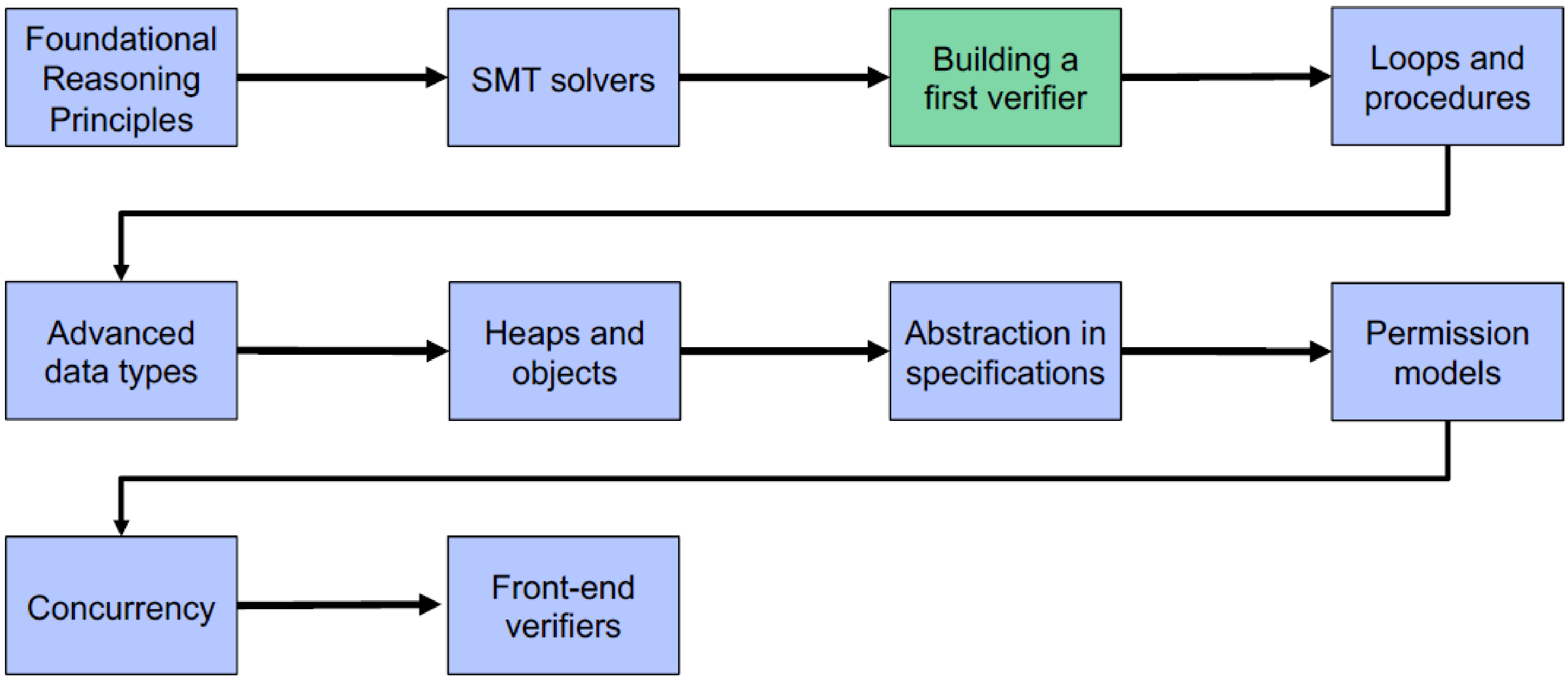
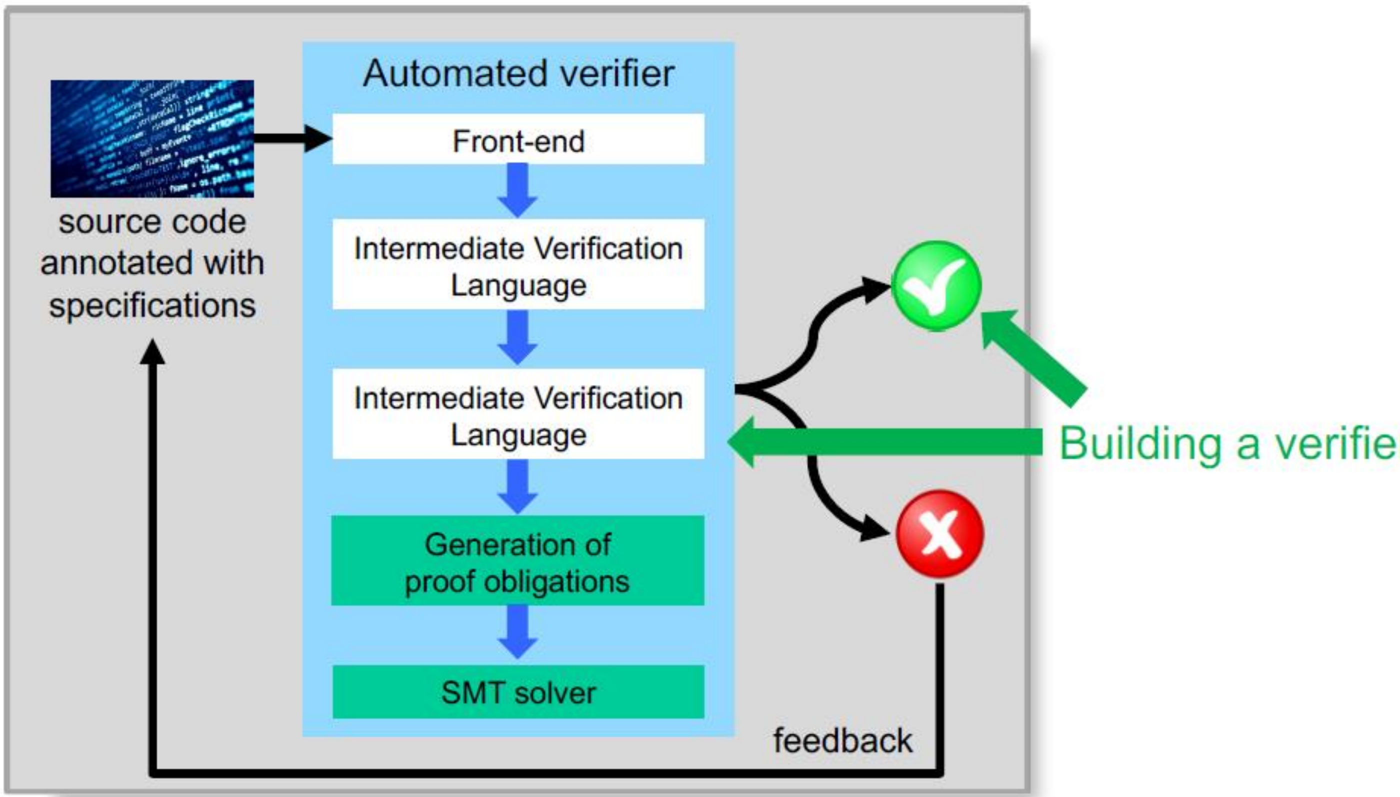
Methodologies for Software Processes

Lecture 4

Tentative course outline



What next?



Automated verifier
Front-end
Intermediate Verification Language
Intermediate Verification Language
Generation of proof obligations
SMT solver
source code annotated with specifications
feedback
Building a verifier

Outline

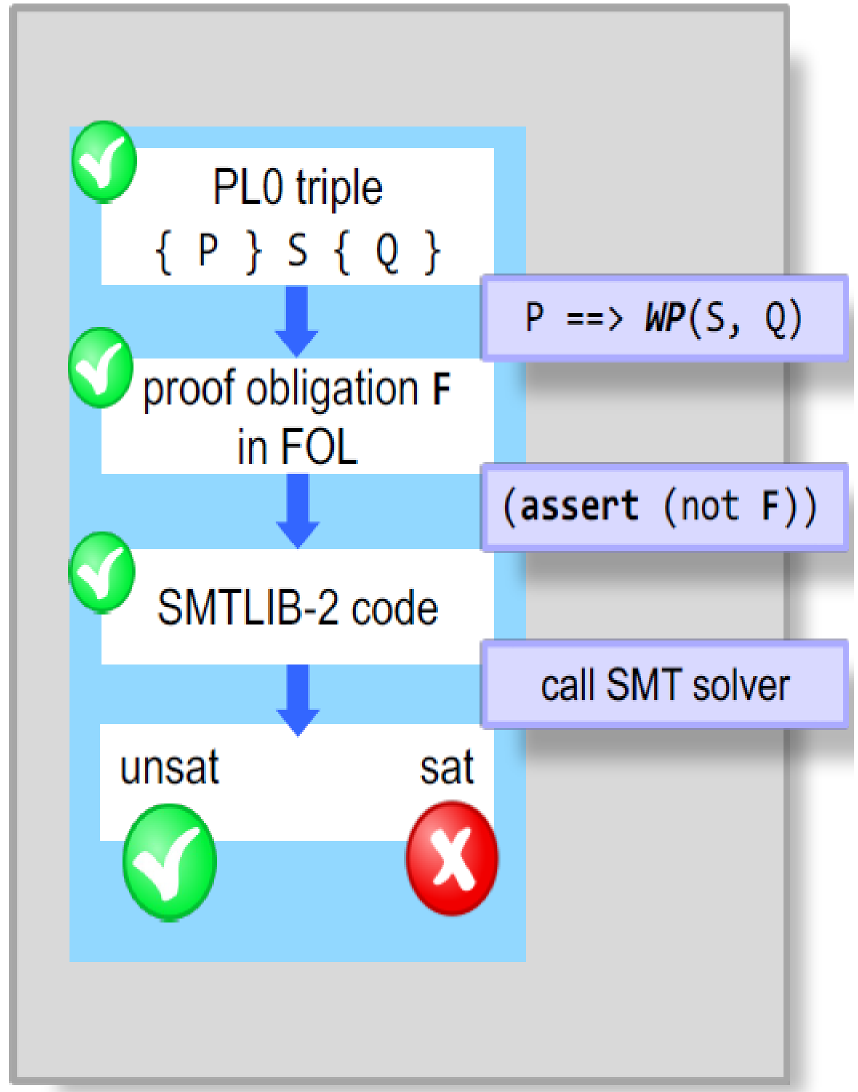
1. The Verification Toolchain

2. Efficient weakest preconditions

3. Error localization

The toolchain so far

- "Verification as compilation"
- Translate verification problems into simpler ones until the answer is trivial
- Wishlist for each translation **A** ➡ **B**
  - **Soundness:** If **B** is valid, then **A** is valid

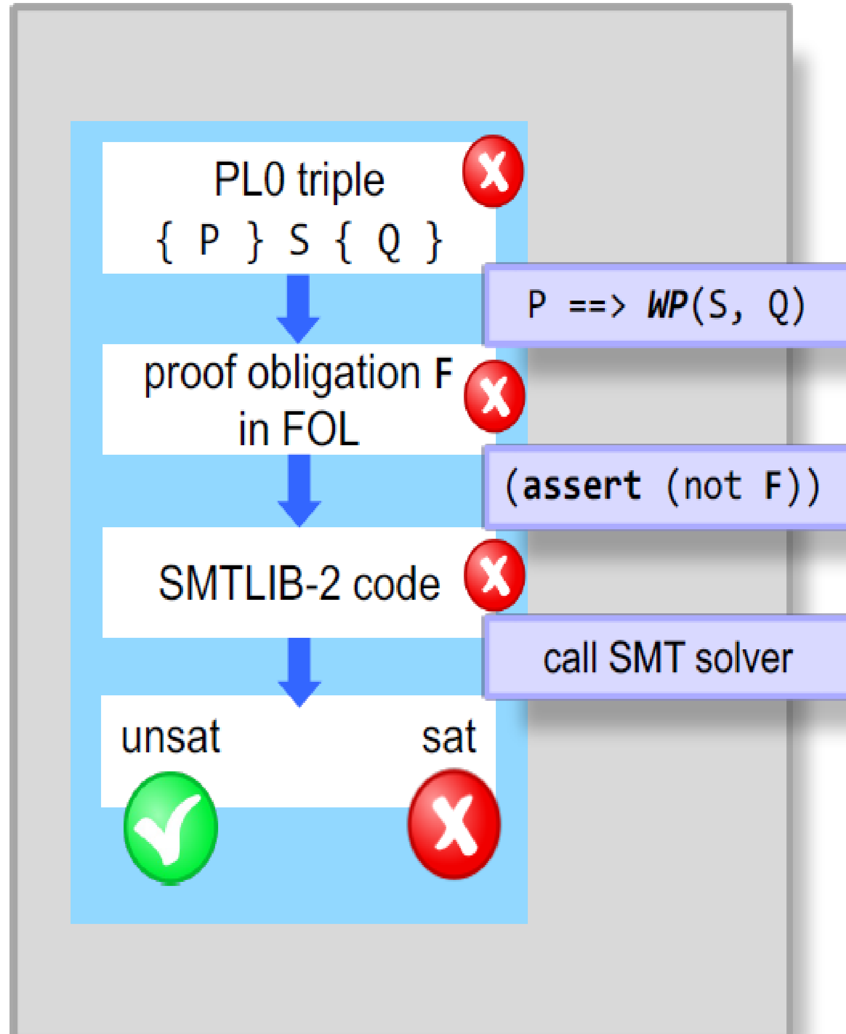


The toolchain so far

- "Verification as compilation"
- Translate verification problems into simpler ones until the answer is trivial
- Wishlist for each translation **A** ➡ **B**
  - **Soundness:** If **B** is valid, then **A** is valid
  - **Completeness:** If **A** is valid, then **B** is valid

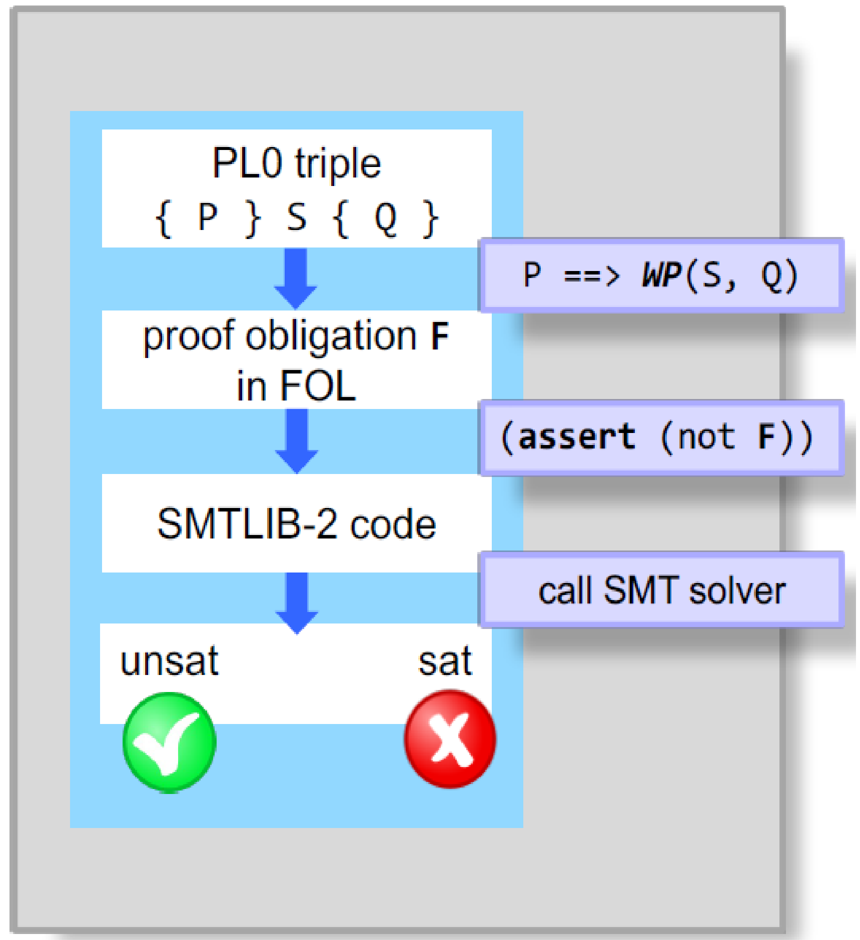
Soundness is *necessary*.

Completeness is desirable.



The toolchain so far

- "Verification as compilation"
- Translate verification problems into simpler ones until the answer is trivial
- Wishlist for each translation **A** ➡ **B**
  - **Soundness:** If **B** is valid, then **A** is valid
  - **Completeness:** If **A** is valid, then **B** is valid
  - **Efficiency:** **B**'s size is reasonable wrt. **A**
  - **Explainability:** We can reconstruct errors in **A** from errors in **B**



Splitting the PL0 Language

Programming Language XPL

- Statements are eXecutable
- Deterministic conditionals
- Specifications via triples

XPL Statements

```
S    ::=    var x  |  x := a  |  S;S
         |  if (b) { S } else { S }
         |  assert b
```

Verification condition

{ P } S { Q } valid

Verification Language PL0

- Statements model verification problems
- Nondeterministic choice
- Verification-specific statements

PL0 Statements

```
S    ::=    var x  |  x := a  |  S;S
         |  S [] S
         |  assert P  |  assume P
```

What is our verification condition for PL0 programs if we have only a statement S (no pre- or postcondition)?

Splitting the PL0 Language

Programming Language XPL

- Statements are eXecutable
- Deterministic conditionals
- Specifications via triples

Verification Language PL0

- Statements model verification problems
- Nondeterministic choice
- Verification-specific statements

XPL Statements

```
S    ::=    var x  |  x := a  |  S;S
         |  if (b) { S } else { S }
         |  assert b
```

PL0 Statements

```
S    ::=    var x  |  x := a  |  S;S
         |  S [] S
         |  assert P  |  assume P
```

Verification condition

{ P } S { Q } valid

Verification condition

WP(S, true) valid

Running example: `triple_min`

```
method triple_min(x: Int, y: Int) returns (z: Int)
requires x >= 0 && y >= 0
ensures z <= 3 * x && z <= 3 * y && (z == 3 * x || z == 3 * y)
{
    z := x - y
    if (z < 0) {
        z := z + y
        z := z + 2 * x
    } else {
        z := z - x
        z := z + 4 * y
    }
}
```

The code examples contain every translation step applied to this program

```
// Step one: Encode the triple { A } S { B } using assume and assert.
// A few steps are needed to get a well-formed Viper program:
// We declare all variables upfront in a preamble
// We put everything in a method without parameters
method main()
{
    // preamble
    var x: Int
    var y: Int
    var z: Int

    // assume precondition
    assume x >= 0 && y >= 0

    // program statement still needs encoding
    z := x - y
    if (z < 0) {
        z := z + y
        z := z + 2 * x
    } else {
        z := z - x
        z := z + 4 * y
    }

    // assert postcondition
    assert z <= 3 * x && z <= 3 * y && (z == 3 * x || z == 3 * y)
}
```

```
// Step two: Encode S in the next language layer (PL0)
// Here, we need to encode the conditional if-then-else using assume and
nondeterministic choice
// To illustrate this in Viper, we need to declare a Boolean variable star.
method main()
{
    // preamble
    var x: Int
    var y: Int
    var z: Int
    var star: Bool // needed to represent nondeterminism

    assume x >= 0 && y >= 0

    z := x - y
    if (star) { // nondeterministic choice S1 [] S2
        assume z < 0
        z := z + y
        z := z + 2 * x
    } else {
        assume !(z < 0)
        z := z - x
        z := z + 4 * y
    }

    assert z <= 3 * x && z <= 3 * y && (z == 3 * x || z == 3 * y)
}
```

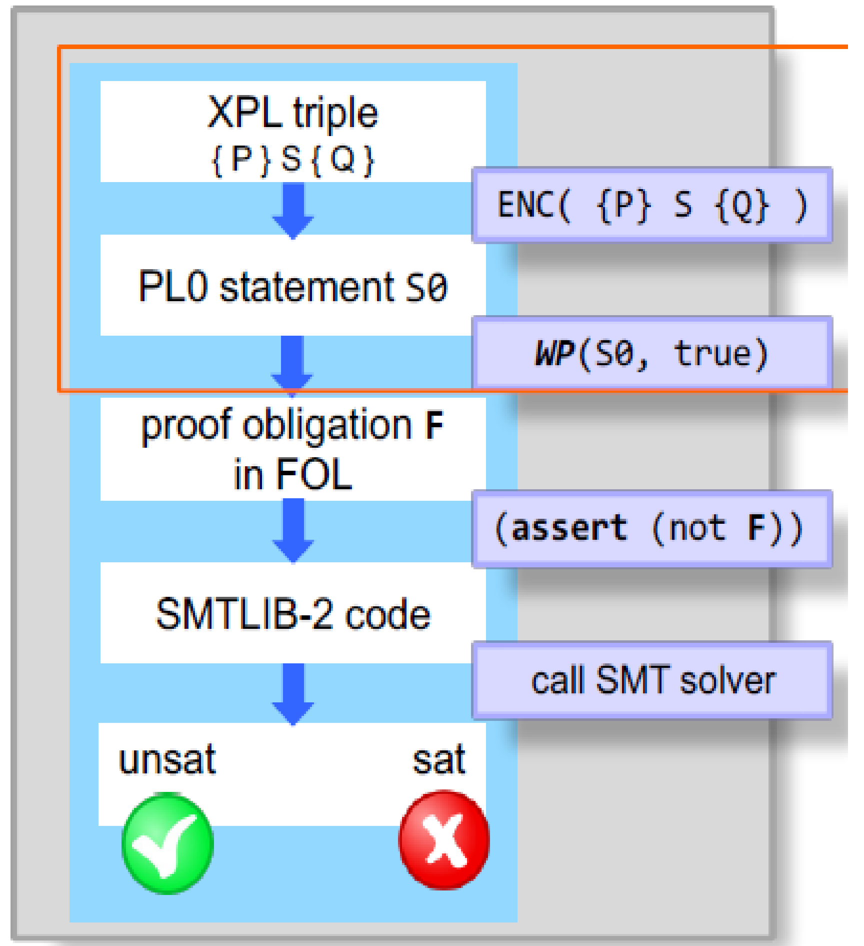
```
// Step three: Encode S in the next language layer
// by transforming the program into
// dynamic single assignment form (DSA)
method main()
{
    // preamble
    var x0: Int
    var y0: Int
    var z0: Int
    var z1: Int
    var z2: Int
    var star: Bool

    assume x0 >= 0 && y0 >= 0

    z0 := x0 - y0
    if (star) {
        assume z0 < 0
        z1 := z0 + y0
        z2 := z1 + 2 * x0
    } else {
        assume !(z0 < 0)
        z1 := z0 - x0
        z2 := z1 + 4 * y0
    }

    assert z2 <= 3 * x0 && z2 <= 3 * y0 && (z2 == 3 * x0 || z2 == 3 * y0)
}
```

The toolchain so far

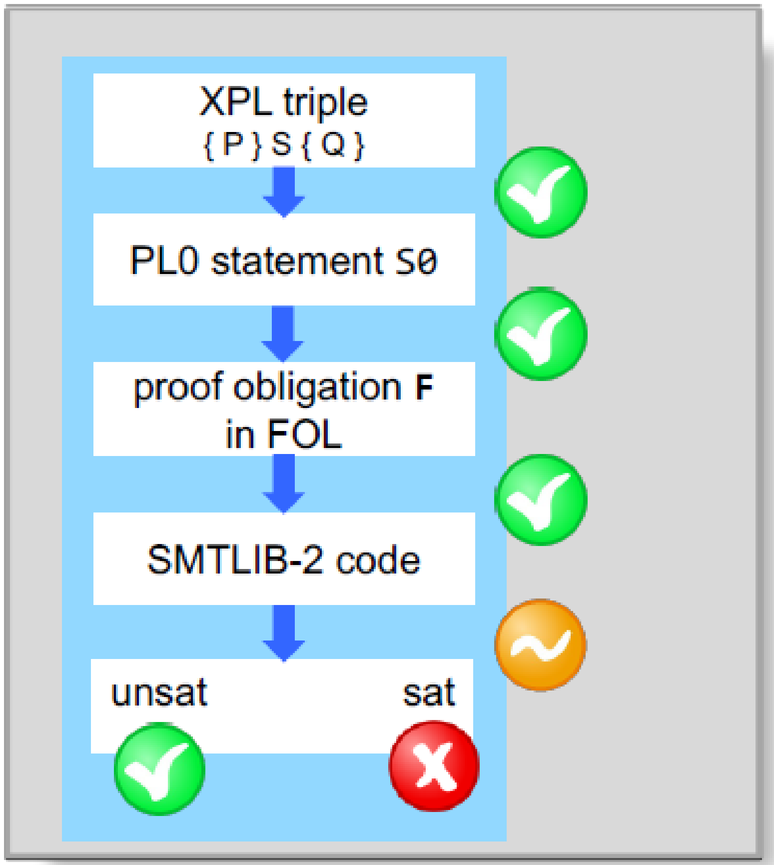
- "Verification as compilation"

- Translate verification problems into simpler ones until the answer is trivial

- Wishlist for each translation **A** ➡ **B**
  - **Soundness:** If **B** is valid, then **A** is valid
  - **Completeness:** If **A** is valid, then **B** is valid
  - **Efficiency: B**'s size is reasonable wrt. **A**
  - **Explainability:** We can reconstruct errors in **A** from errors in **B**



Soundness across the toolchain



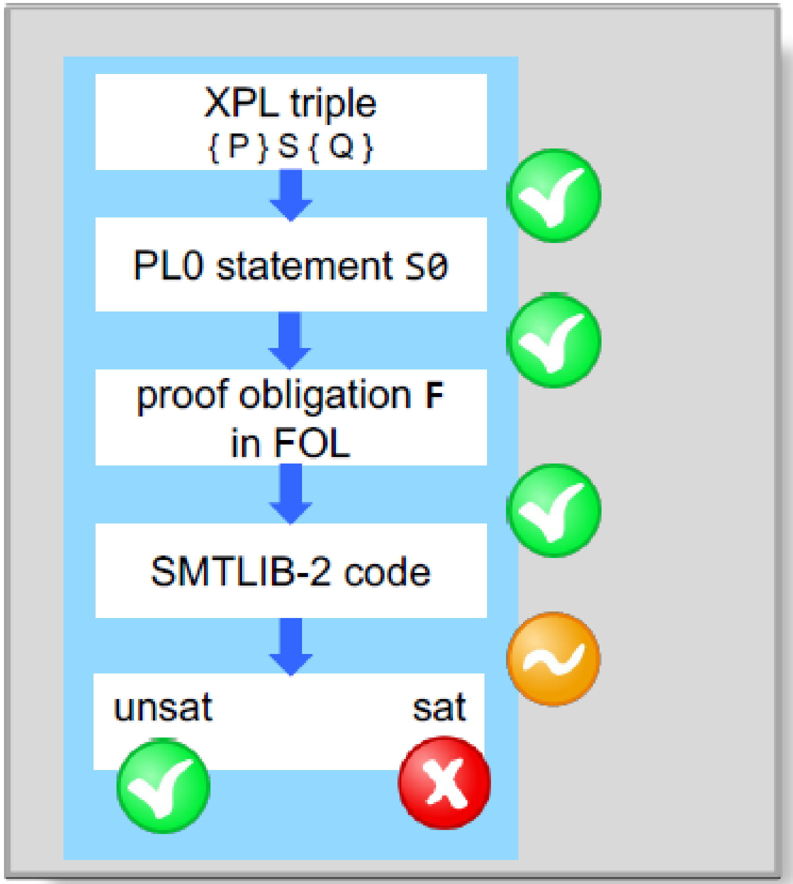
previous exercise

```
{ P } S0 { Q } valid
iff
P ==> WP(S0, Q) (aka F) valid
```

```
F valid iff !F unsatisfiable
```

Sound for formally verified SMT solver (not Z3)

Completeness across the toolchain



previous exercise

```
{ P } S0 { Q } valid
iff
P ==> WP(S0, Q) (aka F) valid
```

```
F valid iff !F unsatisfiable
```

Solver can only be complete for decidable theories

- unknown or non-termination ➔ false negatives

Outline

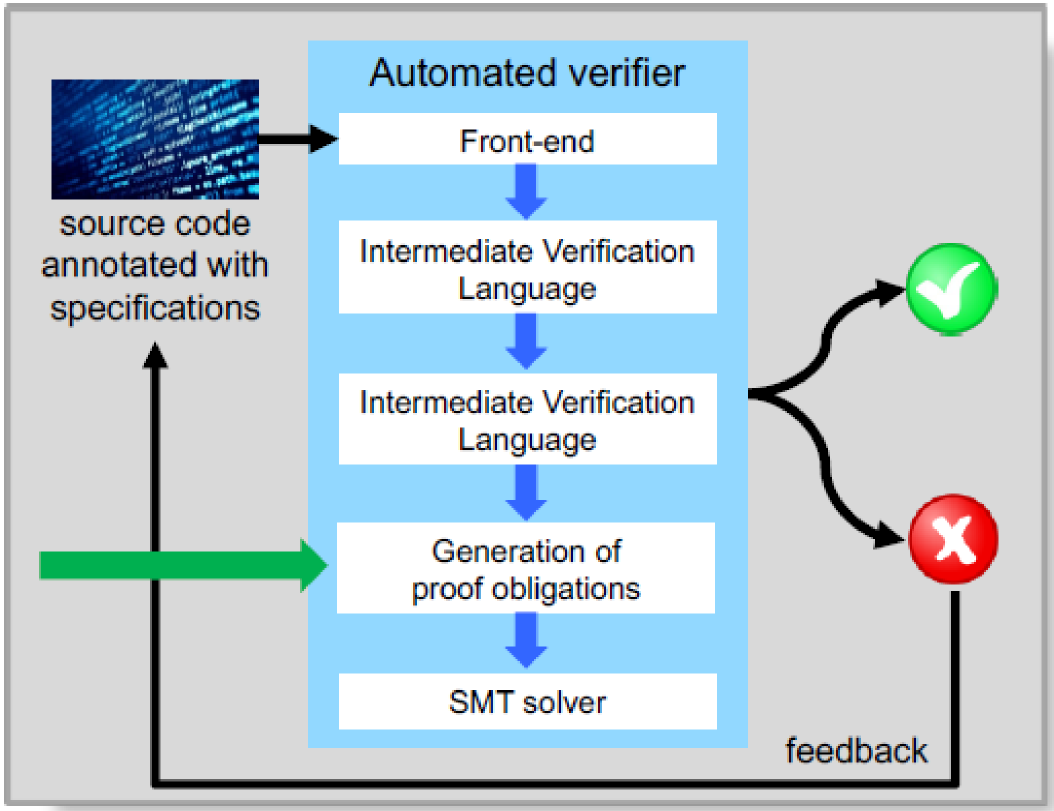
1. The Verification Toolchain

2. Efficient weakest preconditions

3. Error localization

Roadmap

We are here

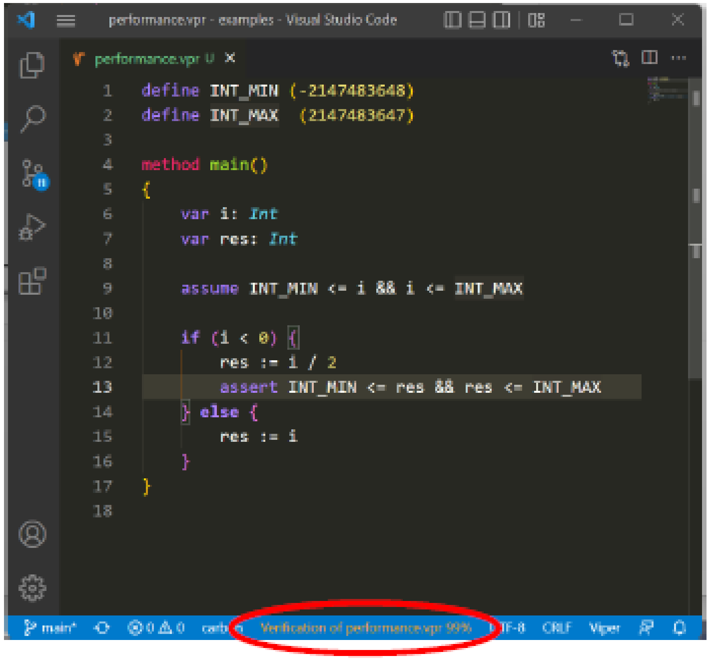


Verifier Performance

- The time consumed by an automated verifier is typically dominated by the SMT solver

- Factors influencing SMT performance
  - Size of verification conditions
  - Theories in the background predicate
  - Effectiveness of heuristics for undecidable theories, particularly quantifier instantiation

- Verification times are flaky
  - Minor changes in VCs can have major impact
  - Verification is often much faster than refutation



Size of Verification Conditions

Compute WP(S, Q) for the programs below; do you notice a pattern?

```
{ TODO }
res := (start + end)/2
{ res * res * res == x }
```

```
{ TODO }
{

  x := (y+z)*(y+z)

} [ ] {

  x := 12

}
{ 0 <= x }
```

Size of Verification Conditions

Expression a is duplicated for each occurrence of variable x

S	*WP*(S, Q)
var x	**forall** x :: Q
x := a	Q[x / a]
assert R	R && Q
assume R	R ==> Q
S1; S2	*WP*(S1, *WP*(S2, Q))
S1 [] S2	*WP*(S1, Q) && *WP*(S2, Q)

Postcondition Q is duplicated for each nondeterministic choice

```
{ (start + end)/2 * (start + end)/2 *
  (start + end)/2 == x }
res := (start + end)/2
{ res * res * res == x }
```

```
{ 0 <= (y+z)*(y+z) ∧ 0 <= 12 }
{
  { 0 <= (y+z)(y+z) }
  x := (y+z)*(y+z)
  { 0 <= x }
} [ ] {
  { 0 <= 12 }
  x := 12
  { 0 <= x }
}
{ 0 <= x }
```

Worst case: VC grows exponentially in the program size

Eliminating duplication from assignments

Idea: add knowledge x == a once and for all instead of substituting every x by a

```
WP(x := a, Q)  ::=  (x == a) ==> Q
```

Example with current ***WP***

```
{ (start + end)/2 * (start + end)/2 *
  (start + end)/2 == x }
res := (start + end)/2
{ res * res * res == x }
```

Example with proposed ***WP***

```
{ res == (start + end)/2 ==>
  res * res * res == x }
res := (start + end)/2
{ res * res * res == x }
```

Is the proposed change of ***WP*** sound?

Soundness of alternative assignment rule

```
{ true }
// ==>
{ (0 == 1 ==> false) }
// ==>
{ x == 0 ==> (x == 1 ==> false) }
x := 0
{ x == 1 ==> false }
x := 1
{ false }
assert false
{ true }
```

Unsound: program verifies even though an assertion fails!

Proposed change

```
WP(x := a, Q)  ::=  (x == a) ==> Q
```

- Issue: the new rule might contradict prior information about x
- Solution: introduce a *fresh* variable

Preliminary sound assignment rule

```
WP(x := a, Q)  ::=  (y == a) ==> Q[x / y]
```

where y is a fresh variable

```
{ true }
==>
{ z == 0 ==> (y == 1 ==> false) }
x := 0;
{ y == 1 ==> false }
x := 1;
{ false }
assert false
{ true }
```

Fixes unsoundness

```
{ y == (start + end)/2 ==>
  y * y * y == x }
res := (start + end)/2
{ res * res * res == x }
```

still avoids duplication

Eliminating redundancy from choice-statements

Similar idea: factor out postcondition using a fresh variable

```
WP(S1 [] S2, Q)  ::=  (B == Q) ==> WP(S1, B) && WP(S2, B)
```

where B is a fresh Boolean variable

```
{ (x == 5 ==> 0 <= x) ∧ 0 <= x }
{
  { x == 5 ==> 0 <= x }
  assume x == 5
  { 0 <= x }
} [] {
  { 0 <= x }
  assert true
  { 0 <= x }
}
{ 0 <= x }
```

```
{ b == (0 <= x) ==> (x == 5 ==> b) ∧ b }
{
  { x == 5 ==> b }
  assume x == 5
  { b }
} [] {
  { b }
  assert true
  { b }
}
{ 0 <= x }
```

Soundness of alternative rule for choices

```
WP(S1 [] S2, Q)  ::=  (B == Q) ==> WP(S1, B) && WP(S2, B)
```

where B is a fresh Boolean variable

Is the proposed change of ***WP*** sound?

- **No**, not in general
- Issue: assignments in S1, S2
  - substitutions [x / a] have no effect on fresh B
  - but: may change postcondition Q
- **Yes**, if S1, S2 contain *no assignments*

```
{ B == (0 <= x) ==> B ∧ B }
{
  { B }
  x := (y+z)*(y+z)
  { B }
} [] {
  { B }
  x := -12
  { B }
}
{ 0 <= x }    // unsound!
```

Towards efficient verification conditions

- **Choices:** sound and efficient rule for programs without assignments

```
WP(S1 [] S2, Q)  ::=  (B == Q) ==> WP(S1, B) && WP(S2, B)     where B is fresh
```

- **Assignments:** sound and efficient rule

```
WP(x := a, Q)  ::=  (y == a) ==> Q[x / y]      where y is fresh
```

- **Observation:** if x does not appear in a (x ∉ FV(a)), then

```
WP(assume x == a, Q) valid    iff    WP(x := a, Q) valid
```

➔ Can we translate PL0 into a reduced verification language without assignments?

The minimal verification language `MVL`

MVL Statements

```
S    ::= assert R
       | assume R
       | S;S
       | S [] S
```

efficient weakest preconditions

`S`	`EWP(S, Q)`
`assert R`	`R && Q`
`assume R`	`R ==> Q`
`S1; S2`	`EWP(S1, EWP(S2, Q))`
`S1 [] S2`	`(B == Q) ==> EWP(S1, B) && EWP(S2, B)` where `B` is fresh

sound without assignments

- `PL0`: ***`WP`***`(S, Q)` is exponential in the size of `S` and `Q`
- `MVL`: ***`EWP`***`(S, Q)` is linear in the size of `S` and `Q`

→ Is there a sound & complete encoding from `PL0` to `MVL`?

From PL0 to MVL

- Main idea:
  1. Eliminate variable declarations (exercise, later)
  2. Make all assignments assign to fresh variables ➔ single static assignment form (SSA)
  3. Replace every assignment `x := a` by `assume x == a` ➔ passification

- Observation: all paths through a PL0 program are finite (no loops / recursion)
- A program is in dynamic single assignment form (DSA)

  iff every assignment on a path assigns to a fresh variable

```
x := 0
x := 1
y := x
```

```
x1 := 0
x2 := 1
y1 := x2
```

```
x := 0
{
  x := (y+z)*(y+z)
} [] {
  x := -12
}
```

```
x1 := 0
{
  x2 := (y1+z1)*(y1+z1)
} [] {
  x2 := -12
}
```

DSA Construction

- Main idea
  - Introduce multiple versions of each variable
  - Always use the latest version

- Assignment
  - Assign to a new version

```
x := 0
x := 1
y := x
```

```
x_1 := 0
x_2 := 1
y_1 := x_2
```

- Choice-statements
  - convert both branches individually
  - synchronize the last version of each variable

```
x := 0
{
  x := (y+z)*(y+z)
  x := 7
} [] {
  x := -12

}
y := x
```

```
x_1 := 0
{
  x_2 := (y_1+z_1)*(y_1+z_1)
  x_3 := 7
} [] {
  x_2 := -12
  x_3 := x_2
}
y := x_3
```

How do we encode variable declarations in MVL?

Hint: try to encode `var x` as a PL0 program first

S	*WP*(S, Q)
var x	**forall** x :: Q
x := a	Q[x / a]
assert R	R && Q
assume R	R ==> Q
S1; S2	*WP*(S1, *WP*(S2, Q))
S1 [] S2	*WP*(S1, Q) && *WP*(S2, Q)

Solution: How do we encode variable declarations in MVL?

Main Idea:

- Declaration “forgets” previous values
- Same effect: Assigning to a fresh variable

S	*WP*(S, Q)
var x	**forall** x :: Q
x := a	Q[x / a]
assert R	R && Q
assume R	R ==> Q
S1; S2	*WP*(S1, *WP*(S2, Q))
S1 [] S2	*WP*(S1, Q) && *WP*(S2, Q)

```
WP(var x, Q) = WP(x := y, Q) ::=  Q[x / y]
```

where y is fresh

```
(wlog. assume VC is in prenex normal form)
valid: forall x :: Q
iff (y fresh)
valid: forall y :: Q[x/y]
iff (y is free, validity implicitly quantifies universally over all free variables)
valid: Q[x / y]
```

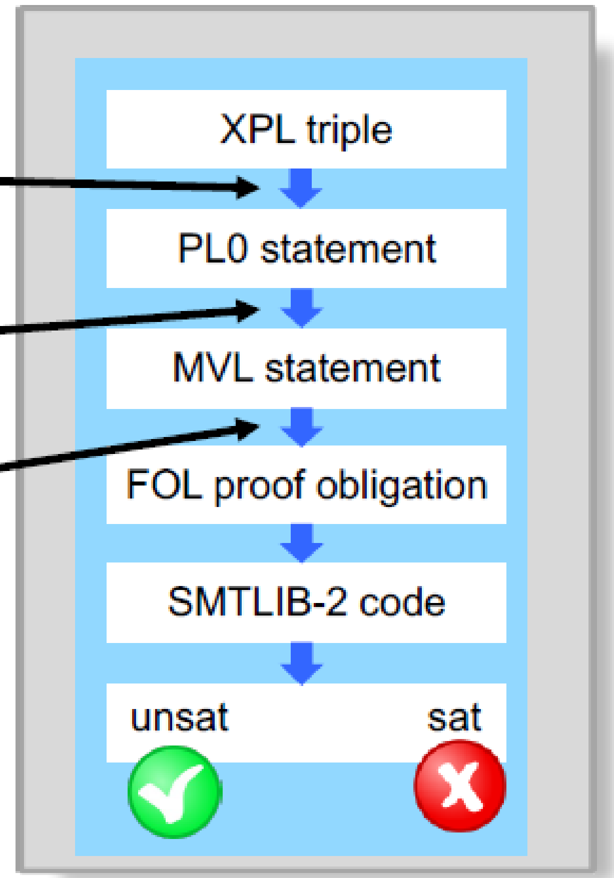
The toolchain so far

Encode

- Pre- and postconditions
- If-statements

- Variable declarations
- DSA transformation
- Passification

Efficient ***WP***



All *encodings* are sound and complete
(not necessarily true for *solvers*)

Size of VCs: linear in the original triple

Outline

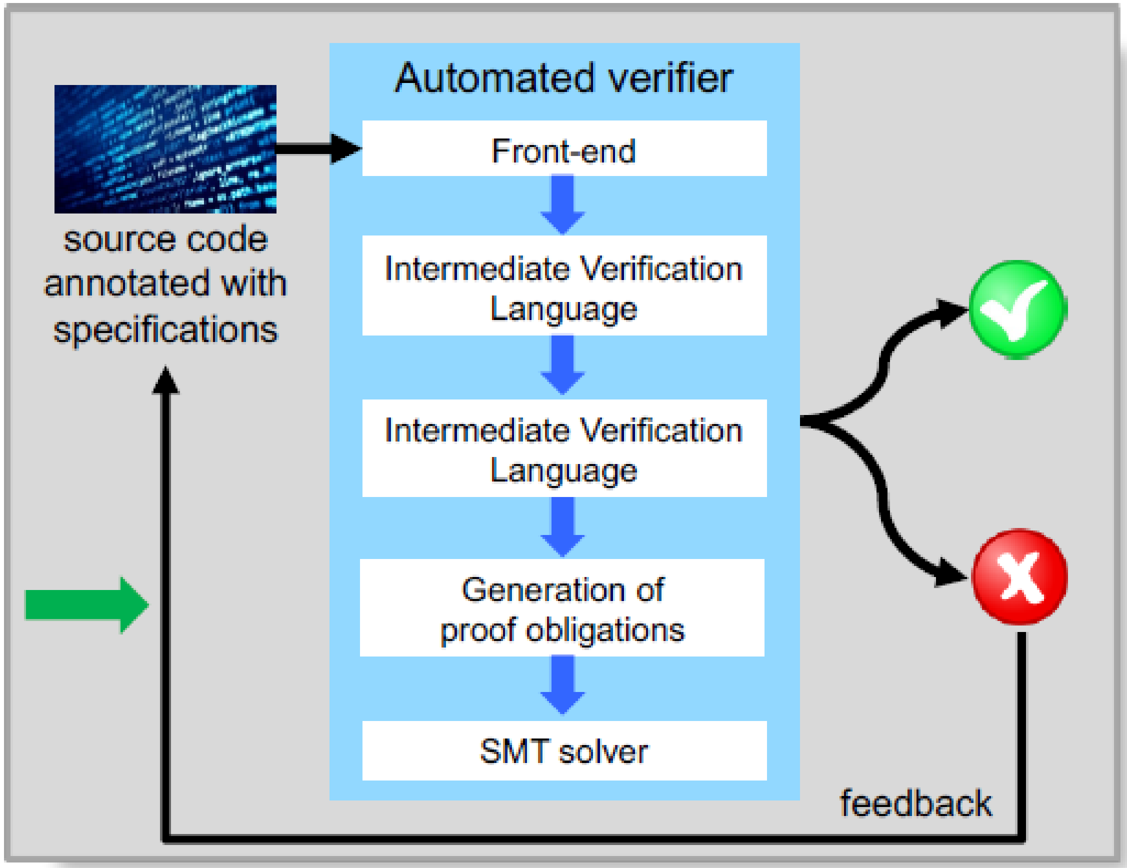
1. The Verification Toolchain

2. Efficient weakest preconditions

3. Error localization

Roadmap

We are here



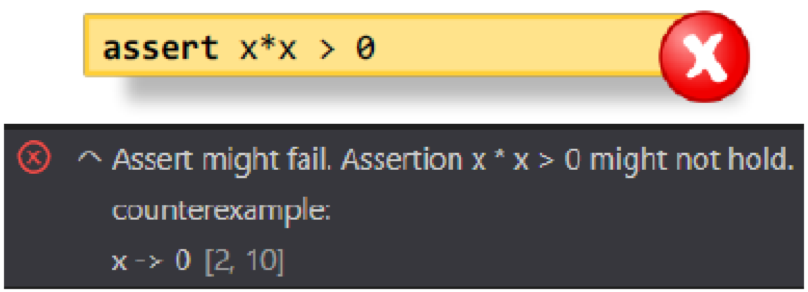
Verification Debugging with Counterexamples

Verification condition : `!(E)WP(S, true)` satisfiable?

- `unsat:`
- `sat:` + model with initial values invalidating VC ➔ counterexample
- `unknown:` + we can often still get a partial model

- Viper command line option

  `--counterexample variables`



Causes for verification failures

- Errors in the implementation
- Errors in the specification
  - Pre- and postconditions
  - Assumptions and assertions
- Incompleteness of the verifier
- Unsoundness of the SMT solver
  - Possible but unlikely for unverified solvers

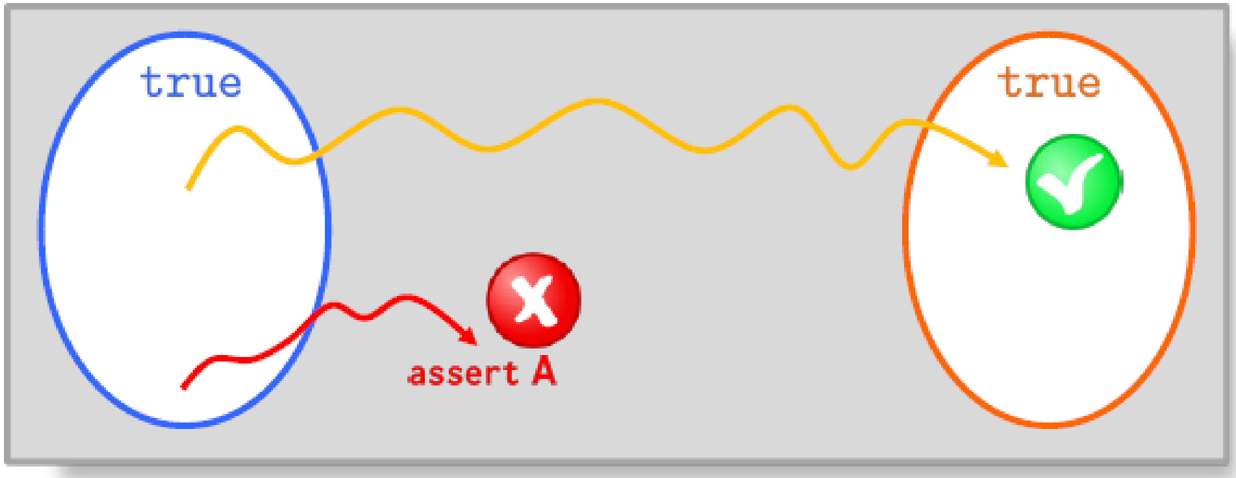
```
{ 0 ≤ b*b – 4*c }
discriminant := b*b – 4*a*c;
x := (-b + √discriminant) / 2
{ a*x² + b*x + c = 0 }
```

```
// Fermat’s last theorem
assert 0<x && 0<y && 0<z ==>
       x*x*x + y*y*y != z*z*z
```

→ Verifiers should help users to localize and fix verification failures

How does verification fail?

Verification condition: ***(E)WP***`(S, true)` valid



If `S` contains no assertions, then ***(E)WP***`(S, true)` is valid.

How many assertions could fail? Which ones should we report?

```
{ (x < 17 ==> x < 26)
  && (x >= 17 ==> x > 42 && x > 17 && x != 16) }
{
  { x < 17 ==> x < 26 }
  assume x < 17;
  { x < 26 }
  assert x < 26
  { true }
} [] {
  { x >= 17 ==> x > 42 && x > 17 && x != 16 }
  assume x >= 17;
  { x > 42 && x > 17 && x != 16 }
  assert x > 42;
  { x > 17 && x != 16 }
  assert x > 17;
  { x != 16 }
  assert x != 16
  { true }
} { true }
```

Solution

```
{ (x < 17 ==> x < 26)
  && (x >= 17 ==> x > 42 && x > 17 && x != 16) }
{
  { x < 17 ==> x < 26 }
  assume x < 17;
  { x < 26 }
  assert x < 26 // never fails
  { true }
} [] {
  { x >= 17 ==> x > 42 && x > 17 && x != 16 }
  assume x >= 17;
  { x > 42 && x > 17 && x != 16 }
  assert x > 42; // can fail ➔ report!
  { x > 17 && x != 16 }
  assert x > 17; // can fail ➔ report?
  { x != 16 }
  assert x != 16 // can fail ➔ report?
  { true }
} { true }
```

Error localization

If `S` contains no assertions, then ***(E)WP***`(S, true)` is valid.

- Goal: report assertions that fail verification
- How to identify failing assertions?
- How many failing assertions should we report?
- How do we deal with dependencies between failures?

```
assert MIN_INT <= x + y
assert x + y <= MAX_INT
res := x + y

assert MIN_INT <= x - y
assert x - y <= MAX_INT
d := x - y

assert d != 0
res := res / d
```

➔A single VC ***EWP***`(S, true)` cannot report which parts of a proof fail

Idea: Split VC at assertions into *multiple* proof obligations

sets of predicates

`S`	***MWP***`(S, M)`
`assert` `R`	`M ∪ {R}`
`assume` `P`	`{P ==> Q	Q ∈ M}`
`S1; S2`	***MWP***`(S1,` ***MWP***`(S2, M))`
`S1 [] S2`	***MWP***`(S1, M) ∪` ***MWP***`(S2, M)`

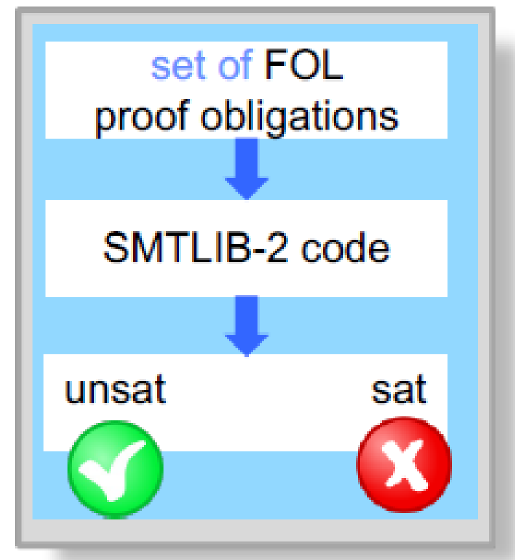
- New verification condition:

  Every `P in` ***MWP***`(S, {})` is valid

- All predicates are implication chains

  `P ==> Q ==> R`

  not valid ➔ `assert R` failed



Exercise: error localization

- Compute ***MWP*(S, {})** for the statement on the right.
- Which of the proof obligations are valid?
- For each *invalid* proof obligation, determine an initial state such that the corresponding assertion fails
- Verify the example on the right in Viper using the Carbon verifier. How many error messages do you get?

```
{
  assert x == 7
} [ ] {
  assert x == 2
  assert x > 0
}
```

```
method foo(x: Int, b: Bool) {
  if(b) {
    assert x == 7
  } else {
    assert x == 2
    assert x > 0
  }
}
```

Solution: error localization

- `MWP(S, {})` = { x == 7, x == 2, x > 0 }

- Since x has an arbitrary value, none of the three proof obligations are valid

- Initial states
  - x == 7 may fail for initial state x == 0
  - x == 2 may fail for initial state x == 0
  - There is no execution in which x > 0 fails because each execution where x is non-positive fails already at the previous assertion

- Viper reports only the first two assertions

```
{
  assert x == 7
} [ ] {
  assert x == 2
  assert x > 0
}
```

```
method foo(x: Int, b: Bool) {
  if(b) {
    assert x == 7
  } else {
    assert x == 2
    assert x > 0
  }
}
```

Avoiding masked verification errors

- ***WP*** and ***MWP*** ignore the order of assertions

```
WP(assert P; assert R, Q)  =  P && R && Q
```

```
MWP(assert P; assert R, M)  =  M∪{ P }∪{ R }
```

```
assert x == 2
assert x > 0
```

```
assert x > 0
assert x == 2
```

- Issue: second assertion should only be checked if it passed the first assertion

- Solution: add an assumption after each assertion

```
assert R
```

→

```
assert R
assume R
```

Avoiding masked verification errors

```
{ x == 2 ==> x > 0, x == 2 }
assert x == 2
{ x == 2 ==> x > 0 }
assume x == 2
{ x > 0 }
assert x > 0
{ }
assume x > 0
{ }
```

Case 1: one assertion fails

```
{ x > 0 ==> x == 2, x > 0 }
assert x > 0
{ x > 0 ==> x == 2 }
assume x > 0
{ x == 2 }
assert x == 2
{ }
assume x == 2
{ }
```

Case 2: both assertions fails

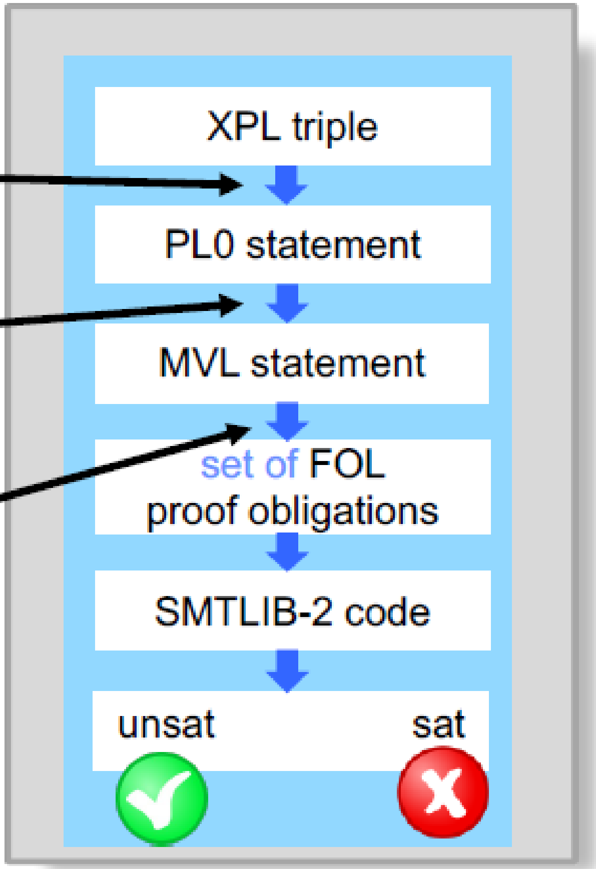
The toolchain so far

Encode

- Pre- and postconditions
- If-statements

- Variable declarations
- DSA transformation
- Passification
- Avoid masked errors

Efficient *MWP*



All *encodings* are sound and complete

The Error Propagation Toolchain

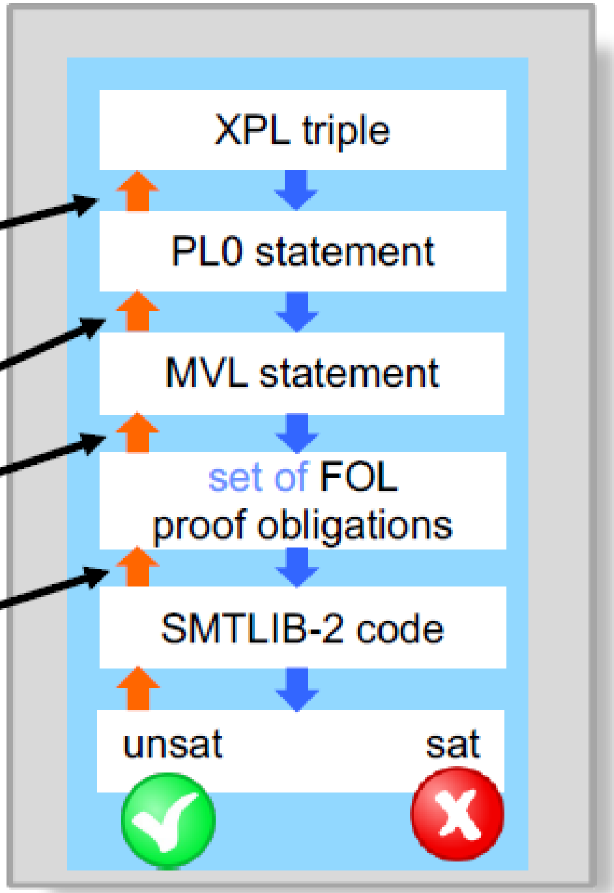
Keep back-translation map from encoding to original ➔ report errors for original problem

- Assertions ➔ postconditions, assertions
- Assume/Choice statements ➔ if-statements

- Versioned variables (DSA) ➔ original variables
- Assumptions ➔ assignments, masked errors

- Proof obligations ➔ assertions

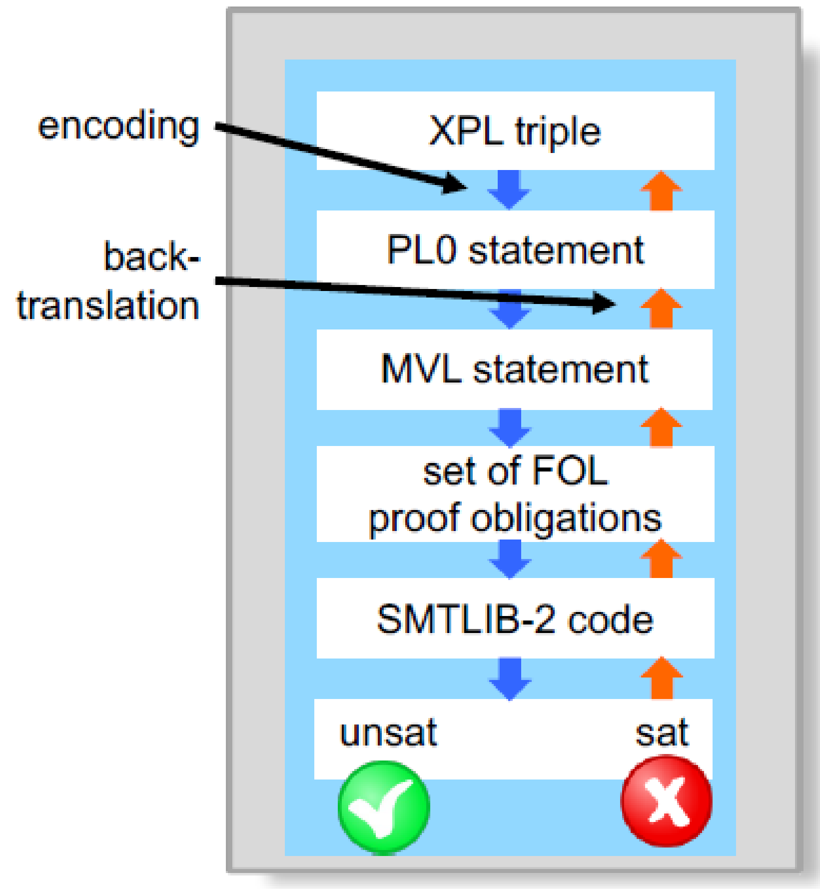
- Solver results ➔ proof obligations



Wrap-up

- "Verification as compilation"

- Wishlist for each translation **A** ➡ **B**
  - Sound encodings
  - Complete encodings
  - Linear-size verification conditions
  - Localize and back-translate errors



Error reporting in Viper

- Viper has two verification backends
  - Counterexamples can be enabled via command line option

- Carbon
  - Uses weakest preconditions, similarly to the technique taught in this course, but uses a more efficient approach
  - Reports multiple verification failures

- Silicon
  - Uses symbolic execution (similar to ***SP***)
  - Reports one verification error per method
  - Default verifier in the IDE