Eclat: Automatic Generation and Classification of Test Inputs

Carlos Pacheco and Michael Ernst
Program Analysis Group
MIT

The Problem

- Suppose you have a program that works
  - It passes an existing test suite
  - Its observable behavior appears correct
- You want improved confidence in the program's reliability
  - Ensure that the program works on different inputs
    - If program’s operation incorrect on some input, fix the program and add a new test case to test suite

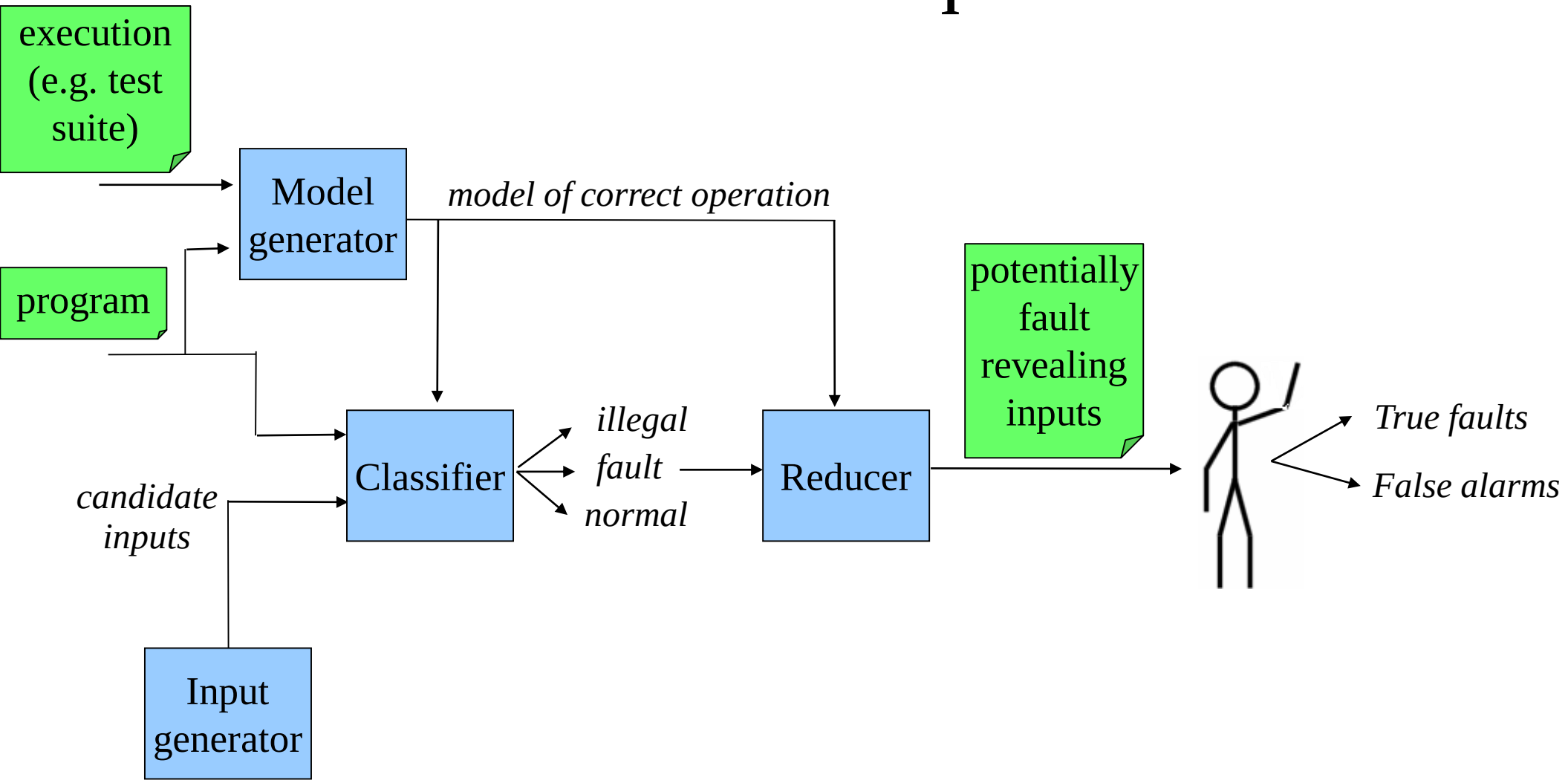
Input Generation

- Can automatically generate test inputs
  - Random generation [Klaessen & Hughes 2002, …]
  - Bounded exhaustive testing [Boyapati et al. 2002]
  - Constraint solving [Korel 1996, Gupta 1998, …]
  - Many more…
- Without automatic tool support, must inspect each resulting input (unless executable spec/oracle exists)
  - Is the input fault-revealing?
  - Is the input useful?

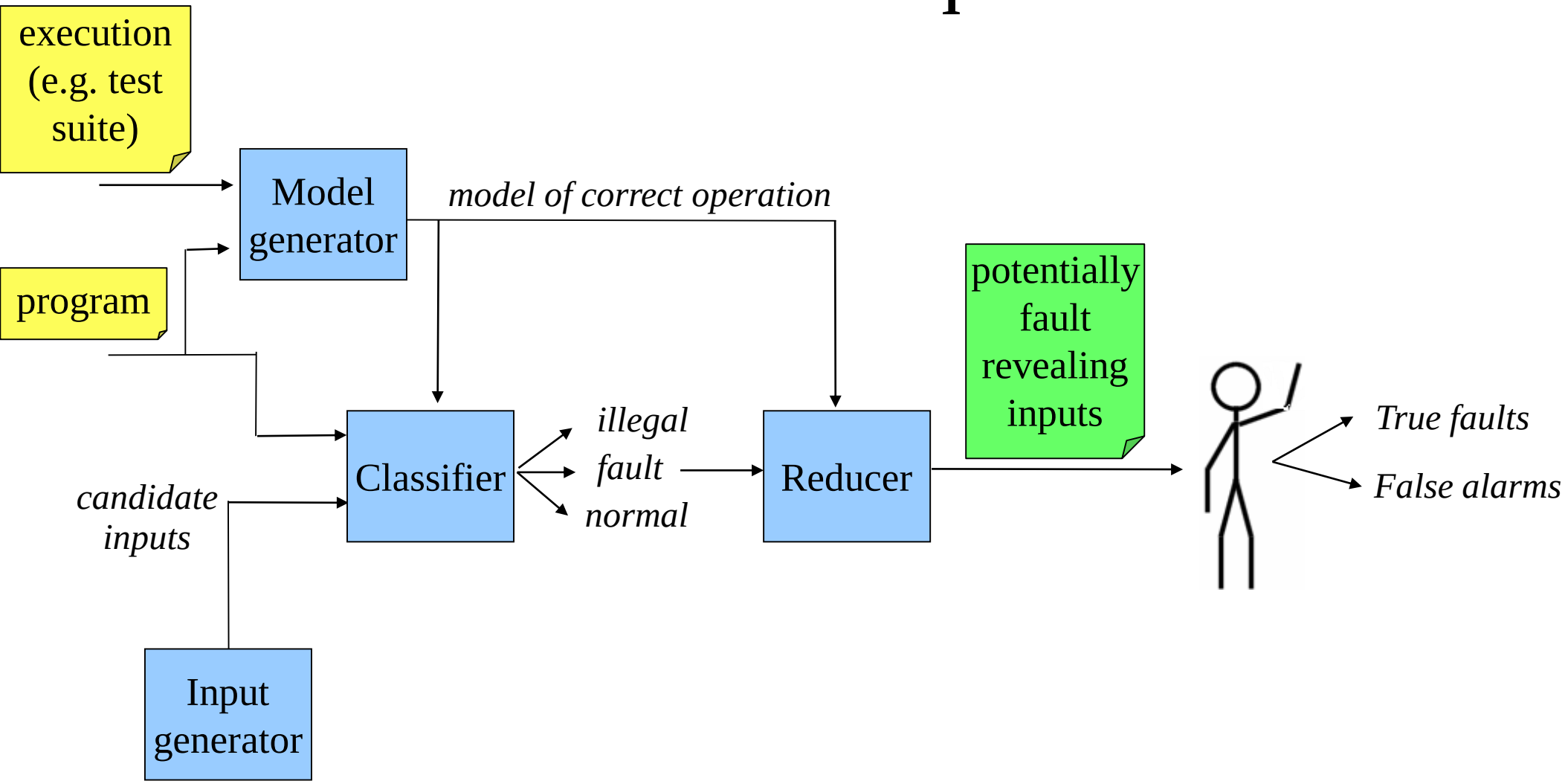
Research Goal

- Help the user select from a large number of inputs, a small “promising” subset:
  - Inputs exhibiting new program behavior
  - Inputs likely to reveal faults

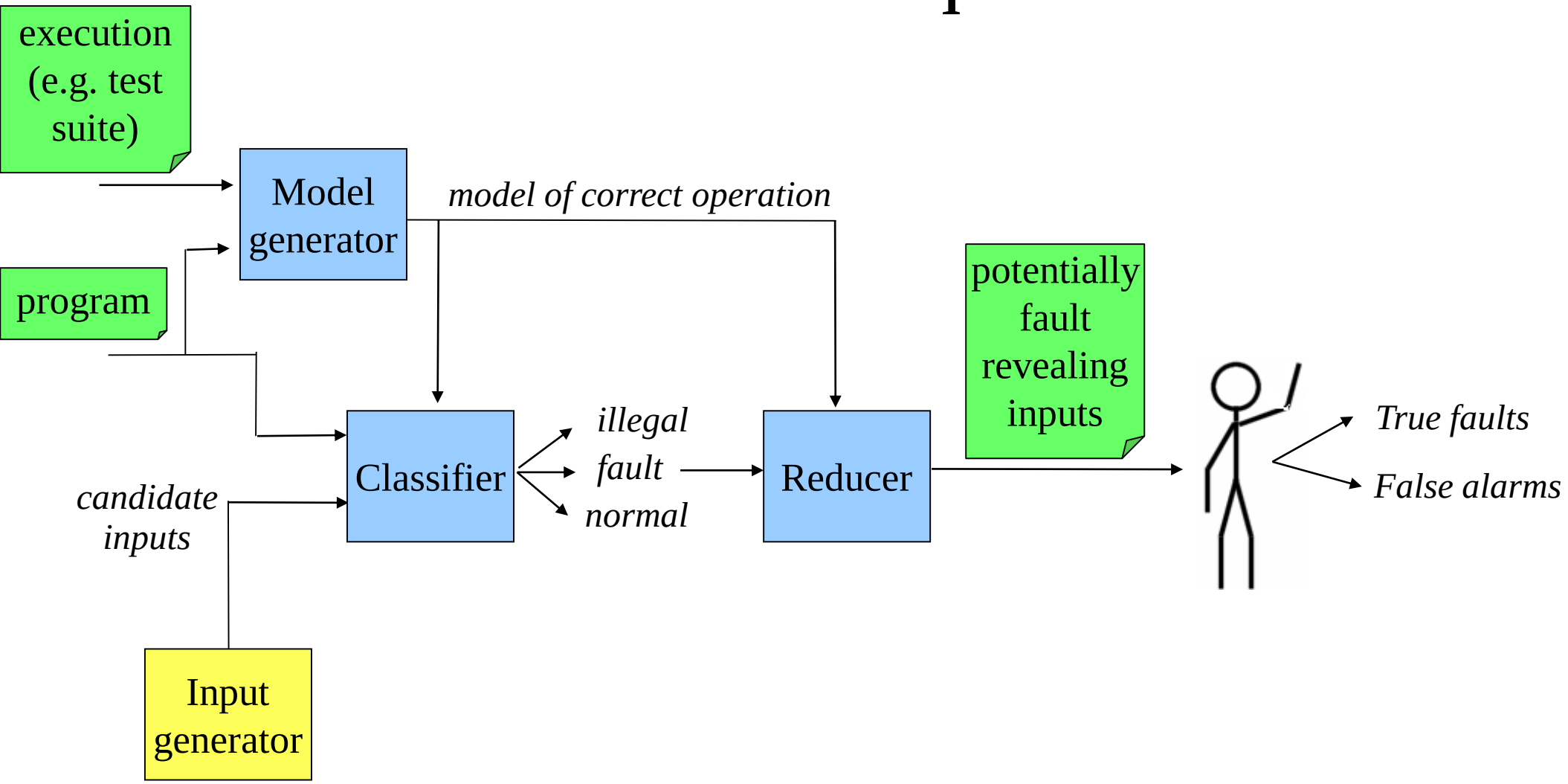
The Technique



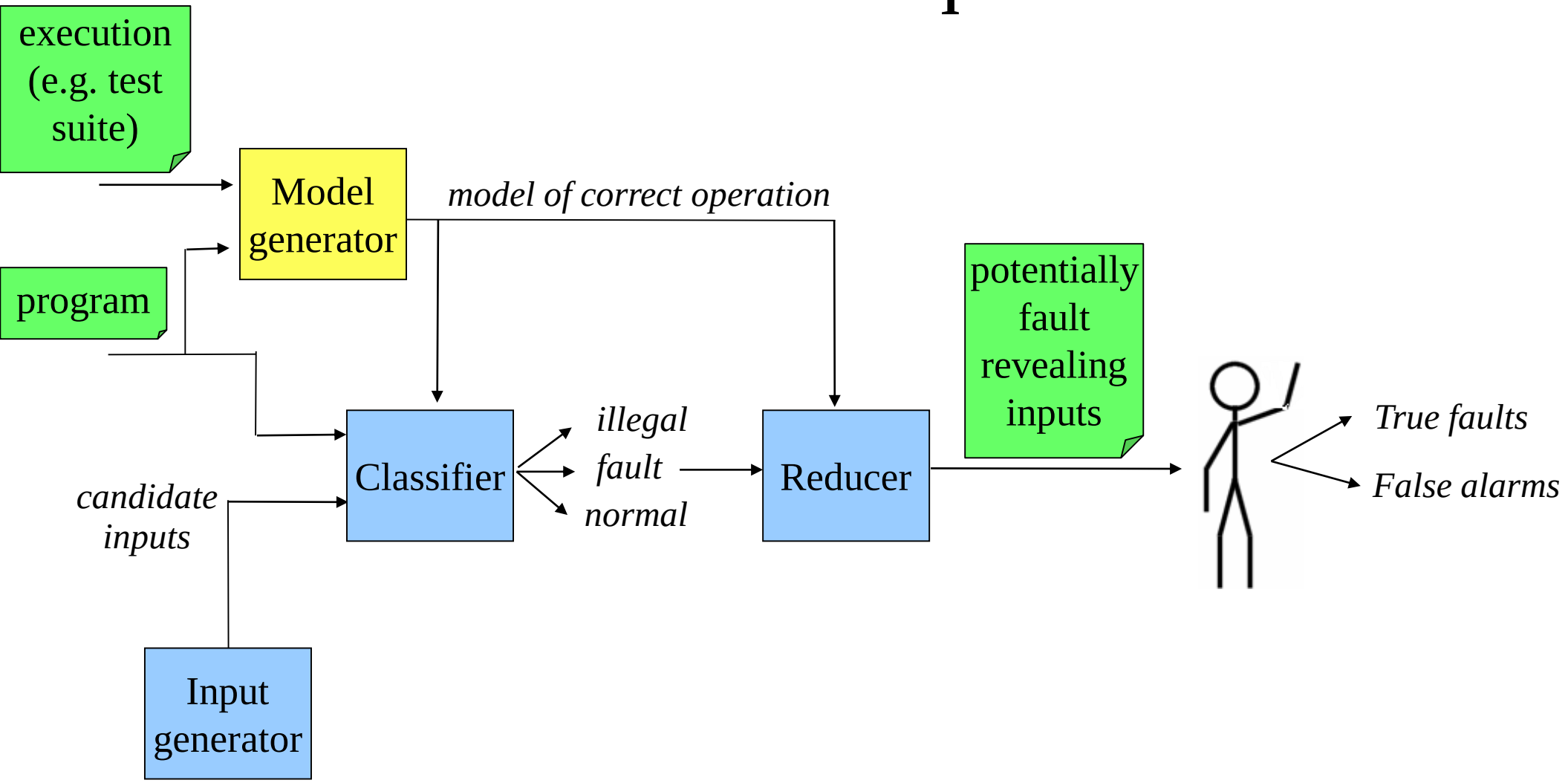
The Technique



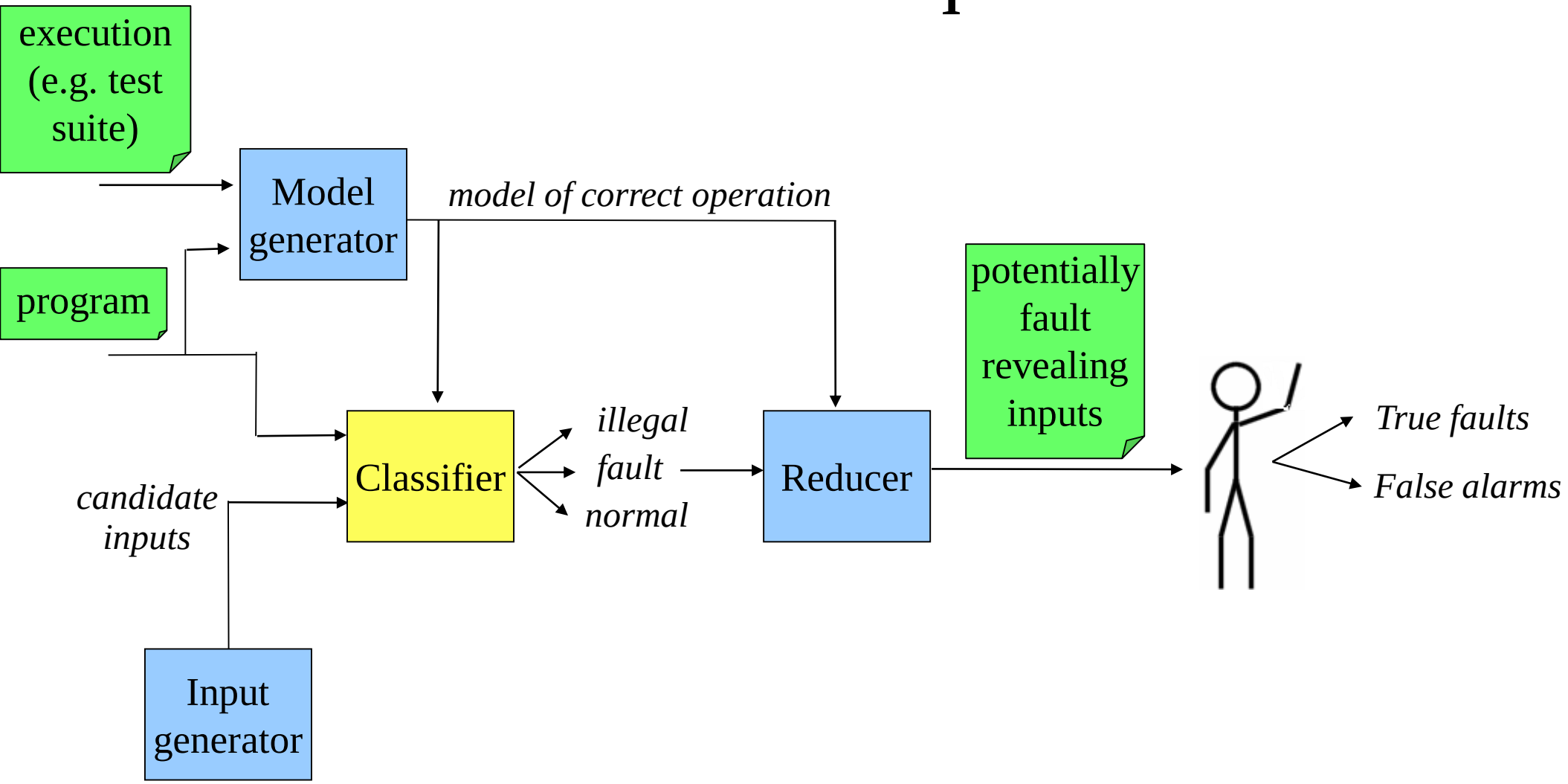
The Technique



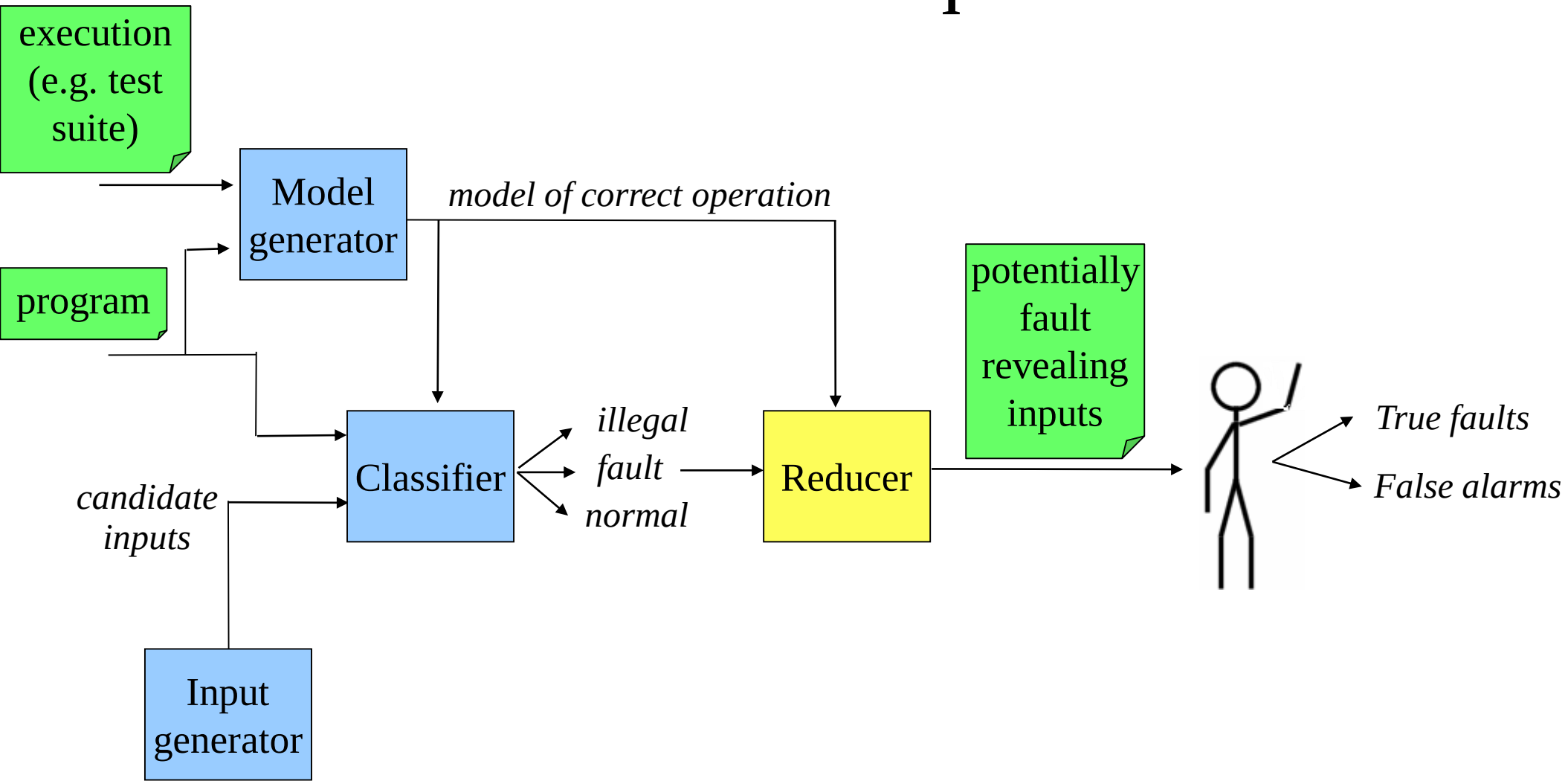
The Technique



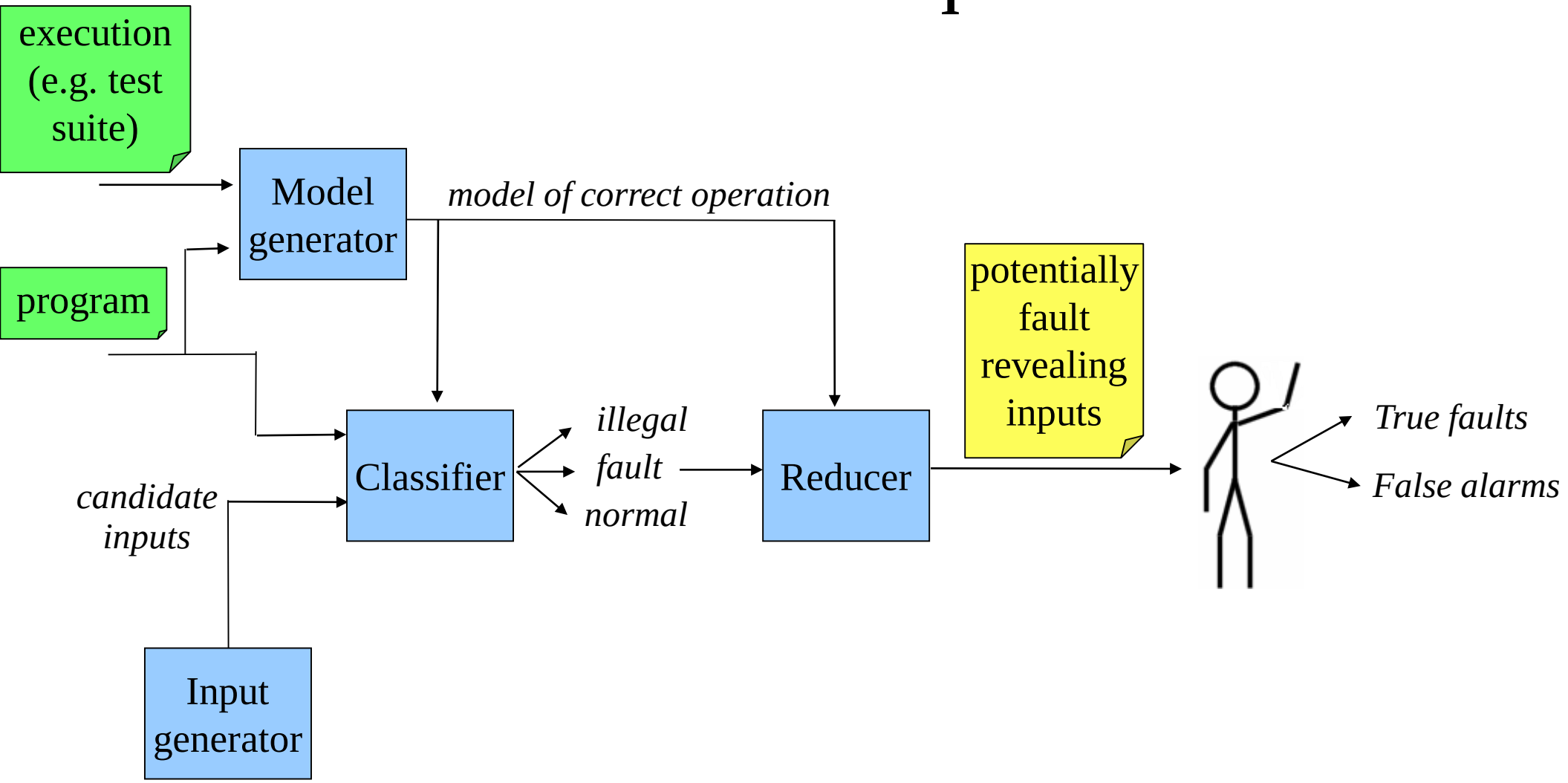
The Technique



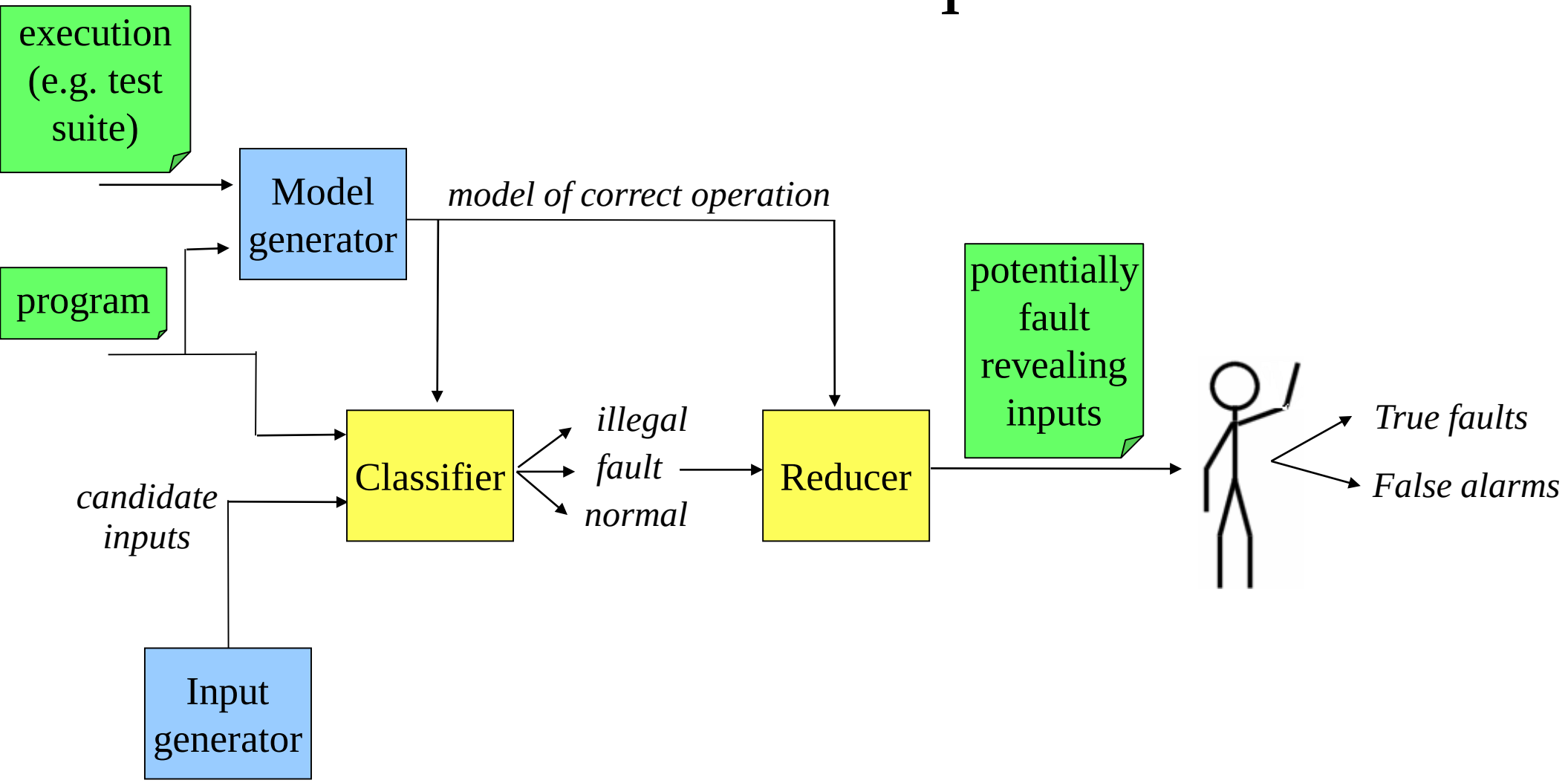
The Technique



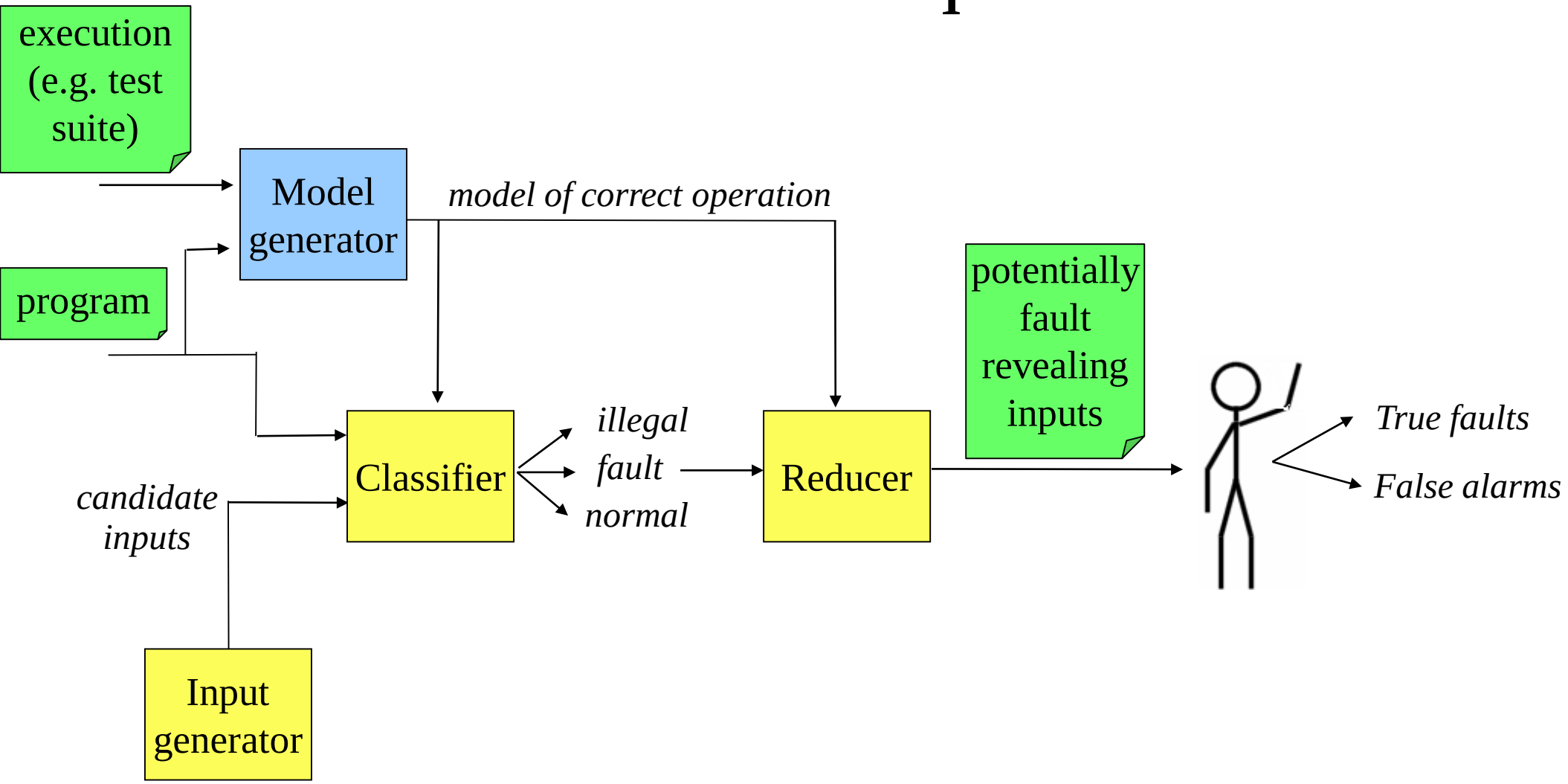
The Technique



The Technique



The Technique



Model Generator

- Our technique uses a *model generator* to produce a model of correct program operation, derived from observing a correct execution

  [Ernst et al. 2001, Ammons et al. 2002, Hankel and Diwan 2003, …]

- Our technique requires
  - Set of properties hold at component boundaries
  - The properties can be evaluated

Example: bounded stack

[Stotts et al. 2002, Xie and Notkin 2003, Csallner and Amaragdakis 2004]

```
public class Stack {

  private int[] elems;
  private int topOfStack;
  private int capacity;

  public Stack() { ... }

  public void push(int k) { ... }

  public void pop() {
     topOfStack --;
  }

  public boolean isMember(int i) {
    ...
  }

...

}
```

```
public class StackTest {
  ...
}
```

Example: bounded stack

```
public class Stack {

  private int[] elems;
  private int topOfStack;
  private int capacity;

  public Stack() { ... }

  public void push(int k) { ... }

  public void pop() {
     topOfStack --;
  }

  public boolean isMember(int i) {
    ...
  }

...

}
```

```
public class StackTest {
  ...
}
```

object properties

capacity == elems.length
elems != null
capacity == 2
topOfStack >= 0

pop: entry properties

elems ∈ { [3,0], [3,2] }

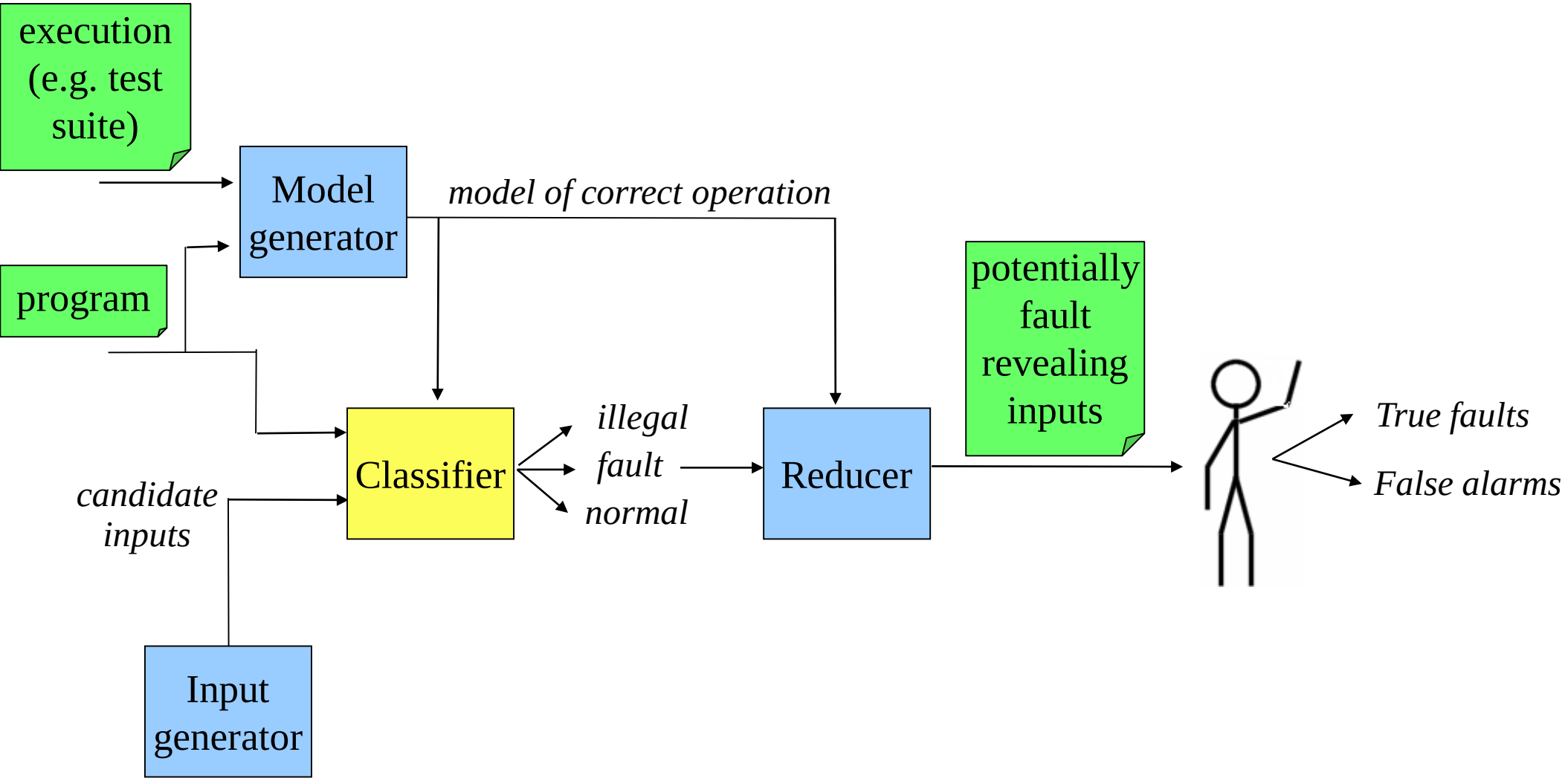
isMember: entry properties

k ∈ elems

isMember: exit properties

elems == *orig*(elems)
orig(k) ∈ elems

Classifier


execution (e.g. test suite)
Model generator
model of correct operation
program
potentially fault revealing inputs
Classifier
illegal
fault
normal
Reducer
True faults
False alarms
candidate inputs
Input generator

Classifier

- Run program on candidate input
- Detect set of violated model properties
- *Classify:*

entry violations?	*exit violations?*	*Classification*
no	no	normal
no	yes	fault
yes	no	normal (new)
yes	yes	illegal

Classifier: normal input

entry violations?	*exit violations?*	*Classification*
no	no	normal
no	yes	fault
yes	no	normal (new)
yes	yes	illegal

```
Stack var1 = new Stack();
var1.push(3);
var1.pop();
```

object properties (all methods)

capacity == elems.length
elems != null
capacity == 2
topOfStack >= 0

pop: entry properties

elems ∈ { [3,0], [3,2] }

isMember: entry properties

k ∈ elems

isMember: exit properties

elems == *orig*(elems)
orig(k) ∈ elems

Classifier: normal input

entry violations?	*exit violations?*	*Classification*
no	no	normal
no	yes	fault
yes	no	normal (new)
yes	yes	illegal

```
Stack var1 = new Stack();
var1.push(3);
var1.pop();
```

object properties (all methods)

capacity == elems.length
elems != null
capacity == 2
topOfStack >= 0

pop: entry properties

elems ∈ { [3,0], [3,2] }

isMember: entry properties

k ∈ elems

isMember: exit properties

elems == *orig*(elems)
orig(k) ∈ elems

Classifier: fault-revealing input

entry violations?	*exit violations?*	*Classification*
no	no	normal
no	yes	fault
yes	no	normal (new)
yes	yes	illegal

```
Stack var1 = new Stack();
var1.push(3);
var1.pop();
var1.pop();
```

object properties (all methods)

capacity == elems.length
elems != null
capacity == 2
topOfStack >= 0

pop: entry properties

elems ∈ { [3,0], [3,2] }

isMember: entry properties

k ∈ elems

isMember: exit properties

elems == *orig*(elems)
orig(k) ∈ elems

Classifier: fault-revealing input

entry violations?	*exit violations?*	*Classification*
no	no	normal
no	yes	fault
yes	no	normal (new)
yes	yes	illegal

```
Stack var1 = new Stack();
var1.push(3);
var1.pop()
var1.pop();
```

object properties (all methods)

capacity == elems.length
elems != null
capacity == 2
topOfStack >= 0 *(on exit)*

pop: entry properties

elems ∈ { [3,0], [3,2] }

isMember: entry properties

k ∈ elems

isMember: exit properties

elems == *orig*(elems)
orig(k) ∈ elems

Classifier: illegal input

entry violations?	*exit violations?*	*Classification*
no	no	normal
no	yes	fault
yes	no	normal (new)
yes	yes	illegal

```
Stack var1 = new Stack();
var1.push(0);
var1.isMember(-5);
```

object properties (all methods)

capacity == elems.length
elems != null
capacity == 2
topOfStack >= 0

pop: entry properties

elems ∈ { [3,0], [3,2] }

isMember: entry properties

k ∈ elems

isMember: exit properties

elems == *orig*(elems)
orig(k) ∈ elems

Classifier: illegal input

entry violations?	*exit violations?*	*Classification*
no	no	normal
no	yes	fault
yes	no	normal (new)
yes	yes	illegal

```
Stack var1 = new Stack();
var1.push(0);
var1.isMember(-5);
```

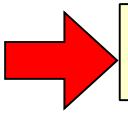
object properties (all methods)

capacity == elems.length
elems != null
capacity == 2
topOfStack >= 0

pop: entry properties

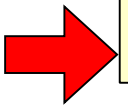
elems ∈ { [3,0], [3,2] }

isMember: entry properties



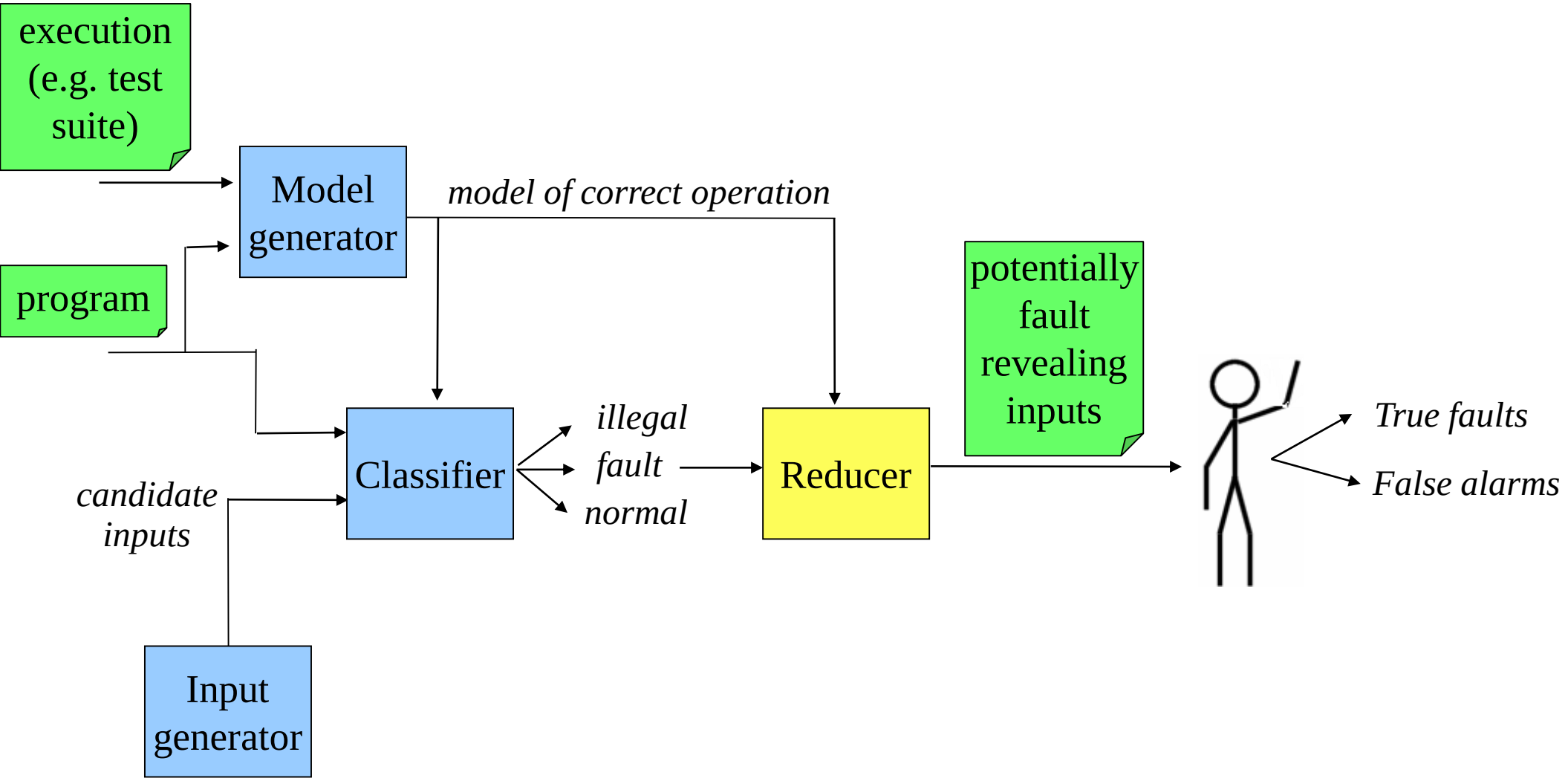
k ∈ elems

isMember: exit properties



elems == *orig*(elems)
***orig*(k) ∈ elems**

Reducer


execution (e.g. test suite)
Model generator
model of correct operation
program
potentially fault revealing inputs
Classifier
illegal
fault
normal
Reducer
True faults
False alarms
candidate inputs
Input generator

Reducer

- Partitions inputs based on the set of properties they violate
- Reports one inputs from each partition
- Inputs in same partition are likely to manifest same faulty behavior

Reducer: example

Two equivalent inputs:

```
Stack var1 = new Stack();
var1.push(3);
var1.pop();
var1.pop();
```

Violation pattern:

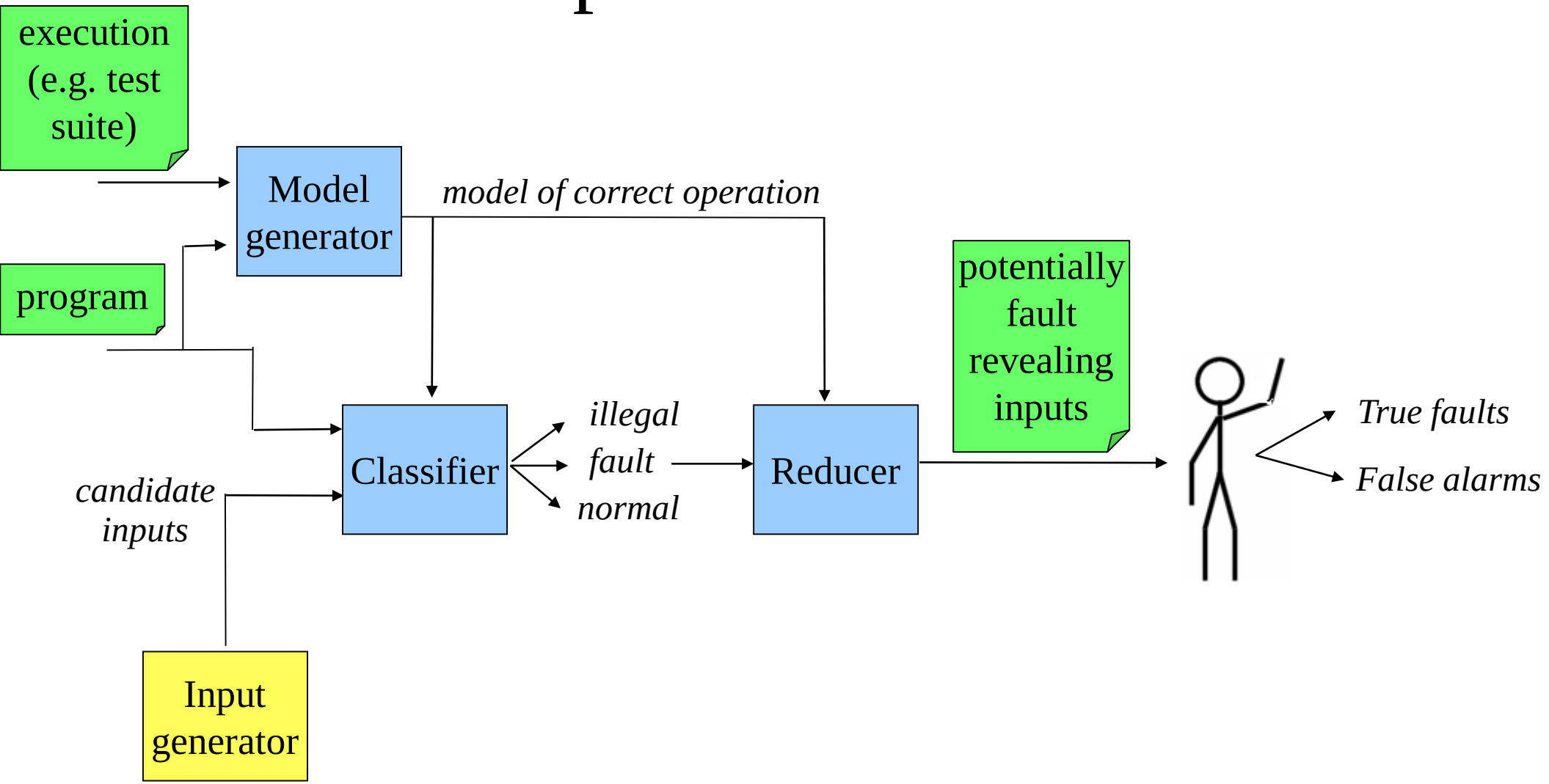
topOfStack >= 0 *(on exit)*

```
Stack var1 = new Stack();
var1.push(0);
var1.pop();
var1.push(3);
var1.pop();
var1.pop();
```

Violation pattern:

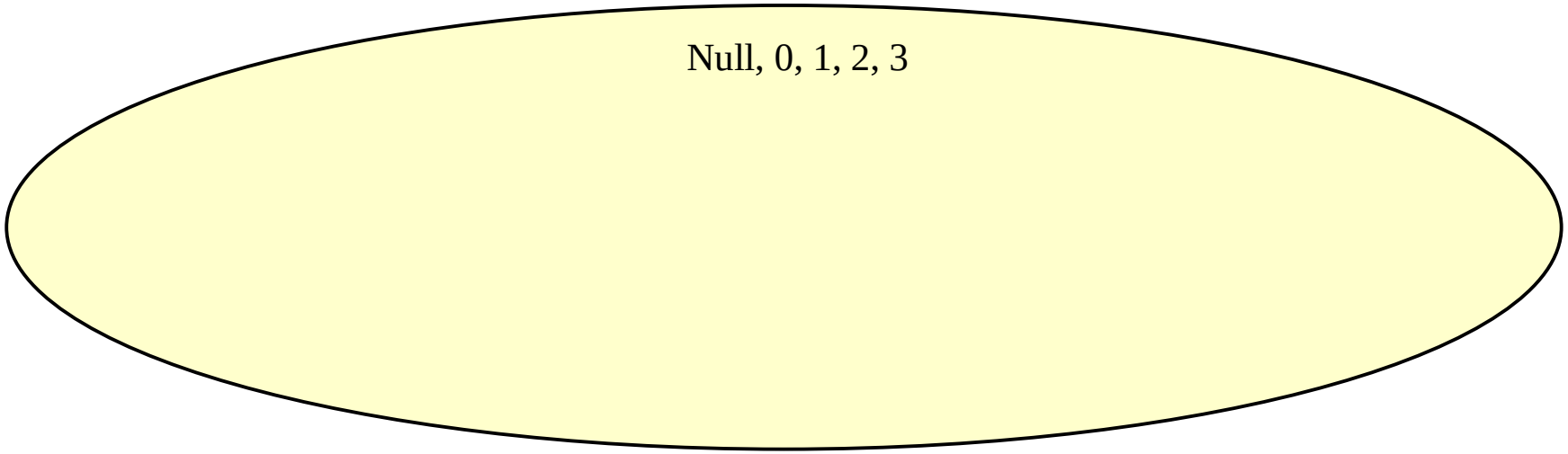
topOfStack >= 0 *(on exit)*

Input Generator



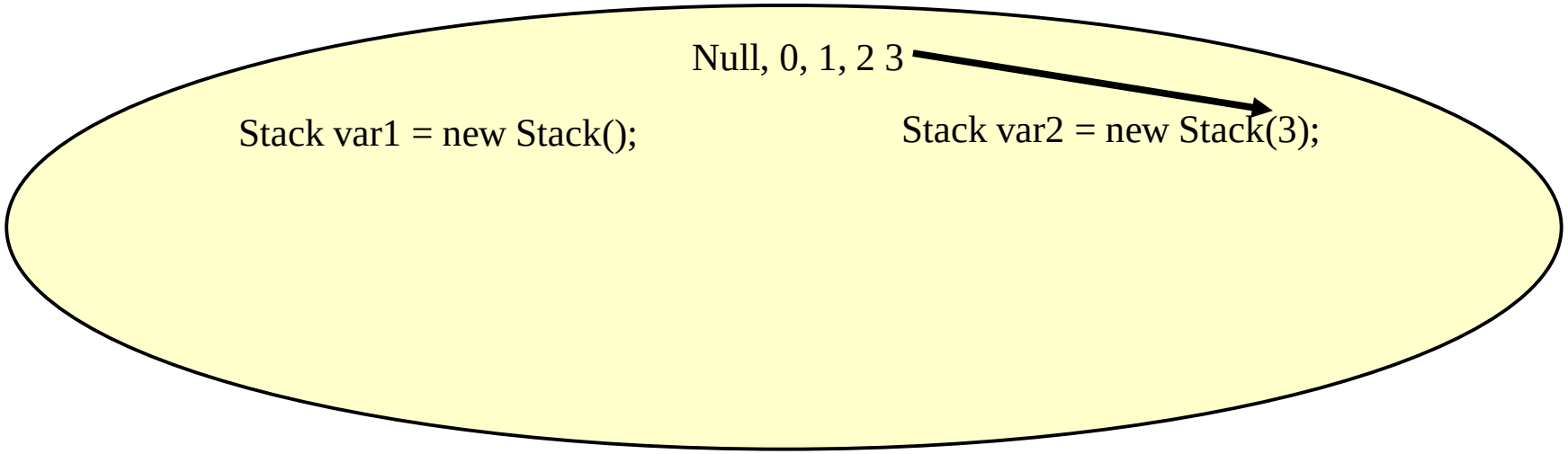
Bottom-up Random Generator

1. pool := a set of primitives (null, 0, 1, etc.)
2. do N times:
   2.1. create new inputs by calling methods/constructors using pool inputs as arguments
   2.2. add resulting inputs to the pool



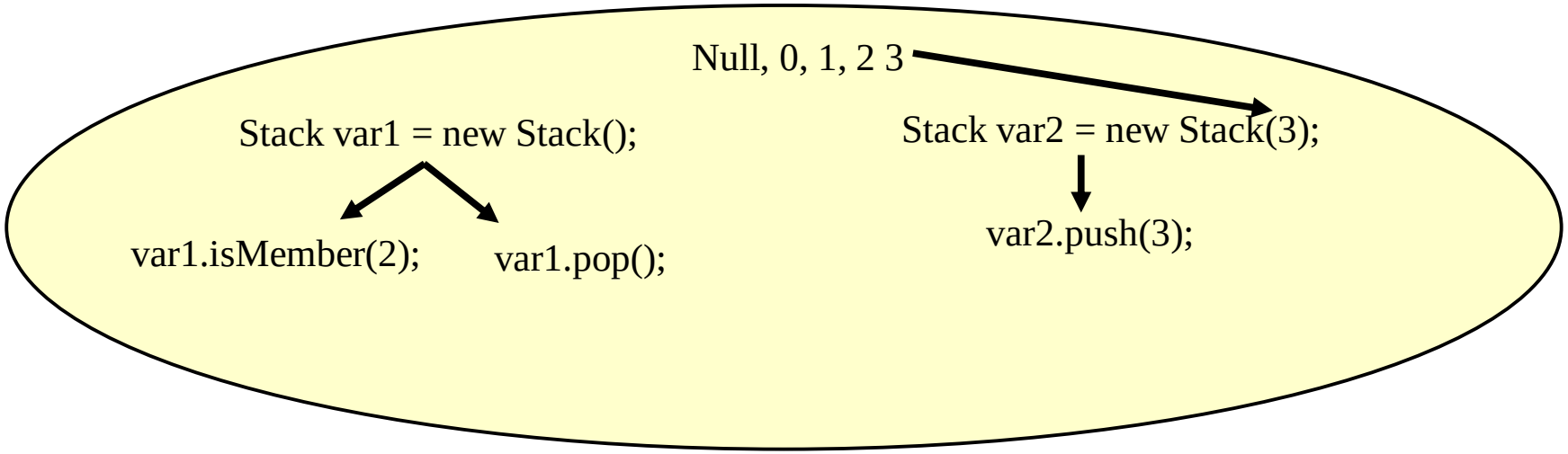
Bottom-up Random Generator

1. pool := a set of primitives (null, 0, 1, etc.)
2. do N times:
   2.1. create new inputs by calling methods/constructors using pool inputs as arguments
   2.2. add resulting inputs to the pool



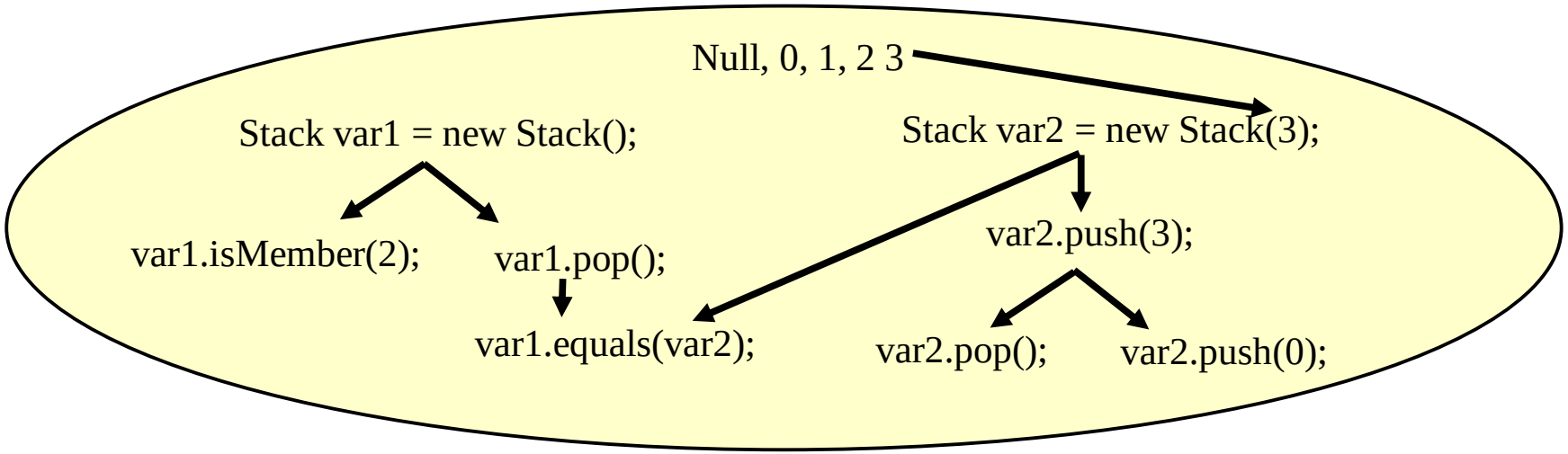
Bottom-up Random Generator

1. pool := a set of primitives (null, 0, 1, etc.)
2. do N times:
   2.1. create new inputs by calling methods/constructors using pool inputs as arguments
   2.2. add resulting inputs to the pool



Bottom-up Random Generator

1. pool := a set of primitives (null, 0, 1, etc.)
2. do N times:
   2.1. create new inputs by calling methods/constructors using pool inputs as arguments
   2.2. add resulting inputs to the pool



Avoiding illegal inputs

- It’s important that the inputs in the pool are legal
  - Inputs in the pool are building blocks for other inputs

Input 1(tests **pop**)

```
Stack s = new Stack();
s.pop();
```

fault-revealing

Input 2 (tests **equals**)

```
Stack s = new Stack();
s.pop();
Stack s2 = new Stack(3);
s.equals(s2);
```

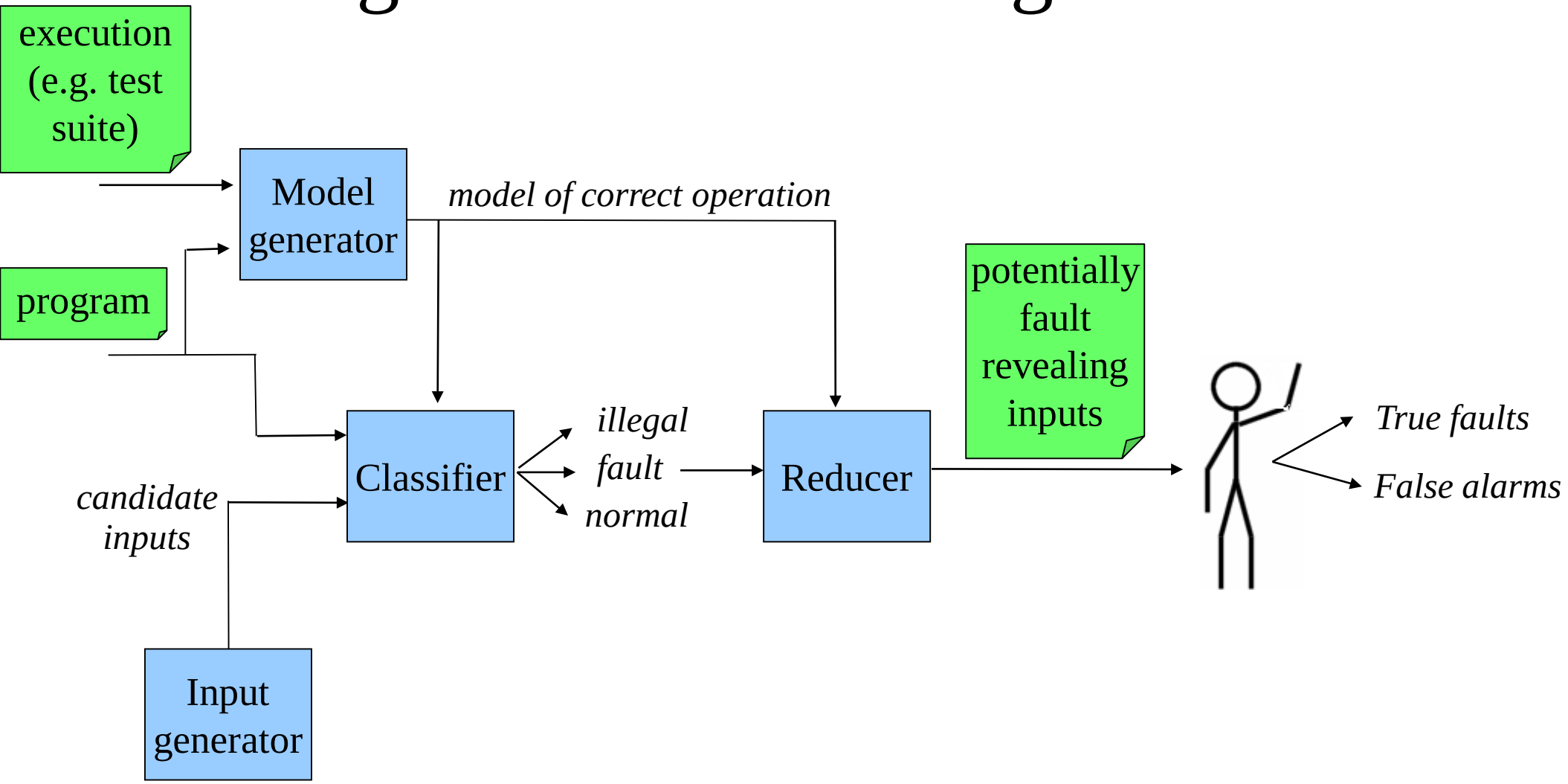
illegal

Input 3 (tests **isMember**)

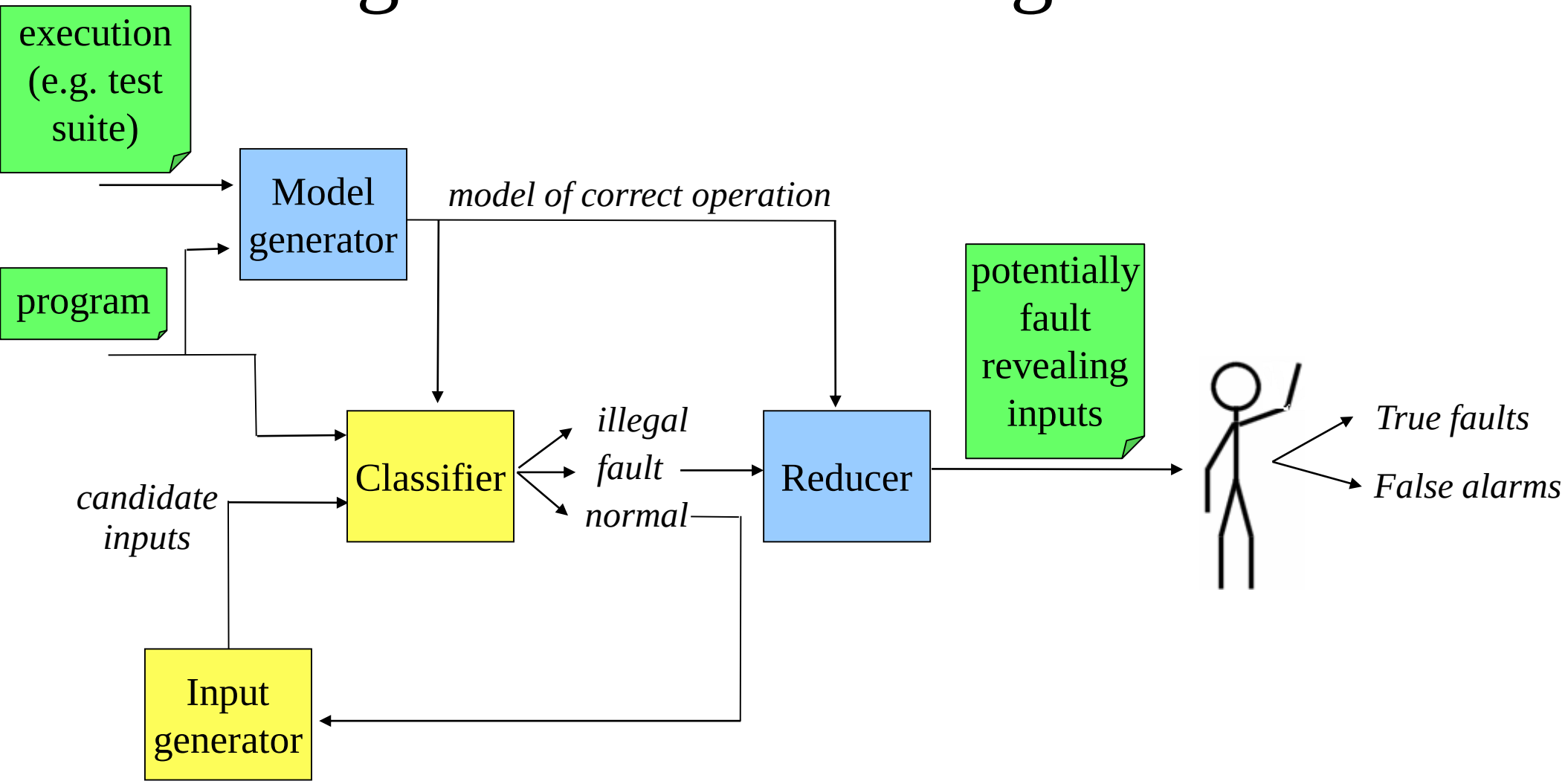
```
Stack s = new Stack();
s.pop();
s.isMember(1);
```

illegal

Using the classifier for generation



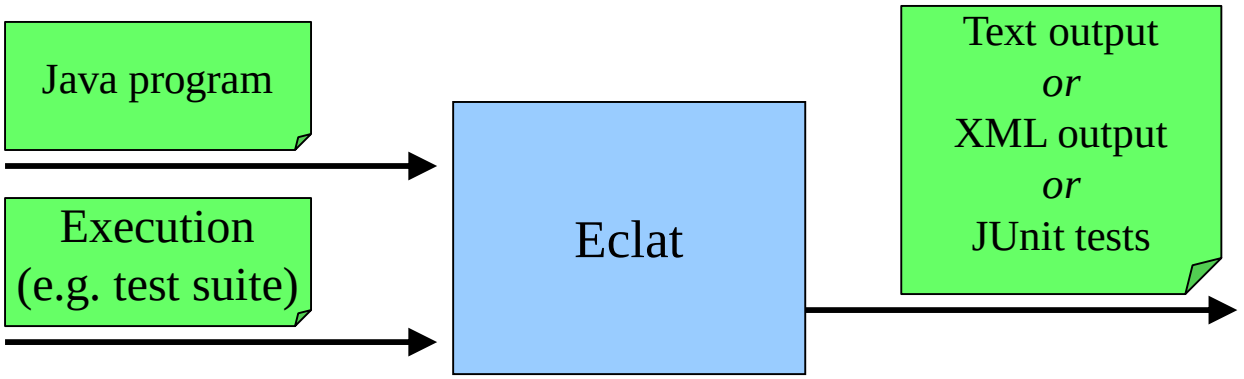
Using the classifier for generation



Enhanced generator

1. pool := a set of primitives (null, 0, 1, etc.)
2. do N times:
   - 2.1. create new inputs by calling methods/constructors using pool inputs as arguments
   - 2.2 classify inputs
   - 2.3 throw away illegal inputs
   - 2.4 save away fault inputs
   - 2.5. add normal inputs to the pool

Eclat



- Eclat generates inputs for Java unit testing
- Eclat uses the Daikon invariant detector to create a model of correct execution
- Each test input is wrapped as a JUnit test
- Eclat proposes assertion checks based on violated properties

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

Eclat's output: example

```
public void test_1_integrate() {

  RatPoly rp1 = new RatPoly(4, 3);
  RatPoly rp2 = new RatPoly(1, 1);
  RatPoly rp3 = rp1.add(rp2);

  checkPreProperties(rp3);
  rp3.integrate(0);
  checkPostProperties(rp3);

}
```

Eclat's output: example

```
public void test_1_integrate() {

  RatPoly rp1 = new RatPoly(4, 3);
  RatPoly rp2 = new RatPoly(1, 1);
  RatPoly rp3 = rp1.add(rp2);

  checkPreProperties(rp3);
  rp3.integrate(0);
  checkPostProperties(rp3);

}
```

Eclat's output: example

```
public void test_1_integrate() {

  RatPoly rp1 = new RatPoly(4, 3);
  RatPoly rp2 = new RatPoly(1, 1);
  RatPoly rp3 = rp1.add(rp2);

  checkPreProperties(rp3);
  rp3.integrate(0);
  checkPostProperties(rp3);

}
```


```
public void checkPostProperties(RatPoly rp) {

  ...

 // on exit: all terms in rp always have non-zero coefficient
 Assert.assertTrue(!allZeroes(rp.terms));

}
```

Eclat's output: example

```
public void test_1_integrate() {

  RatPoly rp1 = new RatPoly(4, 3);
  RatPoly rp2 = new RatPoly(1, 1);
  RatPoly rp3 = rp1.add(rp2);

  checkPreProperties(rp3);
  rp3.integrate(0);
  checkPostProperties(rp3);

}
```

Assertion violation!

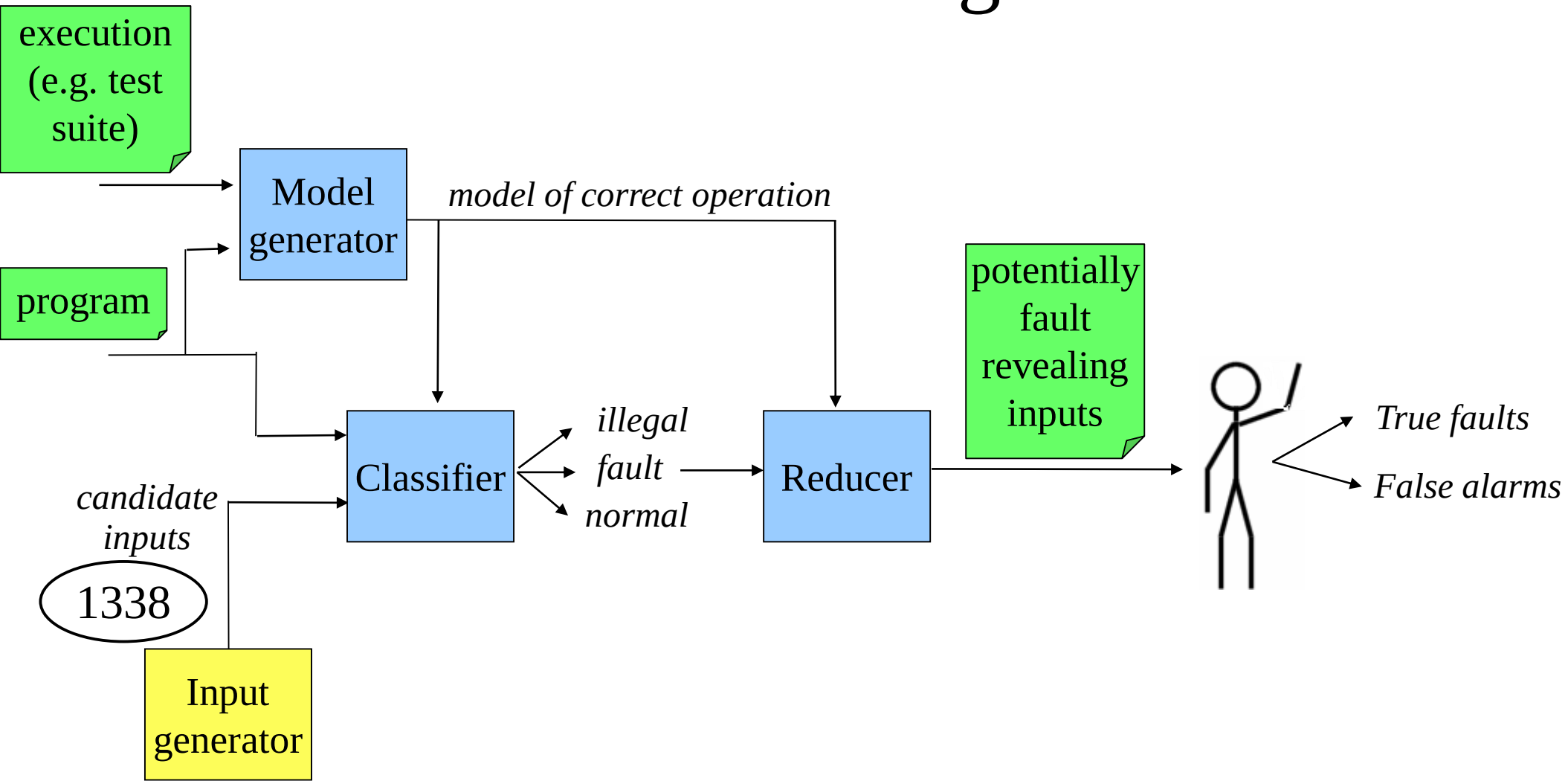
```
public void checkPostProperties(RatPoly rp) {

 ...

 // on exit: all terms in rp always have non-zero coefficient
 Assert.assertTrue(!allZeroes(rp.terms));

}
```

94 inputs violate this property.
Of these, 3 are shown to the user.

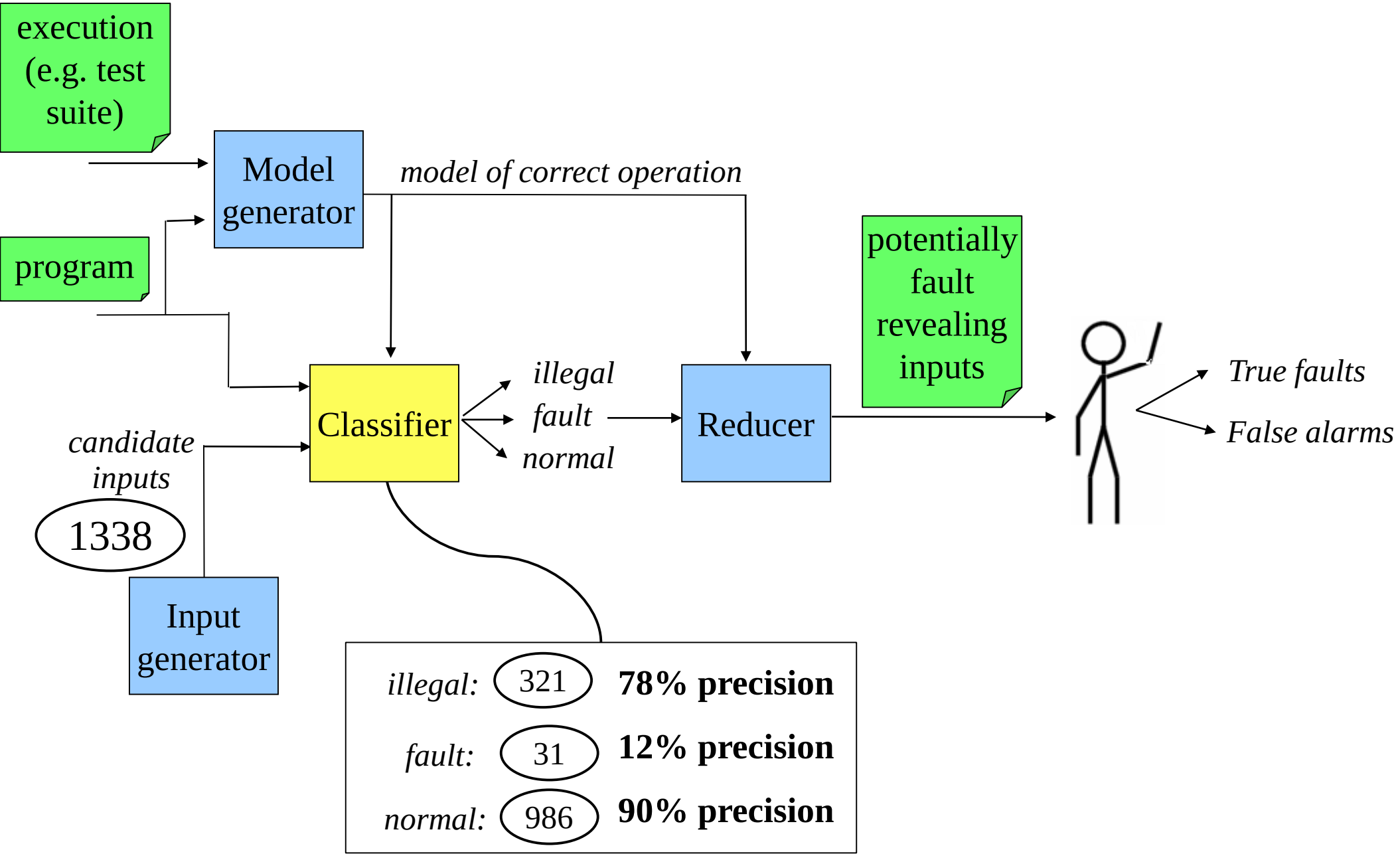
Evaluation

- We used Eclat to generate inputs for
  - 6 families of libraries
  - 64 distinct interfaces
  - 631 implementations
  - 75,000 NCNB lines of code

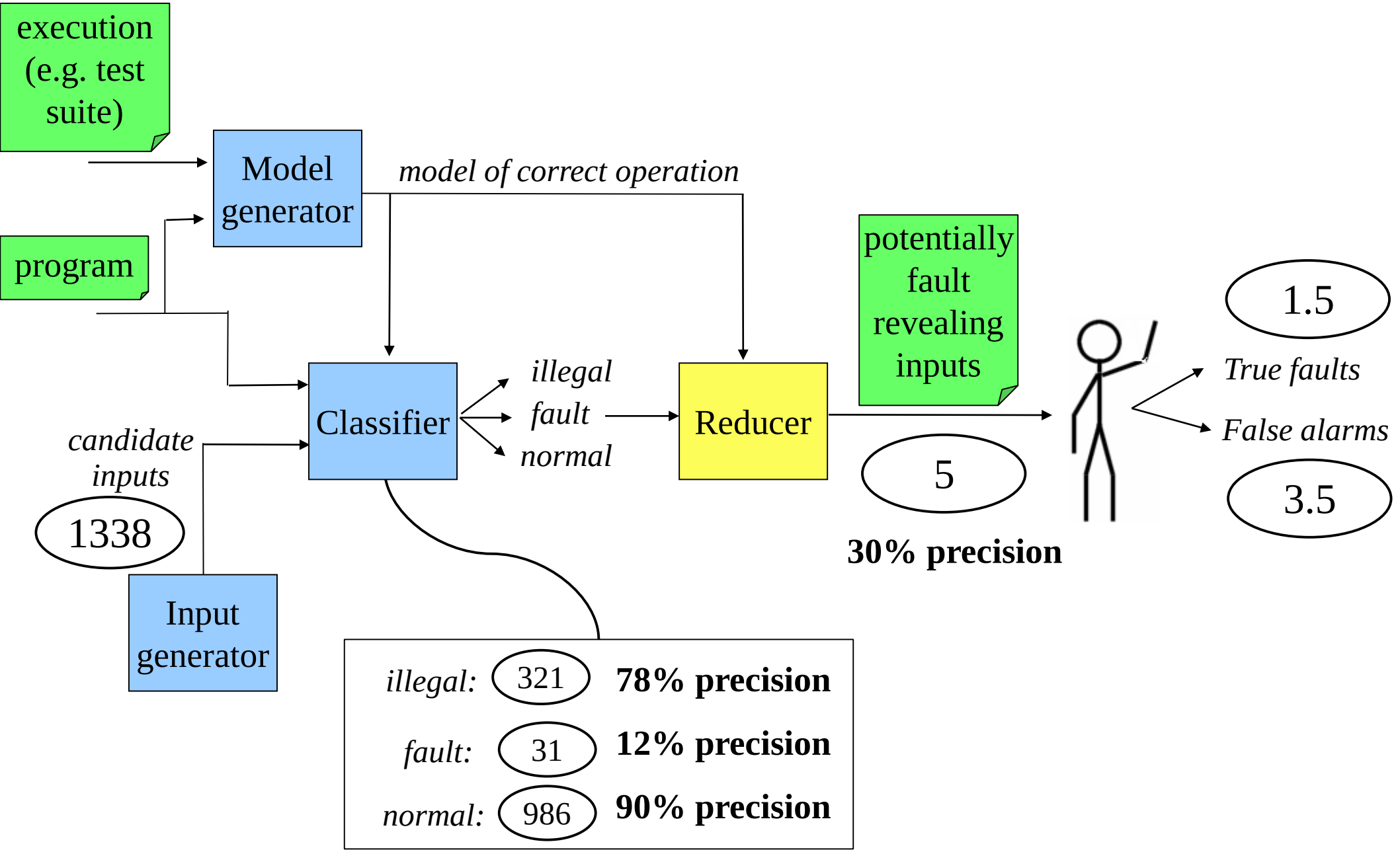
Evaluation results: generator



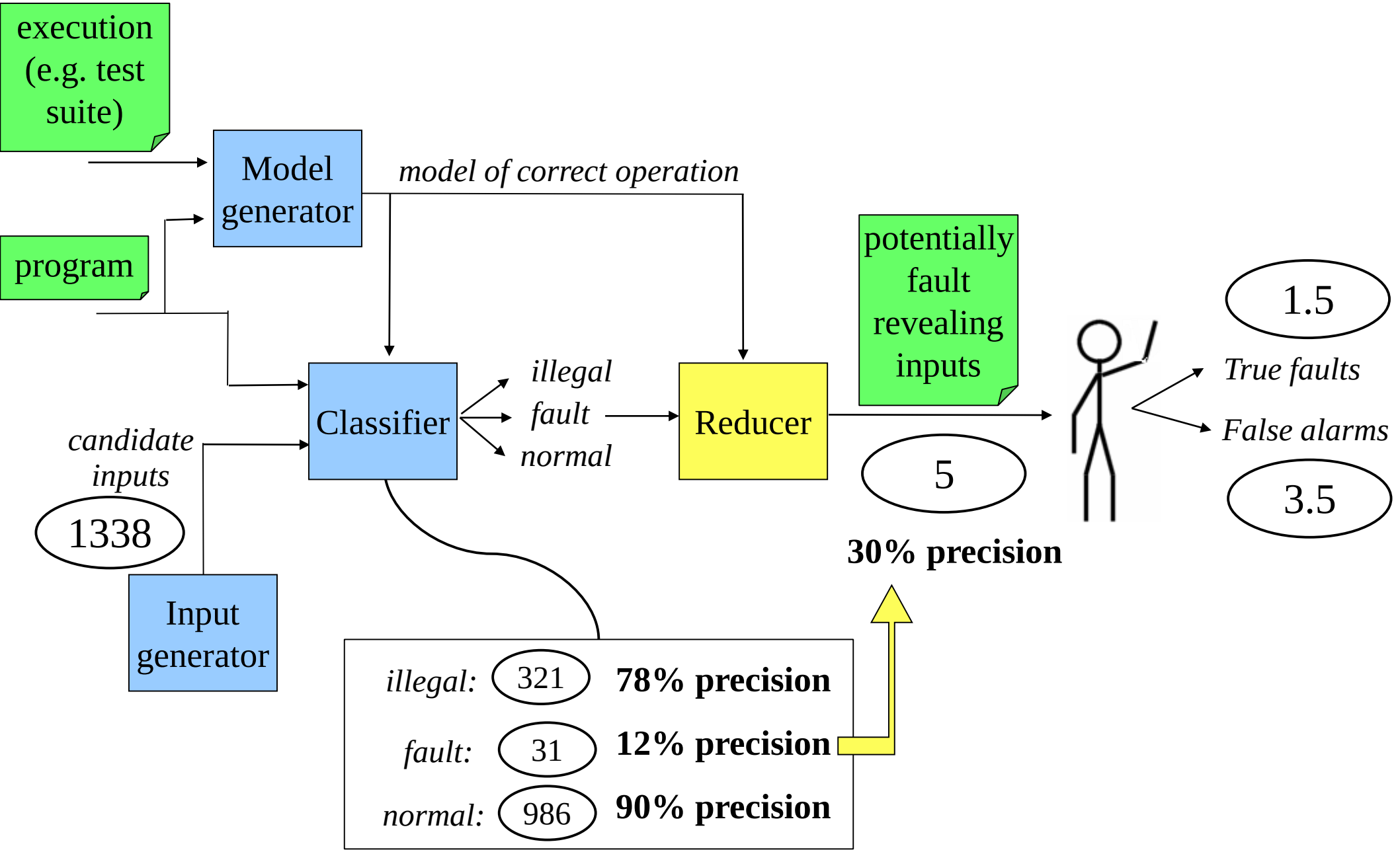
Evaluation results: classifier



Evaluation results: reducer



Evaluation results: reducer



Future Directions

- Incorporate other classification techniques

  [Podgursky et al. 2003, Brun and Ernst 2004, Bowring et al. 2004, …]

- Incorporate other generation strategies

  [Korel 1996, Gupta 1998, Klaessen & Hughes 2002, Boyapati et al. 2002, …]

Conclusion

- Technique to generate new program inputs
  - Inputs likely to reveal faults
  - Inputs not covered by an existing test suite
- Technique is effective in uncovering errors
- Eclat: automatically generates unit tests for Java classes

Eclat:

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

Related Work

- Harder et al. Improving test suites via operational abstraction. *ICSE 2003*.
- Xie and Notkin. Tool-assisted unit test selection based on operational violations. *ASE 2003*.
- Csallner and Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software Practice and Experience*, 2004.