1. Overview

2. Examples

3. Handling of Loops

4. Internal Operation

---

**Theoretical Limitations**

**ESC/Java2 is not sound.**
- Soundness: if $\{P\}c\{Q\}$ does not hold, it cannot be proved.
  - ESC/Java2 may not produce warning on wrong $\{P\}c\{Q\}$.
- Sources of unsoundness:
  - Loops are handled by unrolling, arithmetic is on $\mathbb{Z}$.
  - JML annotation assume adds unverified knowledge.
  - Object invariants are not verified on all existing objects.

**ESC/Java2 is not complete.**
- Completeness: if $\{P\}c\{Q\}$ cannot be proved, it does not hold.
  - ESC/Java2 may produce superfluous warnings.
- Sources of incompleteness:
  - Simplify’s limited reasoning capabilities (arithmetic, quantifiers).
  - JML annotation nowarn to turn off warnings.
  - Potentially not sound.

Not every error is detected, not every warning actually denotes an error.


## Practical Usefulness

- ESC/Java2 detects many (most) programming errors.
  - Array index bound violations.
  - Division by zero.
  - Null-pointer dereferences.
  - Violation of properties depending on linear arithmetic.
  - ...
- Forces programmer to write method contracts.
  - Especially method preconditions.
  - Better documented and better maintainable code.

A useful extension of compiler type checking.

## Use of ESC/Java2

- Command-line interface.
  - `escjava2 [options] File.java`
- Graphical interface.
  - `java -jar esctools2.jar`
- Eclipse 3.5 plugin.
  - See web site.
  - `escjava2 -help`.

## Tutorial Program

```java
class Bag {
    int[] a; int n;

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

    int extractMin() {
        int m = Integer.MAX_VALUE;
        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;
    }
}
```
Tutorial Program: Assumptions

class Bag {
    /**< non_null */ int[] a;
    int n; /**< invariant 0 <= n && n <= a.length; */
    /**< requires input != null; */
    Bag(int[] input) {
        ...
    }
    /**< requires n>0; */
    int extractMin() {
        ...
    }
}

Invariants and preconditions have to be added to pass the checking.

Tutorial Program: Guarantees

/*@ requires n>0; @*/
/*@ ensures n == \old(n)-1; @*/
/*@ ensures (\forall int i; 0 <= i && i < \old(n);
@   \result <= \old(a[i])); @*/
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[0]; // ERROR: a[0] rather than a[i]
        }
    }
    n--;
    a[mindex] = a[n];
    return m;
}

Postconditions may be added (and are checked to some extent).

Tutorial Program: Wrong Guarantees

/*@ requires n>0; @*/
/*@ ensures n == \old(n)-1; @*/
/*@ ensures (\forall int i; 0 <= i && i < \old(n);
@   \result <= \old(a[i])); @*/
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[0]; // ERROR: a[0] rather than a[i]
        }
    }
    n--;
    a[mindex] = a[n];
    return m;
}

But also this program passes the check!

Example Program: Arithmetic1

//@ 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;
}

Masters linear integer arithmetic with inequalities.
**Example Program: Conditional**

```c
/*@ ensures (result == i || result == j || result == k)
@ && (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;
}
```

Masters conditionals.

**Example Program: Sort**

```c
/*@ requires a != null;
@ ensures (\forall int i; 0 <= i && i < a.length-1; a[i] <= a[i+1]); @*/
static void insertSort(int[] a)
{
    int n = a.length;
    for (int i = 1; i < n; i++) {
        int x = a[i];
        int j = i-1;
        while (j >= 0 && a[j] > x) {
            a[j+1] = a[j];
            j = j-1;
        }
        a[j+1] = x;
    }
}
```

Detects many errors in array-based programs.

**Example Program: Arithmetic2**

```c
//@ ensures result == i*i;
static int f1(int i)
{
    return i*(i+1)-i;
}//@ nowarn Post;
//@ ensures result >= 0;
static int f2(int i)
{
    return i*i;
}//@ nowarn Post;
```

Does not master non-linear arithmetic.

**Example Program: Loop**

```c
//@requires n >= 0;
static void loop(final int n)
{
    int i=0;
    while (i < n)
    {
        i = i+1;
    }
    //@ assert i==n;
    //@ assert i<3;
}
```

Does only partially master post-conditions of programs with loops.
1. Overview
2. Examples
3. Handling of Loops
4. Internal Operation

Wolfgang Schreiner http://www.risc.jku.at

Loop Unrolling

We will now use a high-level description of the ESC/Java2 handling of loops by loop unrolling.

- Original program.
  ```
  while (e) c;
  ```
- Unrolling the loop once.
  ```
  if (e) {
    c;
    while (e) c;
  }
  ```
- Unrolling the loop twice.
  ```
  if (e) {
    c;
    if (e) {
      c;
      while (e) c;
    }
  }
  ```

Faithful loop unrolling preserves the meaning of a program.

Verification of Unrolled Program

Let us consider how verification is affected by loop unrolling.

- Original: \{P\} while(e) c \{Q\}
  ```
  P \Rightarrow wp(while(e) c, Q)
  ```

- Unrolled: \{P\} if (e) \{c; if (e) \{c; while (e) c\}\} \{Q\}
  ```
  (P \land \neg e) \Rightarrow Q
  ```

- Assume false: from false, everything can be concluded.
  ```
  assume false: \text{from false, everything can be concluded.}
  ```

Only simplified program is verified by ESC/Java2.

ESC/Java2 Loop Unrolling

- Faithful unrolling
  ```
  \{P\} if(e) \{c; if(e) \{c; while (e) c\}\} \{Q\}
  ```

- ESC/Java2 default unrolling
  ```
  \{P\} if(e) \{c; if(e) \{
    assume false;
  \}\} \{Q\}
  ```

Not unrolled execution of loop is replaced by "assume false".

- assume false: from false, everything can be concluded.
- No more verification takes place in this branch.

Three obligations (1-3) equivalent to original obligation (0).
**Verification of Unrolled Program**

Let us consider the simplified verification problem.

\[
\begin{align*}
\{ P \} \text{if}(e) \{ c; \text{if}(e) \{ \text{assume false} \} \} \{ Q \} \\
(P \land \neg e) \Rightarrow Q & \quad (1) \\
\{ P \land e \} c; \text{if}(e) \{ \text{assume false} \} \{ Q \} \\
\{ P \land e \} c \{ \neg e \Rightarrow Q \} & \quad (2) \\
\{ P \land e \} c \{ e \land \text{false} \Rightarrow Q \} & \quad \Leftrightarrow \{ P \land e \} c \{ \text{true} \} \\
\Leftrightarrow \text{true} \\
\end{align*}
\]

Proof obligation (3) of the original problem is dropped.

**Expressive Power of Simplified Verification**

What does this mean for the whole verification process?

- **Example program:**
  
  ```
  while (e) { c1 } c2
  ```

- **Verified program:**
  
  ```
  if (e) { c1; if (e) { \text{assume false} } } c2
  if (e) { c1; if (e) { \text{assume false} } c2 } \text{else} c2
  if (e) { c1; if (e) { \text{assume false}; c2 } \text{else} c2 } \text{else} c2
  if (e) { c1; if (e) \text{skip} \text{else} c2 } \text{else} c2
  if (e) { c1; if (\neg e) c2 } \text{else} c2
  ```

- In verified program, only runs are considered where
  - loop terminates after at most one iteration, i.e.
  - execution of \( c2 \) is only considered in such program runs.

After a loop, only special contexts are considered for verification.

**Control of Loop Unrolling**

**ESC/Java2 control of loop unrolling**

```
escjava2 -loop n.5
``` 

- Loop is unrolled \( n \) times (default \( n = 1 \)).
- \( .5 \): also loop condition after \( n \)-th unrolling is checked.

- **Preconditions.**
  - All preconditions are checked that arise from the loop expression and the loop body in the first \( n \) iterations.

- **Postconditions.**
  - It is checked whether the postcondition of the loop holds in all executions that require at most \( n \) iterations.

All program paths with more than \( n \) iterations are “cut off”.

---

Wolfgang Schreiner  
http://www.risc.jku.at

21/47
Unsoundness of Loop Unrolling

Unsoundness of strategy can be easily shown.

```java
int i=0;
while (i < 1000)
i = i+1;
//@ assert i < 2;
```

For unrolling with \( n < 1000 \), this postcondition is true.

For any execution, that terminates after at most \( n \) iterations (i.e. none), the postcondition is true.

For true verification of loop programs, reasoning about a loop invariant is required.

---

Internal Operation

Java program is first translated into a much simpler language. Variant of Dijkstra’s guarded command (GC) language.

```
cmd ::= variable = expr | skip | raise | assert expr | assume expr |
    var variable+ in cmd end | cmd ; cmd | cmd ! cmd | cmd [] cmd.
```

Actually, first a sugared version of the language.

```
cmd ::= . . |
    check expr | call p(expr*) | loop { invariant expr } cmd end.
```

Then desugar program, i.e. translate it into core language.

Various desugaring strategies possible.

Then generate verification conditions for program in core language.

Verification conditions are forwarded to theorem prover.

We first discuss the semantics of the core language and then the translation process Java \( \rightarrow \) sugared GC \( \rightarrow \) core GC.
Core Language Semantics

Defined by weakest preconditions.

\[
wp(cmd, N, X)
\]

- Weakest condition on state in which \( cmd \) may be executed such that
  - either \( cmd \) terminates normally in a state in which \( N \) holds,
  - or \( cmd \) terminates exceptionally in a state in which \( X \) holds.
- All commands in the core language terminate.
  - No distinction to weakest liberal precondition.
- Relationship to total correctness.
  \[
  \{ P \} ~ cmd ~ \{ Q \} \iff (P \Rightarrow wp(c, Q, false))
  \]

Two ways how a command may terminate.

Core Language Semantics

\[
wp(x = e, N, X) \iff N[e/x]
\]

\[
wp(skip, N, X) \iff N
\]

\[
wp(raise, N, X) \iff X
\]

\[
wp(assert e, N, X) \iff (e \Rightarrow N) \land \exists X, \exists N, \exists X
\]

\[
wp(c_1; c_2, N, X) \iff wp(c_1, wp(c_2, N, X), X)
\]

Tuple of postconditions has to be considered.

Core Language Semantics

Core Language Semantics

Core Language Semantics

Core Language Semantics

Tuple of postconditions has to be considered.
Core Language Semantics

\[ wp(\text{raise}, N, X) \Leftrightarrow X \]
\[ wp(c_1|c_2, N, X) \Leftrightarrow wp(c_1, N, wp(c_2, N, X)) \]

- Interpretation of \textit{raise} rule
  - The command terminates not normally but exceptionally.
  - Thus the exceptional postcondition \( X \) must hold before the call.
- Interpretation of signal handling rule (\!).
  - If \( c_1 \) terminates normally, the normal postcondition \( N \) must hold (because \( c_2 \) is not executed).
  - If \( c_1 \) terminates exceptionally, it must be in a state such that the execution of \( c_2 \) ensures the required postconditions \( N \) and \( X \).

Note the symmetry of command composition and exception handling.

Example

What is the weakest preconditon such that
\[(x = x + 1; \text{raise}; x = x − 2) \not\rightarrow x = x + 2\]

normally terminates in a state with \( x = 3 \)?

\[ wp(((x = x + 1; x = x − 2) ! x = x + 2), x = 3, false) \]
\[ \Leftrightarrow wp((x = x + 1; x = x − 2), x = 3, wp(x = x + 2, x = 3, false)) \]
\[ \Leftrightarrow wp((x = x + 1; x = x − 2), x = 3, x + 2 = 3) \]
\[ \Leftrightarrow wp((x = x + 1; x = x − 2), x = 3, x = 1) \]
\[ \Leftrightarrow wp(x = x + 1, wp(\text{raise}; x = x − 2), x = 3, x = 1, x = 1) \]
\[ \Leftrightarrow wp(x = x + 1, wp(\text{raise}; wp(x = x − 2, x = 3, x = 1), x = 1), x = 1) \]
\[ \Leftrightarrow wp(x = x + 1, x = 1, x = 1) \]
\[ \Leftrightarrow x + 1 = 1 \]
\[ \Leftrightarrow x = 0 \]

Translation of Java Loops

The guarded command language does not have while loops.

- Translation of while (e) \{ \( c_1 \) \} \( c_2 \)
  \[ \text{loop if } \lnot (\text{e}) \text{ raise; } c_1 \text{ end } ! \ c_2 \]
- Construct \textbf{loop} runs forever.
  - Loop is terminated by signalling an exception in the body.
  - Exception is caught and \( c_2 \) is executed.

Replacement of while loops by core \textbf{loop} and exceptions.
Translation of Java Conditionals
The guarded command language also does not have conditionals.

- Translation of if (e) c1 else c2.
  
  ( assume e ; c1 ) [] ( assume ¬e ; c2 )

- Translation of if (e) c.
  
  ( assume e ; c ) [] ( assume ¬e ; skip )

- Non-deterministic selection of two commands.
  - One of two branches is executed.
  - Each branch is guarded by a condition which can be assumed to be true in that branch.
  - Conditions are mutually exclusive, thus actually only one branch can be executed.

Replacement of conditionals by guarded selection of commands.

Checking Expressions
Handling of preconditions.

  check expr;

- Occurs e.g. in translation of object dereferencing
  
  v = o.f

  check o != null; v = select(o, f)

- Possible translation of check expr.
  1. Treat violation as error.
     assert expr
  2. Ignore violation (user has switched warning off).
     assume expr
  3. Treat violation as runtime exception.
     if (!expr) raise

Translation partially controlled by nowarn annotations.

Procedure Calls
Call of a procedure r that is allowed to modify a variable x.

  call r(e0, e1)

- Translation (simplified): 
  
  var p0, p1 in
  p0 = e0; p1 = e1;
  check precondition (involves p0, p1);
  var x0 in
  x0 = x;
  modify x;
  assume postconditions (involves p0, p1, x0, x);
  end
  end

- modify x desugars to
  var x' in x = x'
end

Reduce complex procedure call rule to simpler constructs.

Loops
Execution of a core loop.

  loop { invariant expr } cmd end

- Handling by loop unrolling.
  
  check expr, cmd;
  check expr, cmd;

  . . .

  check expr, assume false.

- By default, loops are unrolled just once.
  
  escjava2 -loop 1.5

We have already investigated the consequence of this.
Verification Conditions

For program in core language, verification conditions are generated.

- Pretty-print generated verification conditions.
  
  escjava2 -v -ppvc Simple.java

... (OR
  (AND (>= |x| 0) (EQ |@true| |@true|))
  (NOT (>= |x| 0))
  (EQ |@true| |@true|)
)
(EQ |y| (- 0 |x|))
...

Hardly readable, only for understanding details.

Wolfgang Schreiner http://www.risc.jku.at 41/47

Simplify

Simplify(1)

NAME
Simplify -- attempt to prove first-order formulas.

SYNTAX
Simplify [-print] [-ax axfile] [-nosc] [-noprune]
[-help] [-version] [file]

DESCRIPTION
*Simplify* accepts a sequence of first order formulas as input, and attempts to prove each one. *Simplify* does not implement a decision procedure for its inputs: it can sometimes fail to prove a valid formula. But it is conservative in that it never claims that an invalid formula is valid.

Formula Syntax

| formula ::= "(" ( AND | OR ) { formula } ")" |
| "(" NOT formula ")" |
| "(" IMPLIES formula formula ")" |
| "(" IFF formula formula ")" |
| "(" FORALL "(" var* ")" formula ")" |
| "(" EXISTS "(" var* ")" formula ")" |
| "(" PROOF formula* ")" |
| literal |

| literal ::= "(" ( "EQ" | "NEQ" | "<" | "<=" | "<" | "<=" )
  term term ")" |
| "(" "DISTINCT" term term+ ")" |
| "TRUE" | "FALSE" | <propVar>

| term ::= var | integer | "(" func { term } ")" |

The formula

| (DISTINCT term1 ... termN)
represents a conjunction of distinctions between all pairs of terms in the list.

The formula

| (PROOF form1 ... formN)
is sugar for

| (AND (IMPLIES form1 form2) |
  (IMPLIES (AND form1 form2) form3) |
  ... |
  (IMPLIES (AND form1 ... formN-1) formN))

"func"'s are uninterpreted, except for "+", "-", and "*", which represent the obvious operations on integers.

Wolfgang Schreiner http://www.risc.jku.at 42/47

Wolfgang Schreiner http://www.risc.jku.at 43/47

Wolfgang Schreiner http://www.risc.jku.at 44/47
**Default Axioms**

\[
\begin{align*}
&\forall (a \ i \ x \ k) \quad (\text{EQ} \ (\text{select} \ (\text{store} \ a \ i \ x) \ i \ k) \ x) \\
&\forall (a \ i \ n) \\
&\quad (\text{EQ} \ (\text{len} \ (\text{subMap} \ a \ i \ n)) \ n) \\
&\forall (a \ i \ n \ j \ k) \\
&\quad (\text{EQ} \ (\text{select} \ (\text{subMap} \ a \ i \ n) \ j \ k) \ (\text{select} \ a \ (+ \ i \ j) \ k)) \\
&\forall (a \ i \ x) \\
&\quad (\text{EQ} \ (\text{len} \ (\text{store} \ a \ i \ x)) \ (\text{len} \ a)) \\
&\forall (a \ i \ n \ b) \\
&\quad (\text{EQ} \ (\text{len} \ (\text{storeSub} \ a \ i \ n \ b)) \ (\text{len} \ a)) \\
&\forall (v \ i) \\
&\quad (\text{EQ} \ (\text{select} \ (\text{mapFill} \ v) \ i) \ v) \\
&\forall (i \ j \ a \ x \ k) \\
&\quad (\text{OR} \ (\text{EQ} \ i \ j) \ (\text{EQ} \ (\text{select} \ (\text{store} \ a \ i \ x) \ j \ k) \ (\text{select} \ a \ j \ k))) \\
&\forall (j \ i \ a \ n \ b \ k) \\
&\quad (\text{OR} \ (\text{AND} \ (\text{OR} \ (< \ j \ i) \ (\geq \ j \ (+ \ i \ n)))) \\
&\quad (\text{EQ} \ (\text{select} \ (\text{storeSub} \ a \ i \ n \ b) \ j \ k) \ (\text{select} \ a \ j \ k)) \\
&\quad (\text{AND} \ (\geq \ j \ i)) \\
&\quad (< \ j \ (+ \ i \ n)) \\
&\quad (\text{EQ} \ (\text{select} \ (\text{storeSub} \ a \ i \ n \ b) \ j \ k) \ (\text{select} \ b \ (+ \ j \ i) \ k)))
\end{align*}
\]

**Power of Simplify**

Simplify can be used as a “pocket calculator for reasoning”.

- **Prover for first-order logic with equality and integer arithmetic.**
  - For proving formula \( F \), the satisfiability of \( \neg F \) is checked.
  - If \( \neg F \) is not satisfiable, the prover returns “valid”.
  - If \( \neg F \) is satisfiable, the prover returns a counterexample context.
    - Conjunction of literals (atomic formulas, plain or negated) that is believed to satisfy \( \neg F \).
- **Proving strategy is sound.**
  - If \( F \) is reported “valid”, this is the case.
- **Proving strategy is not complete.**
  - A reported counterexample context may be wrong.

**Sound, not complete, highly optimized.**

---

**Conclusions**

- **ESC/Java2 is a good tool for finding program errors.**
  - Reports many/most common programming errors.
  - Forces programmer to write method preconditions/assertions.
  - Stable, acceptably fast.
- **ESC/Java2 is not a verification environment.**
  - Postconditions of methods with loops are not appropriately verified.
  - Arithmetic is treated as arbitrary size, not finite.
- **Resources:**