Static Program Analysis Meets Test Case Generation
Lecture 3

Maria Christakis

MPI-SWS, Germany

Textbook:
Weakest Preconditions

Verification condition

- Logic formula reflecting the correctness of a program
- Typically checked by an automatic theorem prover
- Equivalent formulations can affect performance

Verification condition

- Logic formula reflecting the correctness of a program
- Typically checked by an automatic theorem prover
- Equivalent formulations can affect performance

A technique for computing verification conditions is

weakest preconditions

A simple language

$$S, T \ ::= \ \ Id \coloneqq Expr$$

A simple language

$$
\begin{array}{rcl}
S, T & ::= & Id \coloneqq Expr \\
& | & \textbf{assert}\ Expr
\end{array}
$$

A simple language

$$
\begin{aligned}
S, T \; ::= \quad & Id \coloneqq Expr \\
\mid \quad & \textbf{assert } Expr \\
\mid \quad & \textbf{assume } Expr
\end{aligned}
$$

A simple language

$$
\begin{aligned}
S, T \;::=\;& Id \coloneqq Expr \\
\mid\;& \textbf{assert}\ Expr \\
\mid\;& \textbf{assume}\ Expr \\
\mid\;& S\ ;\ T
\end{aligned}
$$

A simple language

$$
\begin{aligned}
S, T \;::=\; & \; Id \coloneqq Expr \\
\mid \; & \; \textbf{assert } Expr \\
\mid \; & \; \textbf{assume } Expr \\
\mid \; & \; S \; ; \; T \\
\mid \; & \; S \; \square \; T
\end{aligned}
$$

Exercise

Is this simple language rich enough to encode if-statements?

if *B* **then** *S* **else** *T* **end**

$$
\begin{array}{rcl}
S, T & ::= & Id \coloneqq Expr \\
& | & \textbf{assert}\ Expr \\
& | & \textbf{assume}\ Expr \\
& | & S\ ;\ T \\
& | & S\ \square\ T
\end{array}
$$

Exercise

Is this simple language rich enough to encode if-statements?

if B **then** S **else** T **end**

Yes!

$$(\textbf{assume}\ B\ ;\ S)\ \square\ (\textbf{assume}\ \neg B\ ;\ T)$$

Weakest precondition

$$wp(S, Q)$$

Weakest precondition

$$wp(S, Q)$$

characterizes all pre-states from which

Weakest precondition

$$wp(S, Q)$$

characterizes all pre-states from which every non-blocking execution of S

Weakest precondition

$$wp(S, Q)$$

characterizes all pre-states from which
every non-blocking execution of S
does not go wrong and terminates

Weakest precondition

$$wp(S, Q)$$

characterizes all pre-states from which
every non-blocking execution of S
does not go wrong and terminates
in a state satisfying Q

Weakest precondition

$$wp(x := Expr, Q) \equiv Q[x := Expr]$$

Weakest precondition

$$wp(x \coloneqq Expr, Q) \equiv Q[x \coloneqq Expr]$$

Example:

$$wp(x \coloneqq y, 0 < x) \equiv 0 < y$$

Weakest precondition

$$wp(x \coloneqq Expr, Q) \equiv Q[x \coloneqq Expr]$$

Exercise:

$$wp(x \coloneqq x + 1, 0 < x) \equiv$$

Weakest precondition

$$wp(x := Expr, Q) \equiv Q[x := Expr]$$

Exercise:

$$wp(x := x + 1, 0 < x) \equiv 0 < x + 1$$

Weakest precondition

$$wp(x \coloneqq Expr, Q) \equiv Q[x \coloneqq Expr]$$

Exercise:

$$wp(x \coloneqq x + 1, 0 < x) \equiv 0 < x + 1$$

$$wp(x \coloneqq x + 1, z - 2 < y) \equiv$$

Weakest precondition

$$wp(x \coloneqq Expr, Q) \equiv Q[x \coloneqq Expr]$$

Exercise:

$$wp(x \coloneqq x + 1, 0 < x) \equiv 0 < x + 1$$

$$wp(x \coloneqq x + 1, z \; - 2 < y) \equiv z \; - 2 < y$$

Weakest precondition

$$wp(x := Expr, Q) \equiv Q[x := Expr]$$

$$wp(\mathbf{assert}\ Expr, Q) \equiv Expr \land Q$$

Weakest precondition

$$wp(x := Expr, Q) \equiv Q[x := Expr]$$

$$wp(\textbf{assert}\ Expr, Q) \equiv Expr \wedge Q$$

Exercise:

$$wp(\textbf{assert false}, Q) \equiv$$

Weakest precondition

$$wp(x := Expr, Q) \equiv Q[x := Expr]$$

$$wp(\textbf{assert}\ Expr, Q) \equiv Expr \land Q$$

Exercise:

$$wp(\textbf{assert false}, Q) \equiv false$$

Weakest precondition

$$wp(x := Expr, Q) \equiv Q[x := Expr]$$

$$wp(\mathbf{assert}\ Expr, Q) \equiv Expr \wedge Q$$

$$wp(\mathbf{assume}\ Expr, Q) \equiv Expr \Rightarrow Q$$

Weakest precondition

$$wp(x := Expr, Q) \equiv Q[x := Expr]$$

$$wp(\textbf{assert}\ Expr, Q) \equiv Expr \wedge Q$$

$$wp(\textbf{assume}\ Expr, Q) \equiv Expr \Rightarrow Q$$

$$wp(S\ ;\ T, Q) \equiv wp(S, wp(T, Q))$$

Weakest precondition

$$wp(x := Expr, Q) \equiv Q[x := Expr]$$

$$wp(\textbf{assert}\ Expr, Q) \equiv Expr \land Q$$

$$wp(\textbf{assume}\ Expr, Q) \equiv Expr \Rightarrow Q$$

$$wp(S\ ;\ T, Q) \equiv wp(S, wp(T, Q))$$

$$wp(S \ \square\ T, Q) \equiv wp(S, Q) \land wp(T, Q)$$

Exercise

What is the weakest precondition of the if-statement?

```
if B then S else T end
```

$$wp(x := Expr, Q) \equiv Q[x := Expr]$$

$$wp(\textbf{assert}\ Expr, Q) \equiv Expr \wedge Q$$

$$wp(\textbf{assume}\ Expr, Q) \equiv Expr \Rightarrow Q$$

$$wp(S\ ;\ T, Q) \equiv wp(S, wp(T, Q))$$

$$wp(S \,\square\, T, Q) \equiv wp(S, Q) \wedge wp(T, Q)$$

Exercise

What is the weakest precondition of the if-statement?

$$(\textbf{assume}\ B\ ;\ S)\ \square\ (\textbf{assume}\ \neg B\ ;\ T)$$

$$wp(x := Expr, Q) \equiv Q[x := Expr]$$

$$wp(\textbf{assert}\ Expr, Q) \equiv Expr \land Q$$

$$wp(\textbf{assume}\ Expr, Q) \equiv Expr \Rightarrow Q$$

$$wp(S\ ;\ T, Q) \equiv wp(S, wp(T, Q))$$

$$wp(S\ \square\ T, Q) \equiv wp(S, Q) \land wp(T, Q)$$

Exercise

What is the weakest precondition of the if-statement?

$$(\textbf{assume}\ \ B\ ;\ S)\ \square\ (\textbf{assume}\ \neg B\ ;\ T)$$

$$B \Rightarrow wp(S, Q) \land \neg B \Rightarrow wp(T, Q)$$

Verification condition

For a given program *P*, the verification condition is

$$wp(P, true)$$

Verification condition

For a given program P, the verification condition is

$$wp(P, true)$$

This condition is valid if and only if P is free of errors

Exercise

What is the weakest precondition of the program?

$$(\textbf{assume}\ \ B\ ;\ x\ \ \coloneqq 1)\ \square\ (\textbf{assume}\ \neg B\ ;\ x\ \ \coloneqq -1)$$

Assume that Q is $x = 2$.

$$B \Rightarrow wp(S, Q) \wedge \neg B \Rightarrow wp(T, Q)$$

$$wp(x \coloneqq Expr, Q) \equiv Q[x \coloneqq Expr]$$

Exercise

What is the weakest precondition of the program?

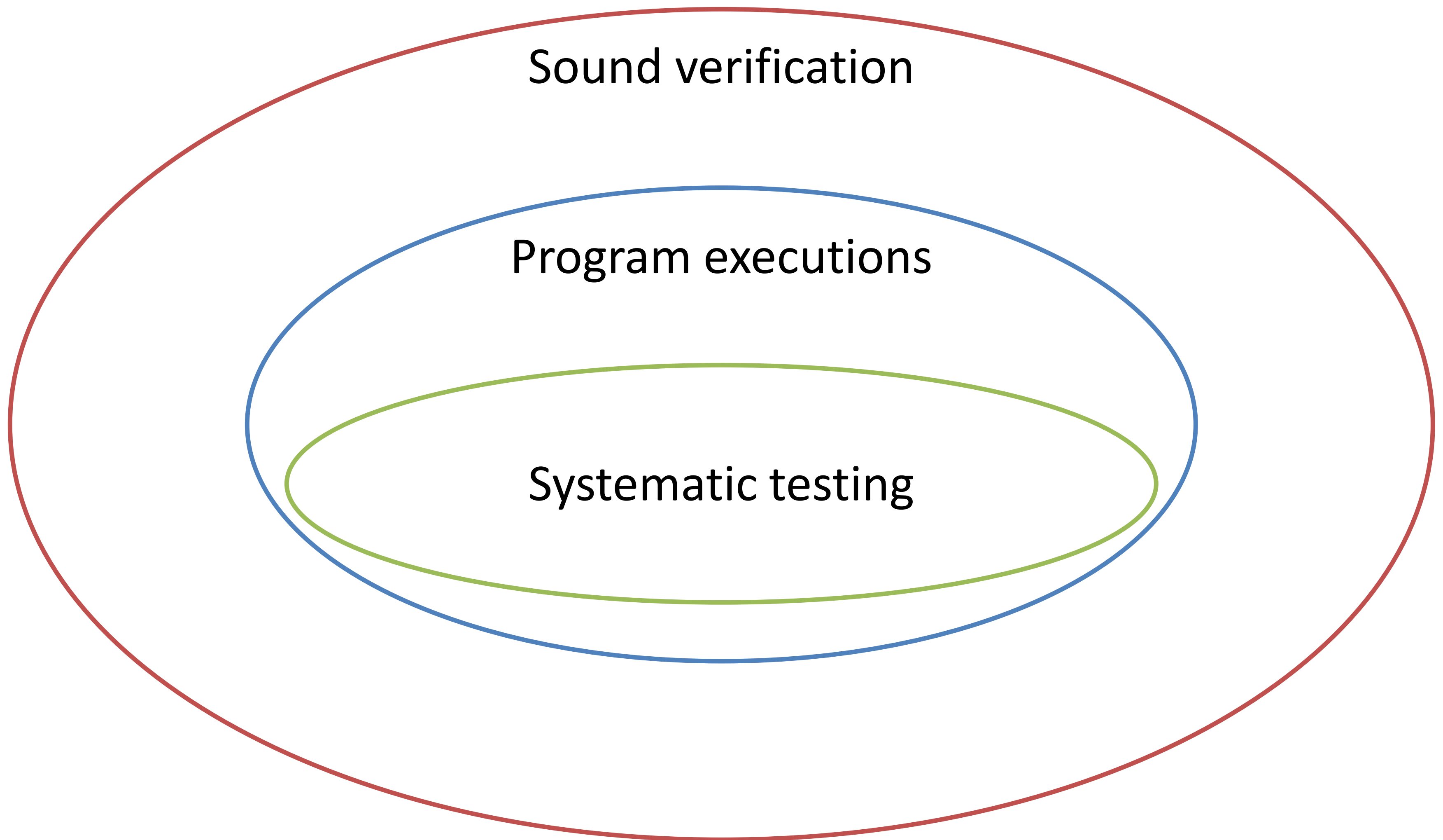
$$(\textbf{assume}\ \ B\ ;\ x\ \ \coloneqq 1)\ \square\ (\textbf{assume}\ \neg B\ ;\ x\ \ \coloneqq -1)$$

Assume that Q is $x = 2$.

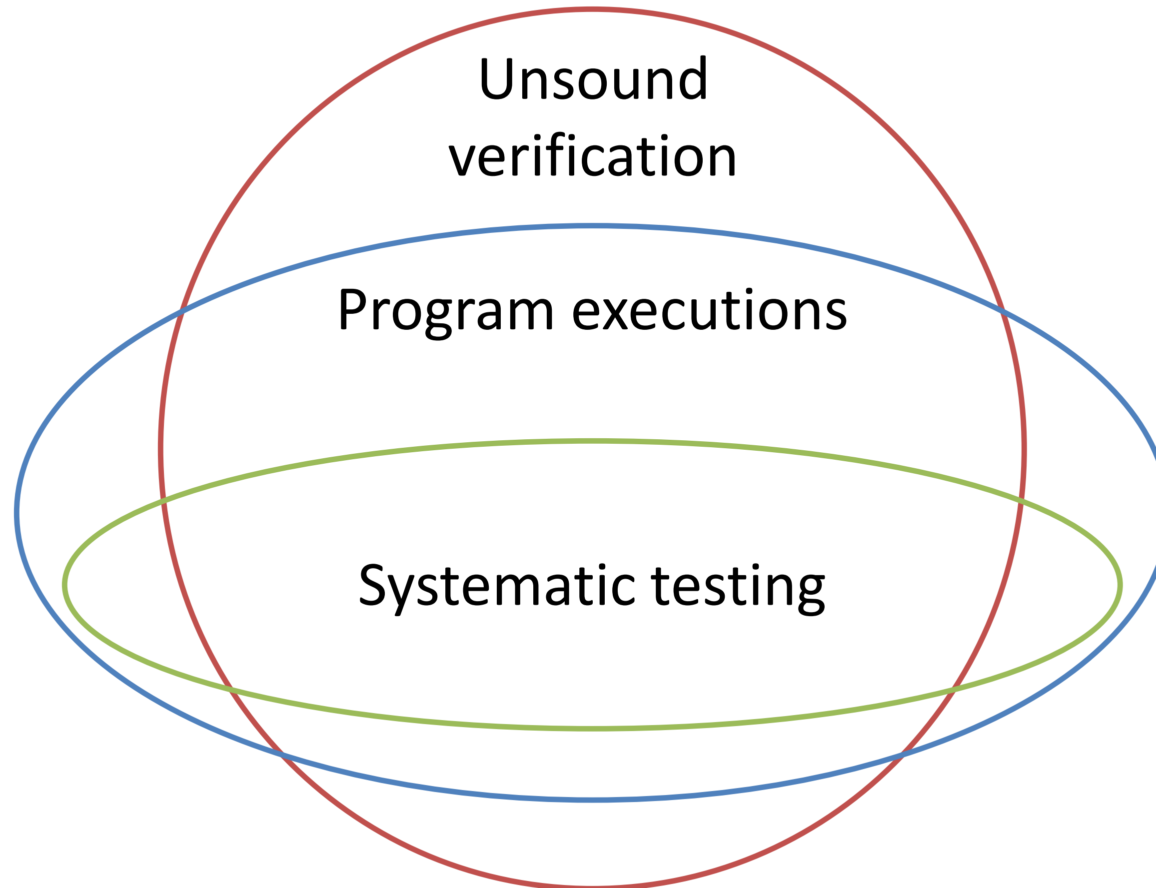
$$false$$

Fun:
Guiding Dynamic Symbolic Execution toward Unverified Program Executions

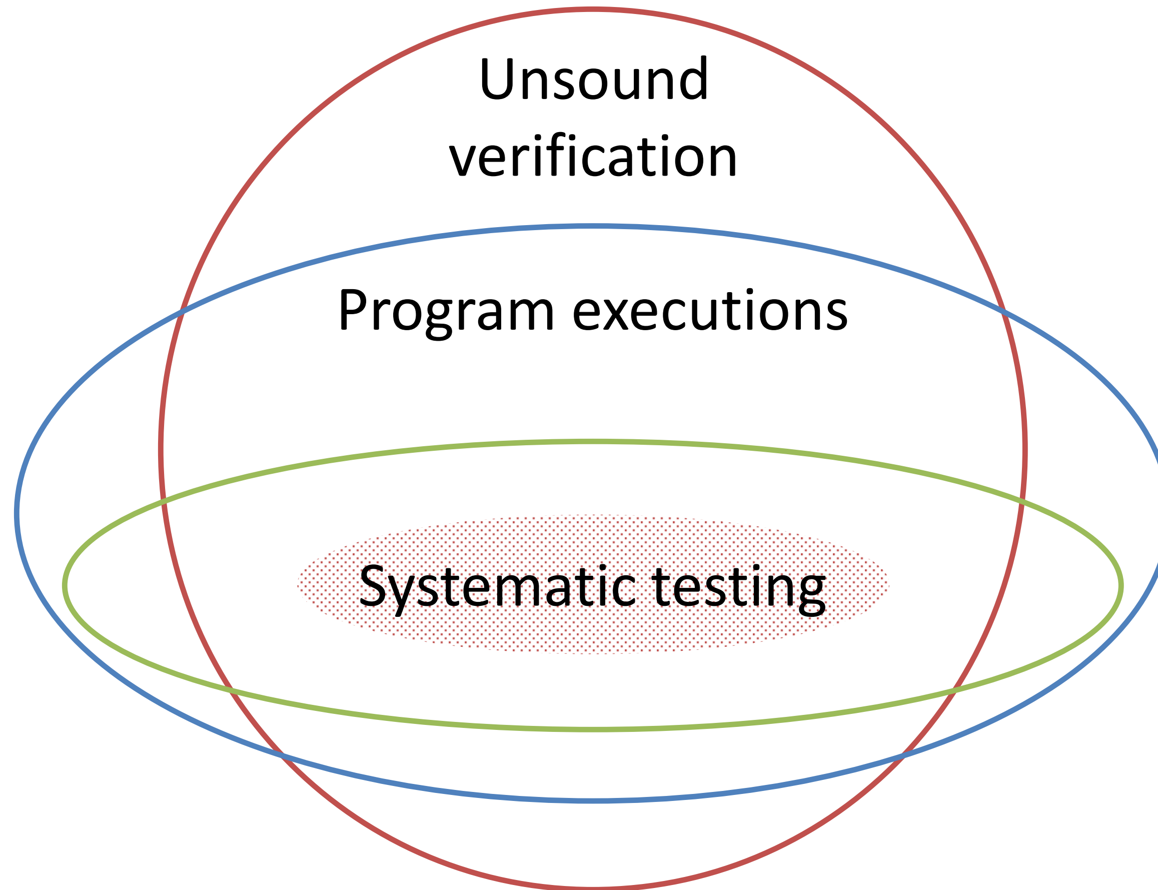
Verification and testing



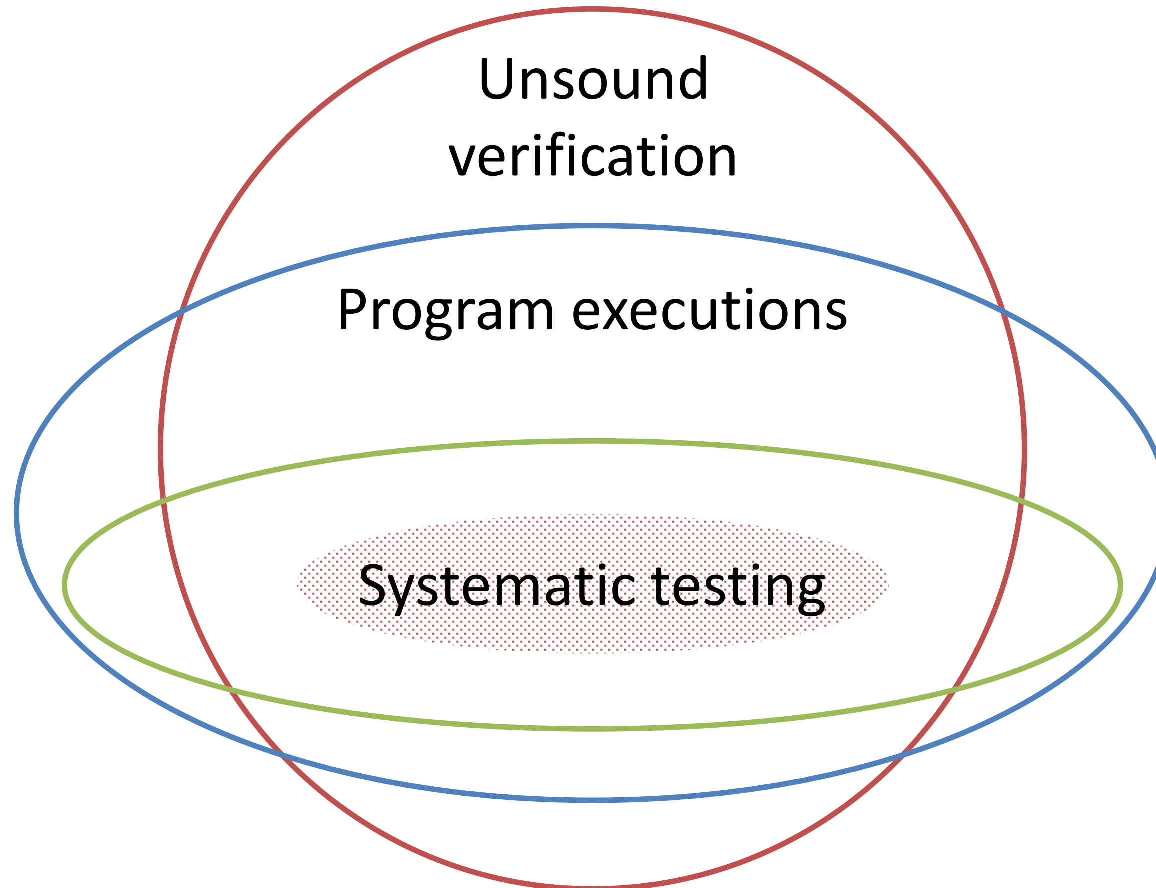
Verification and testing



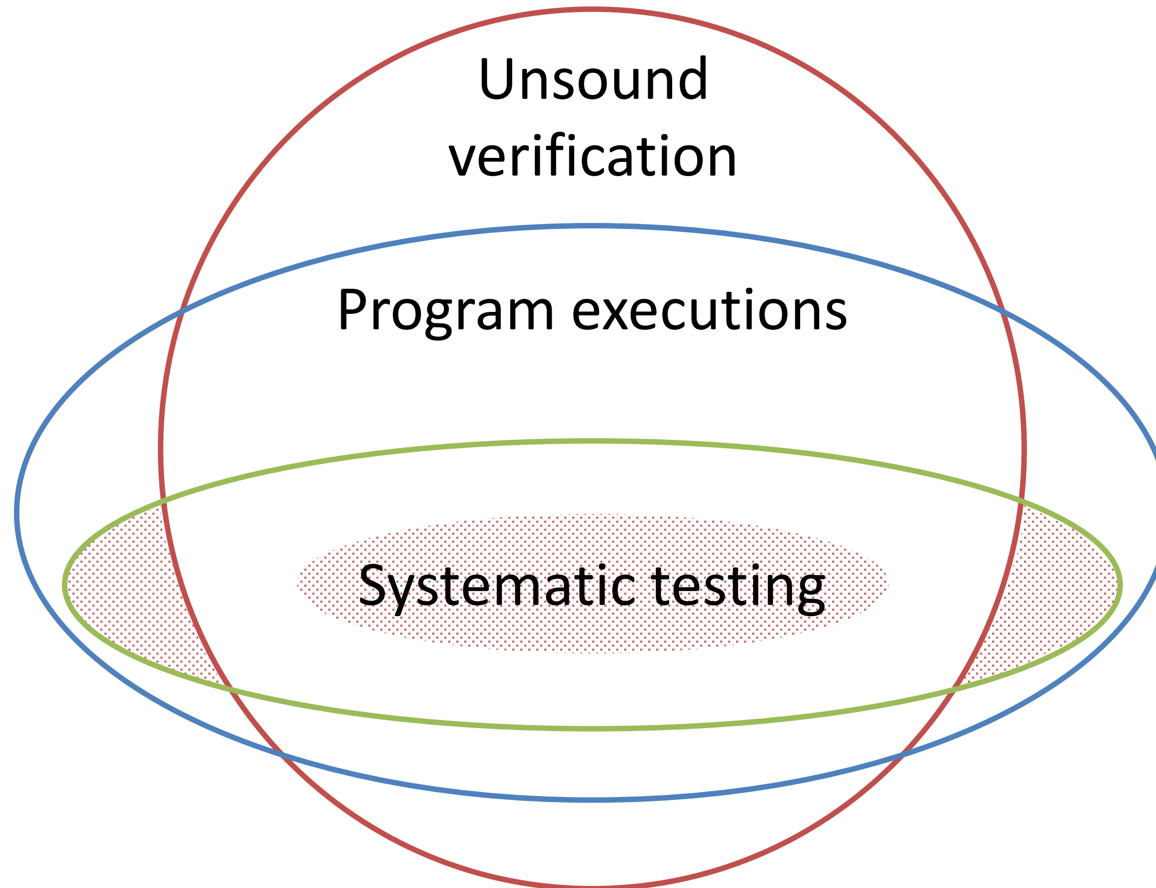
So far…



In our work…



In our work...



Static analysis

- Effective in detecting software errors
- Increasingly applied in industry
- Come in a wide range

Static analysis

- Effective in detecting software errors
- Increasingly applied in industry
- Come in a wide range

but…

Compromises

- Reduce the annotation overhead
- Reduce the number of false positives
- Increase performance
- Preserve modularity

Compromises

- Not checking program properties
- Making unjustified assumptions
- Being unsound

Example

C#

```
void Deposit(int amount) {
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assert balance >= old(balance);
}
```

Example

```
void Deposit(int amount) {
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assert balance >= old(balance);
}
```

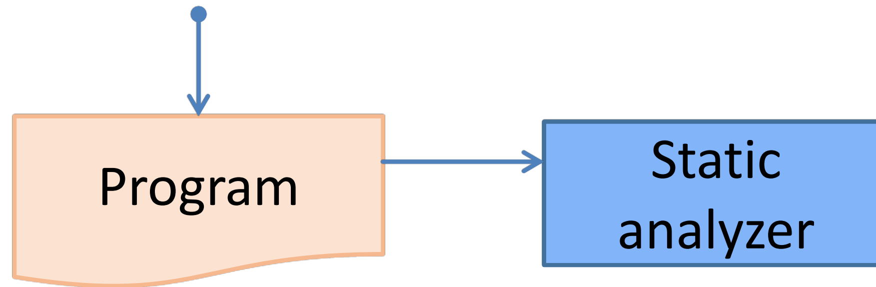
Example

```
void Deposit(int amount) {
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assert balance >= old(balance);
}
```

Consequences

- Absence of errors not guaranteed
- Test effort not reduced

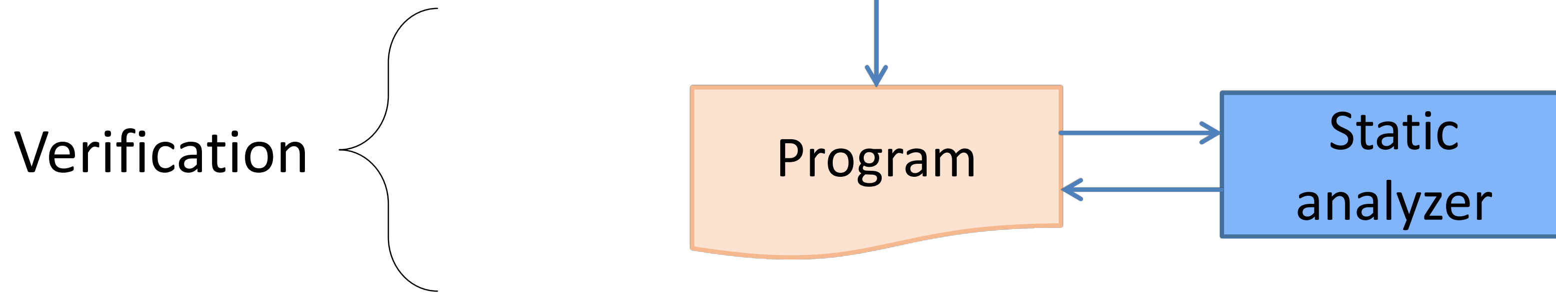
Architecture



Annotations

```
void Deposit(int amount) {
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    assumed noOverflowAdd(balance, amount) as a;
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assert balance >= old(balance) verified a;
}
```

Architecture

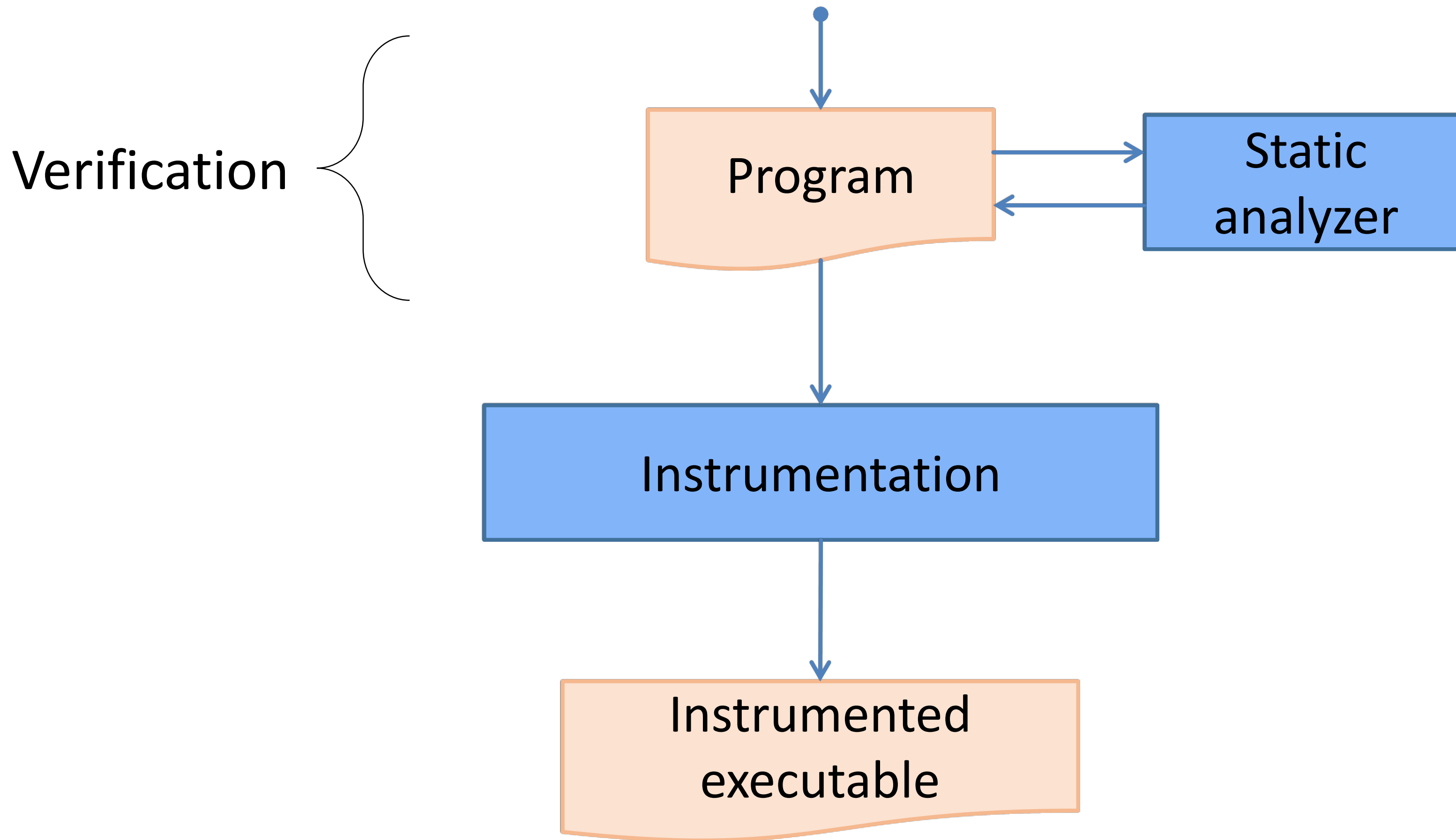


Instrumentation

Rewriter

```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assume a ⟹ balance >= old(balance);
  assert balance >= old(balance);
}
```

Architecture



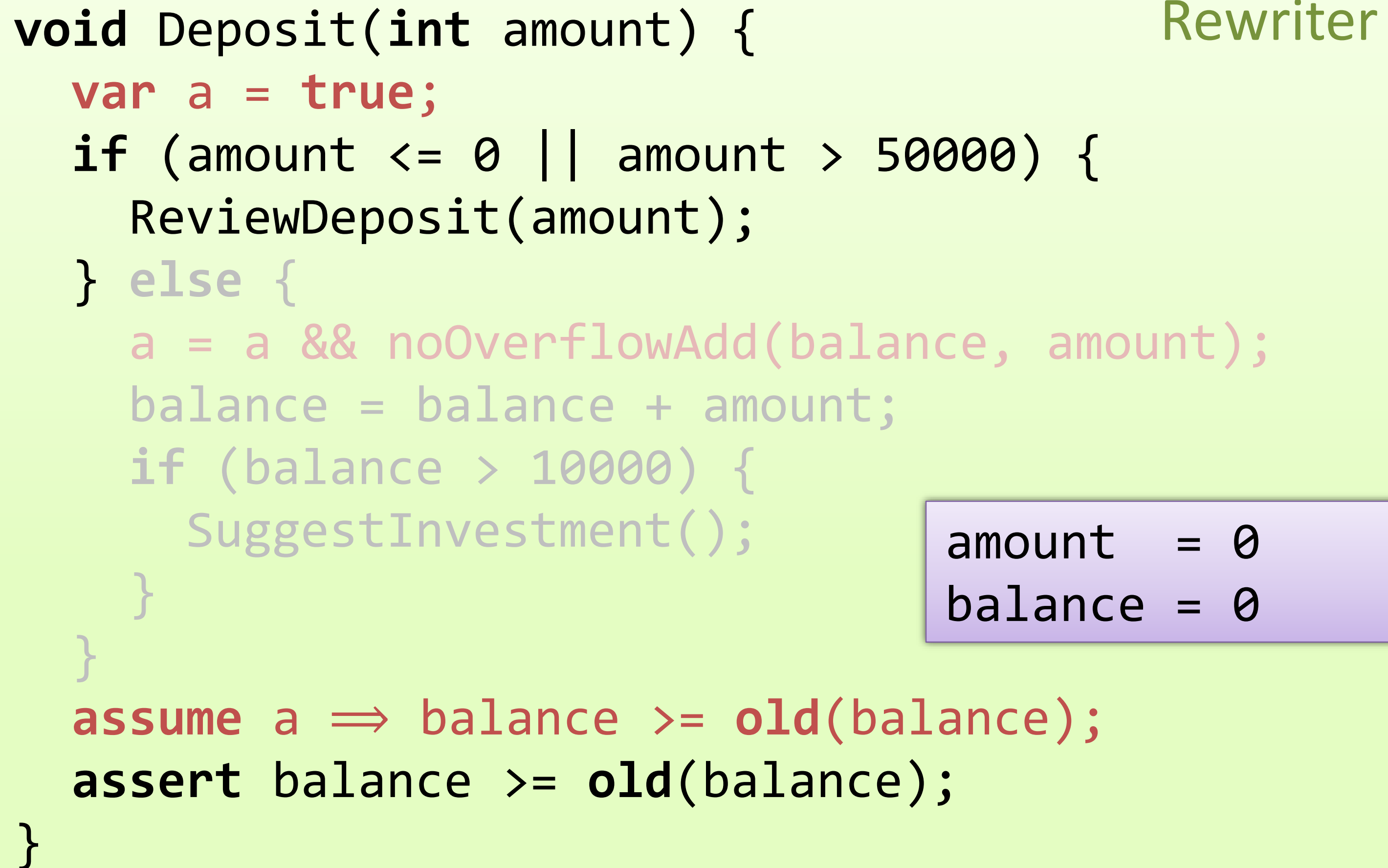
Dynamic test generation

Rewriter

```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assume a ⟹ balance >= old(balance);
  assert balance >= old(balance);
}
```

```
amount  = 0
balance = 0
```

Dynamic test generation



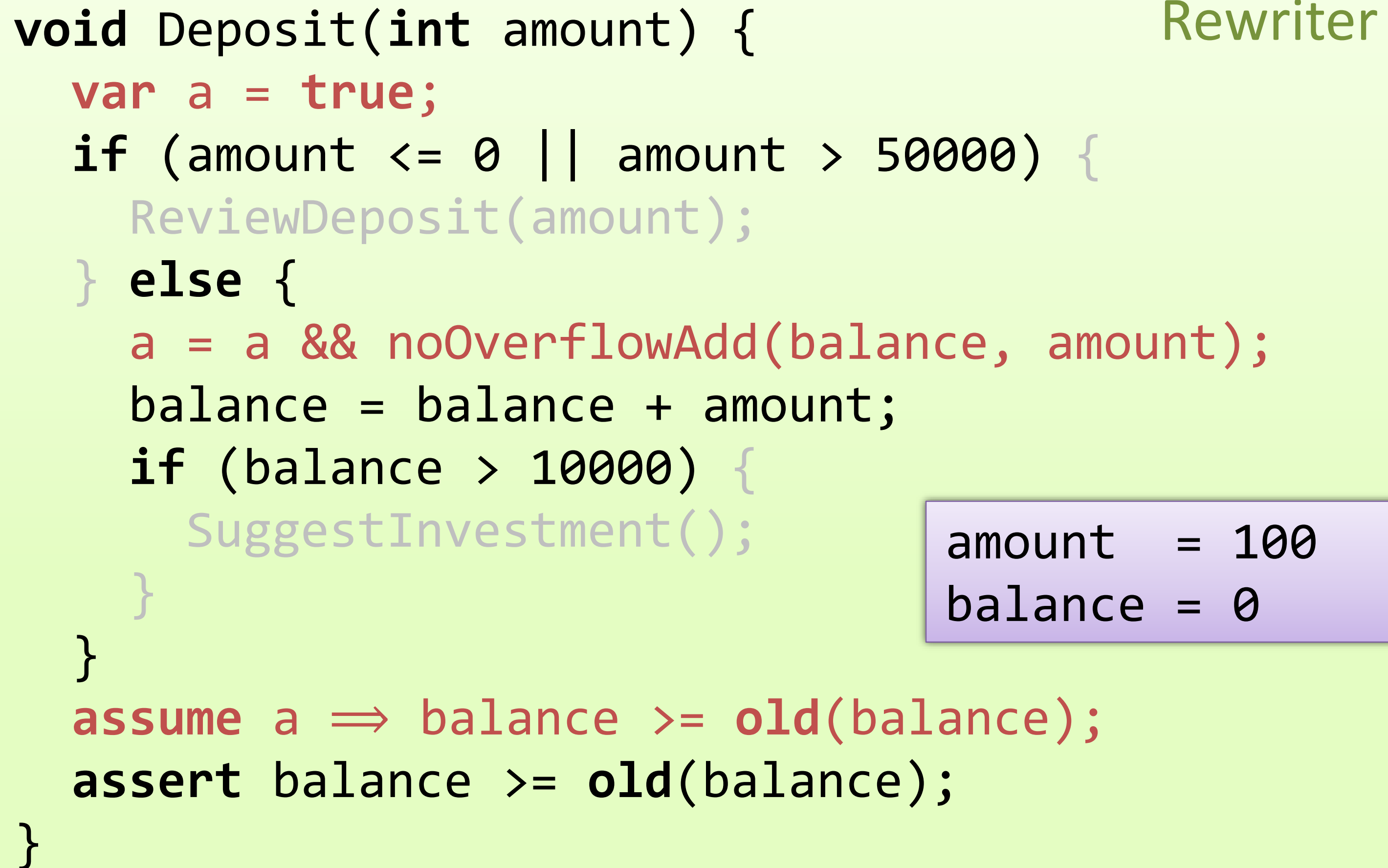
Dynamic test generation

Rewriter

```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assume a ⟹ balance >= old(balance);
  assert balance >= old(balance);
}
```

```
amount  = 100
balance = 0
```

Dynamic test generation



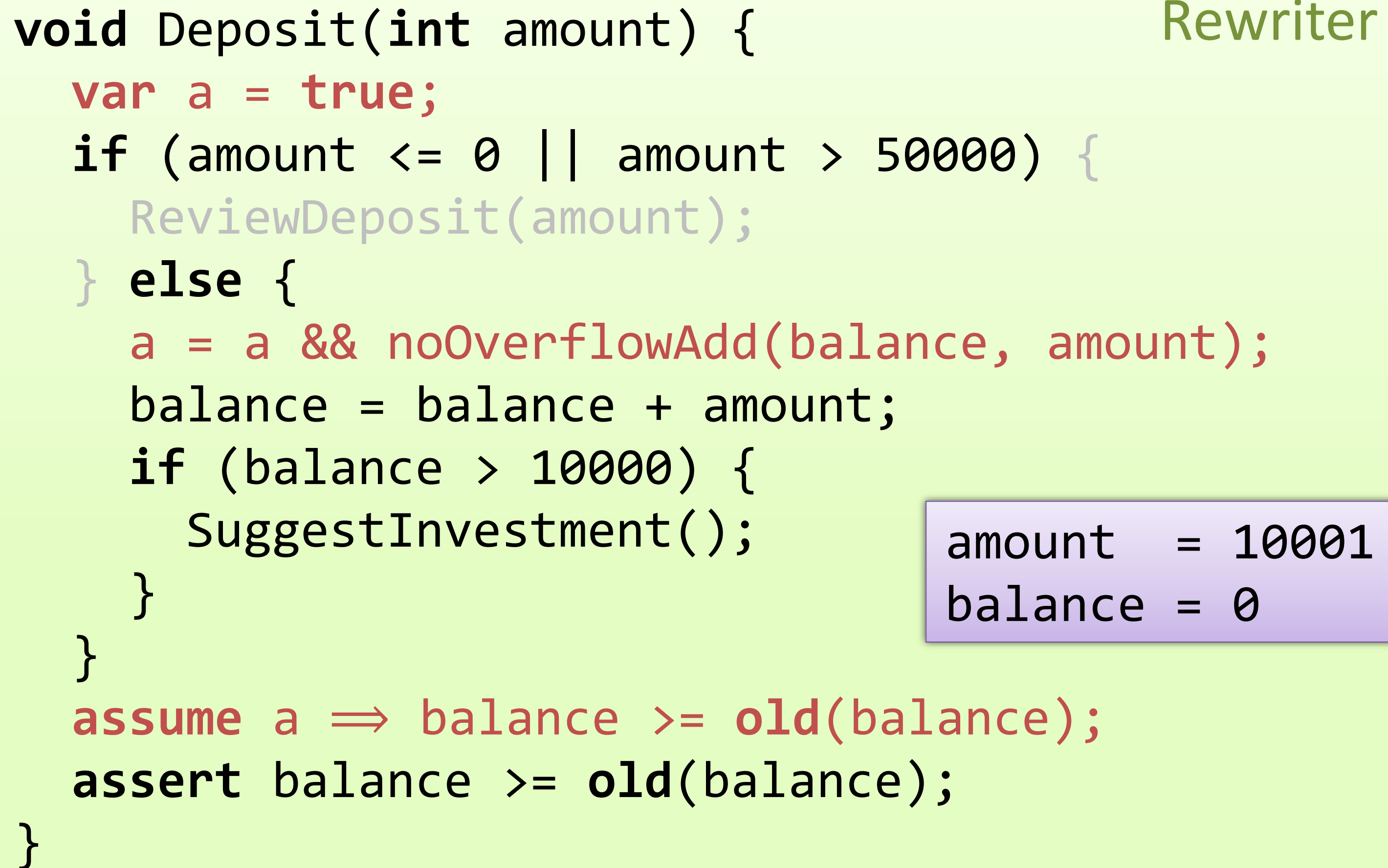
Dynamic test generation

Rewriter

```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assume a ⟹ balance >= old(balance);
  assert balance >= old(balance);
}
```

```
amount  = 10001
balance = 0
```

Dynamic test generation



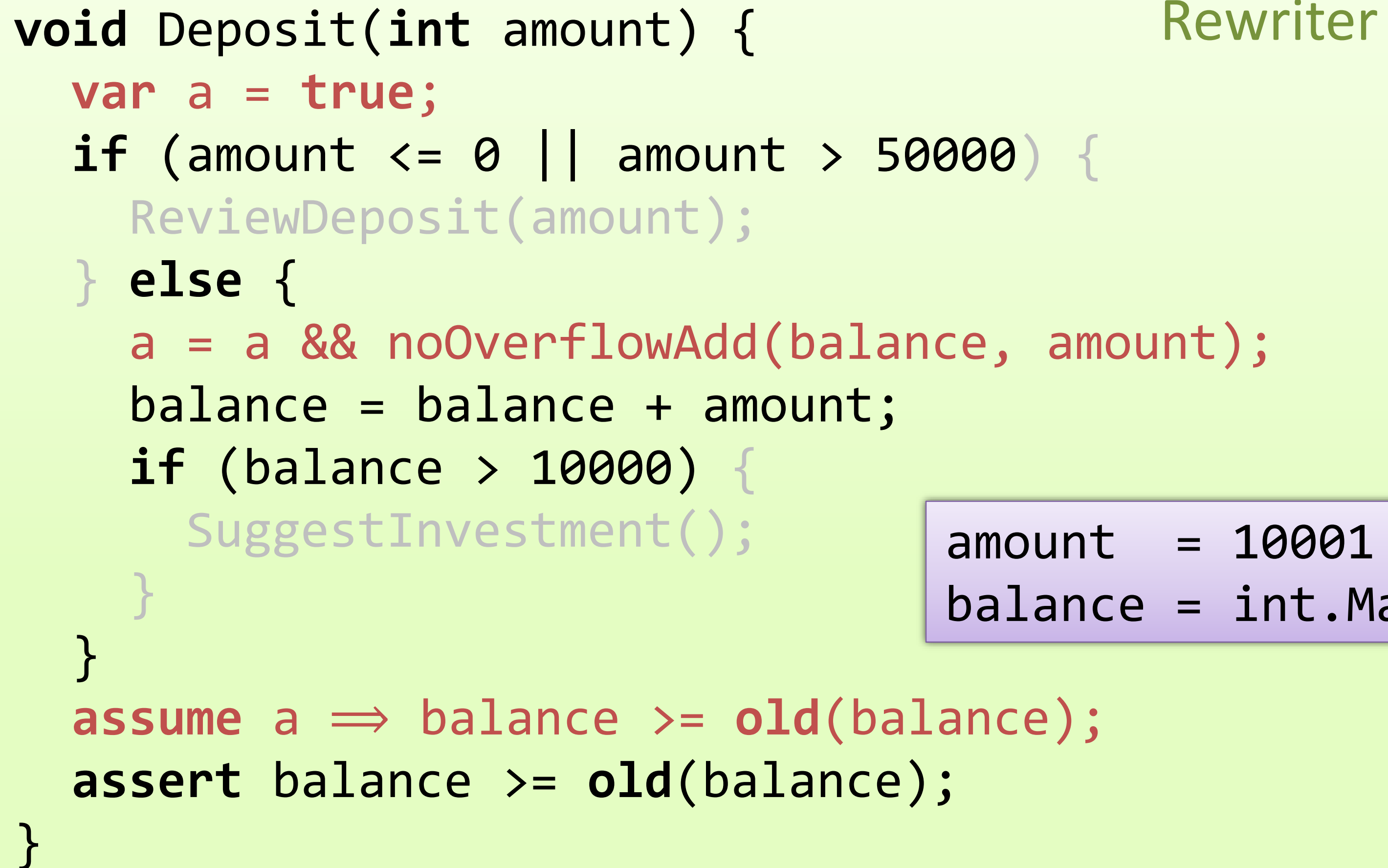
Dynamic test generation

Rewriter

```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assume a ⟹ balance >= old(balance);
  assert balance >= old(balance);
}
```

```
amount  = 10001
balance = int.MaxValue
```

Dynamic test generation



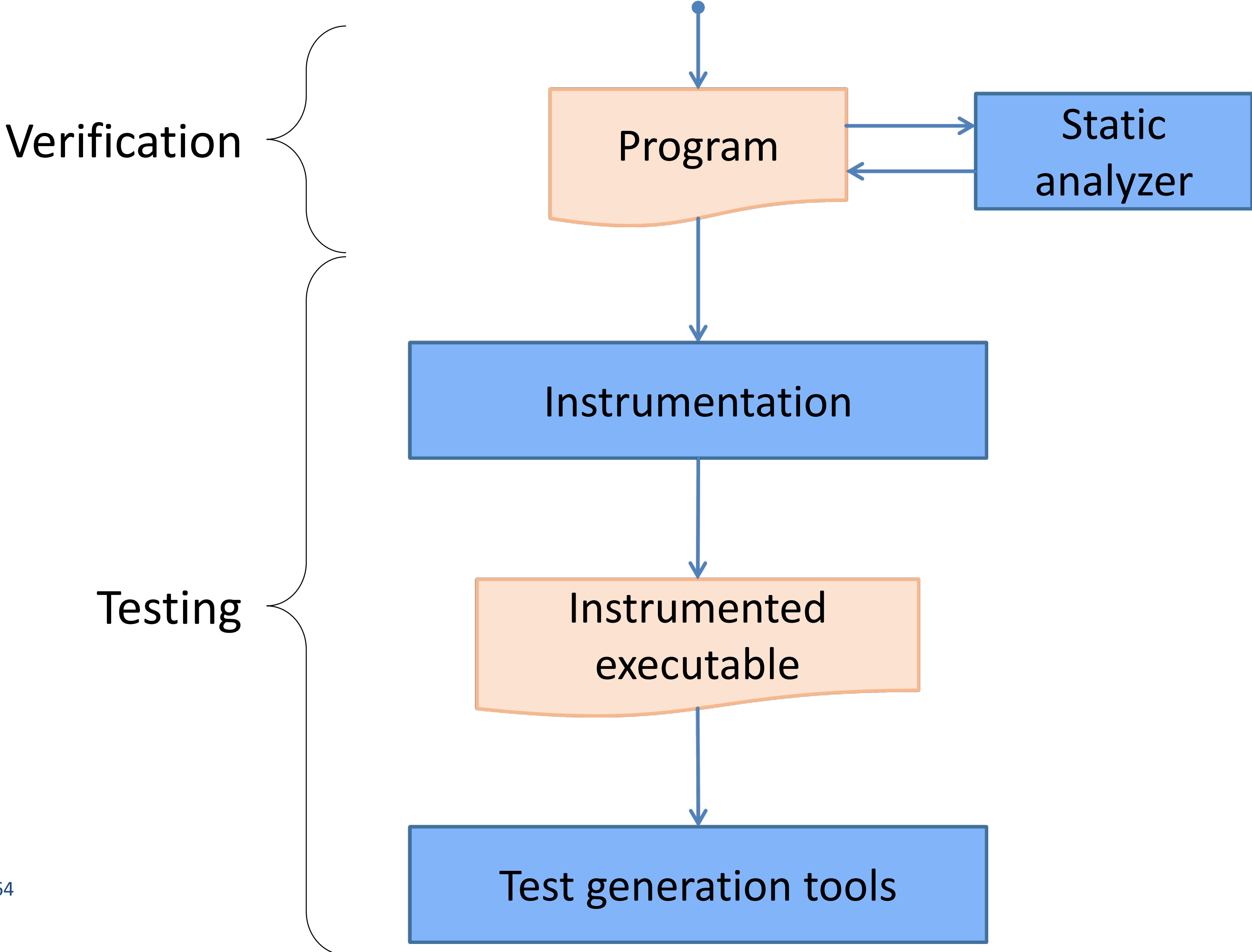
Dynamic test generation

Rewriter

```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assume a ⟹ balance >= old(balance);
  assert balance >= old(balance);
}
```

```
amount  = ...
balance = ...
```

Architecture

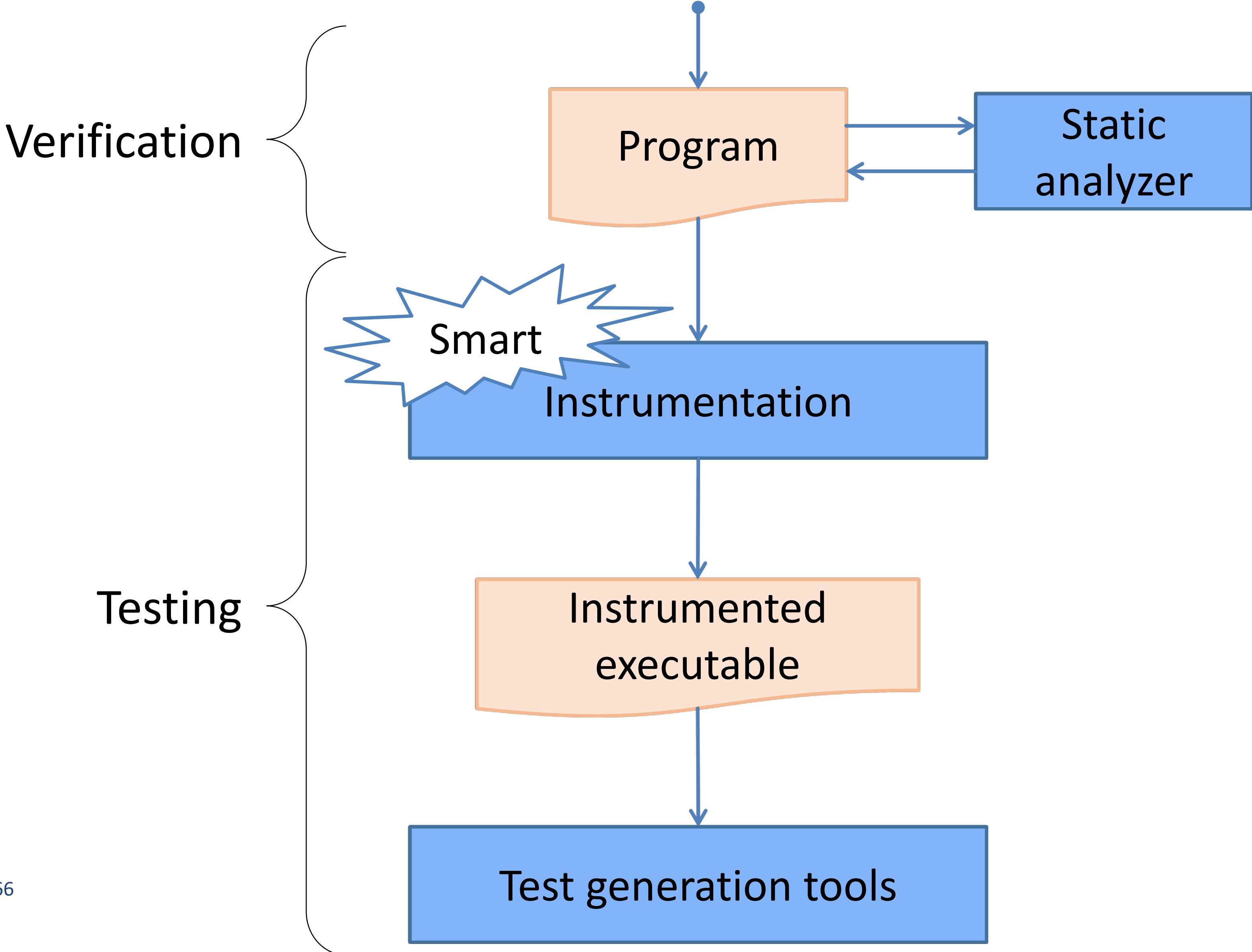


Problem

Rewriter

```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assume a ⟹ balance >= old(balance);
  assert balance >= old(balance);
}
```

Architecture



Smart instrumentation

Idea:

- Propagate conditions about unverified executions higher up in the control flow

Under the hood:

- Compute an abstraction of the program
- Infer conditions about unverified executions
- Instrument the concrete program

Abstraction

```
void Deposit(int amount) {

  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    assumed noOverflowAdd(balance, amount) as a;
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assert balance >= old(balance) verified a;
}
```

Abstraction

```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assert balance >= old(balance) verified a;
}
```

Smart rewriter

Abstraction

```
void Deposit {
  var a = true;
  if (⋆) {
    call Account.ReviewDeposit;
  } else {
    a = a && ⋆;

    if (⋆) {
      call Account.SuggestInvestment;
    }
  }
  assert ⋆ verified a;
}
```

Smart rewriter

Abstraction

```
void Deposit {
  var a = true;
  if (*) {
    call Account.ReviewDeposit;
  } else {
    a = a && *;

    if (*) {
      call Account.SuggestInvestment;
    }
  }
  assert * verified a;
}
```

Smart rewriter

syntactic
non-deterministic
sound

Inference

```
void Deposit {
  var a = true;
  if (★) {
    call Account.ReviewDeposit;
  } else {
    a = a && ★;

    if (★) {
      call Account.SuggestInvestment;
    }
  }
  assert ★ verified a;
}
```

Smart rewriter

Inference

Smart rewriter

```
void Deposit {
  var a = true;
  if (⋆) {

  } else {
    a = a && ⋆;

    if (⋆) {

    }
  }
  assert ⋆ verified a;
}
```

Inference

```
void Deposit {
  var a = true;
  if (*) {

  } else {
    a = a && *;

    if (*) {

    }
  }
  assert * verified a; {false}
}
```

Smart rewriter

unverified executions satisfy **`false`**

Inference

Smart rewriter

```
void Deposit {
  var a = true;
  if (*) {

  } else {
    a = a && *;

    if (*) {

    }
  } {!a}
  assert * verified a; {false}
}
```

unverified executions satisfy !a

Inference

Smart rewriter

```
void Deposit {
  var a = true;
  if (*) {

  } else { {true}
    a = a && *;
    {!a}
    if (*) {
      {!a}
    } {!a}
  } {!a}
  assert * verified a; {false}
}
```

Inference

```
void Deposit {
  var a = true;
  if (⋆) {
    {!a}
  } else { {true}
    a = a && ⋆;
    {!a}
    if (⋆) {
      {!a}
    } {!a}
  } {!a}
  assert ⋆ verified a; {false}
}
```

Smart rewriter

Inference

Smart rewriter

```
void Deposit {
  var a = true; {true}
  if (⋆) {
    {!a}
  } else { {true}
    a = a && ⋆;
    {!a}
    if (⋆) {
      {!a}
    } {!a}
  } {!a}
  assert ⋆ verified a; {false}
}
```

Inference

Smart rewriter

```
void Deposit { {true}
  var a = true; {true}
  if (⋆) {
    {!a}
  } else { {true}
    a = a && ⋆;
    {!a}
    if (⋆) {
      {!a}
    } {!a}
  } {!a}
  assert ⋆ verified a; {false}
}
```

Inference

```
void Deposit { {true}
  var a = true; {true}
  if (⋆) {
    {!a}
  } else { {true}
    a = a && ⋆;
    {!a}
    if (⋆) {
      {!a}
    } {!a}
  } {!a}
  assert ⋆ verified a; {false}
}
```

Smart rewriter

$wp(\textbf{assert} * \textbf{verified}\ A, Q) \equiv A \wedge Q$

Inference

```
void Deposit { {true}
  var a = true; {true}
  if (*) {
    {!a}
  } else { {true}
    a = a && *;
    {!a}
    if (*) {
      {!a}
    } {!a}
  } {!a}
  assert * verified a; {false}
}
```

Smart rewriter

$$MAY(S) \equiv \neg wp(S, true)$$

Inference

Smart rewriter

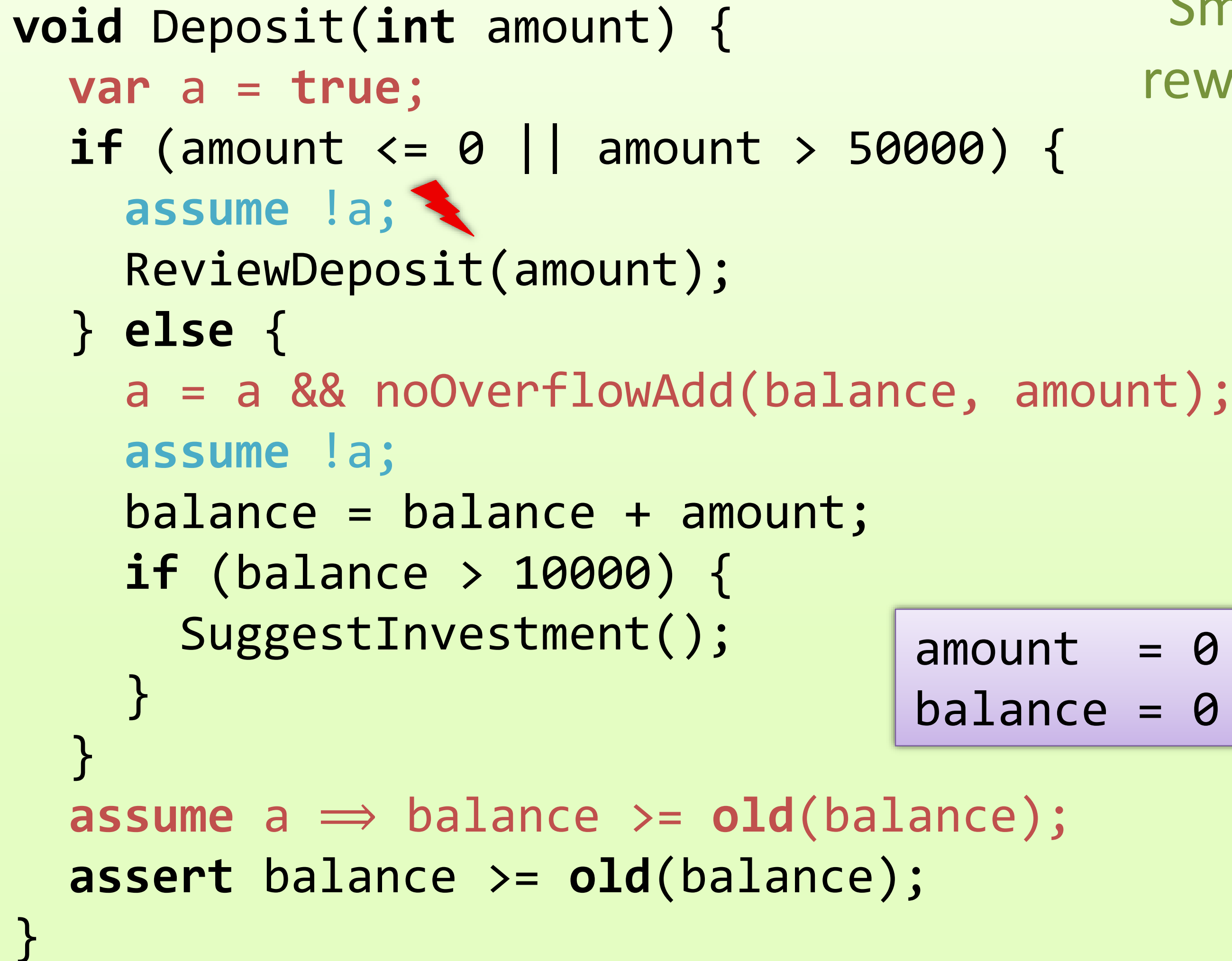
```
void Deposit { {true}
  var a = true; {true}
  if (⋆) {
    {!a}
  } else { {true}
    a = a && ⋆;
    {!a}
    if (⋆) {
      {!a}
    } {!a}
  } {!a}
  assert ⋆ verified a; {false}
}
```

Smart instrumentation

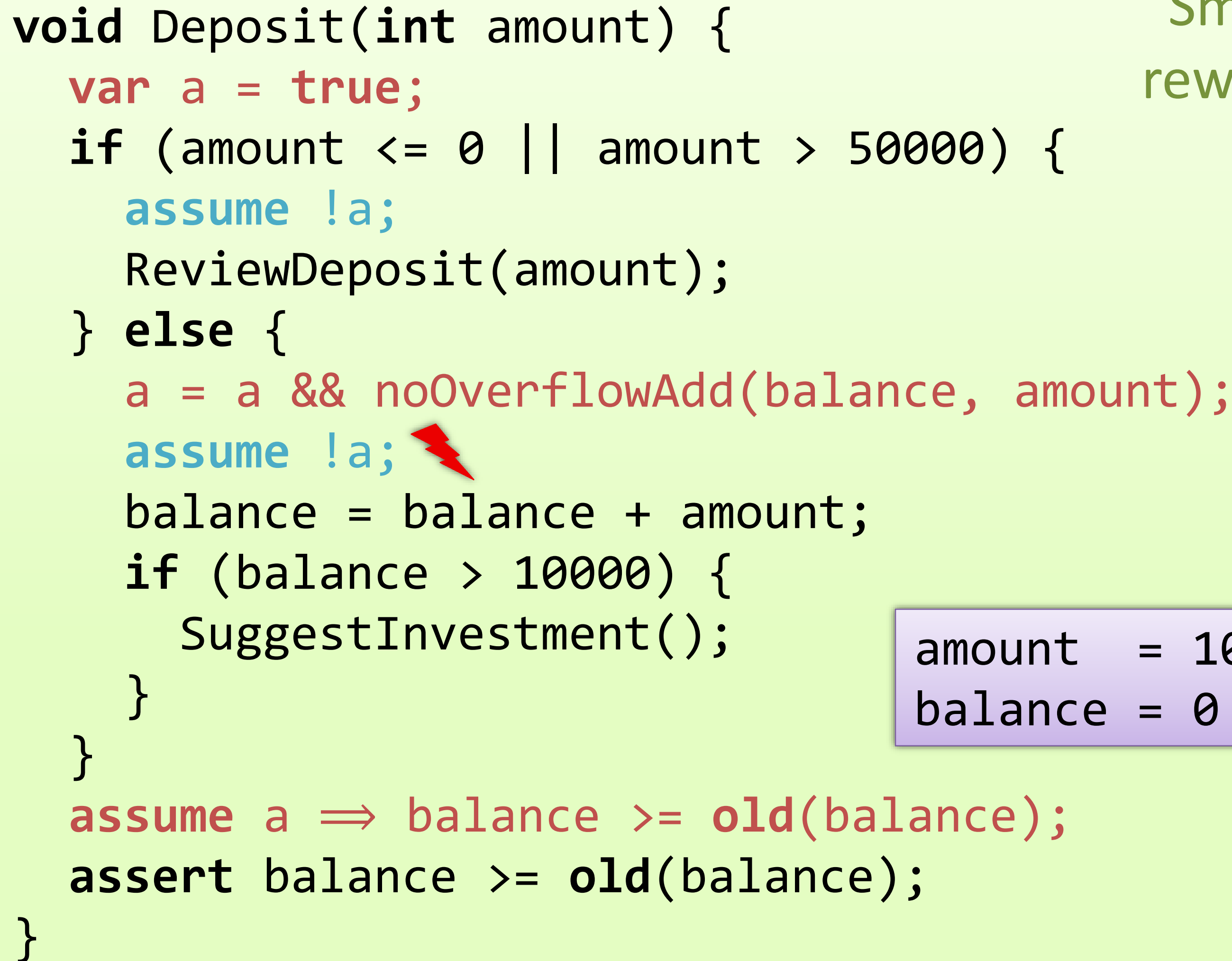
Smart rewriter

```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    assume !a;
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    assume !a;
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assume a ⟹ balance >= old(balance);
  assert balance >= old(balance);
}
```

Dynamic test generation



Dynamic test generation



Dynamic test generation

Smart rewriter

```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    assume !a;
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    assume !a;
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assume a ⟹ balance >= old(balance);
  assert balance >= old(balance);
}
```

```
amount  = 10001
balance = int.MaxValue
```

Dynamic test generation

Smart rewriter

```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    assume !a;
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    assume !a;
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assume a ⟹ balance >= old(balance);
  assert balance >= old(balance);
}
```

```
amount  = 10001
balance = 0
```

Dynamic test generation

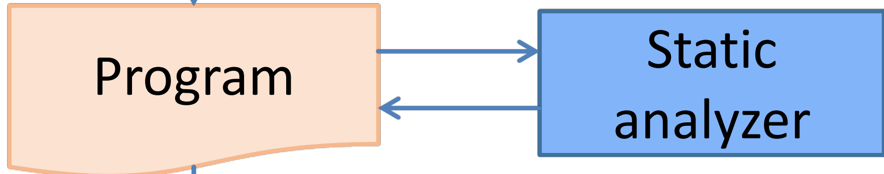
Smart rewriter

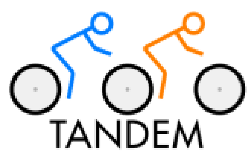
```
void Deposit(int amount) {
  var a = true;
  if (amount <= 0 || amount > 50000) {
    assume !a;
    ReviewDeposit(amount);
  } else {
    a = a && noOverflowAdd(balance, amount);
    assume !a;
    balance = balance + amount;
    if (balance > 10000) {
      SuggestInvestment();
    }
  }
  assume a ⟹ balance >= old(balance);
  assert balance >= old(balance);
}
```

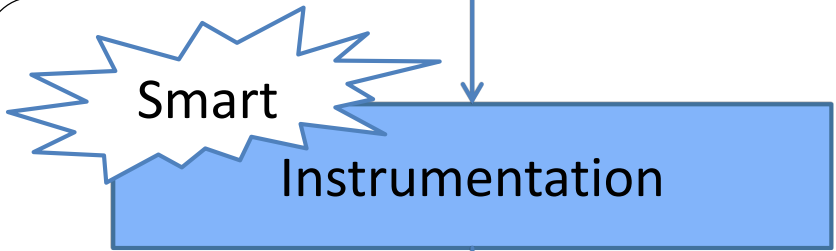
```
amount  = ...
balance = ...
```

Architecture

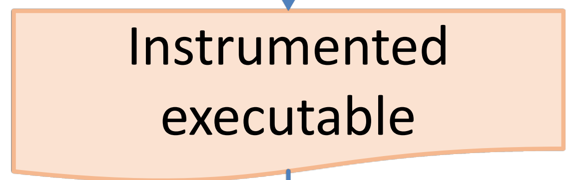
Verification

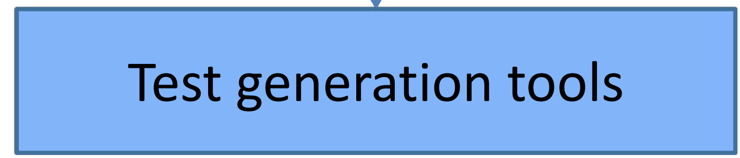






Testing





Evaluation highlights

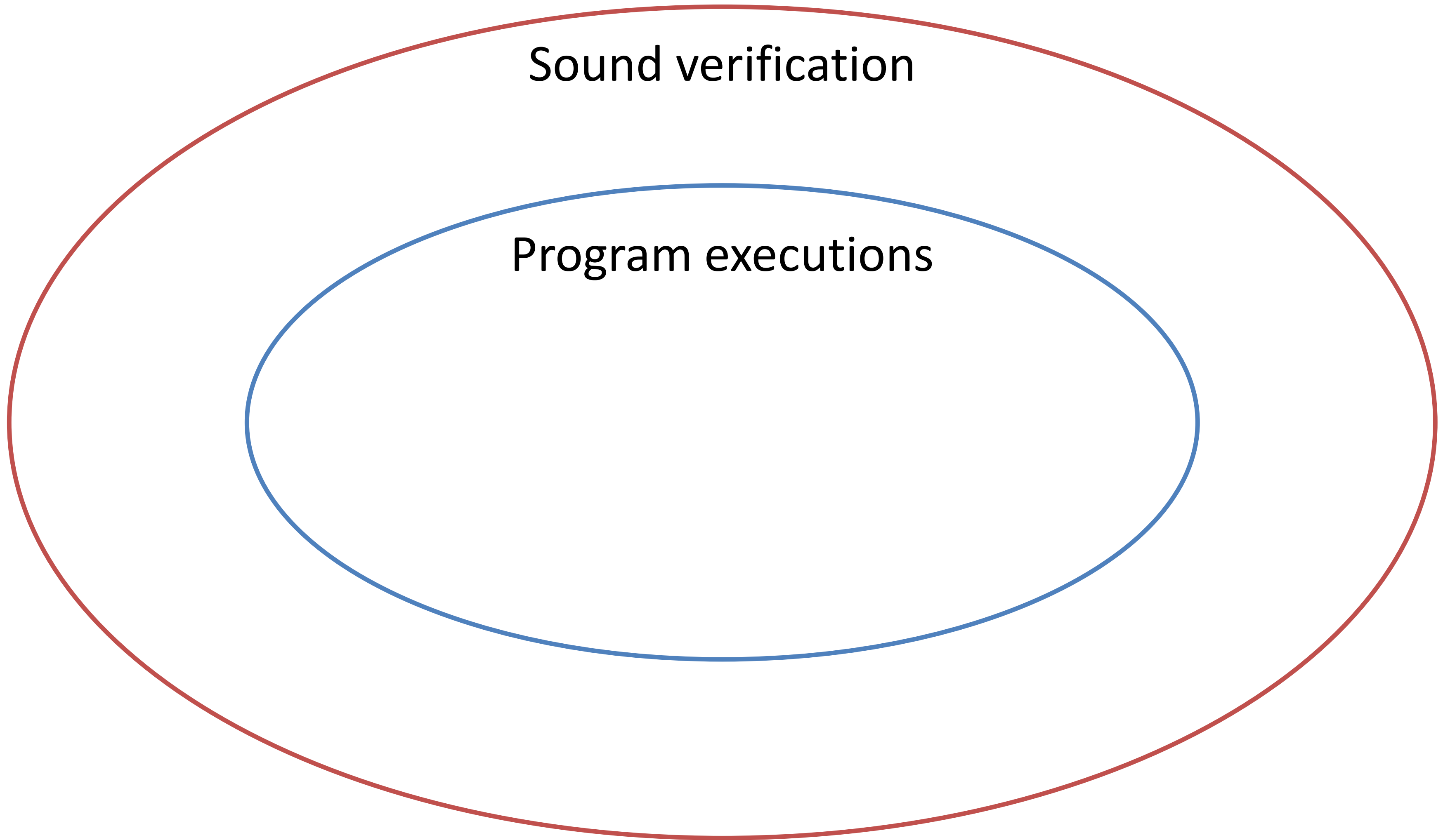
- 19.2% smaller test suites
- 7.1% more unverified executions
- 51.7% shorter testing time
- 29.3% fewer exploration bounds

Takeaways

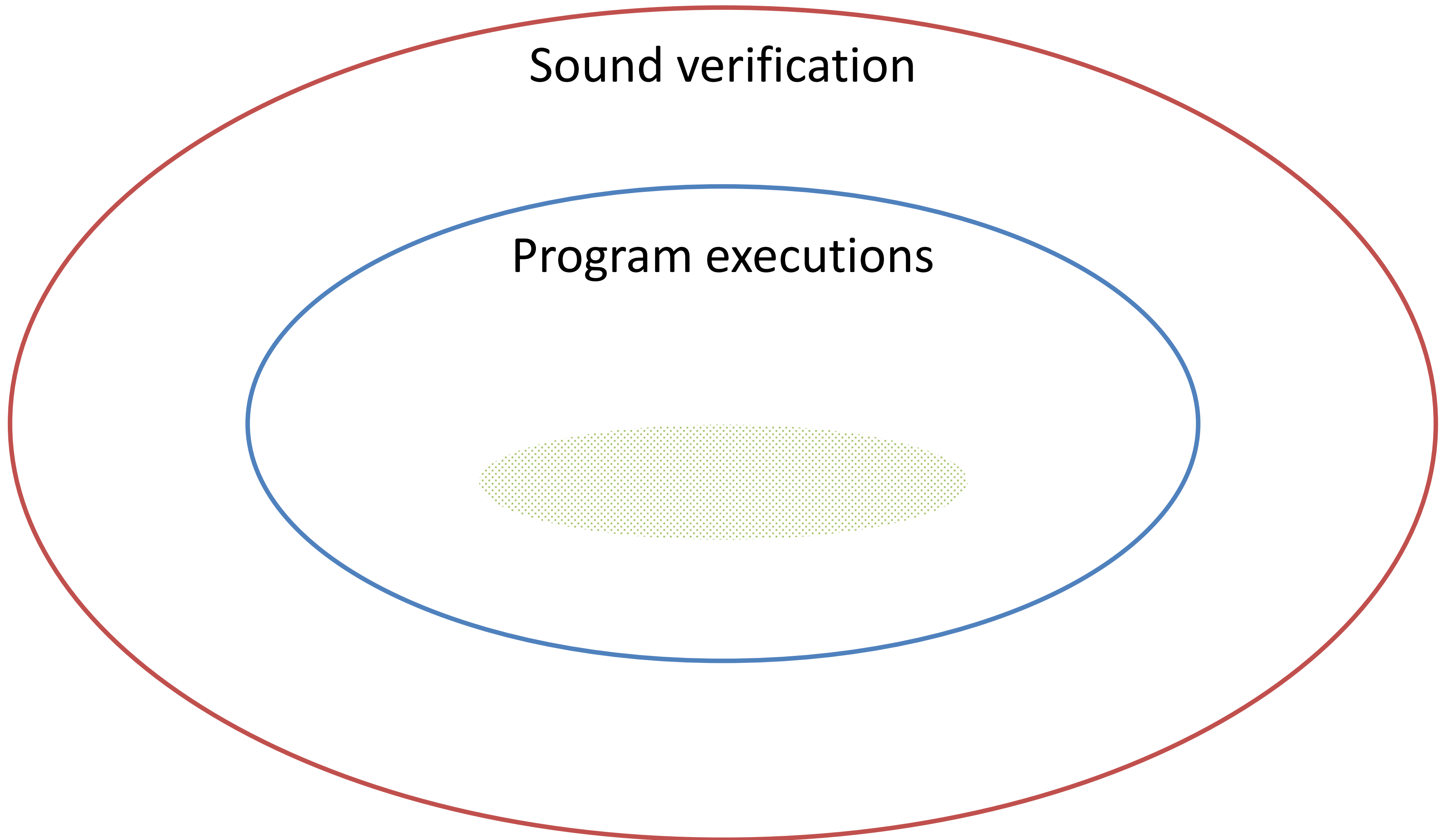
- Verification and systematic testing
  - Tight and effective collaboration
  - Increased software quality
  - Reduced test effort

Fun:
Failure-Directed Program Trimming

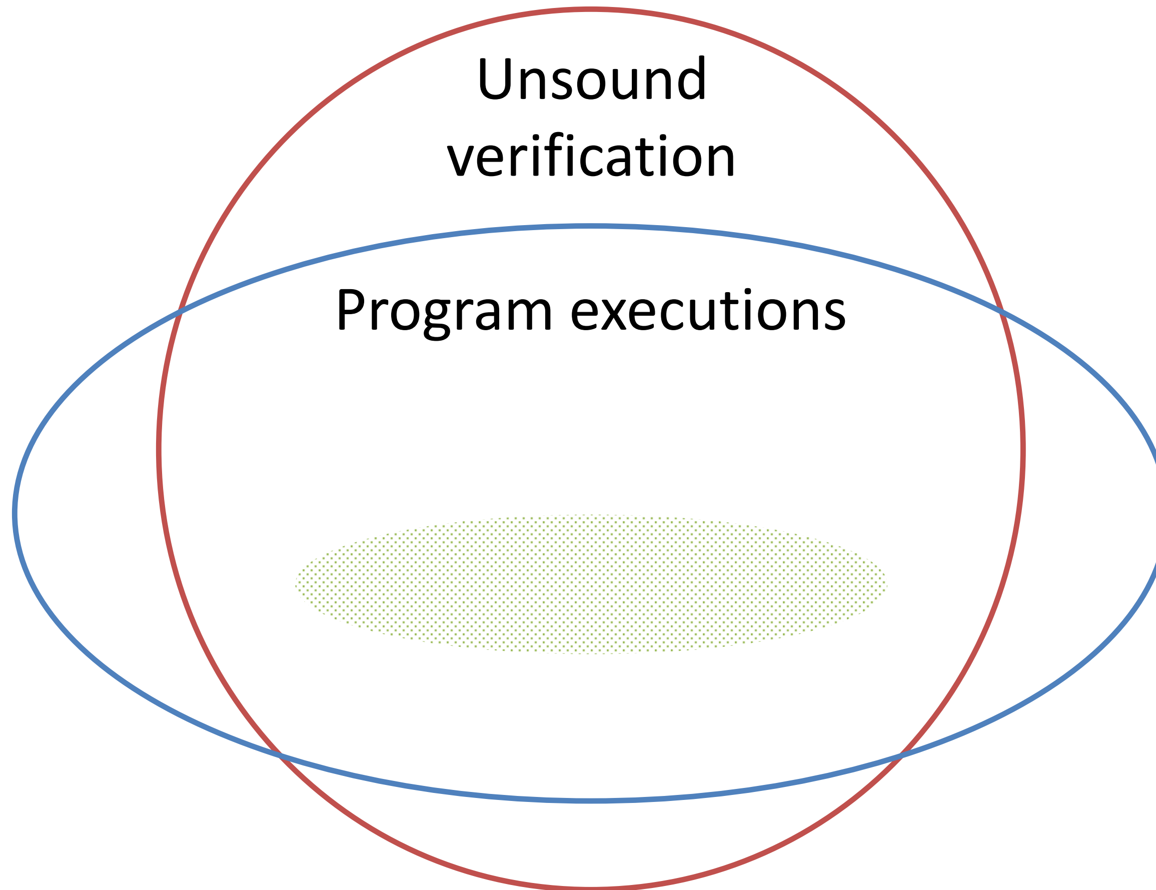
Safe executions



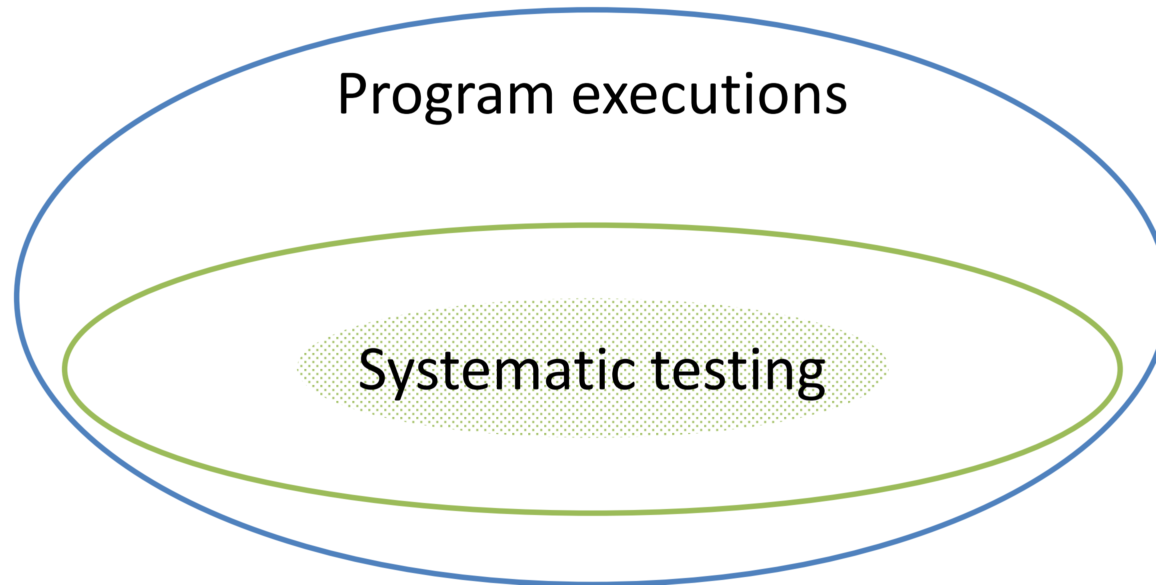
Safe executions



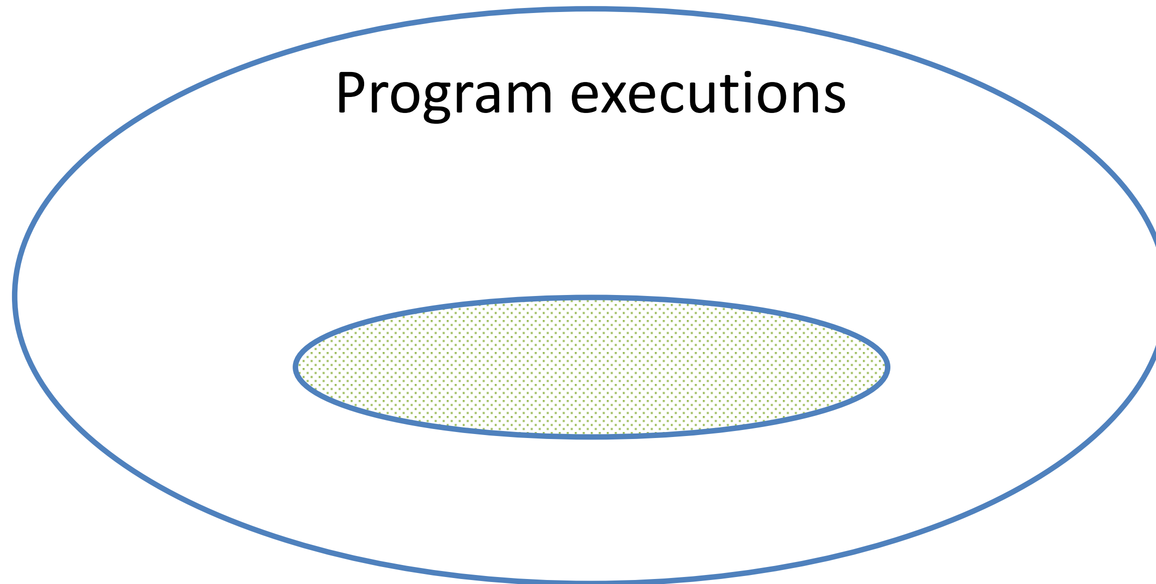
Safe executions



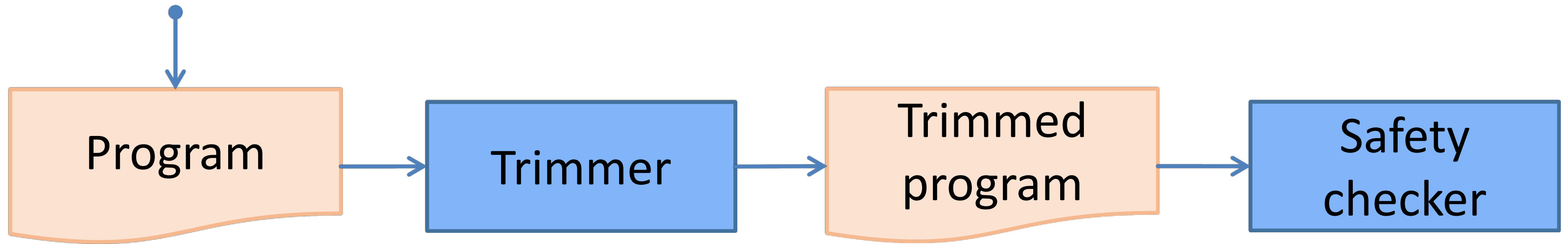
Safe executions



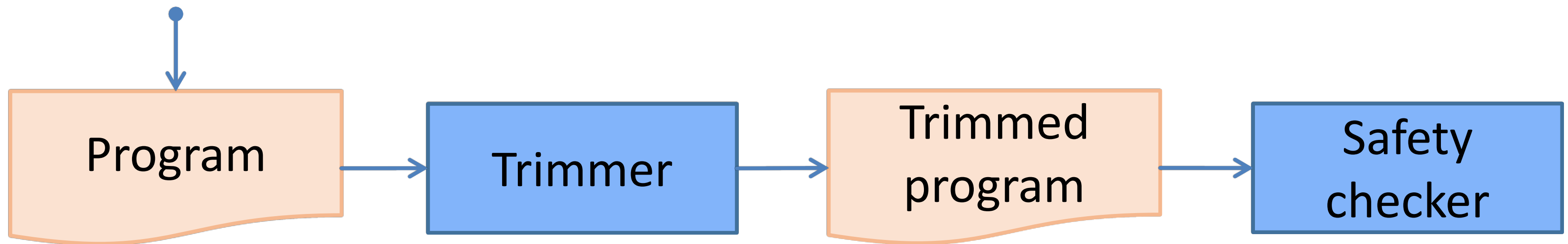
Trimming safe executions



Program trimming



Program trimming



- Simplifies programs
- Makes safety checkers property directed
- Improves scalability and precision of safety checkers

Example

```c
x = *; y = *;                C

if (y > 0) {
  while (x < 10) {
    x = x + y;
  }
} else {
  x = x - 1;
}
assert x > 0;
```

Example

```
x = *; y = *;

if (y > 0) {
  while (x < 10) {
    x = x + y;
  }
} else {
  x = x - 1;
}
assert x > 0;
```

```
x = *; y = *;
assume y <= 0;
x = x - 1;
assert x > 0;
```

Trimmed program

- Given P, program trimming generates P_{tr}
- P and P_{tr} are equi-safe
- P_{tr} contains fewer execution paths than P

Example

```
x = *; y = *;

if (y > 0) {
  while (x < 10) {
    x = x + y;
  }
} else {
  x = x - 1;
}
assert x > 0;
```

```
x = *; y = *;
assume y <= 0;
x = x - 1;
assert x > 0;
```

Example

```
x = *; y = *;                 P

if (y > 0) {
  while (x < 10) {
    x = x + y;
  }
} else {
  x = x - 1;
}
assert x > 0;
```

```
x = *; y = *;                 P_tr
assume y <= 0;
x = x - 1;
assert x > 0;
```

How to trim a program

- Statically infer safety conditions
- Negate the safety conditions to obtain trimming conditions
- Instrument the program with assumptions of the trimming conditions

Example

```
x = ⋆; y = ⋆;        P

if (y > 0) {
  while (x < 10) {
    x = x + y;
  }
} else {
  x = x - 1;
}
assert x > 0;
```

```
x = ⋆; y = ⋆;        Ptr
assume y <= 0;
x = x - 1;
assert x > 0;
```

Example

P

```
x = *; y = *;

if (y > 0) {
  while (x < 10) {
    x = x + y;
  }
} else {
  x = x - 1;
}
assert x > 0;
```

P_{tr}

```
x = *; y = *;
assume y <= 0;
if (y > 0) {
  while (x < 10) {
    x = x + y;
  }
} else {
  x = x - 1;
}
assert x > 0;
```

Condition properties

- Safety conditions are sufficient for correctness
- To be lightweight, we infer stronger safety conditions
- Trimming conditions are necessary for failure
- By definition, assuming the trimming conditions preserves unsafe executions

Benefit 1: Scalability

```
int fact(int n) {
  assume 0 <= n;
  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  return r;
}

void main() {
  int m = ⋆;

  int f = fact(m);
  assert m != 123 || f == 0;
}
```

Benefit 1: Scalability

```
int fact(int n) {
  assume 0 <= n;
  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  return r;
}

void main() {
  int m = *;

  int f = fact(m);
  assert m != 123 || f == 0;
}
```

Benefit 1: Scalability

```
int fact(int n) {
  assume 0 <= n;
  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  return r;
}

void main() {
  int m = *;

  int f = fact(m);
  assert m != 123 || f == 0;
}
```

Benefit 1: Scalability

```
int fact(int n) {
  assume 0 <= n;
  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  return r;
}

void main() {
  int m = *;

  int f = fact(m);
  assert m != 123 || f == 0;
}
```

$m \neq 123 \vee f = 0$

Benefit 1: Scalability

```
int fact(int n) {
  assume 0 <= n;
  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  return r;
}

void main() {
  int m = ⋆;

  int f = fact(m);
  assert m != 123 || f == 0;
}
```

$m \neq 123$

Benefit 1: Scalability

```
int fact(int n) {
  assume 0 <= n;
  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  return r;
}

void main() {
  int m = *;
  assume m == 123;
  int f = fact(m);
  assert m != 123 || f == 0;
}
```

Benefit 1: Scalability

```
int fact(int n) {
  assume 0 <= n;
  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  return r;
}

void main() {
  int m = *;
  assume m == 123;
  int f = fact(m);
  assert m != 123 || f == 0;
}
```

Benefit 1: Scalability

```
int fact(int n) {
  assume 0 <= n;
  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  return r;
}

void main() {
  int m = *;
  assume m == 123;
  int f = fact(m);
  assert m != 123 || f == 0;
}
```

Benefit 2: Precision

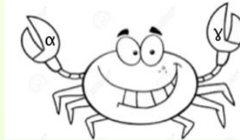
```
int fact(int n) {
  assume 0 <= n;

  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  assert n != 0 || r == 1;
  return r;
}

void main() {
  int m = ⋆;

  int f = fact(m);
}
```

Benefit 2: Precision




```
int fact(int n) {
  assume 0 <= n;

  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  assert n != 0 || r == 1;
  return r;
}

void main() {
  int m = *;

  int f = fact(m);
}
```

Benefit 2: Precision

```
int fact(int n) {
  assume 0 <= n;

  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  assert n != 0 || r == 1;
  return r;
}

void main() {
  int m = ⋆;

  int f = fact(m);
}
```

$n = 0$

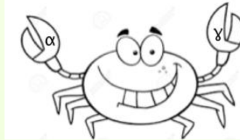
Benefit 2: Precision

```
int fact(int n) {
  assume 0 <= n;
  assume n != 0;
  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  assert n != 0 || r == 1;
  return r;
}

void main() {
  int m = *;

  int f = fact(m);
}
```

Benefit 2: Precision




```
int fact(int n) {
  assume 0 <= n;
  assume n != 0;
  int r = 1;
  if (n != 0) {
    r = n * fact(n - 1);
  }
  assert n != 0 || r == 1;
  return r;
}

void main() {
  int m = ⋆;

  int f = fact(m);
}
```

Inference of safety conditions

- Lightweight, summary-based static analysis
- Similar to a weakest-precondition computation

Inference of safety conditions

- Lightweight, summary-based static analysis
- Similar to a weakest-precondition computation

but…

Inference of safety conditions

- Lightweight, summary-based static analysis
- Similar to a weakest-precondition computation

but…

- Relies on several inference innovations
  - Heap writes
  - Procedure calls

Evaluation highlights

- 26-60% more safe programs
  with three abstract domains of Crab

- The cheapest trimming configuration
  proves 23% more programs safe than
  the most expensive Crab configuration
  in less than half of the time

Evaluation highlights

- 16-18% more safe programs
  with three search strategies of KLEE
- KLEE with trimming explores
  30-39% fewer paths
- The total running time of KLEE
  decreases by 23-36%

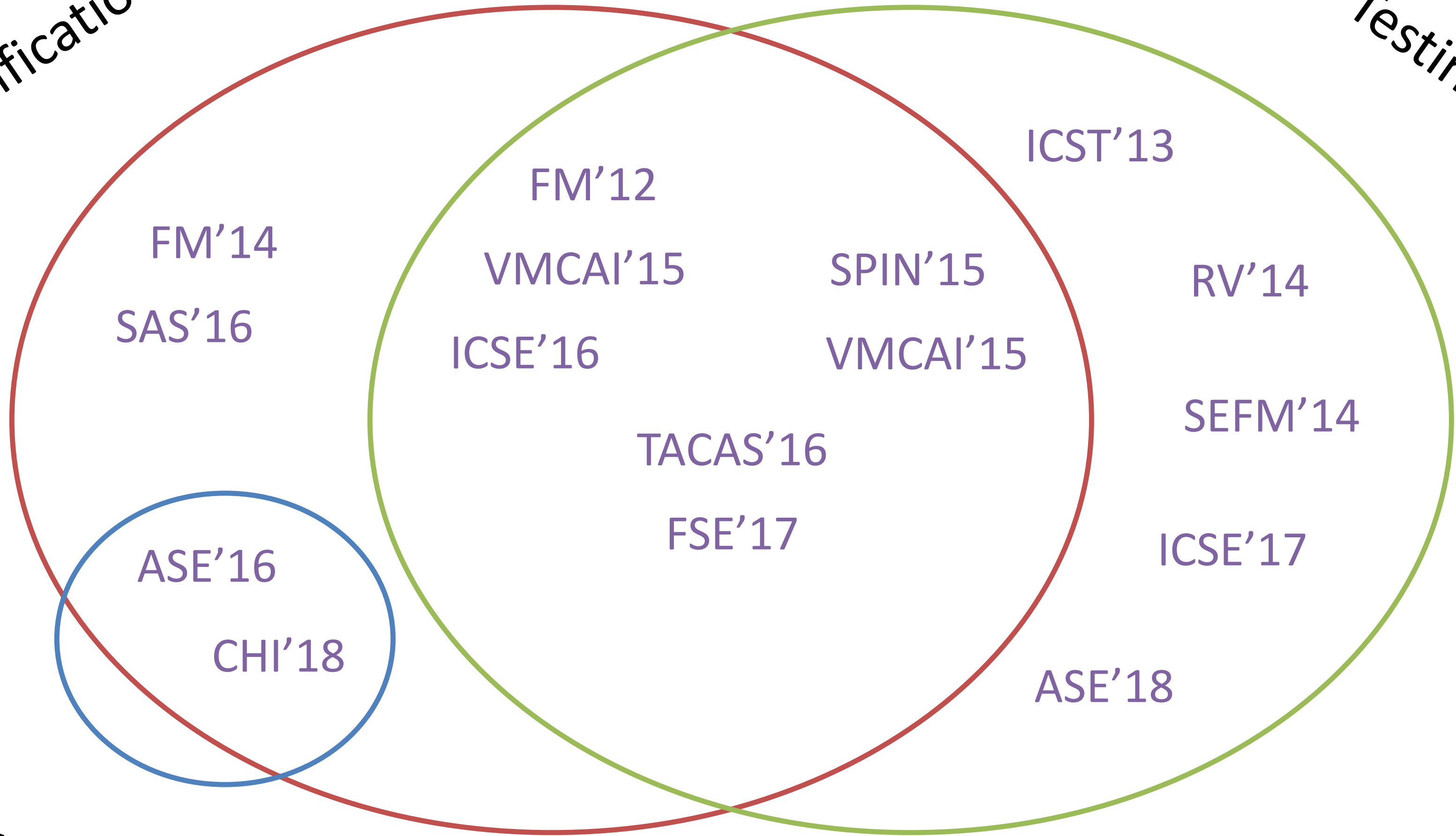
Evaluation highlights

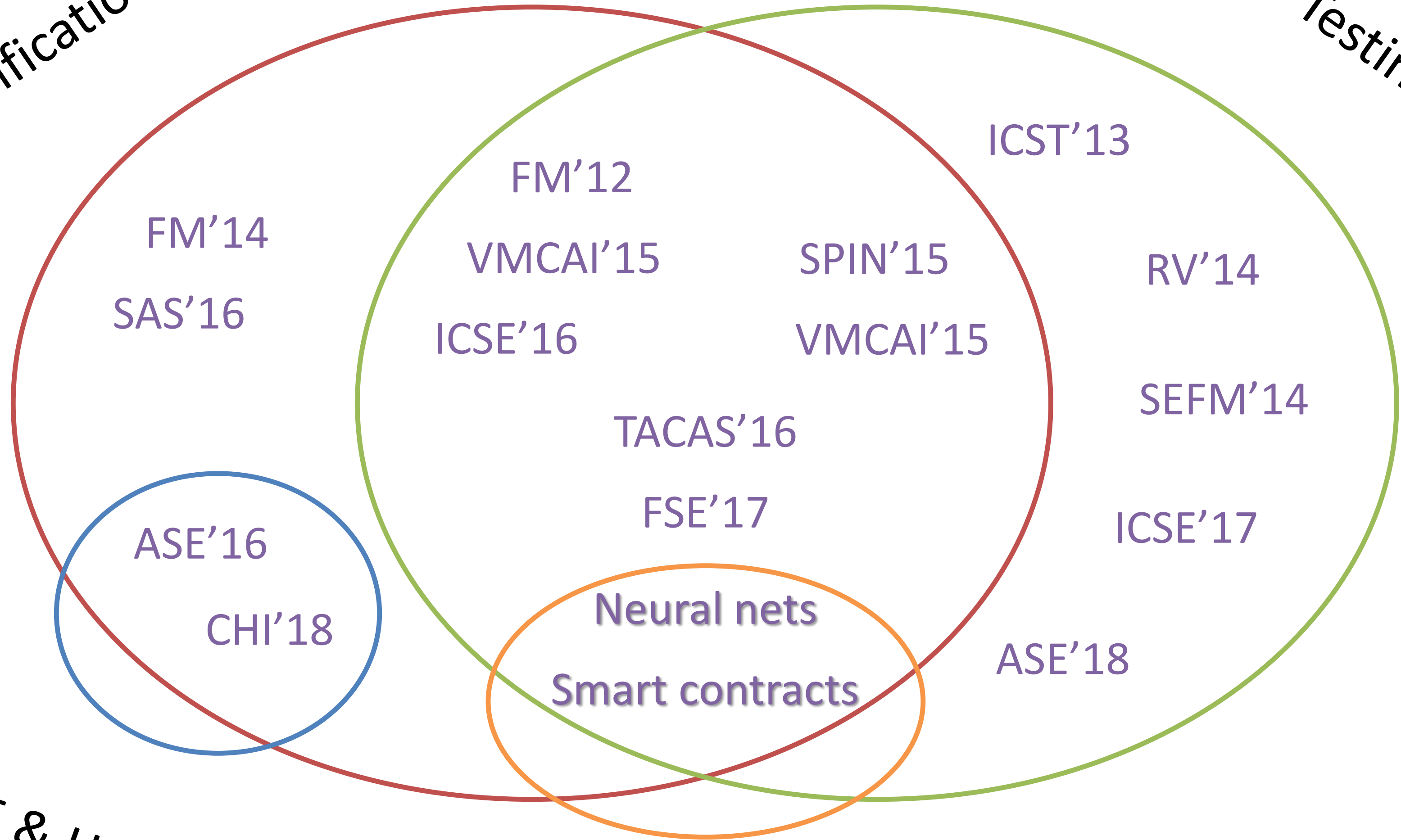
- The total time for trimming all benchmarks

  ranges from 5-9 secs

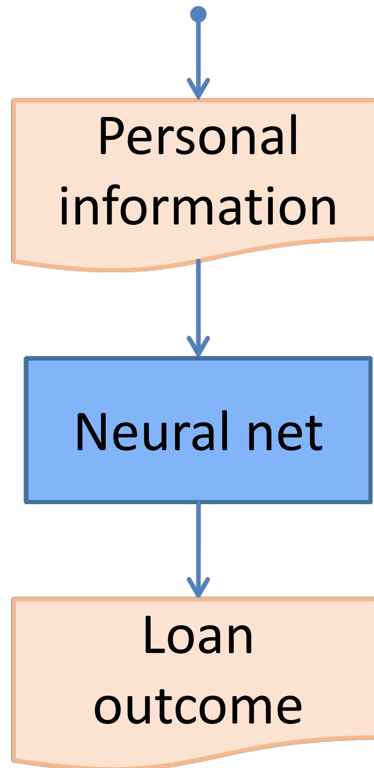
Takeaways

- Program trimming, a program simplification technique
  - Reduce the number of execution paths
  - Preserve unsafe executions of the original program
  - Positively impact efficiency and precision of safety checkers

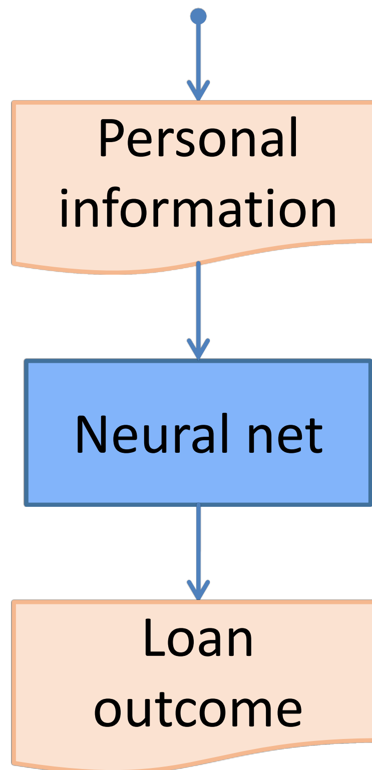

Verification
Testing
ESE & HCI
FM'14
SAS'16
FM'12
VMCAI'15
ICSE'16
SPIN'15
VMCAI'15
TACAS'16
FSE'17
ICST'13
RV'14
SEFM'14
ICSE'17
ASE'18
ASE'16
CHI'18


Verification
Testing
FM'14
SAS'16
FM'12
VMCAI'15
ICSE'16
SPIN'15
VMCAI'15
TACAS'16
FSE'17
ICST'13
RV'14
SEFM'14
ICSE'17
ASE'18
ASE'16
CHI'18
Neural nets
Smart contracts
ESE & HCI
Security & Privacy

Neural nets



Neural nets



Which parts of the input affect which output?

Are there privacy concerns?

Smart contracts

- Manage digital assets according to a protocol
- May jeopardize significant amounts of money
- Have subtle interactions at execution time
- Are written in expressive languages

Smart contracts

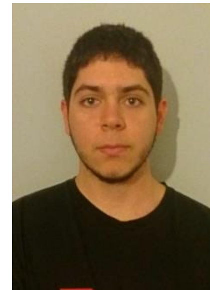
- Manage digital assets according to a protocol
- May jeopardize significant amounts of money
- Have subtle interactions at execution time
- Are written in expressive languages

How to effectively detect security vulnerabilities?

How to help developers understand their possible behaviors?

Shameless plug

I am looking for motivated and talented students and interns!



Static Program Analysis Meets Test Case Generation

Maria Christakis
MPI-SWS, Germany