Extended Static Checking

Software Fiável

Mestrado em Engenharia Informática
Mestrado em Informática
Faculdade de Ciências da Universidade de Lisboa

2015/2016

Vasco T. Vasconcelos, Antónia Lopes
A static analysis technique that relies on automated theorem proving

**Key idea**: A useful extension of compiler type checking

- **Goals:**
  - prevent common errors in programs
  - make it easier to write reliable programs
  - increase programmer’s productivity

- **Considers several classes of errors, namely**
  - the violation of programmer-supplied assertions (preconditions, postconditions, invariants)
  - null dereference, type cast errors, division by 0, and many more
Extended Static Checking

An approach for finding errors in a program, not for proving statements about programs

- The program is analyzed by translating the program into a **verification condition**, i.e., a logical formula that is valid if and only if the program is free from the classes of errors under consideration.
- The verification condition is passed to an automatic theorem prover that mechanically searches for counterexamples.
- Any counterexample reported by the theorem prover is translated into a warning message that the programmer can understand.
The idea of encoding program and property into (first-order) formulas that if true implies that the program satisfies the property has been around since early 1970s

- ESC/Modula-3 was developed at Compaq\(^1\) (1990–1996)
- Extended Static Checker for Java (ESC/Java) was developed at Compaq (1997–2000)
- ESC/Java2 was developed at Univ. Nijmegen (2003–), and by the EU Mobius project (2005–2010)
- Spec# static program verifier Boogie for C# at Microsoft (Redmond) (2004)
- Dafny programming language, designed to support the static verification of programs at Microsoft (Redmond) (2009)

\(^1\)Acquired by Hewlett-Packard in 2002
Extended Static Checker: Pros & Cons

- Theorem prover is a blunt tool
  - It may need help from the user (interaction, annotations)
  - It may diverge (and hence a time limit is usually considered)
- Bug-finder, not full verifier
- Covers much more cases than can be achieved using debugging or testing (and, hence, in practice detects many programming errors)
- Trade-offs to make the checker more usable typically force it to complain about correct programs
- Forces programmers to write method contracts and representation invariants (leads to better documented and easier to maintain code)
- General purpose
- Conceptually elegant
Extended Static Checking vs. Runtime Verification of Contracts

- **compile-time** vs. **run-time** checking

- In runtime verification the correctness of contracts is checked during a specific execution. Hence, results of runtime checking are only as good as the test suite used.

- In extended static checking the correctness of contracts is checked for any possible input and program execution. Hence, it provides higher degree of confidence.
Extended Static Checker for Java

- Uses JML annotations
- Powered by weakest precondition semantics and automatic theorem proving
- Performs modular checking, i.e., to check class $C$, needs only the specifications in the classes that $C$ builds on
- Extended Static Checker for Java version 2 (ESC/Java2):
  
  "The Extended Static Checker for Java version 2 is a programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations."

  http://kind.ucd.ie/products/opensource/ESCJava2/
Available both as a command-line tool and a plugin for the Eclipse development environment (Java 1.5)

Checks for common runtime errors, synchronization errors and violations of programmer-supplied specifications (preconditions, posconditions, invariants)

The assertion language is a variant of JML including new primitives such as `assume φ`

Designed to be useful. It is not sound (it may miss some errors in the program) and is not complete (it may complain about a correct program)
ESC/Java2 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
1. Runtime exceptions: Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore
2. Non null violations
3. Loop and flow specifications
4. Method specification: precondition, postcondition, frame conditions (assignable)
5. Class specification: invariant, initially
6. Exception declaration problems
7. Multithreading: races and deadlocks


ESC/Java2 is not sound

Soundness: If I can prove $\varphi$, then $\varphi$ holds

- ESC/Java2 may approve an incorrect program
- Sources of unsoundness:
  - Pragma `assume` adds unverified knowledge
    
    ```java
    y = x * x + 2 * x + 1;
    //@ assume y >= 0;
    z = new int [y];
    ```
  - Loops are handled by unrolling them, typically once. This avoids the need for programmers to supply invariants.
  - Object invariants are not verified on all existing objects as this would be too expensive.
  - ...

Software Fiável

Extended Static Checking
Completeness: If $\varphi$ holds, then I can prove it

- ESC/Java2 is not complete: it may complain about a correct program
- Sources of incompleteness:
  - Simplify: limited reasoning capabilities (on arithmetic, quantifiers, etc).
  - Java semantics not fully modeled
  - ...
- The pragma `nowarn` can be used to turn off warnings (potentially unsound)
ESC/Java architecture

Annotated Java program → Translator → Verification condition → Automatic theorem prover

Counterexample context → Resource exhausted

Valid

Post processor → Warning messages

[Picture by C. Flanagan]
The Java program is first translated into a much simpler language, a variant of Dijkstra’s guarded command language.

Dijkstra’s guarded command language:

- For describing abstract nondeterministic programs
- The building blocks are guarded commands
- A guarded command is a statement list prefixed by a Boolean expression. Only when this Boolean expression is initially true, is the statement eligible for execution. This Boolean expression is named **guard**.
Guarded commands: Conditional

\begin{verbatim}
if
    B1 → S1
[] B2 → S2
...
[] Bn → Sn
fi
\end{verbatim}

- B1, ..., Bn are boolean expressions (not necessarily disjoint)
- S1, ..., Sn are commands
- Semantics: Pick a guard that is true and execute its command
Guarded commands: Loops

\[
\begin{align*}
\text{do} & \\
& \begin{array}{l}
B_1 \rightarrow S_1 \\
\[ \] B_2 \rightarrow S_2 \\
\ldots \\
\[ \] B_n \rightarrow S_n \\
\text{od}
\end{array}
\end{align*}
\]

- \( B_1, \ldots, B_n \) are boolean expressions (not necessarily disjoint)
- \( S_1, \ldots, S_n \) are commands
- Semantics: Pick a guard that is true and execute its command; repeat until all guards are false
Guarded Commands: Examples

```
    do
      x>y  ->  x:=x−y
      [ ] y>x  ->  y:=y−x
    od

    do
      q1 > q2  ->  m:= q1;  q1:= q2;  q2:= m
      [ ] q2 > q3  ->  m:= q2;  q2:= q3;  q3:= m
      [ ] q3 > q4  ->  m:= q3;  q3:= q4;  q4:= m
    od
```
Weakest preconditions

- WP(S, ψ) — the **weakest precondition** of statement S with respect to the postcondition ψ — is the weakest ϕ such that
  \[\{ϕ\} S \{ψ\}\], i.e.,
  
  for all \(ϕ'\) s.t. \(\{ϕ'\} S \{ψ\}\) we have \(ϕ \Rightarrow ϕ'\)

- In this way
  \[ϕ \Rightarrow WP(S, ψ)\] is equivalent to \(\{ϕ\} S \{ψ\}\)
The internals of ESC/Java: Contracts

//@ requires ϕ;
//@ ensures ψ;
void p() {S};

1. Calculate WP(S, ψ), the weakest precondition of S with respect to the postcondition ψ.
2. Generate VC, the verification condition, $\varphi \Rightarrow WP(S, \psi)$
3. Use automatic theorem proving to check this condition
Example: Weakest precondition for assignment

\{ ?? \} x = 5 \{ y > x \}

- If we want \( y > x \) to be true after, what must be true before?
- Answer: \( WP(x = 5, y > x) \).
- The weakest precondition is obtained by propagating the assignment \( x = 5 \) past the postcondition \( y > x \) (cf. lecture on Hoare logic)

\[
WP(x = 5, y > x) = (y > x)[5/x] = y > 5
\]
Example: Verification condition for assignment

\[
\{y > 0\} \times = 5 \{y > x\}
\]

▶ We know that the weakest precondition \(WP(x = 5, y > x)\) is \(y > 5\), but our precondition is \(y > 0\)

▶ We prove the implication. In Hoare logic, using *left strengthening*:

\[
y > 0 \rightarrow y > 5 \quad \{y > 5\} \times = 5 \{y > x\} \\
\{y > 0\} \times = 5 \{y > x\}
\]

▶ The generated VC is \(y > 0 \rightarrow y > 5\), which ESC/Java passes to a theorem prover
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.

The user can control the amount of loop unrolling ( `-Loop` option).

The user can also supply explicit loop invariants, thus providing for a sound alternative (`loop_invariant` assertion).

The default in ESC/Java is to unroll the loop body once (which evaluates the loop guard twice).
Rules for obtaining weakest preconditions:

- $\text{WP}(\text{skip}, \psi) = \psi$
- $\text{WP}(x := e, \psi) = \psi[x/e]$
- $\text{WP}(S_1; S_2, \psi) = \text{WP}(S_1, \text{WP}(S_2, \psi))$
- $\text{WP}(\text{if } p \text{ then } S_1 \text{ else } S_2, \psi) = p \rightarrow \text{WP}(S_1, \phi) \land \neg p \rightarrow \text{WP}(S_2, \phi)$
- $\text{WP}(\text{while } p \text{ do } S, \psi)$ cannot be expressed as a syntactic construction in terms of weakest preconditions (least fixpoints need to be used and are not computable)
Theorem prover

- Prover for first-order logic with equality and integer arithmetic
  - For proving $F$, the satisfiability of $\neg F$ is checked
- Core: propositional satisfiability check
  - Backtracking to construct satisfying propositional assignment
- Module E-graph for handling equalities
- Module Simplex for handling linear integer arithmetic
- Heuristical instantiation of universally quantified formulas
- Sound, not complete, highly optimized

[Slide by W. Schreiner]
Tutorial: Bag

- We go through the Bag.java example from the manual
- Illustrates:
  - Basic checks performed on unannotated code
  - Addition of preconditions
  - Addition of invariants
  - Addition of postconditions
class Bag {
    private int [] a;
    private int n;

    Bag(int [] input) {
        n = input.length;
        a = new int [n];
        System.arraycopy(input, 0, a, 0, n);
    }

    ...
}
```java
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;
}
```
prompt> $ESCTOOLS_RELEASE/escj escjava/Bag.java
Caution: ...
escjava/Bag.java:6: Warning: Possible null dereference (Null)
    n = input.length;
   ^
escjava/Bag.java:15: Warning: Possible null dereference (Null)
    if (a[i] < m) {
       ^
...
1 caution
6 warnings
Fixing "Possible null dereference"

- Bags cannot be created from null inputs
- For the constructor parameter we use a non_null modifier:
  \[\text{Bag}(\text{int}[\text{input}])\]

  (Notice that, unlike JML, ESC/Java does not assume "non-null by default").

- A consequence of modular checking is that the analysis of the extractMin() method does not know that the constructor leaves the object in a state where \text{a} is not null (ESC/Java does not analyze the body of the constructor).

- Hence, for field \text{a} we use the same modifier:
  \[\text{private int[\text{a}]}\]
Running ESC/Java2 again

prompt> $ESCTOOLS_RELEASE/escj escjava/Bag.java
Caution: ...

escjava/Bag.java:20: Warning: Possible negative array index
a[mindex] = a[n];
   ^

...  
1 caution
1 warning

Two possible solutions: in method extractMin,
  ▶ require n > 0, or
  ▶ return Integer.MAX_VALUE if extractMin is called with n == 0
Solution A: Require a non-empty array

```java
...
//@ 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;
}
}
Solution B: Allow an empty array

```java
int extractMinEvenWhenEmpty() {
    int m = Integer.MAX_VALUE;
    int minIndex = 0;
    for (int i = 0; i < n; i++)
        if (a[i] < m) {
            minIndex = i;
            m = a[i];
        }
    if (n > 0) { // min is in the array
        n--;
        a[minIndex] = a[n];
    }
    return m;
}
```
Even though ESC/Java does not require it, we declare field n as `spec_public`, given that we are using a private field in a public precondition:

```java
private /*@ spec_public @*/ int n;
```

We shall do the same to field a should it become necessary.
prompt> $ESCTOOLS_RELEASE/escj escjava/Bag.java
Caution: ...
escjava/Bag.java:16: Warning: Array index possibly too large
   if (a[i] < m) {
      ^

... 1 caution
2 warnings
The problem is modularity

- The analysis of the `extractMin()` method does not see that `n` is always smaller than the length of `a`.
- Several options:
  - Local assumption: Precondition in every relevant method
  - Global property of the field: Invariant (better)

```plaintext
invariant 0 <= n && n <= a.length;
```
Invariants

- All object invariants are supposed to hold at all routine call boundaries.
- ESC/Java does not fully enforce the discipline, partly because it would be too strict for many programs (classes may need to temporarily breaking an invariant) and partly because such checking would be very expensive.
- Invariants are checked for parameters at call sites and heuristics are used to reduce the set of invariants (see the fine details in the manual).
Temporarily breaking an invariant

- Consider the below variation of the ArrayStack class

```java
//@ invariant elems != null;
public ArrayStack() { clear(); }
private void clear() {
    elems = new Object[DEFAULT_CAPACITY]; size = 0;
}
```

escjava/StackSpecPublic.java:24: Warning: Possible violation of object invariant (Invariant) clear();

- In order to tolerate the temporary violation of the invariant, we classify method `clear` as `helper`:

```java
/*@ helper @*/
private void clear()
```

- Note the pragma can only be applied to methods that cannot be overridden
What about guarantees?

- We have seen that the code raises no exception, but does it behave as expected?
- What do you mean “expected”?
- ExtractMin is supposed to (among other things):
  - Remove one element from the array
  - Return an element less-or-equal than any element in the array

```c
//@ requires n > 0;
//@ ensures n == old(n) - 1;
//@ ensures (forall int i;
//@  0 <= i && i < n; \result <= a[i]);
int extractMin() { ... }
```
Running ESC/Java2 again

prompt> $ESCTOOLS_RELEASE/escj escjava/Bag.java
Caution: ...
1 caution

- ESC/Java2 does not complain about the post condition
- But this does not mean that ESC/Java2 did prove it
- ESC/Java2 is not sound (Not every error is actually detected!)
More guarantees

- ExtractMin is also supposed to return an element *in* the array (notice the use of $\text{old}$)

```java
//@ requires n > 0;
//@ ensures ...;
//@ ensures (\exists int i;
//@ 0 <= i && i <= \old(n);
//@ \result == \old(a[i]));
int extractMin() { ... }
```

- But this is too much for ESC/Java (see next slide)
ESC/Java is not complete

prompt> $ESCTOOLS_RELEASE/escj escjava/Bag.java
...
escjava/Bag.java:32: Warning: Postcondition possibly not established
    }
  ~
Associated declaration is "escjava/Bag.java", line 48, col 5:
   //@ ensures (\exists int i;
   //@ 0 <= i && i <= n; \result == a[i]) ...
...
1 warning

• ESC/Java complains; good!
A do-while loop in disguise

- Recall:

```java
//@ requires n > 0;
int extractMin() {
    int m = Integer.MAX_VALUE;
    int minIndex = 0;
    for (int i = 0; i < n; i++)
        ...
}
```

- How many times will the loop execute?
- **At least** once!
- Let us then write the code properly, so that we ESC/Java can help us
Rewriting our program so that ESC/Java2 may help us

Now that we know that $n > 0$ we do not need to initialize $m$ with `Integer.MAX_VALUE`

We write:

```java
int extractMin() {
    int m = a[0]; 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;
}
```

and ESC/Java2 does not complain about the last post-condition:

$$(\exists i : 0 <= i && i <= old(n); \ result == old(a[i])$$
ESC/Java masters conditionals

```java
/*@ 
@  ensures \result == i || \result == j || 
@  \result == k;
@  ensures \result <= i && \result <= j &&
@  \result <= k;
@*/
static int min (int i, int j, int k) {
    int m = i;
    if (j < m) m = j;
    if (k < m) m = k;
    return m;
}
```

[Example by W. Schreiner]
ESC/Java masters linear arithmetic with inequalities

```java
//@ ensures result == i;
static int f2(int i) {
    int j = i + 1;
    int k = 3 * j;
    return k - 2 * i - 3;
}

//@ requires i < j;
//@ ensures result >= 1;
static int f4(int i, int j) {
    return 2 * j - 2 * i - 1;
}

[Examples by W. Schreiner]
```
ESC/Java cannot handle non-linear arithmetic

```java
//@ ensures \result == i * i;
static int f1(int i) {
    return i * (i + 1) - i;
}

//@ ensures \result >= i;
static int f2(int i) {
    return i * i;
}

Warning: Postcondition possibly not established (Post)
```

Associated declaration is "Arithmetic2.java", line 11, col 5:
```
//@ ensures \result == i * i;
```
Recall the BankAccount class:

```java
//@ requires amount >= 0;
//@ ensures balance() == old(balance()) + amount;
public void deposit (int amount) { balance += amount; }
```

In this case we may use the field rather, than the method, in the `old` expression to content ESC/Java2:

```java
ensures balance() == old(balance) + amount;
```
For complicated bodies, the warning messages give some information about which if-then-else branches caused the warning:

```plaintext
//@ ensures \result > 0;
int m(int i) {
if (i == 0) return 1;
if (i == 2) return 0;
return 4;
}
```

Where is the error?
Trace.java:10: Warning: Postcondition possibly not established (Post)
  }
~
Associated declaration is "escjava/Trace.java", line 1, col 5:
  //@ ensures \result > 0;
~
Execution trace information:
  Executed else branch in "escjava/Trace.java", line 3, col 2.
  Executed then branch in "escjava/Trace.java", line 4, col 14.
  Executed return in "escjava/Trace.java", line 4, col 14.

> Follow the error: else in line 3 → then in line 4 → return in line 4:

> The method reads ensures \result > 0 but the code says return 0
Sometimes when a specification is found to be invalid, ESC/Java2 will produce a counterexample.

A full context will be produced with the -CounterExample option.

These are difficult to read, but can give information about the reason for failure.

They state the formulae that the prover believes to be true; if there is something you think should not be true, that is a hint about the problem.
The Cast warning occurs when ESC/Java2 cannot verify that a ClassCastException will not be thrown.

```java
void m(Object o) {
    String s = (String) o;
}
```

`CastWarning.java:5: Warning: Possible type cast error (Cast)
String s = (String) o;`
non_null pragma versus !null invariant

- For the Bag example, we have set
  ```c
  //@ non_null @*/ int [] a;
  ```

- An alternative is
  ```c
  //@ invariant a != null;
  ```

- What is the difference?
  - non_null pragmas are checked at every assignment to a
  - The invariant pragma is checked only at method calls (entry and exit)
  - Use non_null whenever possible
Possible solutions:

1. Protect the cast with a conditional

```java
void m(Object o) {
    if (o instanceof String)
        String s = (String) o;
}
```

2. Protect the method with a precondition:

```java
//@ requires o instanceof String;
public void m(Object o) {
    String s = (String) o;
}
```
Exceptions

- An Error can be thrown at any time
  - Java requires no declaration in the `throws` clause
  - No checking is performed by ESC/Java2
Checked exceptions

- Checked exceptions (e.g., IOException) are Exceptions that are not RuntimeExceptions
- Java requires the declaration (in the `throws` clauses) of exceptions mentioned in the body of the method
- ESC/Java2 checks during typechecking that throws declarations are correct (similarly to the Java compiler)
Unchecked exceptions

- Java unchecked exceptions (e.g., NullPointerException) are RuntimeExceptions
- Java does not require these to be declared in throws clauses
- ESC/Java2 is stricter than Java—it will issue an Exception warning if an unchecked exception might be explicitly thrown but is not declared in a throws declaration
- Caution: currently ESC/Java2 will assume that an undeclared unchecked exception will not be thrown, even if it is specified in a signals clause—Declare all unchecked exceptions that might be thrown! (e.g., especially when there is no implementation to check)

Slides by David Cok, Joe Kiniry, Erik Poll
Unchecked exceptions

Consider the method:

```java
void m(Object o) {
    if (!(o instanceof String))
        throw new ClassCastException();
}
```

UncheckedException.java:8: Warning: Possible unexpected exception (Exception)

```java

Solution: Declare the exception:

```java
void m(Object o) throws ClassCastException
```
Arrays and subtyping

```java
class Super {}
class Sub extends Super {
    void m() {}
    public static void main(String[] args) {
        Sub[] subArray = new Sub[3];
        Super[] superArray = subArray;
        superArray[0] = new Super();
        subArray[0].m();
    }
}
```

▶ Compiles?
▶ Runs?
The below code generates a ArrayStoreException in line 3 since SuperArray actually refers to a Sub[] object so it is not valid to store an Super reference in it\(^2\)

\begin{verbatim}
Sub[] subArray = new Sub[3];
Super[] superArray = subArray;
superArray[0] = new Super();
\end{verbatim}
Recall class ArrayStack from the lectures on JML

```java
private Object[] elems;
public void push(Object x) {
    if (size == elems.length) grow();
    elems[size] = x;
}
```

Warning: Type of right-hand side possibly not a subtype of array element type (ArrayStore)

```
    elems[size] = x;
   ^
```

The ArrayStore warning occurs when ESC/Java2 cannot verify that the assignment of an object to an array element will not result in an ArrayStoreException
Again, a problem of modularity: ESC/Java does not know the runtime type of array `elems`. Alternatives:

1. Add an invariant stating that the runtime type of array `elems` is always `Object[]`:
   
   ```
   //@ typeof(elems) == type(Object[]);
   ```

2. Add an invariant stating that the runtime type of the values in `elems` is always `Object`:

   ```
   //@ elemtype(typeof(elems)) == type(Object);
   ```
In some cases a precondition (to method `push`) may be what we need:

\[
\text{\texttt{typeof}}(x) \ <: \ \text{\texttt{elemtype}}(\text{\texttt{typeof}}(\texttt{elems}))
\]

- `\text{\texttt{typeof}}` gets the run-time type of an expression
- `\text{\texttt{elemtype}}` gets the base type from an array type
- `\text{\texttt{type}}` gets the type representing the given Java type
- `<:` means “is sub-type of”
- Note: the above solution (precondition to method `push`) is less satisfactory, for it requires clients to know (guess?) the runtime type of `elems`
A **loop_invariant** asserts a predicate that is true prior to each iteration and at the termination of the loop (or else a LoopInv warning is issued).

```java
int max(/*@ non_null @*/ int[] a) {
    int m = Integer.MAX_VALUE;
    //@ loop_invariant (forall int j;
    //@ 0 <= j && j < i; m >= a[j]);
    for (int i = 0; i < a.length; ++i)
        if (m < a[i]) m = a[i];
    return m;
}
```

**Caution:** Loops are checked 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.
Loop unrolling

- 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.

- The default is -Loop 1.5 (the .5 part evaluates loop guard one extra time). A loop \textbf{while (p) do} S; is unrolled to 
  \begin{verbatim}
  if (p) {S; if (p) {assume false ;}};
  \end{verbatim}

- The unrolled part is replaced by \textbf{assume false}; from false everything can be concluded.
Proving termination

- ESC/Java can be used to prove termination as well, but we must provide a variant expression for each loop.
- What is a variant expression for this loop?

```java
for (int i = 0; i < n; i++)
    if (a[i] < m) {
        minIndex = i;
        m = a[i];
    }
```

- We write it just before the loop:

```java
//@ decreases n - i;
for (int i = 0; i < n; i++)
```

- In this case ESC/Java2 is able to prove the assertion.
- If not, then it produces a Decreases warning:

  Warning: Loop variant function possible not decreased (Decreases)
Bibliography