Konzepte objektorientierter Programmierung – Lecture 12 –

Prof. Dr. Peter Müller

Software Component Technology

Wintersemester 05/06

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Agenda for Today

12. Extended Static Checking

12.1 Overview and Demo

12.2 Program Transformations

12.3 Verification Conditions

Objectives

- Application of formal methods
- Verification techniques

[This lecture is partly based on Rustan Leino's lecture 0 at the *Summer school on Formal Models of Software.* Tunisia, 2002/2003]

User's view

Program with specifications → **Program Checker** → Error messages

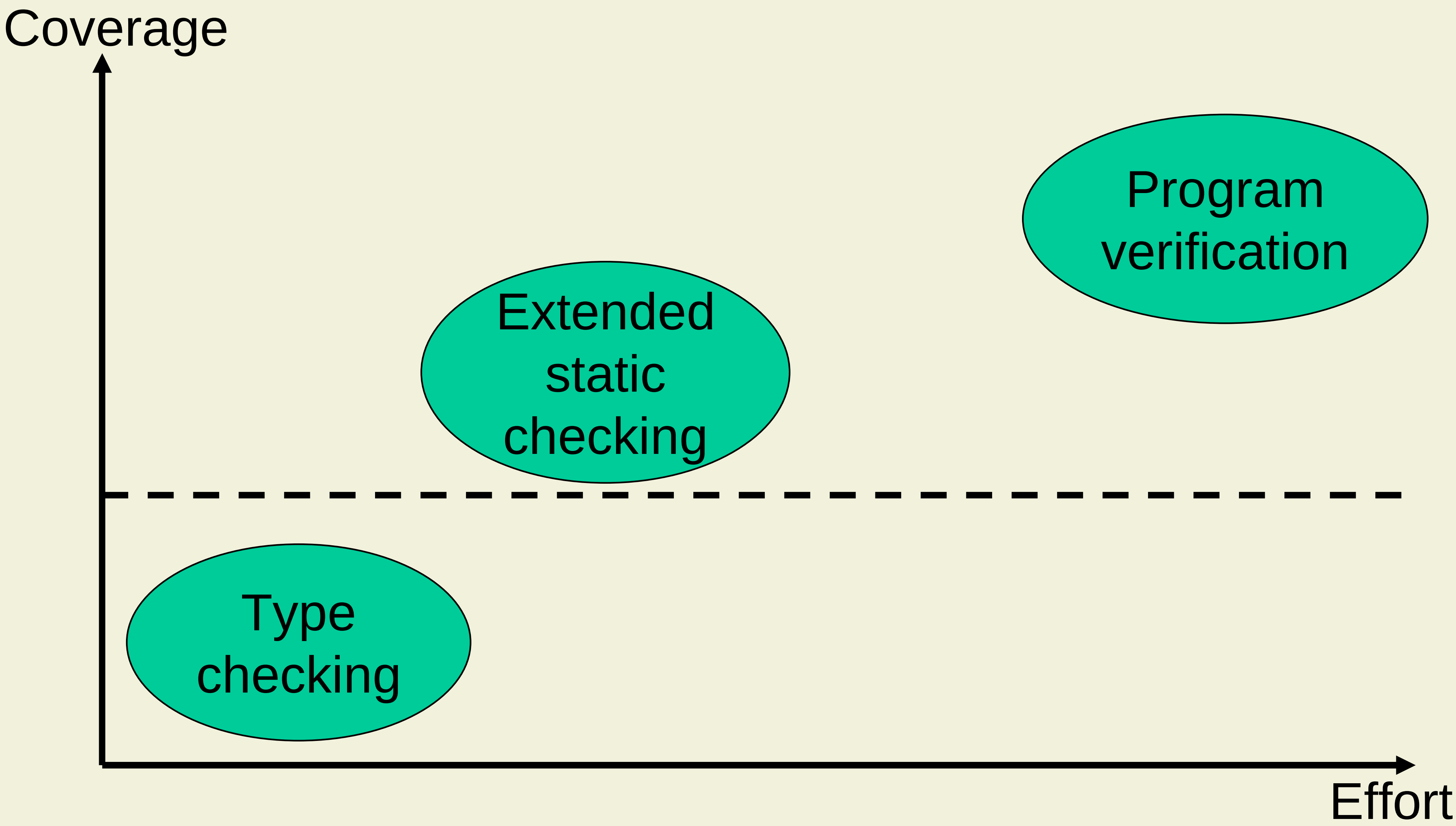
```
public class Bag {
  private /*@non_null*/ int[ ] a;
  private int n;

  //@ invariant 0 <= n && n <= a.length;

  public Bag( /*@non_null*/ int[ ] initial )
   { … }
  …
  …
}
```

Bag.java:18: Array index possibly too large

Extended Static Checking



ESC/Java

- ESC/Java checks programs for coding errors …
  - Null-dereferences
  - Array bounds errors
  - Illegal casts
  - Race conditions, deadlocks
- … and specification violations
  - Simple pre-post specifications
  - Simple invariants

Program Checker Design Tradeoffs

- Objectives
  - Fully automated reasoning
  - As little annotation overhead as possible
  - Performance
- ESC/Java is not sound
  - Errors may be missed
- ESC/Java is not complete
  - Warnings do not always report errors (false alarms)

ESC/Java Demo

```
class Bag {
  int[ ] a;
  int n;


  int extractMin( ) {
    int m = Integer.MAX_VALUE;
    int mindex = 0;
    for ( int i = 1; i <= n; i++ )
      if ( a[ i ] < m )
        { mindex =i; m = a[ i ]; }
    n--;
    a[ mindex ] = a[ n ];
    return m;
}
```

ESC/Java Demo

```
class Bag {
  int[ ] a;        //@ invariant a != null;
  int n;


  int extractMin( ) {
    int m = Integer.MAX_VALUE;
    int mindex = 0;
    for ( int i = 1; i <= n; i++ )
      if ( a[ i ] < m )
        { mindex =i; m = a[ i ]; }
    n--;
    a[ mindex ] = a[ n ];
    return m;
}
```

Warning:
Possible null deference.
Plus other warnings

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ESC/Java Demo

```
class Bag {
  int[ ] a;       //@ invariant a != null;
  int n;          //@ invariant 0 <= n && n <= a.length;


  int extractMin( ) {
    int m = Integer.MAX_VALUE;
    int mindex = 0;
    for ( int i = 1; i <= n; i++ )
      if ( a[ i ] < m )
        { mindex =i; m = a[ i ]; }
    n--;
    a[ mindex ] = a[ n ];
    return m;
}
```

Warning:
Array index possibly too large

ESC/Java Demo

```
class Bag {
  int[ ] a;      //@ invariant a != null;
  int n;         //@ invariant 0 <= n && n <= a.length;

  int extractMin( ) {
    int m = Integer.MAX_VALUE;
    int mindex = 0;
    for ( int i = 0; i < n; i++ )
      if ( a[ i ] < m )
        { mindex =i; m = a[ i ]; }
    n--;
    a[ mindex ] = a[ n ];
    return m;
}
```

Warning:
Array index possibly too large

ESC/Java Demo

```
class Bag {
  int[ ] a;      //@ invariant a != null;
  int n;         //@ invariant 0 <= n && n <= a.length;

  //@ requires n > 0;
  int extractMin( ) {
    int m = Integer.MAX_VALUE;
    int mindex = 0;
    for ( int i = 0; i < n; i++ )
      if ( a[ i ] < m )
        { mindex =i; m = a[ i ]; }
    n--;
    a[ mindex ] = a[ n ];
    return m;
}
```

Warning:
Possible negative array index

Interface Specifications

- Annotation language is tailored towards ESC
  - No method invocations in specifications
  - No data abstraction (no models)
- Simple annotations (non_null)
- Method annotations
  - pre-post specifications
  - modifies clauses
- Object invariants

Assume Annotation

- Assume a condition **without checking**

```
//@ assume E;
```

- Useful to avoid full verification
- Assume is a source of unsoundness

```
int prime( int n ) { … }
void m( ) {
  int p = prime( 5 );
  //@ assume (\forall int i,j; i*j==p && i>0 && j>0 ==> i==1 ¦¦ j==1);
  … // code works only if p is prime
}
```

Assert Annotation

- Assert can be used to write specifications inside method bodies

```
//@ assert E;
```

- Semantics:
  - Evaluate E
  - If E then skip else abort

```
void m1( ) {
  int a = 5 ;
  int b = a - 5 ;
  //@ assume b != 0;
  int c = a / b;
}
```

```
void m2( ) {
  int a = 5 ;
  int b = a - 5 ;
  //@ assert b != 0;
  int c = a / b;
}
```

Tool Architecture

Annotated Java program

Translator

Verification condition

Automatic Theorem Prover

Valid

Resource exhausted

Counterexample context

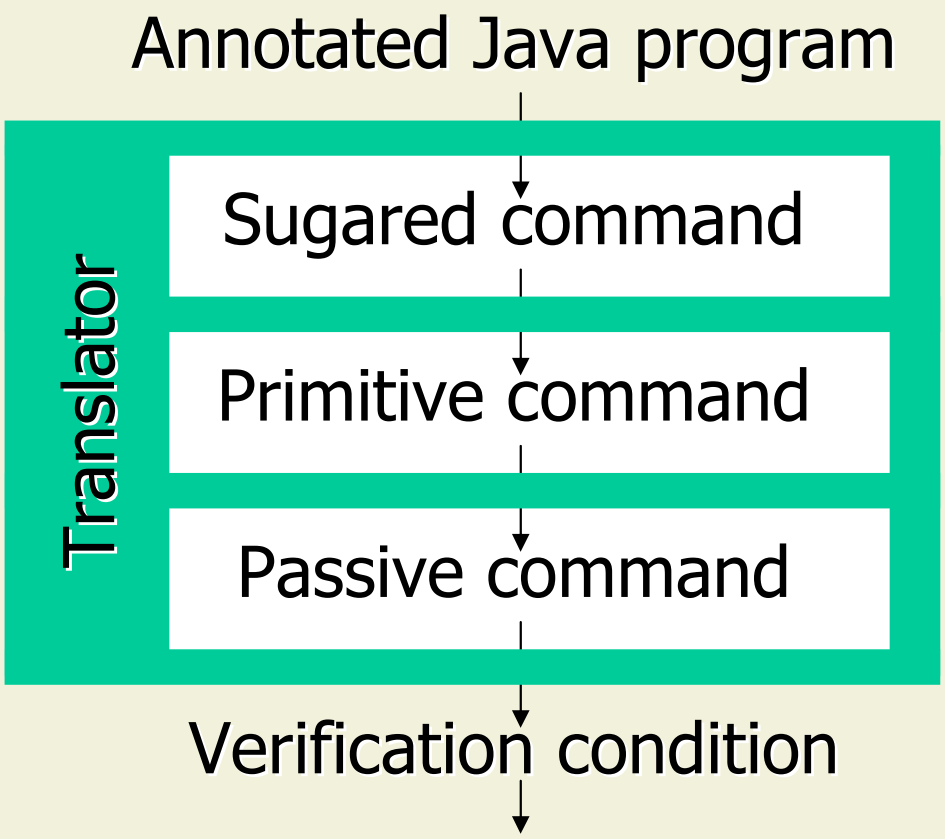
Post Processor

Warning messages

12. Extended Static Checking

Translator

- Java programs are difficult to handle
  - Many different statements and expressions
- Approach
  - **Translate** source program **to guarded command language**
  - **Make simplifications** to facilitate extended static checking



Sugared Commands

- Assume and assert are treated as statements
- If- and switch-statements are replaced by **nondeterministic choice []**
- Only one **loop statement** (with loop invariant)

```
S,T  ::=assert E
       |  assume E
       |  x = E
       |  S ; T
       |  S [] T
       |  loop {inv E} S → T end
       |  call x =  t.m(E)
       |  …
```

Sugared Commands: Example

```
x =  t.f.g;
if (x < 0) x = -x;
/*@ assert x >= 0; */
```

```
tmp = t.f;
x = tmp.g;
if (x < 0)
  x = -x;
else
  ;
/*@ assert x >= 0; */
```

```
assert t ≠ null;
tmp =  select( f,t );
assert tmp ≠ null;
x = select( g,tmp )

(
  assume x < 0;
  x = -x
[]
  assume ¬(x < 0)
);

assert x >= 0
```

From Sugared to Primitive Commands

```
S,T  ::=assert E
       |  assume E
       |  x = E
       |  S ; T
       |  S [] T
       |  loop {inv E} S → T end
       |  call x =  t.m(E)
       |  …
```

→

```
S,T  ::=assert E
       |  assume E
       |  x = E
       |  S ; T
       |  S [] T
       |  …
```

Treatment of Loops

- Reasoning about loops is difficult
- Approach in ESC/Java: Consider **only first iteration** of the loop plus evaluation of condition after first iteration

```
while( n > 0 ) {
  res = res * n;
  n = n - 1;
}
```

```
if (n > 0) {
  res = res * n;
  n = n - 1;
  if (n > 0)
    assume false;
}
```

```
( assume n > 0;
  res = res * n;
  n = n – 1
  ( assume n > 0;
    assume false;
  []
    assume ¬(n > 0)  )
[]
  assume ¬(n > 0)  );
```

Treatment of Loops: Problems

```
int n = 1;
while( true ) {
  int r = 5 / n;
  n = n - 1;
}
```

```
n = 1;
if ( true ) {
  int r = 5 / n;
  n = n - 1;
  if ( true )
    assume false;
}
```

```
n = 1;
(
  assume true;
  int r = 5 / n;
  n = n - 1;
  (
    assume true;
    assume false;
  []
    assume ¬ true  )
[]
  assume ¬true
);
```

Treatment of Loops is Unsound

- **Problems** might occur **in later iterations**
- **Practical experience** shows that most errors are caught in first iteration
- loopSafe option for **sound treatment** of loops
  - Based on specification of **loop invariant**

Treatment of Method Calls

- Method calls are handled by referring to the **specification of the called method**
  - Preconditions have to be **established** (assert)
  - Postconditions can be **assumed** to hold (assume)

```
//@ requires P;
//@ ensures Q;
int m( T t ) { … }
```

```
call x = m( E );
```

- Using specifications enables **modular checking**
  - One method at a time
  - Dynamic method binding is handled by **behavioral subtyping**

Treatment of Method Calls: Example

```
//@ requires P;
//@ ensures Q;
int m( T t ) { … }
```

```
call x = m( E );
```

Pattern

```
var t,\result in
  t = E;
  assert P;
  // execution of m skipped
  assume Q;
  x = \result;
end
```

```
//@ requires d > 0;
//@ ensures \result == ( n / d );
int div( int n, int d ) { … }
```

```
call x = div( 5,3 );
assert x <= 5;
```

Example

```
var n,d,\result in
  n = 5; d = 3;
  assert d > 0;
 // execution of m skipped
  assume \result == ( n / d );
  x = \result;
end;
assert x <= 5
```

Side-Effects and Modifies Clauses

```
//@ requires d > 0;
//@ modifies this.attr;
//@ ensures \result == ( n / d );
int div( int n, int d ) { … }
```

```
this.attr = 5;

call x = div( 5,3 );
assert x <= 5;

assert this.attr == 5;
```

⇒

```
this.attr = 5;

var n,d,\result in
  n = 5; d = 3;
  assert d > 0;
  // execution of m skipped
  assume \result == ( n / d );
  x = \result;
end;
assert x <= 5

assert this.attr == 5;
```

Playing Havoc

```
//@ requires d > 0;
//@ modifies this.attr;
//@ ensures \result == ( n / d );
int div( int n, int d ) { … }
```

```
this.attr = 5;

call x = div( 5,3 );
assert x <= 5;

assert this.attr == 5;
```

```
this.attr = 5;

var n,d,\result in
  n = 5; d = 3;
  assert d > 0;
  var attr0 in
    attr0 = this.attr;
    havoc this.attr;
    assume \result == ( n / d );
    x = \result;
  end;
end;
assert x <= 5

assert this.attr == 5;
```

- **havoc** assigns an **arbitrary value** to a variable

From Imperative to Passive Commands

```
S,T ::= assert E
     |  assume E
     |  x = E
     |  S ; T
     |  S [] T
     |  …
```

```
S,T ::= assert E
     |  assume E
     |  S ; T
     |  S [] T
     |  …
```


ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Eliminating Assignments

- Since **we do not execute** the translated program, we do **not** have to **actually modify the variables**
- It is sufficient for the checker to know that after the assignment v = exp; the condition v == exp holds
- Approach: **Replace assignments by assumptions**

v = exp; → **assume** v == exp

Eliminating Assignments (cont'd)

- **Problem:** v appears on right-hand side

v = v + 1; → **assume** v == v + 1

- **Solution:** New variable v_n for each assignment

(**assume** x < 0;
  x = -x
[]
  assume ¬(x < 0)
);
assert x >= 0

(**assume** x_0 < 0;
  x_1 = $-x_0$;
  x_2 = x_1
[]
  assume ¬(x_0 < 0);
  x_2 = x_0);
assert x_2 >= 0

(**assume** x_0 < 0;
  assume x_1 == $-x_0$;
  assume x_2 == x_1
[]
  assume ¬(x_0 < 0);
  assume x_2 == x_0);
assert x_2 >= 0

12. Extended Static Checking

Weakest Preconditions

- A **Hoare triple**

  $$\{\, \mathbf{P} \,\}\ \mathbf{S}\ \{\, \mathbf{Q} \,\}$$

  says that if command **S** is started in a state satisfying **P**, then **S** terminates **without error** in a state satisfying **Q**

- The **weakest precondition** of a command **S** with respect to a postcondition **Q**, written wp(**S**, **Q**), is the weakest **P** such that

  $$\{\, \mathbf{P} \,\}\ \mathbf{S}\ \{\, \mathbf{Q} \,\}$$

wp for Passive Commands

wp(**assert** E,Q)	=	E && Q
wp(**assume** E, Q)	=	E ==> Q
wp(S;T, Q)	=	wp(S, wp(T,Q))
wp(S [] T, Q)	=	wp(S,Q) && wp(T,Q)

wp Calculation

wp(S;T, Q) = wp(S, wp(T,Q))
wp(S [] T, Q) = wp(S,Q) && wp(T,Q)

wp(**assert** w == 1;
(**assume** i == 5;
[] **assume** w == 1);
, i==5) =

wp(**assert** w == 1; , wp((**assume** i == 5;
[] **assume** w == 1); , i==5)) =

wp(**assert** w == 1; , wp(**assume** i == 5 , i==5) &&
wp(**assume** w == 1 , i==5)) =

wp Calculation (cont'd)

wp(**assert** E,Q) = E && Q
wp(**assume** E, Q)= E ==> Q

wp(**assert** w == 1; , wp(**assume** i == 5 , i==5) &&

wp(**assume** w == 1 , i==5)) =

wp(**assert** w == 1; ,

(i==5 ==> i==5) && (w==1 ==> i==5)) =

w==1 && (i==5 ==> i==5) && (w==1 ==> i==5)

Verification Conditions

- The **verification condition** VC_m for a method m is

$$\boxed{Pre_m \text{ ==> } wp(\ body_m, Post_m\ )}$$

- **Modifies clauses** are **used** to reason about calls, **but not checked**
  - Another source of unsoundness

Verification Conditions (cont'd)

- **Universal background predicate** BP_{Univ} specifies properties of the programming language
  - Example: $(\forall t: \ t <: \ t)$
- **Type-specific background predicate** specifies properties of the program
  - Bag <: java.lang.Object
- The verification condition passed to the theorem prover is

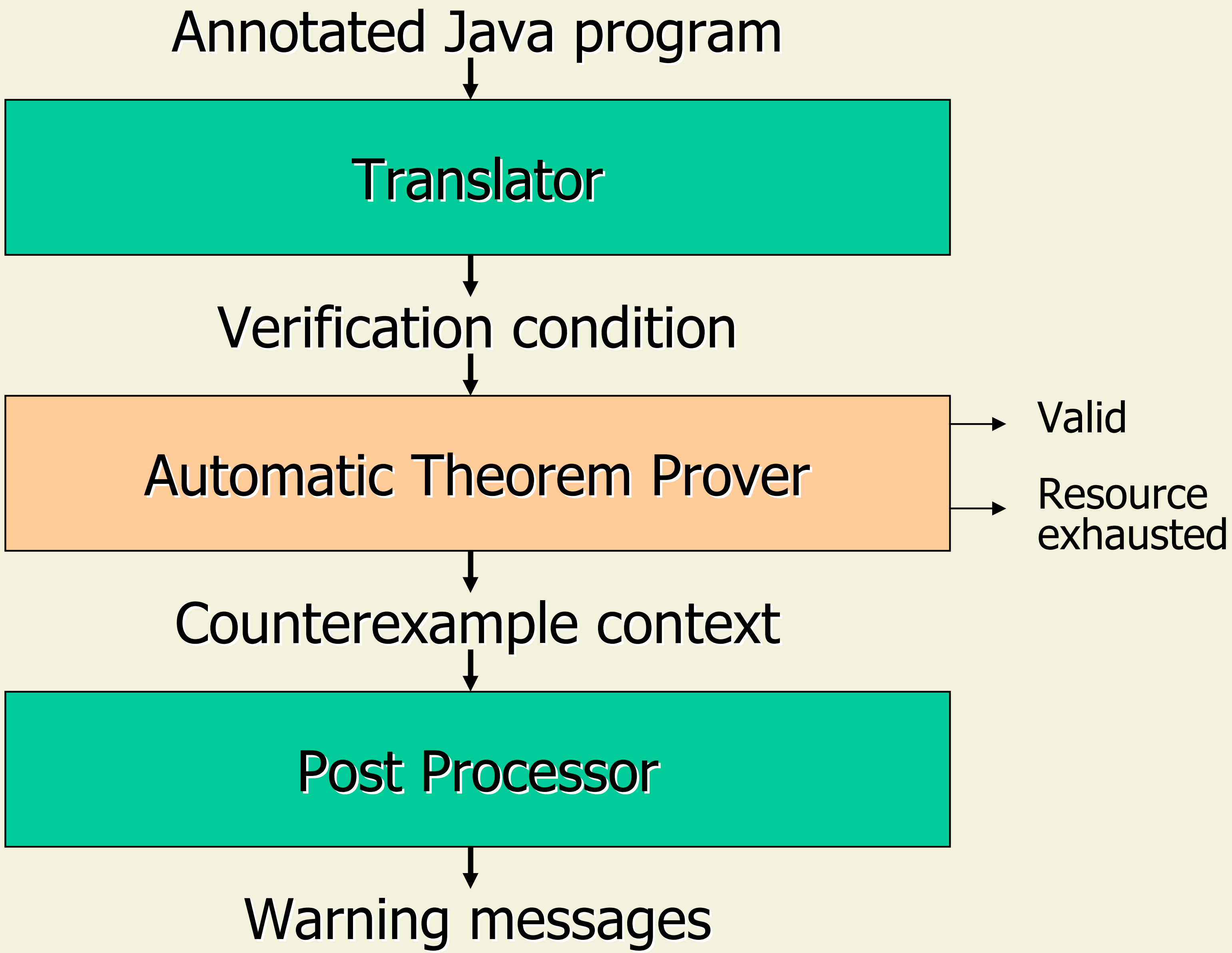
$$\boxed{BP_{Univ} \;\&\&\; BP_T \;==>\; VC_m}$$

Verification Condition Example

```
(AND
 (<: T_T |T_java.lang.Object|)
 (EQ T_T (asChild T_T |T_java.lang.Object|))
 (DISTINCT arrayType |T_boolean| |T_char| |T_byte| |T_short| |T_int|
                 |T_long| |T_float| |T_double| |T_.TYPE|
                 T_T |T_java.lang.Object|)))
(EXPLIES
 (LBLNEG |vc.T
  (IMPLIES
   (AND
    (EQ |elems@
    (EQ elems (
    (< (eClosed
    (EQ LS (asL
    (EQ |alloc@
   (NOT
    (AND
     (EQ |@true
     (OR
      (AND
       (OR
        (AND
         (< |x:2.21| 0)
         (LBLPOS |trace.Then^0,3.15| (EQ |@true| |@true|))
         (EQ |x:3.17| (- 0 |x:2.21|))
```

```
class T {
  static int abs( int x ) {
    if ( x < 0 ) { x = -x; }
    //@ assert x >= 0;
  }
}
```

Tool Architecture



Theorem Prover: “Simplify”

- Automatic: **No user interaction**
- **Refutation based**: To prove φ it will attempt to satisfy ¬φ
  - If this is possible, a counterexample is found, and we know a reason why φ is invalid
  - If it fails to satisfy ¬φ then φ is considered to be valid

Time Limits

- Logic used in Simplify is **semi-decidable**
  - Each procedure that proves all valid formulas loops forever on some invalid ones
- Simplify works with a **time limit**
  - When time limit is reached, counterexample is returned
  - Longer computation might turn out that returned counterexample is inconsistent
- Time limits are a source of **incompleteness**
  - Spurious counterexamples lead to spurious warnings

Experience: Annotations

- Capture common design decisions
- Suggested immediately by warnings
- Overhead: 4-10% of source code
- ~1 annotation per field or parameter
- Most common annotations:
  - non_null
  - Container element types

Experience: Performance

- 50% of all methods: < 0.5 s
- 80% of all methods: < 1 s
- Time limit: 300 s

- Total time for Javafe (~40kloc): 65 min

References

- Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata: *Extended static checking for Java*. PLDI, ACM Press, 2002.
  Available from course web site

- Download
  - ESC/Java (tool, documentation, sources): http://research.compaq.com/SRC/esc
  - Boogie (tool, documentation, sources): http://research.microsoft.com/specsharp/