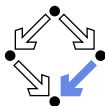


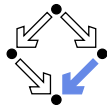
# Verifying Java Programs with KeY

Wolfgang Schreiner  
Wolfgang.Schreiner@risc.jku.at

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



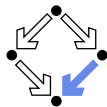
# Verifying Java Programs



- **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

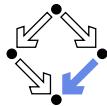


<http://www.key-project.org>

- **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: “Verification of Object-Oriented Software: The KeY Approach”, Springer, 2007. (book)
    - Ahrendt et al: “The KeY Tool”, 2005. (paper)
    - Engel and Roth: “KeY Quicktour for JML”, 2006. (short paper)
- **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, Yices, Z3.

We will only deal with the tool's JML interface “JMLKeY”.

# Dynamic Logic



Further development of Hoare Logic to a modal logic.

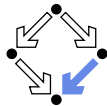
- **Hoare logic:** two separate kinds of statements.
  - Formulas  $P, Q$  constraining program states.
  - Hoare triples  $\{P\}C\{Q\}$  constraining state transitions.
- **Dynamic logic:** single kind of statement.

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.

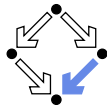
# Dynamic Logic versus Hoare Logic



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.



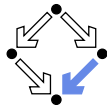
# Advantages of Dynamic Logic

---

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

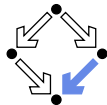
## ■ A core language of commands (non-deterministic):

$X := T$  ... 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.

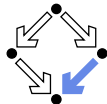
## Command-related rules:

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

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:**

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

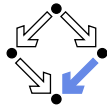
- **Field access rule:**

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

- 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



/zvol/formal/bin/startProver &

The screenshot displays the KeY 2.0.0 software interface. The main window shows a proof tree on the left and a code editor on the right. The code editor contains the following Java code snippet:

```
wellFormed(heap)
& (a.<created> = TRUE | a = null) & inInt(x)
& (!a = null & !a = null)
-> {heapAtPre:=heap || _a:=a || _x:=x}
|<
exc=null;try {result=Linsearch.Main.search(_a,_x)@Linsearch.Main;
}catch (java.lang.Exception e) {
exc=e;
```

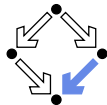
A dialog box titled "The KeY Project" is overlaid on the interface. It contains the following text:

(C) Copyright 2001-2013 Karlsruhe Institute of Technology, Chalmers University of Technology, and Technische Universität Darmstadt  
WWW: <http://key-project.org>  
Version 2.0.0 (Internal: 93bb62e2794d260c5a2f08c5dfdf44df85d638e)

An "OK" button is visible at the bottom of the dialog box.

At the bottom of the KeY window, a status bar indicates: Strategy: Applied 1122 rules (6.6 sec), closed 14 goals, 0 remaining

# A Simple Example



Engel et al: “KeY Quicktour for JML”, 2005.

```
package paycard;

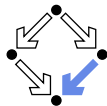
public class PayCard {
    /*@ public invariant log.\inv;
       @ public invariant balance >= 0;
       @ public invariant limit > 0;
       @ public invariant unsucc >= 0;
       @ public invariant log != null;
    */

    /*@ spec_public @*/ int limit=1000;
    /*@ spec_public @*/ int unsucc;
    /*@ spec_public @*/ int id;
    /*@ spec_public @*/ int balance=0;
    /*@ spec_public @*/
        protected LogFile log;
    ...

    /*@ public normal_behavior
       @ requires amount>0;
       @ requires amount+balance<limit && isValid()
       @ ensures \result == true;
       @ ensures balance == amount+\old(balance);
       @ ensures unsucc == \old(unsucc);
       @ assignable balance, unsucc;
       @ also ...
    */

    public boolean charge(int amount)
        throws IllegalArgumentException {
        if (amount <= 0)
            throw new IllegalArgumentException();
        if (balance+amount<limit && isValid()) {
            balance=balance+amount;
            return true;
        }
        ...
    }
}
```

# A Simple Example (Contd)



Proof Management

By Target | By Proof

Contract Targets

- paycard
  - CardException
    - getCause()
    - getMessage()
    - initCause(java.lang.Throwable)
  - LogFile
  - LogRecord
  - PayCard
    - PayCard()
    - PayCard(int)
    - charge(int)
    - chargeAndRecord(int)
    - createJuniorCard()
    - isValid()

Contracts

JML exceptional\_behavior operation contract 0

```
result = self.charge(amount) catch(exc)
pre amount <= 0 & self.<inv>
post !exc = null & ( ( java.lang.Exception::instance(exc) = TRUE -> self.<inv> ) & java.lang.IllegalArgumentException)
mod {}
termination diamond
```

JML normal\_behavior operation contract 0

```
result = self.charge(amount) catch(exc)
pre amount > 0 & ( javaAddInt(amount, self.balance) < self.limit & self.isValid() = TRUE & self.<inv> )
post result = TRUE & amount = amount & ( self.balance = javaAddInt(amount, int)::select(heapAtPre, self, b
mod {(self, balance)} \cup {(self, unsuccessfulOperations)}
termination diamond
```

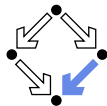
JML normal\_behavior operation contract 1

```
result = self.charge(amount) catch(exc)
pre amount > 0 & ( ( javaAddInt(amount, self.balance) >= self.limit | !self.isValid() = TRUE ) & self.<inv> )
post !result = TRUE & amount = amount & ( self.unsuccessfulOperations = javaAddInt(int)::select(heapAtPre
mod {(self, balance)} \cup {(self, unsuccessfulOperations)}
termination diamond
```

Start Proof Cancel

Generate the proof obligations and choose one for verification.

# A Simple Example (Contd'2)

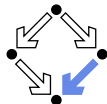


The screenshot displays the KeY 2.0.0 IDE interface. On the left, the 'Proofs' pane shows the environment 'Env. with model paycard@1:40:51 PM #1' and a goal 'paycard.PayCard|paycard.PayCard::charge(int)'. Below it, the 'Proof Search Strategy' and 'Rules' panes are visible, with a 'Proof Tree' containing a single node '1: OPEN GOAL'. The main editor shows the 'Current Goal' as a 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
    & \forallall Field f;
      \forallall 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))
  }
```

At the bottom of the IDE, a status bar reads: 'KeY Integrated Deductive Software Design: Ready (Hint: type F3 to search in proof trees or sequents.)'

The proof obligation in Dynamic Logic.

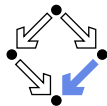


## A Simple Example (Contd'3)

```
==>
  wellFormed(heap)
& !self = null & ...
& ( 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.unsucc
      = int::select(heapAtPre,
                    self,
                    unsucc)
    & self.<inv>))
    & exc = null & ...
```

Press button "Start" (green arrow).

# A Simple Example (Contd'4)



File View Proof Options About

Run Z3

Proof Management

Proofs

Env. with model paycard@1:57:53 PM #1

(paycard.PayCard@paycard.PayCard:charge(int))

Proof Search Strategy Rules Goals

Proof Tree

- 1: One Step Simplification: 4 rules
- 2: impRight
- 3: andLeft
- 4: andLeft
- 5: andLeft
- 6: andLeft
- 7: andLeft
- 8: andLeft
- 9: andLeft
- 10: notLeft
- 11: exc=null
- 12: One Step Simplification: 2 rules
- 13: translateJavaAdd
- 14: eqSymm
- 15: commuteUnion
- 16: eqSymm
- 17: eqSymm
- 18: translateJavaAdd
- 19: polySimp\_homoEq
- 20: polySimp\_mulComm0
- 21: inEqSimp\_gtToGeq
- 22: times\_zero
- 23: add\_zero\_right
- 24: inEqSimp\_ltToLeq
- 25: polySimp\_mulComm0
- 26: polySimp\_addComm1

Inner Node

```
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,
```

Proof closed

Proved.

Statistics:

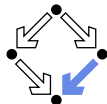
Nodes:373

Branches: 10

OK

Strategy: Applied 363 rules (2.7 sec), closed 10 goals, 0 remaining

Proof runs through automatically.



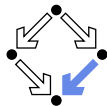
# A Loop Example

```
/*@ 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;
}
```



# A Loop Example (Contd)



KeY 2.0.0

File View Proof Options About

Run Z3

Proof Management

Proofs

Env. with model\_@2:24:32 PM #1

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

Proof Search Strategy Rules Goals

Proof Tree

- Normal Execution ( $a \neq \text{null}$ )
- Null Reference ( $a = \text{null}$ )

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=linearch.Main0.search(_a,_x)@linearch.Main0
}catch (java.lang.Exception e) {
exc=e;
}
}|> { ( exc = null
javaUnaryMinusInt(i)
i--;
j--;
<= j
< a.length
inInt(j)
j] = x)
a.length
j] = x
forall int j;
( 0 <= 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

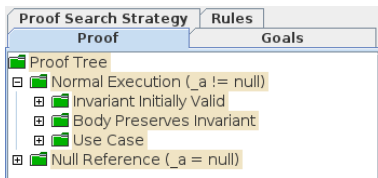
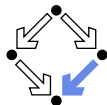
Branches: 11

OK

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

Also this verification is completed automatically.

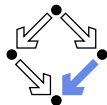
# Proof Structure



- 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)...