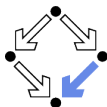


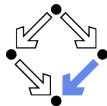
Verifying Java Programs with KeY

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

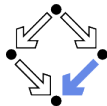


<http://www.key-project.org>

- **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.
 - Simplify, CVC3, CVC4, Yices, Z3.

Now only JML is supported as a specification language.

Dynamic Logic



Further development of Hoare Logic to a modal logic.

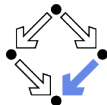
- **Hoare logic:** two separate kinds of statements.
 - Formulas P, Q constraining program states.
 - Hoare triples $\{P\}C\{Q\}$ constraining state transitions.
- **Dynamic logic:** single kind of statement.

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.

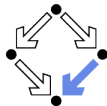
Dynamic Logic versus Hoare Logic



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.

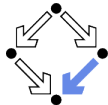


Advantages of Dynamic Logic

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:**

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

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 JMLKeY Prover



> KeY &

KeY 2.6.2 [HEAD]

File View Proof Options About

Run Z3

Proof Goals Proof Search Strategy Info

Proof Tree

- Invariant Initially Valid
- Body Preserves Invariant
- Use Case

Inner Node

```
==>
wellFormed(heap)
& !self = null
& self.<created> = TRUE
```

The KeY Project

© Copyright 2001-2016 Karlsruhe Institute of Technology, Chalmers University of Technology, and Technische Universität Darmstadt

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

Version 2.6.2 (internal: 00c1abfd22b738afe24e89fccc2ee4eec2c38f4a)

OK

```
}> ( \forallall int i;
(0 <= i & i < a.length & inInt(i) -> a[i] <= self.max)
& ( a.length > 0
-> \exists int i;
( 0 <= i
& i < a.length
& inInt(i)
& self.max = a[i]))
& ( self.sum
```

Show tactic info (Inner Nodes only)

Strategy: Applied 2430 rules (6.5 sec), closed 37 goals, 0 remaining

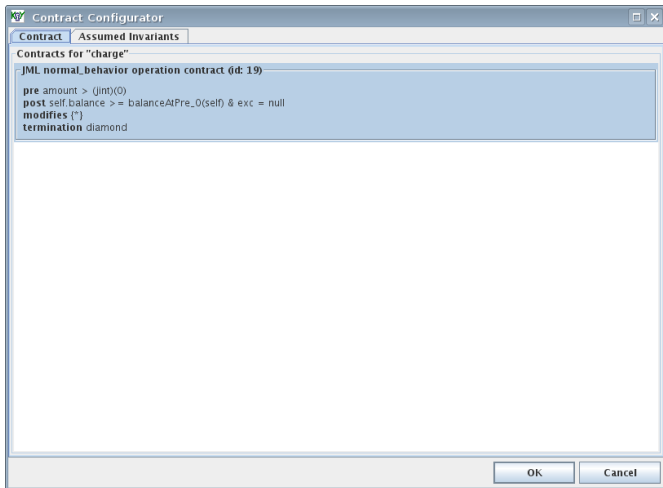
A Simple Example



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)



Generate the proof obligations and choose one for verification.

A Simple Example (Contd'2)



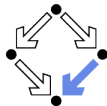
The screenshot shows the KeY 2.0.0 IDE interface. The main window is titled "KeY 2.0.0" and "About". The menu bar includes "File", "View", "Proof", and "Options". The toolbar contains icons for "Run Z3", "Back", "Forward", "Home", "End", "Print", and "Proof Management".

The left sidebar is divided into "Proofs" and "Proof Search Strategy". Under "Proofs", it shows "Env. with model paycard@1:40:51 PM #1" and a goal: `paycard.PayCard|paycard.PayCard::charge(int)`. Under "Proof Search Strategy", there are tabs for "Proof" and "Goals". The "Proof" tab shows a "Proof Tree" with a root node labeled "1: OPEN GOAL".

The main editor area is titled "Current Goal" and contains the following Dynamic Logic formula:

```
wellFormed(heap)
& !self = null
& self.<created> = TRUE
& paycard.PayCard::exactInstance(self) = TRUE
& inInt(amount)
& ( amount > 0
  & ( javaAddInt(amount, self.balance) < self.limit
    & self.isValid() = TRUE
    & self.<inv>))
-> {heapAtPre:=heap || _amount:=amount}
\<{
  exc=null;try {result=self.charge(_amount)(paycard.PayCard);
}catch {java.lang.Exception e} {
  exc=e;
}
}\>
& result = TRUE
& amount = amount
& ( self.balance
  = javaAddInt(amount,
    int::select(heapAtPre, self, balance))
  & ( self.unsuccessfulOperations
    = int::select(heapAtPre,
      self,
      unsuccessfulOperations)
    & self.<inv>))
& exc = null
& \forall Field f;
  \forall java.lang.Object o;
  ( (o, f) \in {(self, balance)}
    \cup {(self, unsuccessfulOperations)}
    | !o = null
    & !boolean::select(heapAtPre, o, <created>) = TRUE
    | o.f = any::select(heapAtPre, o, f))
```

The proof obligation in Dynamic Logic.



A Simple Example (Contd'3)

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

Press button "Start" (green arrow).

A Simple Example (Contd'4)

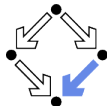


The screenshot displays the KeY 2.0.0 interface. The main window is titled "KeY 2.0.0" and contains several panes:

- File View Proof Options**: A menu bar at the top.
- Run Z3**: A button for running the Z3 solver.
- Proof Management**: A toolbar with icons for proof operations.
- Proofs**: A pane on the left showing the current proof environment: "Env. with model paycard@1:57:53 PM #1" and "(paycard.PayCard[paycard.PayCard:charge(int)])".
- Proof Search Strategy**: A pane below the Proofs pane, showing a "Proof Tree" with a list of rules and goals. The tree includes rules like "1:One Step Simplification: 4 rules" and "12:One Step Simplification: 2 rules", along with various logical and arithmetic rules.
- Inner Node**: A large pane on the right showing the current state of the inner node. It contains a list of conditions in a goal language, such as `wellFormed(heap)`, `& !self = null`, `& self.<created> = TRUE`, `& paycard.PayCard::exactInstance(self) = TRUE`, `& inInt(amount)`, `& (amount > 0`, `& (javaAddInt(amount, self.balance)`, `< self.limit`, `& self.isValid() = TRUE`, and `& self.<inv>)`. Below these conditions is the goal: `-> {heapAtPre:=heap || _amount:=amount}, self.charge(_amount)@paycard.PayCard; option e) ;`.
- Proof closed**: A dialog box in the foreground reporting the proof's completion. It states: "Proved. Statistics: Nodes:373, Branches: 10".
- Status Bar**: At the bottom, it shows "Strategy: Applied 363 rules (2.7 sec), closed 10 goals, 0 remaining".

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 2.0.0

File View Proof Options About

Run Z3

Proof Management

Proofs

Env. with model_@2:24:32 PM #1

lineSearch.Main0[lineSearch.Main0::search([i,int])]

Proof Search Strategy Rules Goals

Proof Tree

- Normal Execution (_a != null)
- Null Reference (_a = null)

Inner Node

```
wellFormed(heap)
& ((a.<created> = TRUE | a = null) & inInt(x))
& (!a = null & !a = null)
-> {heapAtPre:=heap | _a:=a | !_x:=x}
|<
exc=null;try {result=lineSearch.Main0.search(_a,_x)@lineSearch.Main0
}catch (java.lang.Exception e) {
exc=e;
}|> | ( exc = null
javaUnaryMinusInt(i)
i:=j;
j<=j
j<=a.length
iInt(j)
j]=x)
a.length
j]=x
forall int j;
{
j<=j
& j<result
& inInt(j)
-> !a[j]=x)
& exc = null
& forall Field f;
forall java.lang.Object o;
{
o = null
& ! boolean::select(heapAtPre,
o,
<created>)
= TRUE
| o.f
= any::select(heapAtPre, o, f)}
```

Proof closed

Proved.

Statistics:

Nodes: 785

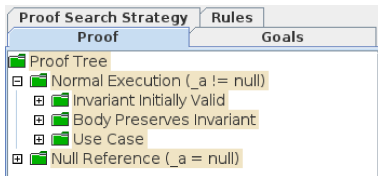
Branches: 11

OK

Strategy: Applied 774 rules (3.5 sec), closed 11 goals, 0 remaining

Also this verification is completed automatically.

Proof Structure



- Multiple conditions:
 - Invariant initially valid.
 - Body preserves invariant.
 - Use case (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 Z3, Yices, CVC3, Simplify”).

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) . . .