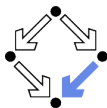
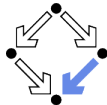


Extended Static Checking with ESC/Java2

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
<https://www.risc.jku.at>





1. Overview

2. Examples

3. Handling of Loops

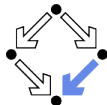
4. Internal Operation



- Latest outcome of a series of projects.
 - Compaq: ESC/Modula-3 (–1996), ESC/Java (–2000).
 - Univ. Nijmegen (–2005), Univ. College Dublin (2005–): ESC/Java2.
 - <https://github.com/GaloisInc/ESCJava2>
- Extended Static Checking for Java.
 - Find programming errors by automated reasoning techniques.
 - Simplified variant of Hoare/weakest precondition calculus.
 - Full Java 1.4 (much of Java 1.5), fully automatic.
 - Feels like type-checking.
 - Uses JML for specification annotations (ESC/Java2).
 - ESC/Modula-3 and ESC/Java had their own annotation language.
- Based on the **Simplify** prover.
 - Greg Nelson et al, written in Modula-3 for ESC/Modula-3.

Finding errors in a program rather than verifying it.

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.

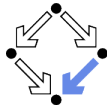
Not every error is detected, not every warning actually denotes an error.

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.



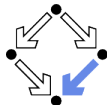
1. Overview

2. Examples

3. Handling of Loops

4. Internal Operation

Tutorial Program



```
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;
    }
}
```

> `escjava2 Bag.java`

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.

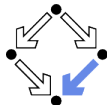
Tutorial Program: Guarantees



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

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

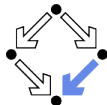


```
//@ ensures \result == i;
static int f2(int i)
{
    int j = i+1;
    int k = 3*j;
    return k-2*i-3;
}

//@ 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



```
/*@ 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



```
/*@ 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



```
//@ 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;
```

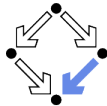
Does not master non-linear arithmetic.

Example Program: Loop



```
//@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.

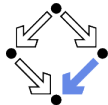


1. Overview

2. Examples

3. Handling of Loops

4. Internal Operation



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;
```

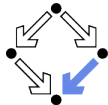
- Unrolling the loop once.

```
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.

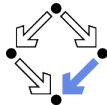
- Original: $\{P\} \text{ while}(e) \ c \ \{Q\}$
 - $\underline{P \Rightarrow \text{wp}(\text{while}(e) \ c, Q)}$ (0)

- Unrolled: $\{P\} \text{ if } (e) \ \{c; \text{ if } (e) \ \{c; \text{ while } (e) \ c\}\} \ \{Q\}$
 - $\underline{(P \wedge \neg e) \Rightarrow Q}$ (1)

- $\{P \wedge e\} \ c; \text{ if } (e) \ \{c; \text{ while } (e) \ c\} \ \{Q\}$
 - $\underline{\{P \wedge e\} \ c \ \{\neg e \Rightarrow Q\}}$ (2)

- $\underline{\{P \wedge e\} \ c \ \{e \Rightarrow \text{wp}(c; \text{ while } (e) \ c, Q)\}}$ (3)

Three obligations (1-3) equivalent to original obligation (0).



ESC/Java2 Loop Unrolling

- Faithful unrolling

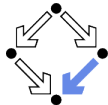
$\{P\} \text{if}(e) \{c; \text{if}(e) \{c; \text{while} (e) c\}\} \{Q\}$

- ESC/Java2 default unrolling

$\{P\} \text{if}(e) \{c; \text{if}(e) \{ \mathbf{assume} \text{ false}; \}\} \{Q\}$

- 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.

Only simplified program is verified by ESC/Java2.



Verification of Unrolled Program

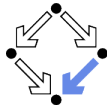
Let us consider the simplified verification problem.

$$\begin{aligned} & \blacksquare \{P\} \text{ if}(e) \{c; \text{ if}(e) \{ \mathbf{assume\ false}\} \} \{Q\} \\ & \quad \blacksquare \frac{(P \wedge \neg e) \Rightarrow Q}{\{P \wedge e\} c; \text{ if}(e) \{ \mathbf{assume\ false}\} \{Q\}} \end{aligned} \tag{1}$$

$$\begin{aligned} & \blacksquare \frac{\{P \wedge e\} c \{\neg e \Rightarrow Q\}}{\{P \wedge e\} c \{e \wedge \mathbf{false} \Rightarrow Q\}} \\ & \quad \Leftrightarrow \{P \wedge e\} c \{\mathbf{true}\} \\ & \quad \Leftrightarrow \mathbf{true} \end{aligned} \tag{2}$$

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

Expressive Power of Simplified Verification



- Checked proof obligations
 - $(P \wedge \neg e) \Rightarrow Q$
 - Postcondition holds, if loop terminates after zero iterations.
 - $\{P \wedge e\} c \{\neg e \Rightarrow Q\}$
 - Postcondition holds, if loop terminates after one iteration.
- Unchecked proof obligation
 - $\{P \wedge e\} c \{e \Rightarrow \text{wp}(c; \text{while } (e) c, Q)\}$
 - Postcondition holds, if loop terminates after **more than one** iteration.

Only partial verification of loops in ESC/Java 2.

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) { assume false } } c2
```

```
if (e) { c1; if (e) { assume false } c2 } else c2
```

```
if (e) { c1; if (e) { assume false; c2 } else c2 } else c2
```

```
if (e) { c1; if (e) skip else c2 } else c2
```

```
if (e) { c1; if (¬e) c2 } else c2
```

- In verified program, only runs are considered where
 - loop terminates after at most one iteration, i.e.
 - execution of c_2 is only considered in such program runs.

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

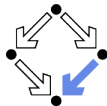
Control of Loop Unrolling



- 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” .

Unsoundness of Loop Unrolling

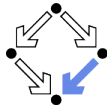


- Unsoundness of strategy can be easily shown.

```
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.

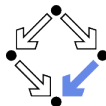


1. Overview

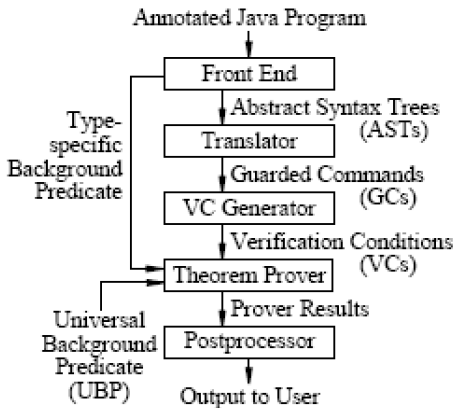
2. Examples

3. Handling of Loops

4. Internal Operation

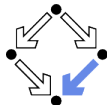


Internal Operation



From Leino et al (2002): Extended Static Checking for Java.

Guarded Commands



Java program is first translated into a much simpler language.

- Variant of **Dijkstra's guarded command (GC) language**.

$$\text{cmd} ::= \text{variable} = \text{expr} \mid \text{skip} \mid \text{raise} \mid \text{assert } \text{expr} \mid \text{assume } \text{expr} \mid \\ \text{var } \text{variable}^+ \text{ in } \text{cmd} \text{ end} \mid \text{cmd} ; \text{cmd} \mid \text{cmd} ! \text{cmd} \mid \text{cmd} [] \text{cmd}.$$

- Actually, first a **sugared** version of the language.

$$\text{cmd} ::= \dots \mid \\ \text{check } \text{expr} \mid \text{call } p(\text{expr}^*) \mid \text{loop } \{ \text{invariant } \text{expr} \} \text{cmd} \text{ end}.$$

- 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 $\text{Java} \rightarrow \text{sugared GC} \rightarrow \text{core GC}$.

Monitoring the Translation



- Print guarded command version of language.

```
escjava2 -pgc Simple.java
```

- Java program.

```
int y; if (x >= 0) y = x; else y = -x;
```

- Guarded command program (simplified).

```
VAR int y IN
{
  ASSUME integralGE(x, 0); y = x;
[]
  ASSUME boolNot(integralGE(x,0)); y = -x;
}
END
```

Low-level program; only necessary for understanding details.

Core Language Semantics



Defined by weakest preconditions.

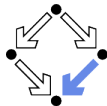
$$wp(cmd, N, X)$$

- 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.

$$\{P\} c \{Q\} \Leftrightarrow (P \Rightarrow wp(c, Q, \text{false}))$$

Two ways how a command may terminate.

Core Language Semantics



$$\text{wp}(x = e, N, X) \Leftrightarrow N[e/x]$$

$$\text{wp}(\text{skip}, N, X) \Leftrightarrow N$$

$$\text{wp}(\text{raise}, N, X) \Leftrightarrow X$$

$$\text{wp}(\text{assert } e, N, X) \Leftrightarrow (e \Rightarrow N) \wedge (\neg e \Rightarrow X)$$

$$\text{wp}(\text{assume } e, N, X) \Leftrightarrow (e \Rightarrow N)$$

$$\text{wp}(\text{var } x_1, \dots, x_n \text{ in } c, N, X) \Leftrightarrow \forall x_1, \dots, x_n : \text{wp}(c, N, X)$$

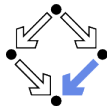
$$\text{wp}(c_1; c_2, N, X) \Leftrightarrow \text{wp}(c_1, \text{wp}(c_2, N, X), X)$$

$$\text{wp}(c_1!c_2, N, X) \Leftrightarrow \text{wp}(c_1, N, \text{wp}(c_2, N, X))$$

$$\text{wp}(c_1 \square c_2, N, X) \Leftrightarrow \text{wp}(c_1, N, X) \wedge \text{wp}(c_2, N, X)$$

Tuple of postconditions has to be considered.

Core Language Semantics



$$\text{wp}(\text{skip}, N, X) \Leftrightarrow N$$

$$\text{wp}(c_1; c_2, N, X) \Leftrightarrow \text{wp}(c_1, \text{wp}(c_2, N, X), X)$$

- Interpretation of **skip** rule
 - The command terminates normally but not exceptionally.
 - Thus the normal postcondition N must hold before the call.
- Interpretation of command composition rule (;).
 - If c_1 terminates exceptionally, the exceptional postcondition X must hold (because c_2 is not executed).
 - If c_1 terminates normally, it must be in a state such that the execution of c_2 ensures the required postconditions N and X .

Slight generalization of the basic rule of the weakest precondition of command composition.

Core Language Semantics



$\text{wp}(\text{raise}, N, X) \Leftrightarrow X$

$\text{wp}(c_1!c_2, N, X) \Leftrightarrow \text{wp}(c_1, N, \text{wp}(c_2, N, X))$

- Interpretation of **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



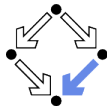
What is the weakest precondition such that

$$(x = x + 1; x = x - 2) ! x = x + 2$$

normally terminates in a state with $x = 3$?

$$\begin{aligned} & \text{wp}(((x = x + 1; x = x - 2) ! x = x + 2), x = 3, \text{false}) \\ \Leftrightarrow & \text{wp}((x = x + 1; x = x - 2), x = 3, \text{wp}(x = x + 2, x = 3, \text{false})) \\ \Leftrightarrow & \text{wp}((x = x + 1; x = x - 2), x = 3, x + 2 = 3) \\ \Leftrightarrow & \text{wp}((x = x + 1; x = x - 2), x = 3, x = 1) \\ \Leftrightarrow & \text{wp}(x = x + 1, \text{wp}(x = x - 2, x = 3, x = 1), x = 1) \\ \Leftrightarrow & \text{wp}(x = x + 1, x - 2 = 3, x = 1) \\ \Leftrightarrow & \text{wp}(x = x + 1, x = 5, x = 1) \\ \Leftrightarrow & x + 1 = 5 \\ & x = 4 \end{aligned}$$

Example

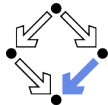


What is the weakest precondition such that

$$(x = x + 1; \mathbf{raise}; x = x - 2) ! x = x + 2$$

normally terminates in a state with $x = 3$?

$$\begin{aligned} & \text{wp}(((x = x + 1; \mathbf{raise}; x = x - 2) ! x = x + 2), x = 3, \text{false}) \\ \Leftrightarrow & \text{wp}((x = x + 1; \mathbf{raise}; x = x - 2), x = 3, \text{wp}(x = x + 2, x = 3, \text{false})) \\ \Leftrightarrow & \text{wp}((x = x + 1; \mathbf{raise}; x = x - 2), x = 3, x + 2 = 3) \\ \Leftrightarrow & \text{wp}((x = x + 1; \mathbf{raise}; x = x - 2), x = 3, x = 1) \\ \Leftrightarrow & \text{wp}(x = x + 1, \text{wp}((\mathbf{raise}; x = x - 2), x = 3, x = 1), x = 1) \\ \Leftrightarrow & \text{wp}(x = x + 1, \text{wp}(\mathbf{raise}; \text{wp}(x = x - 2, x = 3, x = 1), x = 1), x = 1) \\ \Leftrightarrow & \text{wp}(x = x + 1, x = 1, x = 1) \\ \Leftrightarrow & x + 1 = 1 \\ \Leftrightarrow & x = 0 \end{aligned}$$

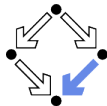


Translation of Java Loops

The guarded command language does not have while loops.

- Translation of `while (e) { c1 } c2`
`loop if ($\neg e$) raise; c1 end ! c2`
- Construct **loop** runs forever.
 - Loop is terminated by signalling an exception in the body.
 - Exception is caught and `c2` is executed.

Replacement of while loops by core **loop** and exceptions.



Translation of Java Conditionals

The guarded command language also does not have conditionals.

- Translation of `if (e) c1 else c2`.
 $(\text{assume } e ; c_1) \square (\text{assume } \neg e ; c_2)$
- Translation of `if (e) c`.
 $(\text{assume } e ; c) \square (\text{assume } \neg e ; \text{skip})$
- 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 $o \neq \text{null}; v = \text{select}(o, f)$

- Possible translation of **check** *expr*.

1. Treat violation as error.

assert *expr*

2. Ignore violation (user has switched warning off).

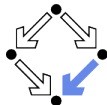
assume *expr*

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 $r(e_0, e_1)$

- Translation (simplified):

```
var  $p_0$   $p_1$  in  
   $p_0 = e_0$ ;  $p_1 = e_1$ ;  
  check precondition (involves  $p_0, p_1$ );  
  var  $x_0$  in  
     $x_0 = x$ ;  
    modify  $x$ ;  
    assume postconditions (involves  $p_0, p_1, x_0, x$ );  
  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|))
  (AND
    (NOT (>= |x| 0))
    (EQ |@true| |@true|)
  )
  (EQ |y| (- 0 |x|))
  ...
)
...
```

Hardly readable, only for understanding details.

Simplify



Simplify(1)

NAME

Simplify -- attempt to prove first-order formulas.

SYNTAX

Simplify [-print] [-ax axfile] [-nosc] [-noprunes]
[-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 } ")"
```

Formula Syntax



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.

Default Axioms



```
(FORALL (a i x k)
  (EQ (select (store a i x) i k) x))
(FORALL (a i n)
  (EQ (len (subMap a i n)) n))
(FORALL (a i n j k)
  (EQ (select (subMap a i n) j k) (select a (+ i j) k)))
(FORALL (a i x)
  (EQ (len (store a i x)) (len a)))
(FORALL (a i n b)
  (EQ (len (storeSub a i n b)) (len a)))
(FORALL (v i)
  (EQ (select (mapFill v) i) v))
(FORALL (i j a x k)
  (OR (EQ i j) (EQ (select (store a i x) j k) (select a j k))))
(FORALL (j i a n b k)
  (OR (AND (OR (< j i) (>= j (+ i n)))
    (EQ (select (storeSub a i n b) j k) (select a j k)))
    (AND (>= j i)
      (< j (+ i n))
      (EQ (select (storeSub a i n b) j k) (select b (- j i) k))))))
```

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:
 - Surveys: Extended Static Checking for Java (2002); ESC/Java2: Uniting ESC/Java and JML (2004).
 - Manual: ESC/Java User Manual (2000), ESC/Java2 Implementation Notes (2004).
 - Guarded Commands: Checking Java Programs via Guarded Commands (1999).
 - Simplify: A Theorem Prover for Program Checking (2003).