Extended Static Checking for Java

Lukas Erlacher

TU München - Seminar Verification

14. Juli 2011
Motivation for static checking

ESC/Java example

ESC/JAVA architecture
VC generator
Simplify

JML + ESC/Java annotation language
JML
What ESC/Java checks
Motivation for static checking

Why check a program’s behaviour?
Motivation for static checking

Why check a program’s behaviour?
- Errors / program does not do what we want
- Testing is incomplete and unsound
- Testing is expensive
Motivation for static checking

Why check a program’s behaviour?
- Errors / program does not do what we want
- Testing is incomplete and unsound
- Testing is expensive

Why static checking?
Motivation for static checking

Why check a program’s behaviour?
- Errors / program does not do what we want
- Testing is incomplete and unsound
- Testing is expensive

Why static checking?
- Does not require executing program
- Can cover all code paths
Motivation for static checking

Why check a program’s behaviour?
- Errors / program does not do what we want
- Testing is incomplete and unsound
- Testing is expensive

Why static checking?
- Does not require executing program
- Can cover all code paths

Why ESC/JAVA?
Motivation for static checking

Why check a program’s behaviour?
- Errors / program does not do what we want
- Testing is incomplete and unsound
- Testing is expensive

Why static checking?
- Does not require executing program
- Can cover all code paths

Why ESC/JAVA?
- First static checker for Java
- Architecture and working principle very clear and structured
- Is applicable in practice
- Annotation language allows to specify design that can also be checked
What is static checking?
What is static checking?

- No checking: Program execution breaks on segfault / null pointer dereference / array bounds violation.
What is static checking?

- No checking: Program execution breaks on segfault / null pointer dereference / array bounds violation.
- Type checking: Compiler notices illegal code and violation of specification embedded in type information.
What is static checking?

- No checking: Program execution breaks on segfault / null pointer dereference / array bounds violation.
- Type checking: Compiler notices illegal code and violation of specification embedded in type information.
- Primitive static checking: Flags easily-detected “suspicious” code such as use of uninitialized variables or unreachable code.
What is static checking?

- **No checking**: Program execution breaks on segfault / null pointer dereference / array bounds violation.
- **Type checking**: Compiler notices illegal code and violation of specification embedded in type information.
- **Primitive static checking**: Flags easily-detected “suspicious” code such as use of uninitialized variables or unreachable code.
- **Formal methods**: Formally prove that program is correct.
What is static checking?

- No checking: Program execution breaks on segfault / null pointer dereference / array bounds violation.
- Type checking: Compiler notices illegal code and violation of specification embedded in type information.
- Primitive static checking: Flags easily-detected “suspicious” code such as use of uninitialized variables or unreachable code.
- Formal methods: Formally prove that program is correct.

Extended static checking uses annotations and generic formal methods to show whether a program behaves within the constraints of its specification.
Comparison of checking methods

- Coverage
- Effort
- Type checking
- Extended static checking
- Program verification
- Decidability ceiling

Motivation for static checking
Motivation for static checking

ESC/JAVA history

- Developed at Compaq Systems Research by Flanagan, Leino, Lillibridge, Nelson, Saxe, and Stata
- Descended from ESC/Modula-3
- Developed as practical tool to check programs for semantic errors, specification violations, and synchronization errors in concurrent programs
- Exploits the space between fast, but primitive syntactic checkers like lint and comprehensive, but costly formal program verification
public class Bag {
    int[] elements;
    int size;

    Bag(int[] input) {
        size = input.length;
        elements = new int[size];
        System.arraycopy(input, 0, elements, 0, size);
    }
}
public class Bag {
    int[] elements;
    int size;

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

Bag.java:6: Warning: Possible null dereference (Null)
size = input.length;
^
public class Bag {
  /*@non_null*/ int[] elements;
  int size;

  Bag(/*@non_null*/ int[] input) {
    size = input.length;
    elements = new int[size];
    System.arraycopy(input, 0, elements, 0, size);
  }

  ..
 }

Bag.java:6: Warning: Possible null dereference (Null)
  size = input.length;
  ~
```java
public class Bag {
    /*@non_null*/ int[] elements;
    int size;

    Bag(/*@non_null*/ int[] input) {
        size = input.length;
        elements = new int[size];
        System.arraycopy(input, 0, elements, 0, size);
    }
}
```
ESC/Java example

```java
public class Bag {
    int[] elements; int size;

    int extractMin() {
        int m = Integer.MAX_VALUE;
        int mindex = 0;
        for (int i = 0; i < size; i++) {
            if (elements[i] < m) {
                mindex = i;
                m = elements[i];
            }
        }
        size--;
        elements[mindex] = elements[size];
        return m;
    }
}
```
public class Bag {
    int[] elements; int size;
    ...
    int extractMin() {
        int m = Integer.MAX_VALUE;
        int mindex = 0;
        for (int i = 0; i < size; i++) {
            if (elements[i] < m) {
                mindex = i;
                m = elements[i];
            }
        }
        size--;
        elements[mindex] = elements[size];
        return m;
    }
}
ESC/Java example

```java
/*@invariant size >= 0 && size <= elements.length; */
public class Bag {
    int[] elements; int size;
    
    int extractMin() {
        int m = Integer.MAX_VALUE;
        int mindex = 0;
        for (int i = 0; i < size; i++) {
            if (elements[i] < m) {
                mindex = i;
                m = elements[i];
            }
        }
        size--;
        elements[mindex] = elements[size];
        return m;
    }
}
```
non_null: Forces assigners to always assign a valid instance - allows users to assume that instance is always valid

invariant: introduces the invariant as precondition and post-condition to every method call

precondition: forces caller to establish precondition before calling

postcondition: forces method to establish post-condition before returning
The basic steps in ESC/Java’s operation.
Guarded Command Language

- Originally designed by Dijkstra (1975)
- Contains only variable declarations and assignments, assertions, assumptions, and constructs to handle sequential composition, branching, and exceptions
- Routines are translated into guarded commands that capture the relevant semantics of the routine. Guarded command “goes wrong” when it hits an assertion that evaluates to false.
- Soundness: A guarded command $G$ translated from a routine $R$ goes wrong iff $R$ can be invoked from a state satisfying its stated preconditions and then behaves erroneously by causing an error or terminating in a state violating its specified postconditions.
VC generator

- Verification condition: First-order predicate that holds for precisely the program states from which execution of a guarded command does not go wrong.
- Weakest liberal precondition ($wlp$) derived directly from a routine’s GC
- Global information (about Java) and class-scope information forms “Background predicate” ($BP$)

$BP \Rightarrow wlp$
Simplify

- Automatic theorem prover developed for ESC/JAVA
- Verifies the $BP \Rightarrow wlp$ predicate
- Limited runtime, caution issued if exceeded
- Results used by post-processor to generate warnings
- Incomplete (cannot prove all valid formulas), but sound (does not erroneously prove invalid formulas)
The annotation language is used to specify usage contracts, encode design properties that are not expressed in the programme code, and assist ESC/JAVA. Annotations are called “pragmas”:

- Basic pragmas: nowarn / assume, assert / unreachable
- Routine pragmas: requires, modifies, ensures, exsures, also...
- Invariant pragmas: non_null, invariant, axiom, loop_invariant
- Accessibility pragmas: spec_public, readable_if, uninitialized
- Ghost variable pragmas: ghost, set
- Synchronization pragmas: monitored_by, monitored
Specification expressions

- Superset of side-effect-free Java expressions, plus syntax to express lock hierarchy and type expressions
- Additional keywords: `\old`, `\modifies`, `\typeof`, `\lockset`
JML

- Java Modelling Language, inspired by ESC/JAVA annotation language
- Allows to specify behaviour and contracts of Java programs and APIs
- Used by a big ecosystem of static checkers, testing engines, documentation tools
- Readable and writeable by Java programmers
What ESC/JAVA checks

- Errors: Runtime type errors (array assignment, cast), unchecked exceptions, array bounds violations, null dereference, zero division
- Concurrency problems: deadlocks, races
- Violated invariants, pre and post-conditions, loop invariants
- Violated assertions, non-null pragmas, accessibility pragmas

ESC/JAVA does not check:

- Whether a loop invariant holds past the first iteration of a loop
- Arithmetic overflow