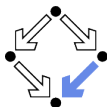


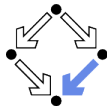
Verifying Java Programs with KeY

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

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



Verifying Java Programs



- **Extended static checking of Java programs:**
 - Even if no error is reported, a program may violate its specification.
 - Unsound calculus for verifying while loops.
 - Even correct programs may trigger error reports:
 - Incomplete calculus for verifying while loops.
 - Incomplete calculus in automatic decision procedure (Simplify).
- **Verification of Java programs:**
 - Sound verification calculus.
 - Not unfolding of loops, but loop reasoning based on invariants.
 - Loop invariants must be typically provided by user.
 - Automatic generation of verification conditions.
 - From JML-annotated Java program, proof obligations are derived.
 - Human-guided proofs of these conditions (using a proof assistant).
 - Simple conditions automatically proved by automatic procedure.

We will now deal with an integrated environment for this purpose.

The KeY Tool

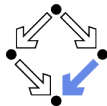


<http://www.key-project.org>

- **KeY:** environment for verification of JavaCard programs.
 - Subset of Java for smartcard applications and embedded systems.
 - Universities of Karlsruhe, Koblenz, Chalmers, 1998–
 - Beckert et al: “Deductive Software Verification – The KeY Book: From Theory to Practice”, Springer, 2016.
 - “Chapter 16: Formal Verification with KeY: A Tutorial”
- **Specification languages:** OCL and JML.
 - Original: OCL (Object Constraint Language), part of UML standard.
 - Later added: JML (Java Modeling Language).
- **Logical framework:** Dynamic Logic (DL).
 - Successor/generalization of Hoare Logic.
 - Integrated prover with interfaces to external decision procedures.
 - Z3, CVC4, CVC5.

Now only JML is supported as a specification language.

Dynamic Logic



Further development of Hoare Logic to a modal logic.

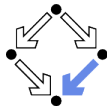
- **Hoare logic:** two separate kinds of statements.
 - Formulas P, Q constraining program states.
 - Hoare triples $\{P\}C\{Q\}$ constraining state transitions.
- **Dynamic logic:** single kind of statement.

Predicate logic formulas extended by two kinds of modalities.

- $[C]Q$ ($\Leftrightarrow \neg\langle C\rangle\neg Q$)
 - Every state that can be reached by the execution of C satisfies Q .
 - The statement is trivially true, if C does not terminate.
- $\langle C\rangle Q$ ($\Leftrightarrow \neg[C]\neg Q$)
 - There exists some state that can be reached by the execution of C and that satisfies Q .
 - The statement is only true, if C terminates.

States and state transitions can be described by DL formulas.

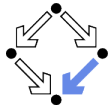
Dynamic Logic versus Hoare Logic



Hoare triple $\{P\}C\{Q\}$ can be expressed as a DL formula.

- **Partial correctness interpretation:** $P \Rightarrow [C]Q$
 - If P holds in the current state and the execution of C reaches another state, then Q holds in that state.
 - Equivalent to the partial correctness interpretation of $\{P\}C\{Q\}$.
- **Total correctness interpretation:** $P \Rightarrow \langle C \rangle Q$
 - If P holds in the current state, then there exists another state that can be reached by the execution of C in which Q holds.
 - If C is deterministic, there exists at most one such state; then equivalent to the total correctness interpretation of $\{P\}C\{Q\}$.

For deterministic programs, the interpretations coincide.

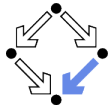


Advantages of Dynamic Logic

Modal formulas can also occur in the context of quantifiers.

- **Hoare Logic:** $\{x = a\} y := x * x \{x = a \wedge y = a^2\}$
 - Use of free mathematical variable a to denote the “old” value of x .
- **Dynamic logic:** $\forall a : x = a \Rightarrow [y := x * x] x = a \wedge y = a^2$
 - Quantifiers can be used to restrict the scopes of mathematical variables across state transitions.

Set of DL formulas is closed under the usual logical operations.



A Calculus for Dynamic Logic

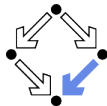
- A core language of commands (non-deterministic):

$X := T$... assignment
 $C_1; C_2$... sequential composition
 $C_1 \cup C_2$... non-deterministic choice
 C^* ... iteration (zero or more times)
 $F?$... test (blocks if F is false)

- A high-level language of commands (deterministic):

skip = true?
abort = false?
 $X := T$
 $C_1; C_2$
if F **then** C_1 **else** C_2 = $(F?; C_1) \cup ((\neg F)?; C_2)$
if F **then** C = $(F?; C) \cup (\neg F)?$
while F **do** C = $(F?; C)^*; (\neg F)?$

A calculus is defined for dynamic logic with the core command language.



A Calculus for Dynamic Logic

■ Basic rules:

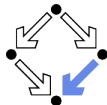
- Rules for predicate logic extended by general rules for modalities.

■ Command-related rules:

- $$\frac{\Gamma \vdash F[T/X]}{\Gamma \vdash [X := T]F}$$
- $$\frac{\Gamma \vdash [C_1][C_2]F}{\Gamma \vdash [C_1; C_2]F}$$
- $$\frac{\Gamma \vdash [C_1]F \quad \Gamma \vdash [C_2]F}{\Gamma \vdash [C_1 \cup C_2]F}$$
- $$\frac{\Gamma \vdash F \Rightarrow [C]F}{\Gamma \vdash F \Rightarrow [C^*]F}$$
- $$\frac{\Gamma \vdash F \Rightarrow G}{\Gamma \vdash [F?]G}$$

From these, Hoare-like rules for the high-level language can be derived.

Objects and Updates



Calculus has to deal with the pointer semantics of Java objects.

- **Aliasing:** two variables o, o' may refer to the same object.
 - Field assignment $o.a := T$ may also affect the value of $o'.a$.
- **Update formulas:** $\{o.a \leftarrow T\}F$
 - Truth value of F in state after the assignment $o.a := T$.

- **Field assignment rule:**

$$\frac{\Gamma \vdash \{o.a \leftarrow T\}F}{\Gamma \vdash [o.a := T]F}$$

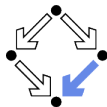
- **Field access rule:**

$$\frac{\Gamma, o = o' \vdash F(T) \quad \Gamma, o \neq o' \vdash F(o'.a)}{\Gamma \vdash \{o.a \leftarrow T\}F(o'.a)}$$

- Case distinction depending on whether o and o' refer to same object.
- Only applied as last resort (after all other rules of the calculus).

Considerable complication of verifications.

The KeY Prover



> KeY &

The KeY Project

2.12

© Copyright 2001-2023 Karlsruhe Institute of Technology, Chalmers University of Technology, and Technische Universität Darmstadt.

WWW: <http://key-project.org/>

Version 2.12.0 (internal: 639802ce88962694ec35fa6e9573fceb3cf280)

```
int sum;
int max;

/*@ normal_behaviour
  @ requires (\forallall int i; 0 <= i && i < a.length;
  @ assignable sum, max;
  @ ensures (\forallall int i; 0 <= i && i < a.length; a
  @ ensures a.length > 0
  ==> (\exists int i; 0 <= i && i < a.length
  @ ensures sum == (\sum int i; 0 <= i && i < a.length
  @ ensures sum <= a.length * max;
  @*/

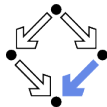
void sumAndMax(int[] a) {
  sum = 0;
  max = 0;
  int k = 0;

  /*@ loop_invariant
  @ 0 <= k && k <= a.length
  @ && (\forallall int i; 0 <= i && i < k; a[i] <= max
  @ && (k == 0 ==> max == 0)
  @ && (k > 0 ==> (\exists int i; 0 <= i && i < k
  @ && sum == (\sum int i; 0 <= i && i < k; a[i]))
  @ && sum <= k * max;
  @
  @ assignable sum, max;
  @ decreases a.length - k;
  @*/

  while(k < a.length) {
    if(max < a[k]) {
      max = a[k];
    }
    sum += a[k];
    k++;
  }
}
```

Strategy: Applied 2730 rules (2.2 sec), closed 50 goals, 0 remaining

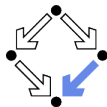
A Simple Example



File/Load Example/Getting Started/Sum and Max

```
class SumAndMax {
  int sum; int max;
  /*@ requires (\forall int i;
    @ 0 <= i && i < a.length; 0 <= a[i]);
    @ assignable sum, max;
    @ ensures (\forall int i;
    @ 0 <= i && i < a.length; a[i] <= max);
    @ ensures (a.length > 0 ==>
    @ (\exists int i;
    @ 0 <= i && i < a.length;
    @ max == a[i]));
    @ ensures sum == (\sum int i;
    @ 0 <= i && i < a.length; a[i]);
    @ ensures sum <= a.length * max;
  */
  void sumAndMax(int[] a) {
    sum = 0;
    max = 0;
    int k = 0;
    /*@ loop_invariant
      @ 0 <= k && k <= a.length
      @ && (\forall int i;
      @ 0 <= i && i < k; a[i] <= max)
      @ && (k == 0 ==> max == 0)
      @ && (k > 0 ==> (\exists int i;
      @ 0 <= i && i < k; max == a[i]))
      @ && sum == (\sum int i;
      @ 0 <= i && i < k; a[i])
      @ && sum <= k * max;
      @ assignable sum, max;
      @ decreases a.length - k;
    */
    while (k < a.length) {
      if (max < a[k]) max = a[k];
      sum += a[k];
      k++;
    } } }
}
```

A Simple Example (Contd)



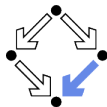
The screenshot shows a window titled "Proof Management" with two tabs: "By Target" and "By Proof". The "By Target" tab is active. On the left, under "Contract Targets", there is a tree view with "SumAndMax" and "sumAndMax(int[])" selected. The main area, titled "Contracts", contains the following JML code:

```
JML normal_behavior operation contract 0
self.sumAndMax(a) catch(exc)
pre  $\forall$  int i; (0  $\leq$  i  $\wedge$  i < a.length  $\wedge$  inInt(i)  $\rightarrow$  0  $\leq$  a[i])  $\wedge$  (self.<inv>  $\wedge$   $\neg$ a = null)
post  $\forall$  int i; (0  $\leq$  i  $\wedge$  i < a.length  $\wedge$  inInt(i)  $\rightarrow$  a[i]  $\leq$  self.max)  $\wedge$  { (a.length > 0  $\rightarrow$   $\exists$  int i; (0  $\leq$  i  $\wedge$  i < a.length  $\wedge$  inInt(i)  $\wedge$  self.max = a[i])  $\wedge$  (self.sum = bsum(int i);)
mod {(self, SumAndMax::$sum)}  $\cup$  {(self, SumAndMax::$max)}
termination diamond
```

At the bottom right, there are two buttons: "Start Proof" and "Cancel".

Generate the proof obligations and choose one for verification.

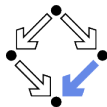
A Simple Example (Contd'2)



The screenshot displays the KeY 2.12.0 IDE interface. The main window shows the source code for the `SunAndMax` class, which includes a `sumAndMax` method with several annotations: `requires`, `assignable`, `ensures`, `loop_invariant`, and `decreases`. The `Sequent` window on the left shows the proof obligations generated for the `sumAndMax` method, including the `wellformed` condition and the `try` block. A tooltip is visible over the `forall` quantifier in the proof obligation, showing the origin of the term and the operator hash.

```
wellformed(heap)
A self = null
A self.<created> = TRUE
A SunAndMax::exactInstance(self) = TRUE
A ((o = null & a.<created> = TRUE)+SC+)
A measuredByEmpty
A (( V int i;
  { (0 ≤ i & i < a.length)+SC+ A inInt(i) - 0 ≤ a[i]
  A ((self.<inv>+impl)+SC+ ((a = null)+impl)+SC+)+SC+
- (heapAtPre->heap [i]_a=a)
  \<{
    exc = null;
    try {
      self.sumAndMax(_a)@SunAndMax;
    } catch (java.Lang.Throwable e) {
      exc = e;
    } \> { V int i;
      /- @k k <= a.length+SC+ A inInt(i)
      Operator Hash: 1358657652
      A (( self.sum
        = bsum(int i);(0, a.length, a[i])
        A (( self.sum < a.length * self.max
          A self.<inv>+impl)+SC+)+SC+)+SC+)+SC+
      A (exc = null)+impl+
      A V Field f;
      V java.Lang.Object o;
      ( (o, f) * ((self, SunAndMax::sum)
        U ((self, SunAndMax::max)
          V m0 = null
          A m0.<created>@heapAtPre = TRUE
          V o, f = o. @heapAtPre])
```

The proof obligation in Dynamic Logic.

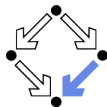


A Simple Example (Contd'3)

```
==>
  wellFormed(heap)
  & ...
  & (( \forall int i;
      ((0 <= i & i < a.length) & inInt(i) -> 0 <= a[i])
      & ((self_25.<inv> & (!a = null))))))
-> {heapAtPre_0:=heap || _a:=a}
  \<{
    exc_25=null;try {
      self_25.sumAndMax(_a)@SumAndMax;
    } catch (java.lang.Throwable e) { exc_25=e; }
  }> ( (\forall int i;
      ( (0 <= i & i < a.length) & inInt(i) -> a[i] <= self_25.max)
      & (( ( a.length > 0
          -> \exists int i;
              (( (0 <= i & i < a.length) & inInt(i) & self_25.max = a[i])))
          & (( self_25.sum = javaCastInt(bsum{int i;}(0, a.length, a[i]))
              & (( self_25.sum <= javaMulInt(a.length, self_25.max)
                  & self_25.<inv>))))))))
      & (exc_25 = null)
      & \forall Field f;
        \forall java.lang.Object o;
          ( (o, f) \in {(self_25, SumAndMax::$sum)}
            \cup {(self_25, SumAndMax::$max)}
            | !o = null
            & !o.<created>@heapAtPre_0 = TRUE
            | o.f = o.f@heapAtPre_0))
```

Press button “Start/stop automated proof search” (green arrow).

A Simple Example (Contd'4)



KeY 2.12.0

File View Proof Options Origin Tracking Proof Management

Run CVCS

Loaded Proofs

Proofs

Env. with model src@5:44:36 PM

SumAndMax(SumAndMax::sumAndMax(JML,normal_behaviour.opera

Goals Proof Proof Slicing Proof Search Strategy

Proof

Proof Tree

- Invariant Initially Valid
- Body Preserves Invariant
- Use Case

Inner Node

wellFormed

```
A ~self f = nu
A self <-creat
A SumAndMax::
A ((a = null)
A ensuredBy
A (( ~ int
A ((self
~ (heapAtPre
  ~ (
    exc = nu
    try {
      self.s
    } catch
    exc = <
  }) ( ~
  A ((
    A ((
      A (exc = null)wimpl
      A ~ Field f;
      ~ java.lang.Object o;
      ( o, f) * ((self, SumAndMax::$sum)
        U ((self, SumAndMax::$max))
      ~ ~ = null
      A ~ ~-created@heapAtPre = TRUE
```

Proof Statistics

Proved.

Nodes	2,780
Branches	50
Interactive steps	0
Symbolic execution steps	219
Automode time	3032ms
Avg. time per step	1,091ms

Rule applications

Quantifier instantiations	12
One-step Simplifier apps	360
SMT solver apps	0
Dependency Contract apps	0
Operation Contract apps	0
Block/Loop Contract apps	0
Loop invariant apps	1
Merge Rule apps	0
Total rule apps	4,842

Close Export as CSV Export as HTML

Save proof Save proof bundle

Source

```
SumAndMax.java
5  /**
6  */
7  /* normal_behaviour
8  @ requires (\forallall int i; 0 <= i && i < a.length;
9  @ assignable sus, max;
10 @ ensures (\forallall int i; 0 <= i && i < a.length; a
11 @ ensures [a.length > 0
12 ==> (\exists int i; 0 <= i && i < a.length
13 @ ensures sum = (\sum int i; 0 <= i && i < a.length
14 @ ensures sum <= a.length * max;
15 */
16 void sumAndMax(int[] a) {
17     sum = 0;
18     max = 0;
19     int k = 0;
20
21     /* loop_invariant
22     @ 0 = k && k <= a.length
23     @ k (\forallall int i; 0 <= i && i < k; a[i] <= m
24     @ k > 0 ==> max = 0)
25     @ k (\exists int i; 0 <= i && i < k
26     @ k sum = (\sum int i; 0 <= i && i < k; a[i])
27     @ k sum <= k * max;
28     @
29     @ assignable sus, max;
30     @ decreases a.length - k;
31     */
32     while(k < a.length) {
33         if(max < a[k]) {
34             max = a[k];
35         }
36         sum += a[k];
37         k++;
38     }
39 }
40
```

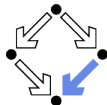
Show Postcondition/Assignable

Strategy: Applied 2730 rules (2.0 sec), closed 59 goals, 0 remaining

Java 0! Proof Caching

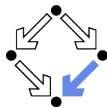
The proof runs through automatically.

Linear Search



```
/*@ requires a != null;
   @ assignable \nothing;
   @ ensures
   @   (\result == -1 &&
   @     (\forall int j; 0 <= j && j < a.length; a[j] != x)) ||
   @   (0 <= \result && \result < a.length && a[\result] == x &&
   @     (\forall int j; 0 <= j && j < \result; a[j] != x));
   @*/
public static int search(int[] a, int x) {
    int n = a.length; int i = 0; int r = -1;
    /*@ loop_invariant
       @   a != null && n == a.length && 0 <= i && i <= n &&
       @   (\forall int j; 0 <= j && j < i; a[j] != x) &&
       @   (r == -1 || (r == i && i < n && a[r] == x));
       @ decreases r == -1 ? n-i : 0;
       @ assignable r, i; // required by KeY, not legal JML
       @*/
    while (r == -1 && i < n) {
        if (a[i] == x) r = i; else i = i+1;
    }
    return r;
}
```


Linear Search (Contd)



KeY 2.12.0

File View Proof Options Origin Tracking Proof Management

Run CVS

Loaded Proofs

Proofs

- with model [research@95:48:16 PM
- research.Main@research.Main:search([],int[],JML operation contract.0
- with model [research@95:48:59 PM
- research.Main@research.Main:search([],int[],JML operation contract.0
- with model [research@95:49:28 PM
- research.Main@research.Main:search([],int[],JML operation contract.0

Goals Proof Proof Slicing Proof Search Strategy

Proof

- Normal Execution (a != null)
- Invariant Initially Valid
- Body Preserves Invariant
- Use Case
- Null Reference (a == null)

Proof Statistics

Wellformed

measuredBy

exc = null

try {

result

exc = null

catch {

exc = null

}

Proof Statistics

Proved.

Nodes	843
Branches	15
Interactive steps	0
Symbolic execution steps	110
Automode time	1197ms
Avg. time per step	1.421ms

Rule applications

Quantifier instantiations	1
One-step Simplifier apps	129
SMT solver apps	0
Dependency Contract apps	0
Operation Contract apps	0
Block/Loop Contract apps	0
Loop invariant apps	1
Merge Rule apps	0
Total rule apps	2,062

Close Export as CSV Export as HTML

Save proof Save proof bundle

Source

```
1 package linearsch;
2
3 public class Main0
4 {
5     /* requires a != null;
6      * assignable \nothing;
7      * ensures
8      *   (\result == -1 &&
9      *   (\forallall int j; 0 <= j && j < a.length; a[j] != x))
10     *   (\forallall int i; 0 <= i && i < a.length && a[i] == x);
11     */
12     public static int search(int[] a, int x)
13     {
14         int n = a.length;
15         int i = 0;
16         int r = -1;
17         /* Loop invariant
18          *   a != null && n == a.length &&
19          *   0 <= i && i <= n &&
20          *   (\forallall int j; 0 <= j && j < i; a[j] != x) &&
21          *   (x == -1 || (x == i && i < n && a[i] == x));
22          * decreases r := -1 ? n-1 : 0;
23          * assignable r,i;
24          */
25         while (x == -1 && i < n)
26         {
27             if (a[i] == x)
28                 r = i;
29             else
30                 i = i+1;
31         }
32         return r;
33     }
34 }
35 }
36
```

Strategy: Applied 828 rules (1.1 sec), closed 15 goals, 0 remaining

Java 00 Proof Caching

Also this verification is completed automatically.

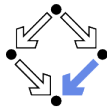
Proof Structure



- Multiple conditions (Tactlet option “javaLoopTreatment::teaching”):
 - Invariant Initially Valid.
 - Body Preserves Invariant.
 - Use Case (on loop exit, invariant implies postcondition).
- If proof fails, elaborate which part causes trouble and potentially correct program, specification, loop annotations.

For a successful proof, in general multiple iterations of automatic proof search (button “Start”) and invocation of separate SMT solvers required (button “Run CVC5”).

Summary



- Various academic approaches to verifying Java(Card) programs.
 - Jack: <http://www-sop.inria.fr/everest/soft/Jack/jack.html>
 - Jive: <http://www.pm.inf.ethz.ch/research/jive>
 - Mobius: <http://kindsoftware.com/products/opensource/Mobius/>
- Do not yet scale to verification of full Java applications.
 - General language/program model is too complex.
 - Simplifying assumptions about program may be made.
 - Possibly only special properties may be verified.
- Nevertheless very helpful for reasoning on Java in the small.
 - Much beyond Hoare calculus on programs in toy languages.
 - Probably all examples in this course can be solved automatically by the use of the KeY prover and its integrated SMT solvers.
- Enforce clearer understanding of language features.
 - Perhaps constructs with complex reasoning are not a good idea. . .

In a not too distant future, customers might demand that some critical code is shipped with formal certificates (correctness proofs) . . .