ESC/Java - Extended Static Checking for Java

Group 1
Adriana Nunes 99172
Francisco Carvalho 99219
José João Ferreira 99259

Topics

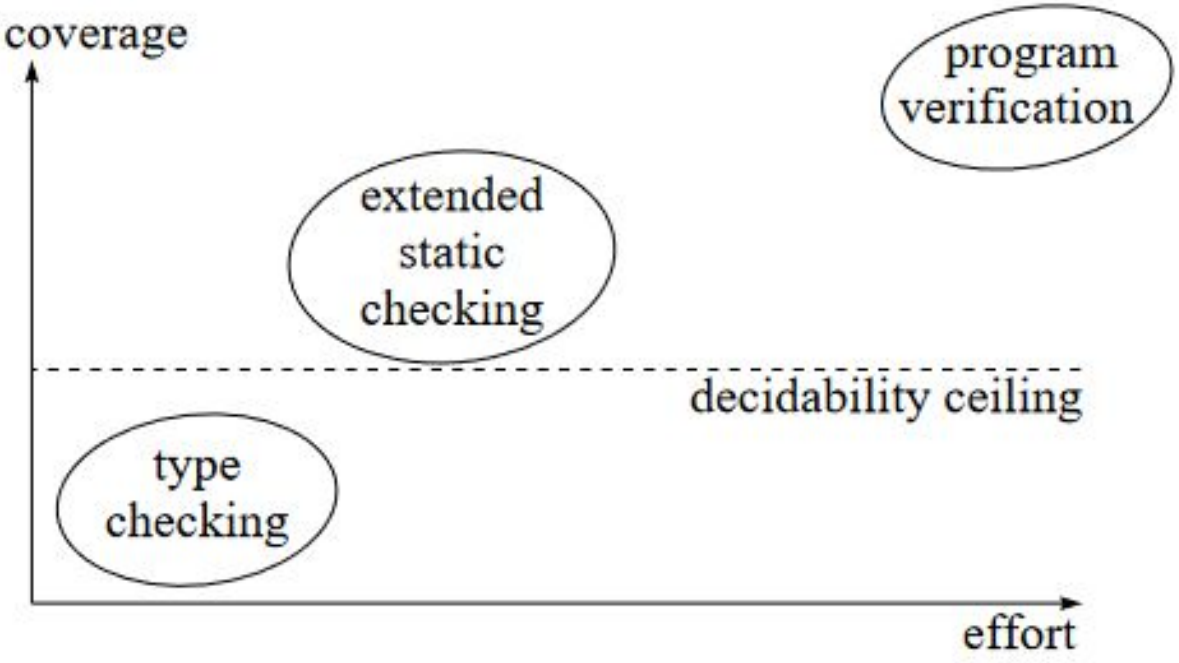
- Introduction
- Using ESC/Java
- Architecture
- Annotation Language
- Performance
- Experience
- Related Work
- Conclusions

Introduction

Introduction

- Static verification tool
- Automatic theorem-prover
- Tries to be a documentation tool
- Warns about problems with the specification (done by the user) and also synchronization

Introduction



Using ESC/Java

Using ESC/Java

```
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, srcPos:0, elements, destPos: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  }
```

Using ESC/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  }
```

```
$ escjava Bag.java
```

```
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];
                             ^
```

Using ESC/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  }
```

```
Bag.java:6: Warning: Possible null dereference (Null)
size = input.length;
           ^
//@ requires input ≠ null
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];
                             ^
```

Using ESC/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  }
```

```
Bag.java:6: Warning: Possible null dereference (Null)
size = input.length;
           ^
//@ requires input ≠ null

Bag.java:15: Warning: Possible null dereference (Null)
if (elements[i] < min) {
            ^
/*@non_null*/

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];
                              ^
/*@non_null*/

Bag.java:21: Warning: Possible negative array index (...
elements[minIndex] = elements[size];
                              ^
```

Using ESC/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  }
```

```
Bag.java:6: Warning: Possible null dereference (Null)
size = input.length;
           ^
//@ requires input ≠ null

Bag.java:15: Warning: Possible null dereference (Null)
if (elements[i] < min) {
            ^
/*@non_null*/

Bag.java:15: Warning: Array index possibly too large (...
if (elements[i] < min) {
            ^
//@ invariant 0 ⩽ size && size ⩽ elements.length

Bag.java:21: Warning: Possible null dereference (Null)
elements[minIndex] = elements[size];
                             ^
/*@non_null*/

Bag.java:21: Warning: Possible negative array index (...
elements[minIndex] = elements[size];
                             ^
//@ invariant 0 ⩽ size && size ⩽ elements.length
```

Using ESC/Java

```
1   class Bag {
2       int size;
3       //@ invariant 0 ≤ size && size ≤ elements.length
4       /*@non null*/ int[] elements;
5
6       //@ requires input ≠ null
7       Bag(int[] input) {
8           size = input.length;
9           elements = new int[size];
10          System.arraycopy(input, 0, elements, 0, size);
11      }
12
13      int extractMin() {
14          int min = Integer.MAX_VALUE;
15          int minIndex = 0;
16          for (int i = 1; i ≤ size; i++) {
17              if (elements[i] < min) {
18                  min = elements[i];
19                  minIndex = i;
20              }
21          }
22          size--;
23          elements[minIndex] = elements[size];
24          return min;
25      }
26  }
```

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

```
if (size ⩾ 0) {
    elements[minIndex] = elements[size];
}
```

$ escjava Bag.java

```
Bag.java:17: Warning: Array index possibly too large (...
if (elements[i] < min) {
            ^

Bag.java:23: Warning: Possible negative array index (...
elements[minIndex] = elements[size];
                             ^
```

Using ESC/Java

```
1   class Bag {
2       int size;
3       //@ invariant 0 ≤ size && size ≤ elements.length
4       /*@non null*/ int[] elements;
5
6       //@ requires input ≠ null
7       Bag(int[] input) {
8           size = input.length;
9           elements = new int[size];
10          System.arraycopy(input, 0, elements, 0, size);
11      }
12
13      int extractMin() {
14          int min = Integer.MAX_VALUE;
15          int minIndex = 0;
16          for (int i = 0; i < size; i++) {
17              if (elements[i] < min) {
18                  min = elements[i];
19                  minIndex = i;
20              }
21          }
22          size--;
23          if (size ≥ 0) {
24              elements[minIndex] = elements[size];
25          }
26          return min;
27      }
28  }
```

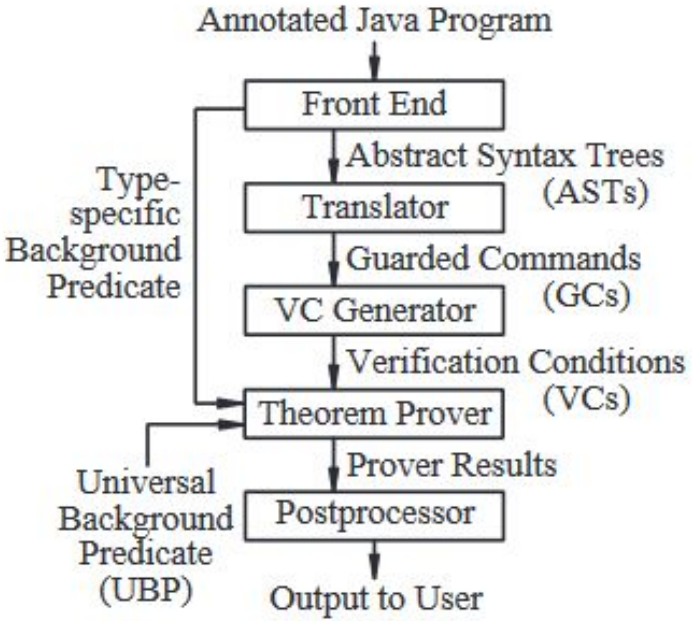
$ escjava Bag.java

```
Bag.java:27: Warning: Possible violation of object invariant
    }
    ^
Associated declaration is "Bag.java", line 3, col 6:
    //@ invariant 0 ≤ size && size ≤ elements.length
        ^
Possibly relevant items from the counterexample context:
    brokenObj == this
(brokenObj* refers to the object for which the invariant
is broken.)
```

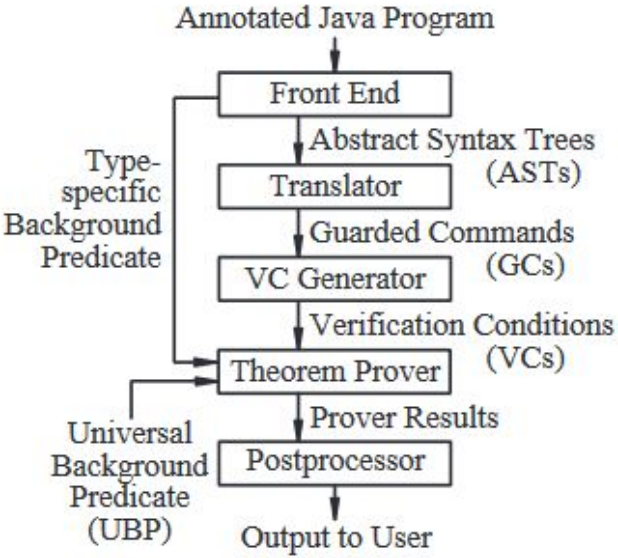
```
if (size > 0) {
    size--;
    elements[minIndex] = elements[size];
}
```

Architecture

Architecture



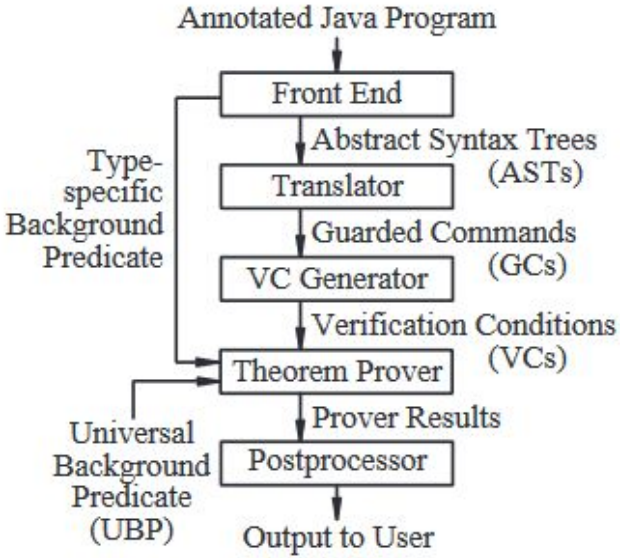
Architecture



Frontend

Similar to a normal Java compiler, but also parses and type checks ESC/Java annotations and Java source code. Produces ASTs and Type-specific background Predicates, a formula in first-order logic that encoding information about the types and fields that routines in that class use.

Architecture

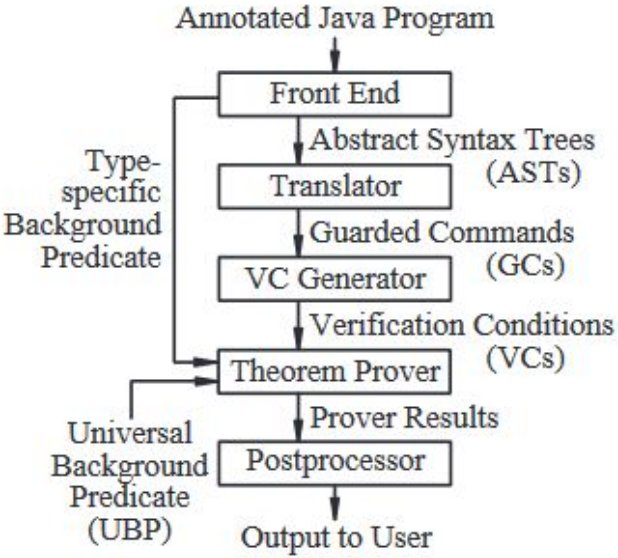


Translator

Translates each routine body to be checked into a language based on Dijkstra's guarded commands. These guarded commands include commands of the form assert E, where E is a boolean expression.

- **Modular checking:** translates the routine call to the specification
- **Overflow:** arithmetic overflow not modeled
- **Loops:** an approximation of the semantics

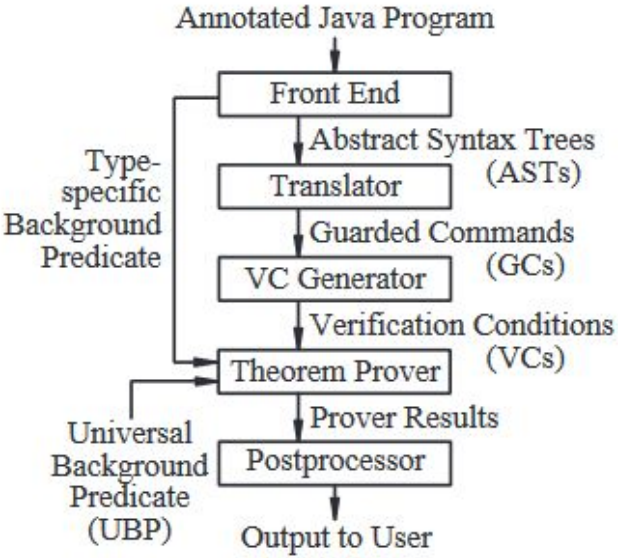
Architecture



VC Generator

Generates verification conditions for each guarded command.
A VC is a predicate in first-order logic that holds for precisely those program states from which no execution of the guarded command can go wrong.

Architecture



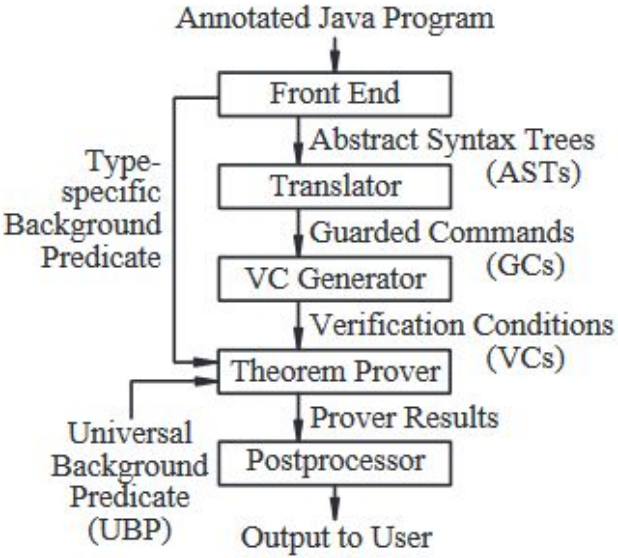
Theorem Prover

Invokes an automatic theorem prover *Simplify* on the conjecture:

$$UBP \land BP \Rightarrow VC$$

- **VC:** verification condition for the routine
- **BP:** the Type-specific Background Predicate for the class T in which the routine is defined
- **UBP:** Universal Background Predicate, encodes some general facts about the semantics of Java

Architecture



Postprocessor

Produces detailed warnings with the help of *Simplify* when the prover is unable to prove verification conditions. The warnings are produced from labels:

```
(LBLNEG |Null@0.6.16| (NEQ |input:0.5.12| null))
```

It can warn multiple counterexample contexts, time limits exceeded and incompleteness.

Annotation Language

Annotation Language: General Design Considerations

- They tried to make the annotation language as “java-like” as possible (can act as primary documentation)
- format: **`@expression+`**
- Abstract variables are not supported (but it supports ghost fields)

Runtime specifications

```
requires P
modifies P
ensures Q
exsures (T,x)R
```

Specification is inherited by default
Can be overwritten with **`also_<expr>`**
(more info in the manual)

Annotation Language: Object Invariants

Operation Boundaries

- Each routine is a separate operation
- Unless it has the modifier **`helper`**

Invariant Enforcement

- Invariants checks are performed at call sites only for arguments and static fields

Invariant Placement

- Invariants must be declared in the class that declares the obj

Constructors

- **`this`** is allowed not to hold invariants in the case of a constructor exit (class is not built)

Annotation Language: Ghost Fields

Java-like fields that are not seen by the compiler

```
//@ ghost public \TYPE elementType
//@ requires \typeof (obj ) <: elementType
public void addElement(Object obj );
```

Escape hatches

`nowarn` – suppresses certain warnings
`@assume P` – assume that P condition holds without checking it

Performance & Experience

Performance

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

- **Program Tested:** *Javafe*
- **Program Size:** 41 thousand lines of code and 2331 routines
- **Processor Used:** 667 MHz Alpha processor
- **Routine Size Distribution:** Most routines in *Javafe* are under 50 lines of code
- **Results:**
  - Majority of routines are checked in less than 10 seconds
  - Only one routine could not be verified within the default five-minute time limit

Experience

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

- ESC/Java is a checker that relies on annotations providing specifications for each routine
- Roughly 40–100 annotations are required per thousand lines of code
- A programmer can annotate 300 to 600 LOC per hour (of an existing program)
- **Possible solution:** write annotations and run checker early in the development cycle

Experience

Mercator (web-crawler)

- **Annotation Effort:** Two developers annotated and checked 4 packages, containing 7 KLOC in 6 hours using ESC/Java.
- **Bug Detection:** ESC/Java detected a previously-undetected bug in the hash table implementation related to a null pointer that was not accounted for in the checkpointing code.

Javafe (Java front-end)

- **Annotation Effort:** One of the researchers spent 3 weeks annotating 30 KLOC of ESC/Java's Front End.
- **Bug Detection:** 12 previously undetected errors were found initially, followed by 12 more after annotating during development.

Experience

Other user experience

- ESC/Java is available for download for free.
- The research team has received more than 100 emails from users in the past year.
- Users have shared success stories of ESC/Java detecting surprising errors in their code.
- Some users have tried using the checker for full functional correctness verification.
- Users faced difficulties due to the incompleteness of Simplify.
- Effective use involves knowing when to use assume annotations for cost efficiency.

Related Work & Conclusions

Related Work

- **ESC/Modula-3 (previous work):** Not so simpler to use, but missed less errors than ESC/Java;

- **PREfix with Symbolic Execution Technique:** Bug finding tool for C and C++. Doesn't have an annotation language, but users could sort warnings by weight;

- **Abstract Interpretation Technique:** A more established technique that uses heuristics to iteratively build up an abstract model of a program. Used successfully in many applications, including space-rocket controllers;

- **Symbolic Model Checking Technique:** Uses predicate abstraction to reason about a given (infinite-state) program as a finite-state system that is model checked. Used in tools such as *Bandera* and *Java PathFinder 2*.

Conclusions

- For two years until the publication, the checker was used to **check a variety of programs**, including moderately large systems;
- The experience of users and developers supports the thesis that ESC/Java can **detect real** and **significant software defects**;
- The performance of the ESC/Java is sufficient for an **interactive use on most methods**, but not the most complex ones;
- Since the annotation language is Java-like and uses object invariants, it helps keeping it **intuitive**;
- Feedback from users suggests that the tool has **not reached the desired level of cost-effectiveness**;
- The authors find this tool suitable for **use in a classroom** setting as a resource for reinforcing lessons on modularity, good design and verification.

Thank you