Extended Static Checking for Java

Presented by:

Tomás Nascimento, Rómulo Kaidussis, Pedro Lobo

Outline

Introduction

Proposed Solution

Evaluation

Related Work

Conclusion

Questions

Introduction - What is ESC/Java?

Compile-time program checker

Finds common programming errors

Programmer writes annotations

Checker finds inconsistencies between annotations and code

Introduction - Why is ESC/Java Needed?

Software development and maintenance are costly

ESC/Java detects defects early in the development process

Helps reduce this cost

Introduction - What is Static Checking?

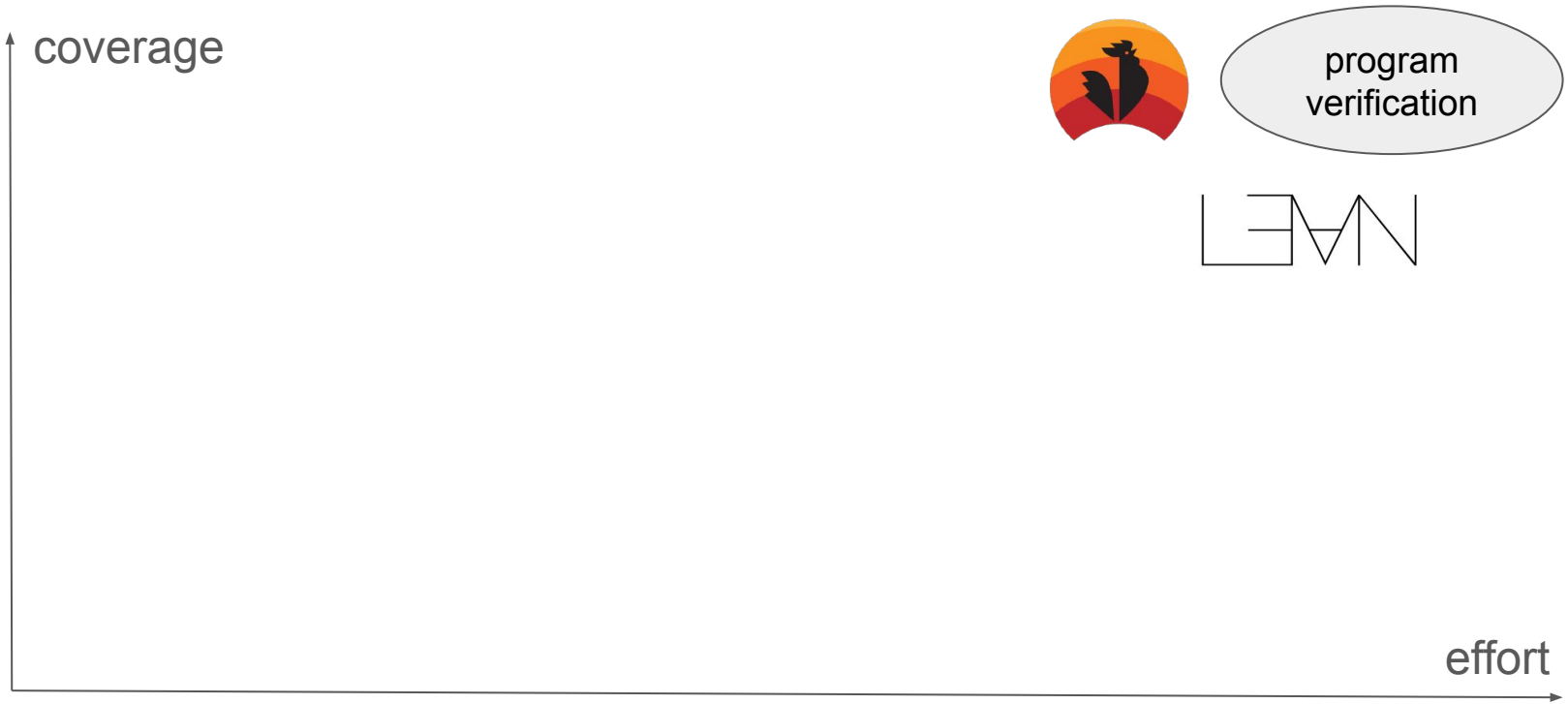
ESC/Java performs extended static checking

Static because the program doesn’t need to be run to perform the checking

Extended because it catches more errors than are caught by conventional static checkers

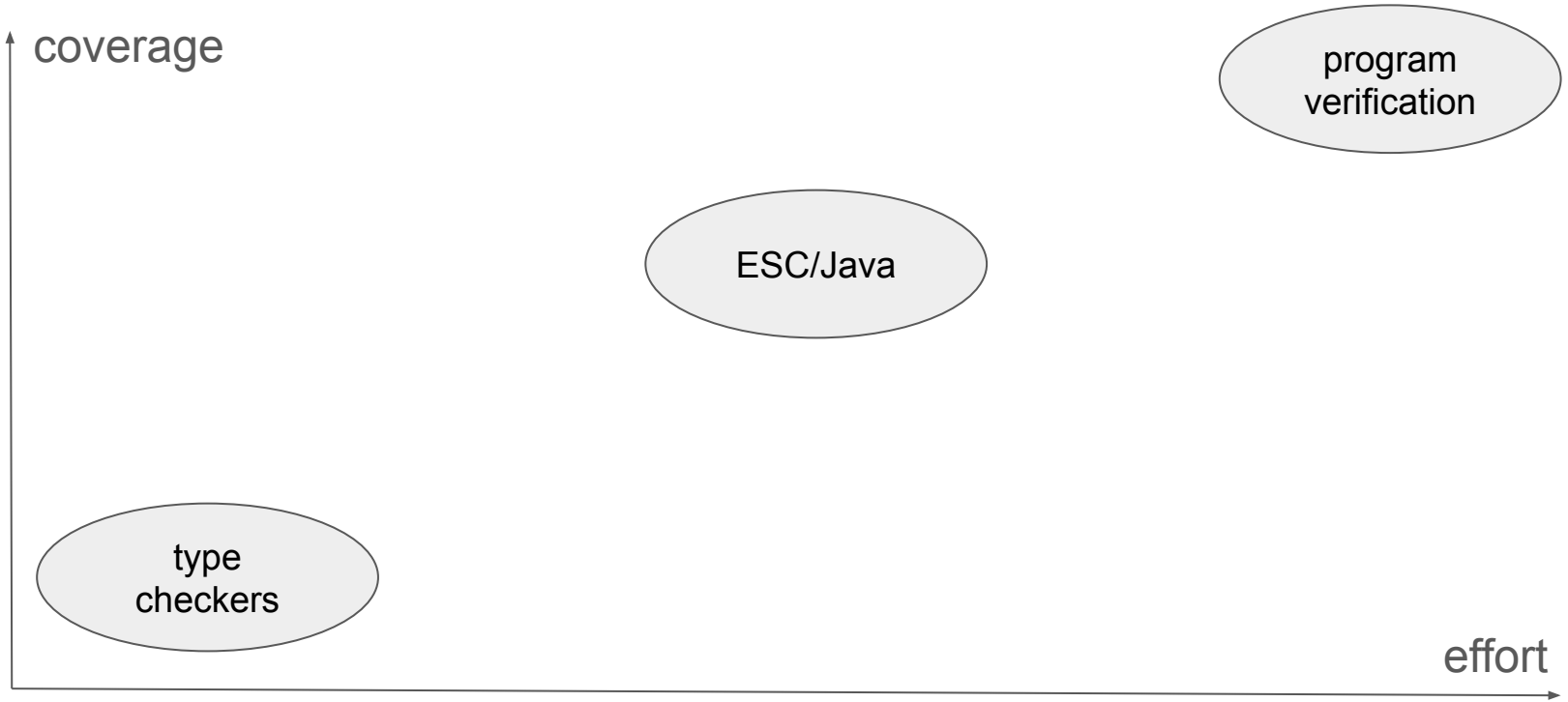
Introduction - The World of Static Checkers

Introduction - The World of Static Checkers



Introduction - The World of Static Checkers

Introduction - The World of Static Checkers



Introduction - The World of Static Checkers

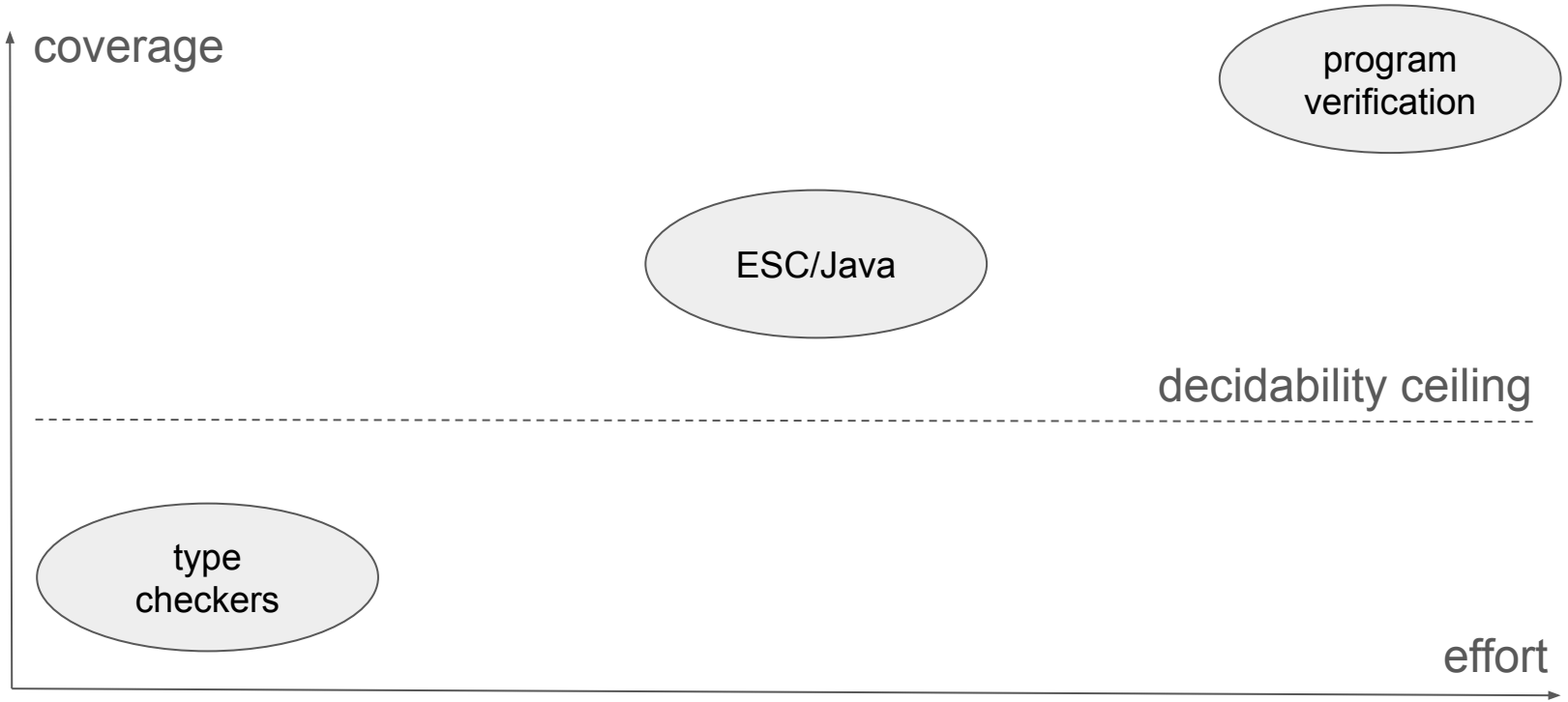
What is an ideal static checker?

Ideal static checker is both sound and complete.

Soundness: Catches all errors

Completeness: Only reports true errors, no false positives

Introduction - The World of Static Checkers



Introduction - Modular Checking

ESC/Java operates on a “piece” of the program at a time.

A piece is a single routine (a method or a constructor).

No need to have the whole source code to run the checker.

It scales better.

Proposed Solution - Practical Example

```
1:  class Bag {
2:      int size;
3:      int[] elements;    // valid:  elements[0..size-1]
4:      //@ requires input != null
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: }
```

```
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: Possible negative array index (...
    elements[minIndex] = elements[size];
                                  ^
```

Proposed Solution - Practical Example

The errors indicate that null may be dereferenced if `elements` is set to null.

In the constructor method, `elements` is explicitly set to a non-null value.

The access modifier of the `elements` field is package-private, allowing it to be set to null by other code in the same package.

Declaring `elements` as private would not eliminate the warnings.

```
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: Possible negative array index (...
    elements[minIndex] = elements[size];
                         ^
```

Proposed Solution - Practical Example

ESC/Java checks methods in isolation.

To conclude that the field could not be set to null, ESC/Java would need to examine all other class methods.

The warnings highlight missing useful documentation.

To specify that `elements` is never null, the user can annotate the declaration of `elements`.

ESC/Java will generate a warning if it suspects null is being assigned to a field annotated as non-null.

```
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: Possible negative array index (...
    elements[minIndex] = elements[size];
                         ^
```

Proposed Solution - Practical Example

The remaining errors suggest a possible out-of-bounds access.

An off-by-one error is detected in the loop, where `i <= size` is checked instead of `i < size`.

In the `extractMin` function, if `size` is 0, an attempt to index `elements` into position -1 occurs.

Specifying an object invariant for `size` helps clarify the programmer's intention for its maintenance at routine boundaries.

Rerunning the checker after fixing the bugs reveals errors indicating violation of object boundaries in indexing instructions.

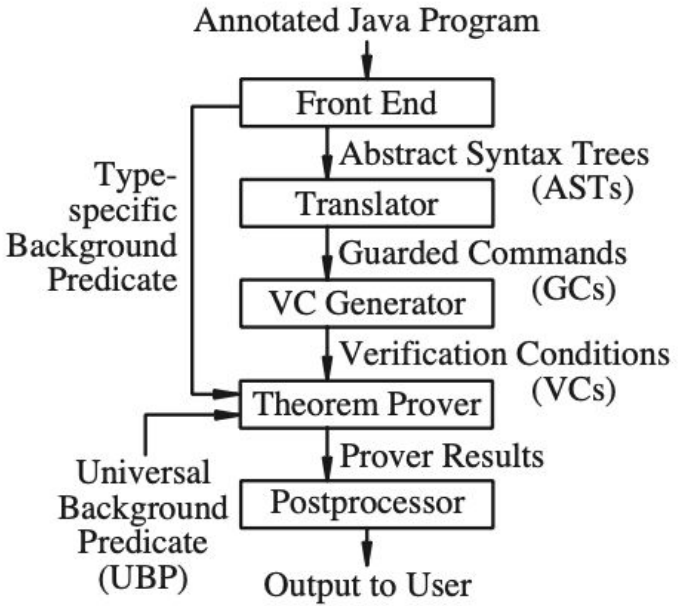
After resolving these issues, the tool reports no more errors, though this doesn't guarantee the absence of all bugs.

```
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: Possible negative array index (...
    elements[minIndex] = elements[size];
                         ^
```

Architecture

The frontend generates an abstract syntax tree and a type-specific background predicate in first-order logic for class methods.

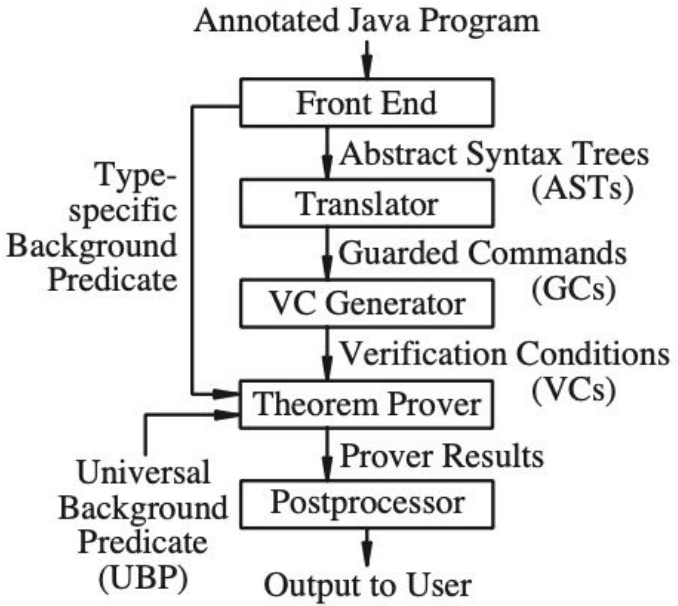
The translator converts method bodies into simple language with boolean assertions as guarded commands, which fail if their preconditions are false, allowing methods to be invoked correctly but potentially violate postconditions.



Architecture

A precise semantics for loops can be defined using weakest fixpoints of predicate transformers or loop invariants, which are conditions that verify at the start, after each iteration, and at the end of the loop.

However, since finding sufficiently expressive loop invariants is difficult, ESC/Java approximates loop semantics by unrolling them a fixed number of times.



Architecture - Postprocessing and Cautionary Measures

Verification conditions, first-order logic predicates, ensure no command execution errors.

Theorem prover attempts to verify conditions using Java semantics and specific class predicates.

Postprocessor warns user with counterexamples if conditions are not provable.

Time-limited postprocessor analysis may cause ESC/Java to caution incomplete checks.

Annotation Language

- Simple and close to Java
  - easy to learn
  - annotation are readable to new users
  - serves as documentation

- Annotations capture significant programmer decisions without needing to document properties that are tedious to specify.

Annotation Language

- Allows the user to specify routine specifications:
  - Precondition
  - Postcondition
  - Exceptional Postcondition – happens when an exception is thrown
  - Modifies List – indicates the variables that are modified

Annotation Language

- Limitation: Modifies List is impractical
  - ESC/Java doesn’t check that an implementation obeys its modifies list
  - it is impossible to write down all variables that a routine may change, since they may be out of scope or even reside in not yet written subclasses

- Overridden methods inherit specifications from the methods they override.
- Users can additionally extend the preconditions, modifies list and postconditions of overridden methods.

Annotation Language

- Users can also specify object invariants
  - checking all methods to respect these object invariants, results in a huge performance penalty

- Pure Functions (i.e. square root function) can ignore object invariants
  - they don’t use the object whose invariant is being considered

- Disallows function calls that are likely to be an error – passing invalid objects as arguments
- Allows calls to pure functions that don’t make use of the object

Annotation Language

- The annotation language isn’t expressive enough to check correct programs that seem to have a bug.

- Escape hatches tells the checker:
  - to suppress warnings about the source line where the annotation appears
  - to assume holds without checking it

Performance

- ESC/Java was applied to the Javafe project, where the source code contained routines of various sizes

Routine size	# of routines	Percentage checked within time limit 0.1s	1s	10s	1min	5mins
0–10	1720	27	90	100	100	100
10–20	525	1	74	99	100	100
20–50	162	0	33	94	99	100
50–100	35	0	0	74	94	100
100–200	17	0	0	53	82	94
200–500	5	0	0	0	80	100
500–1000	1	0	0	0	0	100
total	2331	20	80	98	> 99	> 99

- Performance of the tool was satisfactory:
  - 1 routine wasn't verified within the maximum 5 minute limit
  - majority of routines were checked in ~1 second

Experience

- One of biggest the costs of running the tool is having the programmer write the annotations

- Experience concludes that only a few annotations are needed – 40-100 per KLOC

Annotation type	Annotations per KLOC	
	Javafe	Mercator
`non_null`	8	6
`invariant`	14	10
`requires`	28	16
`ensures`	26	2
`modifies`	4	0
`assume`	1	11
`nowarn`	6	0
other	4	1
total	94	48

- Annotations can be inserted using an iterative process, where the checker is run and, with its feedback, the programmer can refine or add more annotations.

Related Work

- Includes projects like Euclid, Eiffel and Vault
  - try to verify programs including constructs to express pre and postconditions of procedures
  - these assertions are performed at runtime and not in a static way

- Refinement types – types that are expressive enough to specify an object invariant
  - exist in functional languages
  - recently started to being ported to imperative languages

- Symbolic execution – program analysis technique which explores multiple execution paths at the same time.
  - technique behind tools like PREfix, a tool that finds common programming errors in C/C++ code.

Conclusions

- ESC/Java – checker can detect real and significant software defects
- Trade-off between soundness and usefulness:
  - sacrifices soundness to reduce the annotation cost or to improve performance
- Offers an intuitive and close to Java annotation language
- Even though it catches many bugs, the users mainly complain about:
  - having to annotate the program is a burden
  - the occurrence of false-positives is high
- Authors state that the tool is suitable to use in a classroom:
  - resource for reinforcing lessons on modularity, good design, and verification

Questions?

```
1:  class Bag {
2:      int size;
3:      int[] elements;    // valid:  elements[0..size-1]
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: }
```

```
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: Possible negative array index (...
    elements[minIndex] = elements[size];
                                  ^
```