# 272: Software Engineering
# Winter 2024

Instructor: Tevfik Bultan

## Lecture: Extended Static Checking

# A Simple Class and Its Contract in JML

```
class BankAccount {
  int: balance;
  //@ invariant balance >= 0;

  withdraw(int: i) {
    //@ requires balance >= i and i >= 0;
    balance = balance - i;
    //@ ensures balance == \old(balance) - i;
  }
  deposit(int: i) {
    //@ requires i >= 0;
    balance = balance + i;
    //@ ensures balance == \old(balance) + i;
  }

  boolean isEmpty() {
    return balance == 0;
    //@ ensures result == (balance == 0);
  }
}
```

# Same Example, Contract Written as Assertions

```
class BankAccount {
  int: balance;
  withdraw(int: i) {
    int oldbalance = balance;
    assert(balance >= i and i >= 0);
    balance = balance - i;
    assert(balance == oldbalance - i);
  }
  deposit(int: i) {
    int oldbalance = balance;
    assert(i >= 0);
    balance = balance + i;
    assert(balance == oldbalance + i);
  }

  boolean isEmpty() {
    boolean result = (balance == 0);
    assert (result == (balance == 0));
    return result;
  }
}
```

Question: Where do we put the class invariant?
Answer: Add as a conjunction to pre and post-conditions

# Dynamic Contract Monitoring

- We can do dynamic contract monitoring for such specifications using runtime monitoring tools
- When the contract fails we know that there is an error in the implementation
  - We can identify who is responsible for the contract violation (i.e., the caller or the callee)
- Note that the contract monitoring is dynamic, i.e., it is done during the program execution
  - If we do not observe a contract violation for a set of executions, that does not mean that a contract violation will never happen.
- But some of the implementation code is so close to the pre and post-conditions specified in the contract, it looks like we should be able to ***prove*** that the implementation is correct with respect to the contract
  - Proving the implementation correct with respect to the contract means proving that there will never be a contract violation for any execution of the program!

# Example

- Here is the question:
  - If we assume that the pre-condition holds, then does the implementation guarantee that the post-condition is satisfied?
  - I.e., if the pre-condition holds, then is it guaranteed that the assertion that checks the post-condition will not cause an assertion failure?

```
withdraw(int: i) {

  int oldbalance = balance;
  assert(balance >= i and i >= 0);

  balance = balance - i;

  assert(balance == oldbalance - i);

}
```

this is the implementation part

these are the assertions that come from the contract specification

# Extended Static Checking

- Extended Static Checking (ESC) is a static analysis technique that relies on automated theorem proving

- Goals of  Extended Static Checking:
  - prevent common errors in programs
  - make it easier to write dependable programs
  - increase programmer productivity

# Extended Static Checking

- Help programmer in writing dependable programs by generating warning messages pointing out potential errors at compile time

- The programmer annotates the code using design-by-contract style assertions written in Java Modeling Language (JML)
  - Assertions can be added to clarify the contract of a method based on the warnings generated by the ESC tool

- Big difference between ESC and Dynamic design by contract monitoring tools:
  - ESC checks errors at compile time, before running the program, for ***all*** possible executions of the program!

# Extended Static Checking

- Extended Static Checking (ESC) approach and tools was developed by researchers at
  - Systems Research Center (SRC) (aka HP SRC Classic, Compaq's Systems Research Center, Digital Equipment Corporation Systems Research Center)

- There are other projects that build on ESC approach
  - ESC/Java 2
    - https://kindsoftware.com/products/opensource/ESCJava2/
  - Spec#
    - http://research.microsoft.com/projects/specsharp/
  - Dafny
    - https://dafny.org/

# Extended Static Checking

- Suggested reading:
  - ``Extended static checking for Java." Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. In Proc. of PLDI 2002.

- Other References
  - ``Applications of extended static checking." K.R.M. Leino (In proceedings of SAS'01, ed. Patrick Cousot, volume 2126 of Springer LNCS, 2001.)
  - ``Extended Static Checking: a Ten-Year Perspective." K.R.M. Leino (In proceedings of the Schloss Dagstuhl tenth-anniversary conference, ed. Reinhard Wilhelm, volume 2000 of Springer LNCS, 2001.)
  - ``ESC/Java User's Manual.'' K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Technical Note 2000-002, Compaq Systems Research Center, October 2000.
- The slides below are based on these references

# Extended Static Checking

- It is known that  there are a lot of properties that cannot be checked statically:
  - For example there is no way to automatically check that
    - a program will halt for all possible inputs
    - there exists an execution for a program in which a certain program statement is executed
  - These problems are undecidable!
- So what can we do? Possible approaches:
  - Look at properties that can be checked statically
    - type checking looks at type-errors
  - Check properties dynamically by instrumenting the program
    - Dynamic Design-by-Contract tools that monitor contract violations at runtime

# Extended Static Checking

- ESC uses a different approach
  - Check for properties that are undecidable but use approximate verification techniques
    - If ESC reports that “the program is correct” this may not imply that the program is really correct (i.e., ESC Java may report “false negatives”)
    - If ESC reports that “the program is incorrect” this may not imply that there is really an error in the program (i.e., ESC Java may report “false positives”)
      - ESC/Java generates warning messages which indicate that there *may be* errors, but does not imply that *there is* an error
  - Ask for user help
    - User can remove spurious warning messages by annotating the code with assertions that are strong enough to imply the properties that are being checked

# Extended Static Checking

- As mentioned in the previous slide there are two types of approximate verification one can do
  - Use approximate analysis that allows false positives
    - When verifier finds an error that means that there could be an error in the program, it does not mean that there is guaranteed to be an error
    - Note that now the problem is trivially decidable: A verification tool can always claim that there is an error in any program
      - probably nobody will use such a tool
  - Use approximate analysis that allows false negatives
    - If the verifier does not find an error in the program that does not imply that there are no errors in the program, it just means that the verifier was not able to find an error

# Extended Static Checking

- A verification tool is sound if it generates an error whenever a program has an error (i.e., it does not generate false negatives)

- A sound verification tool is also complete if every error it reports is a genuine error rather than a false positive

- For undecidable verification problems we know that we can not develop both sound and complete verification tools
  - We can develop sound but incomplete tools which generate false positives but no false negatives
  - We can develop complete but unsound tools which generate false negatives but no false positives

# Completeness and Soundness in Logic

- In mathematical logic
  - completeness means that all truths should be provable
  - soundness means that nothing false should be provable

- If we define the task of program verification as proving theorems of the form “program P is correct”
  - Soundness becomes equivalent to never saying a program is correct when it has an error (i.e., no false negatives)
  - Completeness becomes equivalent to always being able to say a program is correct whenever it is indeed correct (i.e., no false positives)

# Extended Static Checking

- Unfortunately, ESC/Java allows both kinds of approximations

- This means that
  - When ESC/Java generates a warning you have to validate that there is indeed an error
  - When ESC/Java does not generate any warnings this does not imply that your program is free of errors

- Well, it seems like ESC/Java does not guarantee anything
- But you can think of ESC/Java as a testing tool, which actually covers much more cases than you can achieve using testing, unfortunately it requires you to validate the results

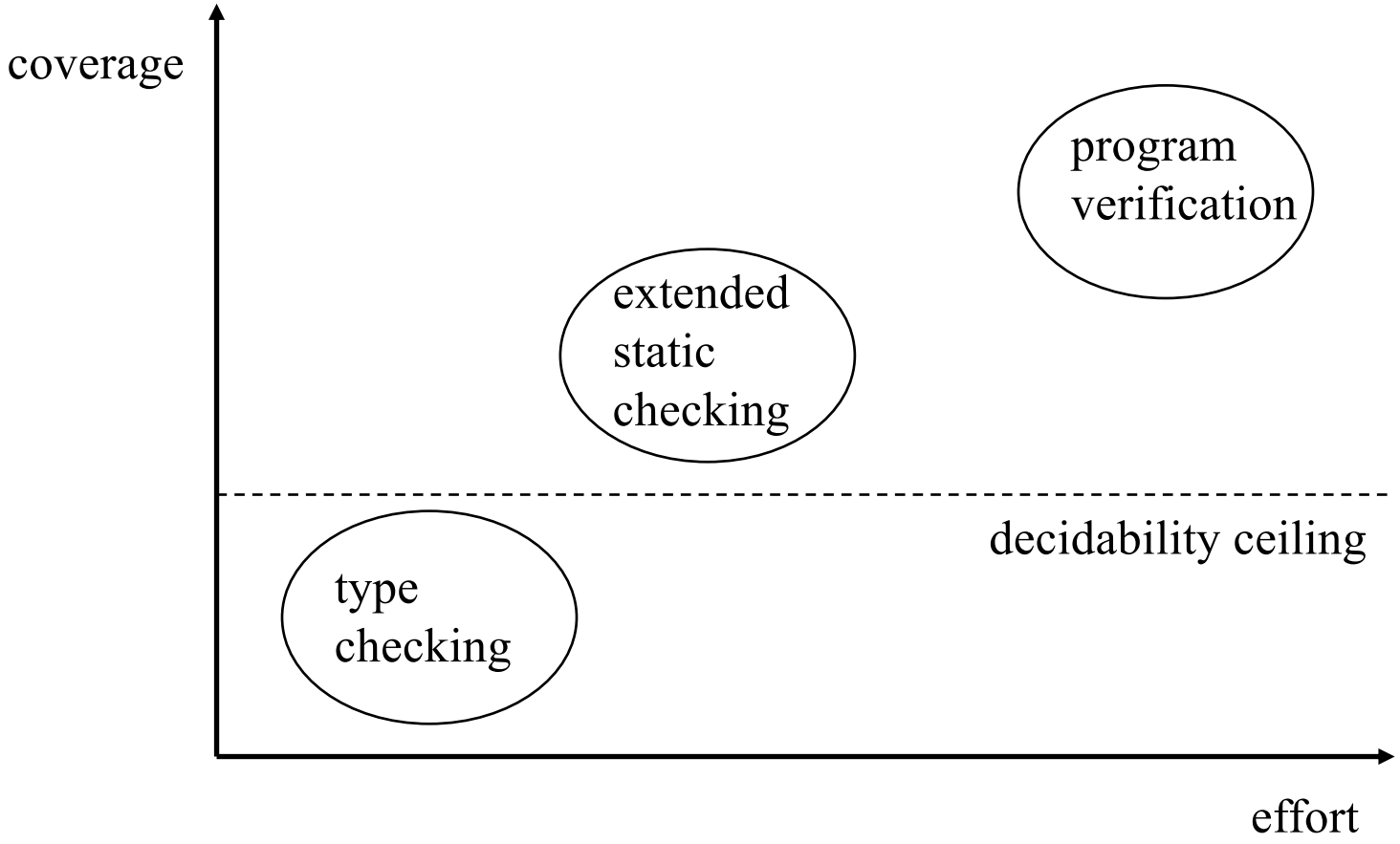
# Extended Static Checking

- ESC/Java also restricts the types of properties that it checks

- Verification of general properties of programs may require a lot of input from the programmer
  - Programmer may need to enter a lot of assertions (such as loop invariants, etc.) to remove the false positives.

- ESC/Java tries to achieve a balance between the properties it covers and the effort required by the programmer
  - In this sense it is different than the more ambitious program verification tools based on Hoare Logic and automated theorem provers

# Extended Static Checking



# Extended Static Checking

- Two extended static checkers were developed at SRC
  - ESC/Modula-3
  - ESC/Java, I will talk about ESC/Java

- ESC/Java used an automated theorem prover called Simplify
  - Simplify has various theories related to programs embedded in it and it checks validity or satisfiability of verification conditions
  - Since then there has been a lot of work on Satisfiability Modula Theories (SMT) (which are a specific type of theorem provers). For example Z3 is a SMT-solver from Microsoft that is widely used in program analysis

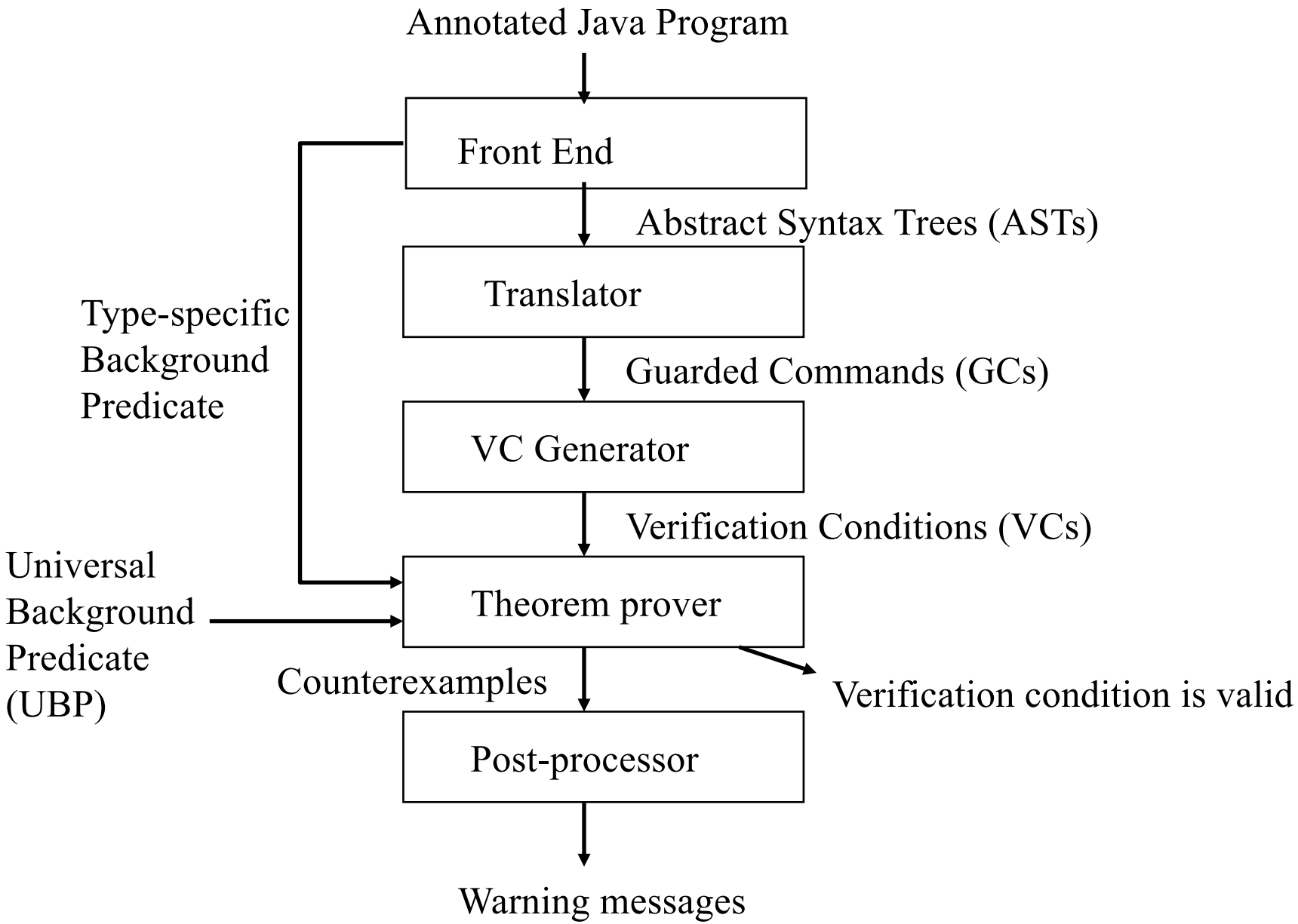
- ESC/Java uses the Java Modeling Language (JML) as the specification language

# Extended Static Checking

- Given a program, ESC tool generates a logical formula, called a ***verification condition***, that is valid if and only if the program is free of the classes of errors under consideration

- An automated theorem prover is used to check if the negation of the verification condition is satisfiable
  - Any satisfying assignment to the negation of the verification condition is a ***counter-example*** behavior that demonstrates a bug

# ESC/Java Architecture



# Types of Errors

- ESC/Java checks three types of errors

1. Common programming errors such as null dereferences, array index bound errors, type-cast errors, division by zero, etc.

2. Common synchronization errors such as race conditions and deadlocks

3. Violations of program annotations, i.e., static checking of contracts such as preconditions, postconditions or invariants

# An Example: Bag.java

```
1: class Bag {
2:   int size;
3:   int[] elements;
4:
5:   Bag(int[] input) {
6:     size = input.length;
7:     elements = new int[size];
8:     System.arraycopy(input,0,elements,0,size);
9:   }
10:
```

```
11:   int extractMin() {
12:     int min = Integer.MAX_VALUE;
13:     int minIndex = 0;
14:     for (int i=1; i <=size; i++) {
15:       if (elements[i] < min) {
16:         min = elements[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
21:     elements[minIndex] = elements[size];
22:     return min;
23:   }
24: }
```

# ESC/Java

- If you run ESC/Java on Bag.java

```
% escjava Bag.java
```

- You get the following warning messages

```
Bag.java:6: Warning: Possible null dereference (Null)
    size = input.length;
                 ^
Bag.java:15: Warning: Possible null dereference (Null)
      if (elements[i] < min) {
                  ^
Bag.java:15: Warning: Array index possibly too large (..
      if (elements[i] < min) {
                  ^
Bag.java:21: Warning: Possible null dereference (Null)
    elements[minIndex] = elements[size];
                                  ^
Bag.java:21: Warning: Array index possibly too large (..
    elements[minIndex] = elements[size];
                                  ^
```

# Dealing with Warnings/Errors

- First warning comes from the fact that if the constructor is called with a null argument this will cause an error

- There are two possible solutions
  - Change the constructor so that if a null reference is passed it creates an empty Bag
  - Add the fact that the constructor accepts a non-null reference as an input to the contract of the constructor as a pre-condition

# ESC/Java Annotations

- Choosing the second option we add the precondition to the constructor by adding the following statement between lines 4 and 5:

```
4.1: //@ requires input != null
```

- The @ sign indicates that this is an annotation, not an ordinary comment

- The keyword requires indicates that this is a precondition to the Bag constructor

# Dealing with Warnings/Errors

- The second and fourth warnings extractMin may dereference null if called when the field elements is null
- Although this may seem like a spurious error based on the code in Bag.java, since elements is not a private field and client code and subclasses can modify it, causing an error
- Even if elements was declared private, ESC/Java will still generate the warning since ESC/Java checks each method in isolation
  - By checking each method in isolation ESC/Java implements ***modular verification***

# Annotations

- To specify the design decision that elements array is always nonnull we change the line 3 as follows:

```
3':   /*@non_null*/ int[] elements;
```

- When ESC/Java sees this annotation it will generate a warning whenever it sees that a null value can be assigned to elements, it will also check that constructors initialize elements to a non-null value

- In a way the fact that elements field is non-null becomes part of the class invariant

- Parameters of procedures can also be declared non-null

# Annotations

- Method parameters can also be restricted to be non-null
- For example we could have changed the constructor for Bag as:

```
5':   Bag(/*@non_null*/ int[] input) {
```

- This condition becomes part of the precondition for the constructor and the effect of the above annotation is equivalent to adding the line 4.1 we showed above

# Dealing with Warnings/Errors

- The remaining two warnings are about possible array bound errors
- We can remove these warnings by declaring a class invariant (called object invariant in the ESC/Java paper)
- We add the following line between lines 2 and 3

```
2.1: //@ invariant 0 <= size && size <= elements.length
```

- ESC/Java will statically check that the class invariant is established at the end of the constructor, and assuming that it holds at the entry to a method it will hold at the exit

# Dealing with Warnings/Errors

- When we run the ESC/Java on the modified program, it still generates the following warning messages

```
Bag.java:15: Warning: Array index possibly too large (..
      if (elements[i] < min) {
                  ^
Bag.java:21: Warning: Array index possibly too large (..
    elements[minIndex] = elements[size];
                                  ^
```

- Looking at the code, we see that actually there is an error in the loop, we change

to

```
14:    for (int i=1; i <=size; i++) {

14':    for (int i=0; i <size; i++) {
```

# Dealing with Warnings/Errors

- However, there is another error, in line 21 elements array can be accessed with a negative index
  - What happens if the size is 0 when extractMin is called
  - This will cause size to become –1

- This can be fixed by changing the lines between 20 and 22 as:

```
20.1:   if (size >= 0) {
21:         elements[minIndex] = elements[size];
21.1:   }
```

# Dealing with Warnings/Errors

- Running ESC/Java on the modified code generates the following warnings

```
Bag.java:26: Warning: Possible violation of object invariant
  }
  ^
Associated declaration is "bag.java", line 3, col 6:
  //@ invariant 0 <= size && size <= elements.length
      ^
Possible relevant items from the counterexample context:
  brokenObj == this
(brokenObj* refers to the object for which the invariant is
broken.)
```

- This warning has three parts
  1. Invariant may be broken at the end of extractMin
  2. Second part shows which invariant is involved
  3. Third part says that the object for which the invariant is broken is this rather than some other Bag instance

# Dealing with Warnings/Errors

- Actually, our previous fix was not very good, we should have used the guard to also protect the decrement of size. So the new code is

```
19.1:   if (size > 0) {
20:         size--;
21:         elements[minIndex] = elements[size];
21.1:   }
```

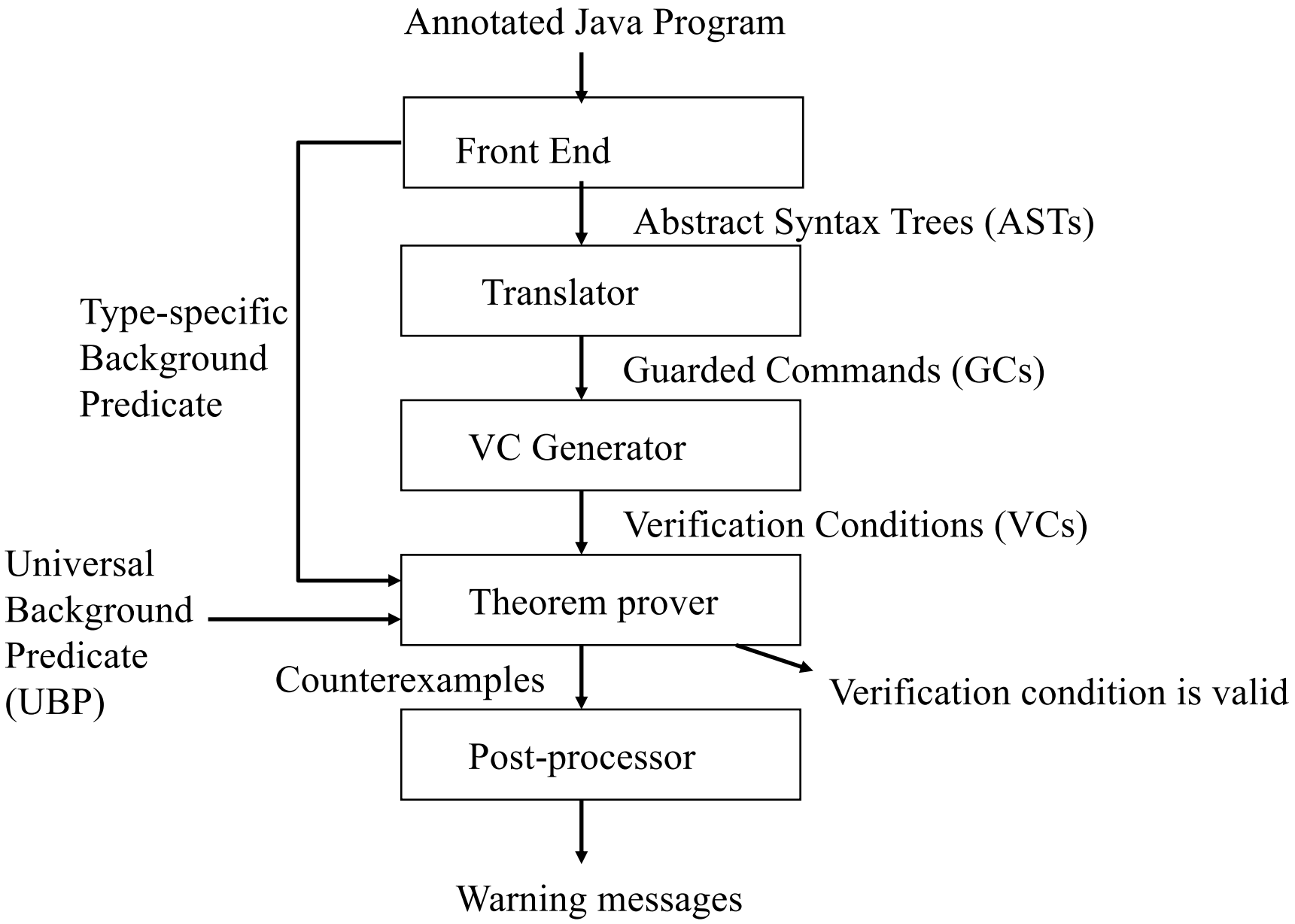
- Running the checker again does not produce any warning messages
  - which means that ESC/Java cannot find any more errors
    - this does not guarantee that the program is free of errors

# An Example: Bag.java

```
1: class Bag {
2:   int size;
2.1: //@ invariant 0 <= size
                 && size <= elements.length
3':   /*@non_null*/ int[] elements;
4:
4.1: //@ requires input != null
5':   Bag(/*@non_null*/ int[] input) {
6:     size = input.length;
7:      elements = new int[size];
8:      System.arraycopy(input,0,elements,0,size);
9:   }
10:
```

```
11:   int extractMin() {
12:      int min = Integer.MAX_VALUE;
13:      int minIndex = 0;
14':     for (int i=0; i <size; i++) {
15:        if (elements[i] < min) {
16:          min = elements[i];
17:           minIndex = i;
18:        }
19:      }
19.1:   if (size > 0) {
20:        size--;
21:        elements[minIndex] = elements[size];
21.1:   }
22:      return min;
23:   }
24: }
```

# How does ESC/Java Work?



# Front End

- Front end for ESC/Java is just like a Java compiler but it also parses and type checks annotations as well as Java code

- The front end produces
  - Abstract Syntax Trees (ASTs) and
  - a type-specific background predicate for each class whose methods are to be checked

# Front End

- The type-specific background predicate is a formula in first-order logic encoding information about the types and fields that methods in that class use

- For example, if T is a final class (i.e., T cannot have any subclasses) then the type-specific background predicate for class T or any client of class T will include the conjunct

  $(\forall S :: S <: T \Rightarrow S = T)$

- The relation <: is the subtype relation, i.e., S <:T is true if S is subtype of T
- The above predicate says that the only subtype of T is itself, i.e., T is a final class

# Translator

- The next stage translates each method body to be checked into a simple language based on Dijkstra's guarded commands (GCs)
- A simple guarded command syntax is

  *guard* $\rightarrow$ *guarded-list*

which means that: execute the guarded-list if the guard evaluates to true.

For example:

$x \geq y \rightarrow max := x$

- Guarded commands allow non-determinism where typically "[]" is used as the non-deterministic choice operator

if $x \geq y \rightarrow max := x$

[] $y \geq x \rightarrow max := y$

fi

# Translator

- ESC/Java's command language includes commands such as

  assert E

where E is a boolean expression

- Execution of a guarded command is said to *go wrong* if control reaches a subcommand of the form assert E when E is false

# Translator

- Ideally the translator should translate the body of a method M into a guarded command G such that
  - G has at least one potential execution that starts in a state satisfying the background predicate of M's class and goes wrong

  if and only if
  - there is at least one way that M can be invoked from a state satisfying its preconditions and then behave erroneously by, for example, dereferencing null or terminating in a state that violates its specified post-conditions
- Unfortunately for ESC/Java neither "if" nor the "only if" part holds
  - The fact that "if" part does not hold means that the tools is unsound
  - The fact that the "only if" part does not hold means that the tool is incomplete
  - So, ESC Java can report both false negatives and false positives

# Modular Verification

- ESC/Java uses a modular verification strategy
- ESC/Java uses Design-by-contract style specifications to achieve this
- When ESC/Java produces the guarded command for a method M it translates each method call in M according to the specification (of its contract) rather than the implementation of the called method
- Hence, the resulting non-deterministic guarded command G may be able to go wrong in ways involving behaviors of called routines that are permitted by their specification but can never occur with the actual implementations
- Note that this is a sign of incompleteness, i.e., ESC/Java can generate false positives

# Modular Verification

- Modular verification modularizes and hopefully simplifies the verification task
  - specifications are likely to be simpler than the implementations

- Another nice side effect of the modular verification is that the methods are verified against the specifications of the methods that they are using
  - In the future if the implementation of a method that is called by method M changes but if its specification remains the same we do not have to verify M again since the verification is based on the specification not the implementation of the called methods

# Overflows

- ESC/Java does not model arithmetic overflows
  - Allowing addition of two integers generate a negative value (i.e. considering the possibility of overflow) generates a lot of spurious warnings, hence they do not consider the possibility of an overflow

- Note that this is a sign of unsoundness
  - There are programs which may have overflow and cause null dereference because of that for which ESC/Java will not report any error

# Loops

- A precise semantics of loops can be defined as the least fixpoints of weakest-preconditions (predicate-transformers)

- Unfortunately least fixpoints characterizing the semantics of loops are
  - uncomputable
  - and they are hard to compute even for restricted cases

- Therefore ESC/Java approximates the semantics of loops by unrolling them a fixed number of times and replacing the remaining iterations by code that terminates without ever producing an error
  - This misses errors (causing unsoundness) that occur only in or after later iterations of a loop

# Loops

- The user can control the amount of loop unrolling
- Or the user can substitute a sound alternative translation for loops that relies on the explicit loop invariants provided by the user
- The default in ESC/Java is to unroll the loop body once (which evaluates the loop guard twice)
  - In a case study unrolling loops twice increased the execution time 20% but generated only one new interesting warning
  - In the same case study five unrollings doubled the time but produced no new warnings
- In their experience ESC/Java developers said that even experts have difficulty in providing correct and sufficiently strong loop invariants

# Verification Condition Generator

- Verification Condition (VC) Generator generates verification conditions from the guarded commands

- The output of the VC generator for a guarded command G is a predicate in first-order logic that holds for precisely those programs states from which no execution of the command G can go wrong

- VC generator in ESC/Java is an efficient ***weakest-precondition generator***

# VC Generator

- A big part of VC Generator is dealing with Java semantics
- Consider the following program fragment

```
class T extends S { ... }

t = (T) s;
```

where t is a variable of type T and s is a variable of type S.

- The Java type-cast expression (T) s returns the value of s after checking that this value is assignable to type T

# VC Generator

- The background predicate includes
  - a relation $<:_1$ which models the direct subtype relation
    - for example T $<:_1$ S

- The background predicate defines the subtype relation <: which is the reflexive transitive closure of $<:_1$

- The background predicate also includes a predicate called “is”, where is(o,U) means that the value o is assignable to type U, defined as:
  - $(\forall$ o, U :: is(o, U) $\equiv$ o = null $\vee$ typeof(o) <: U)

  where typeof maps non-null objects to their types

# VC Generator

- Then given the code

```
class T extends S { ... }

t = (T) s;
```

the generated guarded command will be:

assert is(s, T); t =s

hence, the command explicitly checks if the value of s is assignable to type T

- When the weakest condition is generated it looks like

$T <:_1 S \wedge (\forall o, U :: is(o, U) \equiv o = null \vee typeof(o) <: U) ... \Rightarrow ...is(s,T) \wedge ...$

# Theorem Prover

- After generating the verification condition ESC/Java uses the automated theorem prover Simplify to check the validity of the following formula:

$$UBP \wedge BP_T \Rightarrow VC_M$$

where

UBP is the universal background predicate

$BP_T$ is the type-specific background predicate for the class T in which method M is defined

$VC_M$ is the generated verification condition for M

- The above formula is valid if and only if the method M has no errors
  - As we said this is not exactly true due to the limitations in the generation of the verification condition

# Post Processor

- The final stage postprocesses the theorem prover's output, producing warnings when the prover is unable to prove the verification conditions

- Simplify theorem prover has some features which help the postprocessor to provide some warnings rather then just printing a message that the method is not correct

- When Simplify cannot prove the validity of a formula, it finds and reports one or more counter-example contexts

# Post Processor

- A counter example context is a conjunction of conditions
  - that collectively imply the negation of the verification condition
  - and have not been shown by the prover to be mutually inconsistent
- These conditions are mapped to the program source code and translated to warnings by the postprocessor
- There maybe more than one reason for the verification of a method to fail
- ESC/Java reports multiple (different) warning messages based on the counterexample contexts
- There is a limit (10) on the number of warnings it produces
- Simplify can produce spurious counterexamples which result in spurious warnings

# Annotation Language

- Annotations are written as Java comments that begin with a @ sign

  /*@ ... * /

  //@ ...

- Expressions contained in annotations are side-effect free Java expressions

- The annotation language used by ESC/Java is a subset (with minor differences) of JML (Java Modeling Language)

- In ESC/Java manual they call annotations pragmas

# Method Specifications

- Method specification use contain the following parts:

**requires E;**

E denotes a boolean expression that is a precondition of the method; ESC/Java will assume that E holds initially when checking the implementation of the routine, and will issue a warning if it cannot establish that E holds at a call site.

**assignable S;**

S denotes a nonempty comma-separated list of lvalues; ESC/Java will assume that calls to the method modify only the lvalues listed in S.

ESC/Java does not check and hence, does not warn about implementations that modify more targets than S allows. However in the verification condition generator ESC/Java uses the information modifies list.

# Method Specifications

**ensures E;**

E denotes a boolean expression that is a normal (i.e. non-exceptional) postcondition of the method; ESC/Java will assume that E holds just after each call site the method, and will issue a warning if it cannot prove from the method implementation that E holds whenever the method terminates normally.

In expression E keyword \result refers to the value returned, if any, and \old(P) refers to the value of the expression P at the method entry.

# Method Specifications

**signals (T t) E;**

t is an exception, and E denotes a boolean expression that is an exceptional postcondition of the method; ESC/Java will assume that E holds whenever the a call to the routine completes abruptly by throwing an exception t, and will issue a warning if it cannot prove from the method implementation that E holds whenever the method terminates abruptly by throwing an exception t whose type is a subtype of T.

# Other Specifications

- In addition to pre and post conditions you can declare assertions at any program point:

**assert E;**

E denotes a boolean expression; ESC/Java will issue a warning if it cannot establish that E is true whenever control reaches this assertion

# Other Specifications

**loop_invariant E;**

This annotation may appear only just before a Java for, while, or do statement. ESC/Java will check that E holds at the start of each iteration of the loop.

- When a loop invariant is declared ESC/Java will check that it holds initially and after one execution. There is a –loop option which can be used to change the default option and force ESC/Java to check for more than one iteration

# Object Invariants

**invariant E;**

E denotes a boolean expression that is an object invariant of the class within whose declaration the annotation occurs. If E does not mention “this”, the invariant is called a *static invariant*, and is assumed on entry to implementations, checked at call sites, assumed upon call returns, and checked on exit from implementations. If E mentions this, the invariant is called an *instance invariant*. An instance invariant is assumed to hold for all objects of the class on entry to an implementation and is checked to hold for all objects of the class on exit from an implementation. At a call site, an instance invariant is checked only for those objects passed in the parameters of the call and in static fields. A call is assumed not to falsify the instance invariant for any object.

# Object Invariant

- Object invariant may be broken during the execution of a method of the class that the object belongs
- What happens if the method being checked calls another method?
- ESC/Java enforces that the object invariant should be established before calling another method
- This can be overwritten by declaring a class helper
  - Helper methods are assumed to be part of the original method and object invariant is not checked before or after calling a helper

# Ghost fields

- Ghost fields are provided in Java so that the programmer can define an abstract state of an object. These are auxiliary variables used for specification, they do not effect the program execution.

**ghost M S v;**

S is a specification type, v is an identifier, and M is a sequence of modifiers including public; this annotation is like an ordinary Java variable declaration M S v; except that it makes the declaration visible only to ESC/Java, and not to the compiler; such variables are called *ghost variables*.

**set D = E;**

D refers to a ghost field of some object or class and E is a specification expression containing no quantifiers or labels; this annotation has the analogous meaning to the Java assignment statement D = E;

# Escape Hatches

- Since ESC/Java can generate spurious warning messages, sometimes one may want to turn off warning some messages

**nowarn L;**

L denotes a possibly empty comma-separated list of warning types; ESC/Java will suppress any warning messages of the types in L (or of all types, if L is empty) at the line where the annotation appears

**assume E;** or **axiom E;**

E denotes a boolean expression; ESC/Java will assume that E is true whenever control reaches the annotation and ignores the remainder of all execution paths in which E is false.