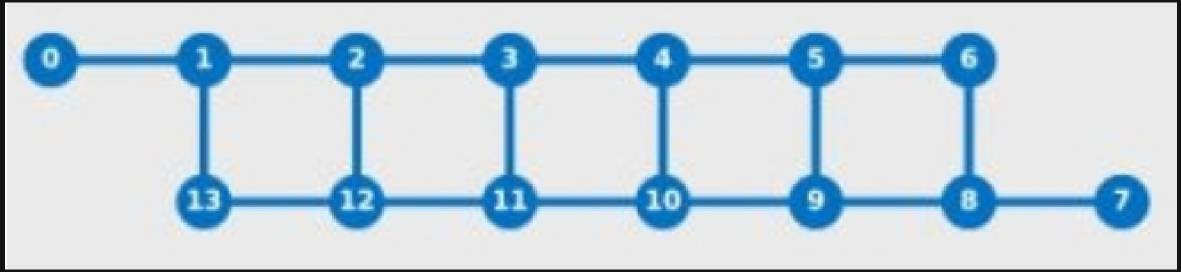
- Change of coupling map
- Change of gate size
- Change of optimization level
- Change of backend

Metamorphic Transformations

IBM Stuttgart, Germany



IBM Melbourne, Australia



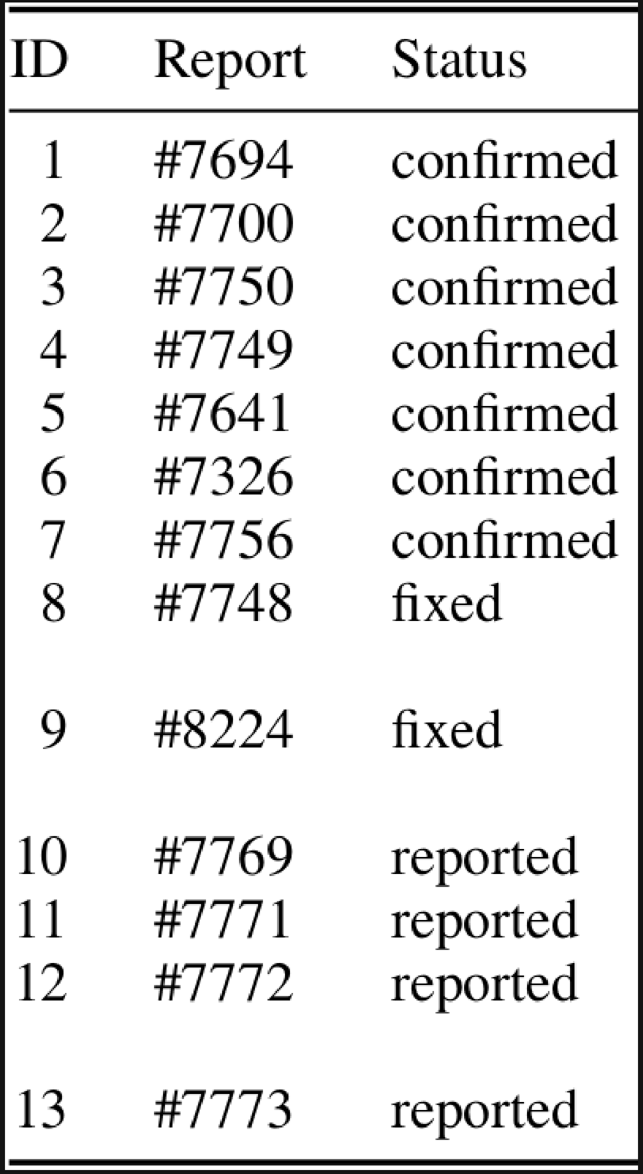
Comparing Behavior

- **Expected output relationship: Equivalence modulo changes in distribution**
  - E.g., changing qubit order will change measured bitstrings
- **Two oracles**
  - Crash vs. non-crash
  - Distribution differences (via Kolmogorov-Smirnov test)

Evaluation

- **Target: IBM’s Qiskit quantum computing platform**
- **48-hour run**
  - 8,360 generated programs
  - Same number of follow-up programs
    - 23.2% of follow-up programs crash
    - 0.7% of non-crashing have distribution differences

Effectiveness



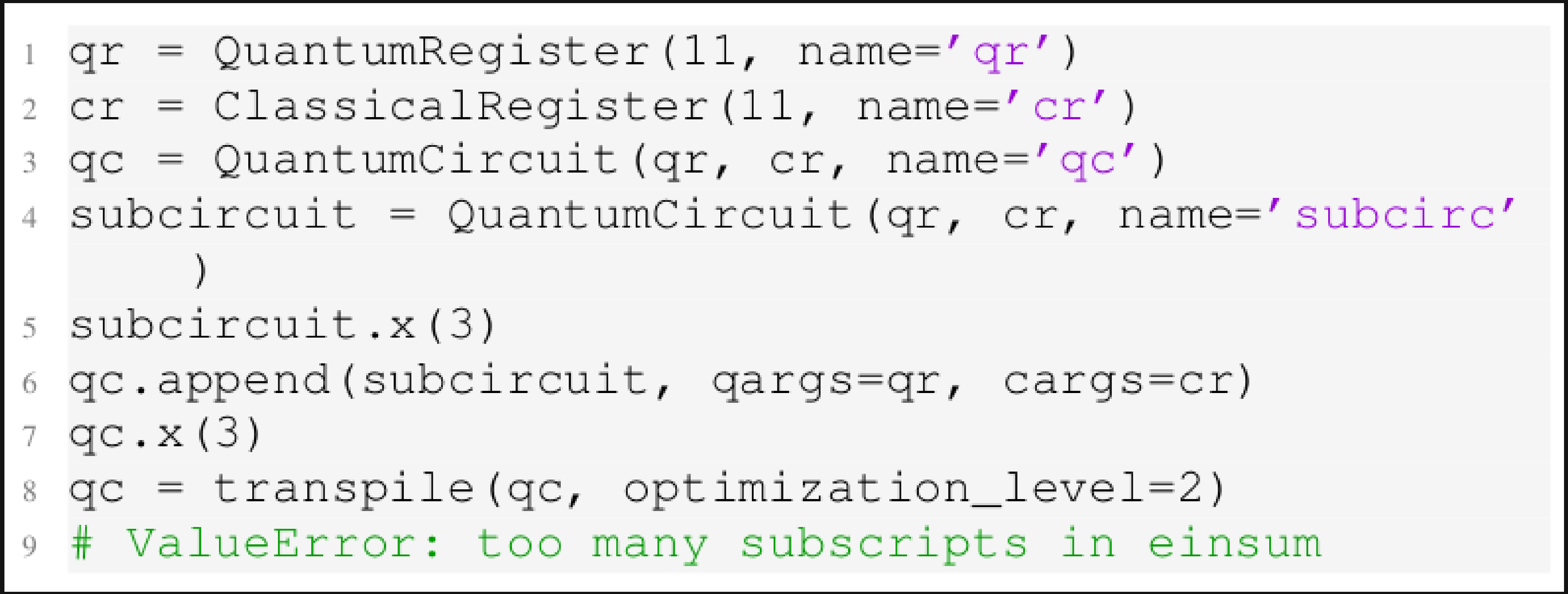
ID	Report	Status
1	#7694	confirmed
2	#7700	confirmed
3	#7750	confirmed
4	#7749	confirmed
5	#7641	confirmed
6	#7326	confirmed
7	#7756	confirmed
8	#7748	fixed
9	#8224	fixed
10	#7769	reported
11	#7771	reported
12	#7772	reported
13	#7773	reported

Bugs filed after

- Automated clustering of warnings
- Delta-debugging to reduce bug-triggering program

Example

Detected by **changing optimization level** and **injecting null-effect operation**



```
qr = QuantumRegister(11, name='qr')
cr = ClassicalRegister(11, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
subcircuit = QuantumCircuit(qr, cr, name='subcirc')
subcircuit.x(3)
qc.append(subcircuit, qargs=qr, cargs=cr)
qc.x(3)
qc = transpile(qc, optimization_level=2)
# ValueError: too many subscripts in einsum
```

This Talk

- Interactive Metamorphic Testing of Debuggers [ISSTA’19]
- MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform [ICSE’23]
- Lessons learned and open challenges ←

  [ICSE’24, ’25, etc. ?]

Lessons Learned

Key ingredient:

Metamorphic transformations

- Inherently domain-specific
- Relies on some “model” of the program-under-test
  - E.g., debuggers transform programs and debugging actions into a debugging trace

Lessons Learned

Key ingredient:
Metamorphic transformations

- Inherently domain-specific
- Relies on some “model” of the program-under-test
  - E.g., debuggers transform programs and debugging actions into a debugging trace

The better the transformations,
the more bugs you find

Lessons Learned (2)

Vaguely specified programs:
Difficult to define precise metamorphic oracles

- Negative example:

  Testing *git* version control system

  - Many underspecified corner cases
  - Failed to effectively test it

Lessons Learned (2)

Vaguely specified programs:
Difficult to define precise metamorphic oracles

- Negative example:

  Testing *git* version control system

  - Many underspecified corner cases
  - Failed to effectively test it

Make sure to know (at least parts of) the program's intended behavior

Lessons Learned (3)

Programs that operate on programs:
Excellent target for metamorphic testing

- Indended semantics are (relatively) clearly defined
- Can derive metamorphic relationships from PL semantics

Lessons Learned (3)

Programs that operate on programs:
Excellent target for metamorphic testing

- Indended semantics are (relatively) clearly defined
- Can derive metamorphic relationships from PL semantics

More developer tools are waiting to be tested

Open Challenges

- **False positives**
  - Debugger testing: 29/59 warnings
  - MorphQ: All warnings due to distribution differences
- **Automate creation of metamorphic relationships**
  - Initial evidence that ML-based prediction may help *

* Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code (Bareiß et al., 2022)

Summary

- Interactive Metamorphic Testing of Debuggers [ISSTA'19]
- MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform [ICSE'23]
- Lessons learned and open challenges [ICSE'24, '25, etc. ?]

Thanks!