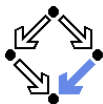
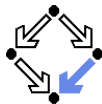


Verifying Java Programs with KeY

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

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



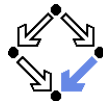


Verifying Java Programs

- **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

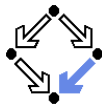


<http://www.key-project.org>

- **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, ICS, CVC3, CVCLite, Yices, ...

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

Dynamic Logic



Further development of Hoare Logic to a modal logic.

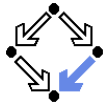
- **Hoare logic:** two separate kinds of statements.
 - Formulas P, Q constraining program states.
 - Hoare triples $\{P\}C\{Q\}$ constraining state transitions.
- **Dynamic logic:** single kind of statement.

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.

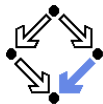
Dynamic Logic versus Hoare Logic



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:

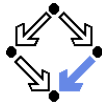
- Rules for predicate logic extended by general rules for modalities.

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:**

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

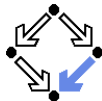
- **Field access rule:**

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

- 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 &

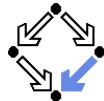
The screenshot displays the KeY Prover interface. The main window is titled "KeY -- Prover" and contains several panes:

- Tasks:** Shows the current model and goals, such as "with model paycard@17:12:06 #1" and "Ensures Post Condition PO (using only invariants from Pa)".
- Proof Search Strategy:** Includes tabs for "Proof", "Goals", and "User Constraint".
- Proof Tree:** A hierarchical tree of proof nodes. The root is "Normal Execution (logArray != null)". It branches into "Invariant Initially Valid", "Body Preserves Invariant", and "Normal Execution (logArray != null)". The latter further branches into "Normal Execution (logArray != null)", "Normal Execution (var != null)", "Normal Execution (lr != null)", "Normal Execution (max)", and "self_LogFile_lv_0.k". The "self_LogFile_lv_0.k" node is highlighted in blue and labeled "B91: OPEN GOAL". Below it are nodes for "self_LogFile_lv_0.k", "Null Reference (max =)", "Null Reference (lr = null)", "Null Reference (var = null)", "Index Out of Bounds (var !=)", "l_0 <= 2 FALSE", "Null Reference (logArray = null)", "Null Reference (logArray = null)", "Use Case", "Termination", "Null Reference (logArray = null)", and "Index Out of Bounds (logArray != null, but)".
- Current Goal:** A list of logical assertions in JML-like syntax, including:

```
self_LogFile_lv_0.logArray[i_0].balance >= 1 + max_0.balance,  
self_LogFile_lv_0.logArray[i_0].<created> = TRUE,  
i_0 <= 2,  
{i:=i_0 || max:=max_0}anon(i, max),  
i_0 >= 0,  
\forallall jint j;  
( j <= -1  
 | j >= i_0  
 | self_LogFile_lv_0.logArray[j].balance <= max_0.balance),  
self_LogFile_lv_0.<created> = TRUE,  
self_LogFile_lv_0.logArray.<created> = TRUE,  
self_LogFile_lv_0.logArray.length = 3,  
self_LogFile_lv_0.currentRecord <= 2,  
self_LogFile_lv_0.currentRecord >= 0,  
\forallall int index_lv_0;  
( index_lv_0 <= -1  
 | index_lv_0 >= 3  
 | !self_LogFile_lv_0.logArray[index_lv_0] = null),  
inReachableState  
==>  
max_0 = null,  
self_LogFile_lv_0.logArray = null,  
self_LogFile_lv_0 = null,  
self_LogFile_lv_0.logArray[i_0] = null,  
{i:=i_0 || max:=max_0}anon(i, max)  
& ( (jint)(1 + i_0) >= 0  
 & ( self_LogFile_lv_0.logArray.length >= (jint)(1 + i_0)  
 & ( !self_LogFile_lv_0.logArray[i_0] = null  
 & \forallall jint j;  
 cut_direct  
 local_cut  
 Apply rules automatically here  
 to clipboard  
 Create abbreviation  
 gArray[j].balance  
 gArray[i_0].balance))
```

The status bar at the bottom indicates "Integrated Deductive Software Design: Ready".

A Simple Example



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

```
package paycard;

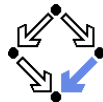
public class PayCard {

    /*@ public instance invariant
       @   log != null
       @   && balance >=0
       @   && limit >0
       @   && unsuccessfulOperations >=0;
    */

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

    /*@
       @ public normal_behavior
       @ requires amount>0 ;
       @ assignable
       @   unsuccessfulOperations, balance;
       @ ensures balance >= \old(balance);
    */
    public boolean charge(int amount) {
        if (this.balance+amount>=this.limit) {
            this.unsuccessfulOperations++;
            return false;
        } else {
            this.balance=this.balance+amount;
            return true;
        }
    }
    ...
}
```

A Simple Example (Contd)



The screenshot shows the JML Specification Browser window. On the left, a tree view shows the class hierarchy: java.lang.Exception, java.lang.Object, java.lang.Throwable, org.jmlspecs, and org.jmlspecs.paycard. The PayCard class is selected. The middle pane shows the methods for PayCard, with the charge method selected. The right pane shows the proof obligations for the charge method, including a normal_behavior specification and a class specification for PayCard. At the bottom, there are checkboxes for 'Use all applicable invariants' and 'Add invariants to postcondition', and buttons for 'Load Proof Obligation' and 'Cancel'.

JML Specification Browser

Classes

- java
 - lang
 - Exception
 - Object
 - Throwable
 - org
 - jmlspecs
 - paycard
 - CardException
 - ChargeUI
 - IssueCardUI
 - LimitedIntContainer
 - LogFile
 - LogRecord
 - PayCard
 - PayCardJunior
 - Start

Show Inherited Methods

Methods

- void <clinit>()
- void PayCard(int limit)
- void PayCard()
- int available()
- void charge(int amount)**
- String infoCardMsg()

Proof Obligations

- normal_behavior speccase for method charge
- in context PayCard
- requires: !self.PayCard = null
- & self.PayCard.<created> = TRUE
- & amount > 0
- Class specification for class PayCard

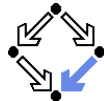
Use all applicable invariants

Add invariants to postcondition

Load Proof Obligation **Cancel**

Generate and load the proof obligations.

A Simple Example (Contd'2)

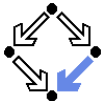


The screenshot shows the KeY Prover interface. The 'Current Goal' pane contains the following code:

```
inReachableState
-> \forall int amount_lv;
  {amount:=amount_lv}
  \forall paycard.PayCard self_PayCard_lv;
  {self_PayCard:=self_PayCard_lv}
  {_old13:=self_PayCard.balance}
  (
    !self_PayCard = null
    & self_PayCard.<created> = TRUE
    & amount > 0
    & ( !self_PayCard.log = null
      & self_PayCard.log.logArray.length = paycard.LogFile.logFileSize
      & self_PayCard.balance
      = self_PayCard.log.logArray[self_PayCard.log.currentRecord].balance
      & ( !self_PayCard.log.currentRecord < self_PayCard.log.logArray.length
        & ( !self_PayCard.log.currentRecord >= 0
          & !self_PayCard.log.logArray = null
          & \forall int index_lv1;
            {_idx1:=index_lv1}
            (
              0 <= _idx1
              & _idx1 < self_PayCard.log.logArray.length
              -> !self_PayCard.log.logArray[_idx1] = null)
            & paycard.LogFile.logFileSize
            = self_PayCard.log.logArray.length)))
      & self_PayCard.balance >= 0
      & self_PayCard.time > 0
      & self_PayCard.available@(paycard.PayCard)() >= 0
      & self_PayCard.unsuccessfulOperations >= 0)
  -> \{ {
    self_PayCard.charge(amount)@paycard.PayCard;
  } }> self_PayCard.balance >= _old13
```

The 'Proof Search Strategy' pane shows a 'Proof Tree' with a single node labeled '1: OPEN GOAL'.

Proof obligation in Dynamic Logic.

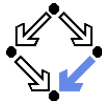


A Simple Example (Contd'3)

```
==>
inReachableState
-> \forall int amount_lv;
    {amount:=amount_lv}
    \forall paycard.PayCard self_PayCard_lv;
        {self_PayCard:=self_PayCard_lv}
        {_old13:=self_PayCard.balance}
        (
            !self_PayCard = null
            & self_PayCard.<created> = TRUE
            & amount > 0
            & ( !self_PayCard.log = null
                & ...
                & self_PayCard.balance >= 0
                & self_PayCard.limit > 0
                & self_PayCard.available@(paycard.PayCard)() >= 0
                & self_PayCard.unsuccessfulOperations >= 0)
        -> \{ {
            self_PayCard.charge(amount)@paycard.PayCard;
        }
        }\> self_PayCard.balance >= _old13
```

Press button "Start automated proof search".

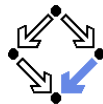
A Simple Example (Contd'4)



The screenshot displays the KeY-Proof application window. The main area shows a proof tree with two nodes, both of which are highlighted in green. The top node is labeled 'Inner Node' and contains a complex logical expression involving a lambda function and several conditions. The bottom node is also highlighted in green. A dialog box titled 'Proof closed' is overlaid on the proof tree, displaying the following statistics: 'Proved.', 'Statistics: Nodes:647', and 'Branches: 14'. The dialog box has an 'OK' button. The bottom status bar of the application indicates 'Strategy: Applied 633 rules (2.1 sec), closed 14 goals, 0 remaining'.

Proof runs through automatically.

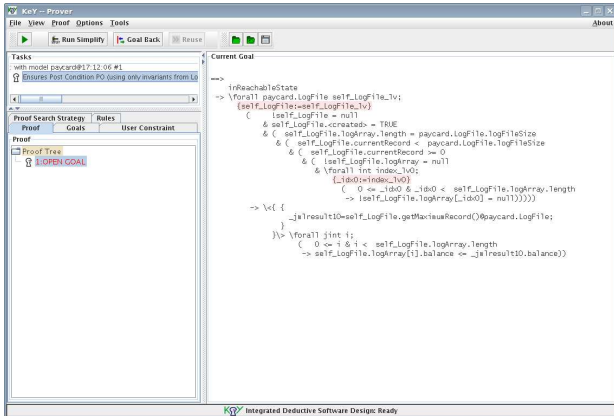
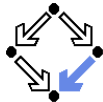
A Loop Example



```
public class LogFile {
    /*@ public invariant
       @ logArray.length
       @ == logFileSize &&
       @ currentRecord < logFileSize
       @ && currentRecord >= 0 &&
       @ \nonnullElements(logArray);
    */
    private /*@ spec_public @*/
        static int logFileSize = 3;
    private /*@ spec_public @*/
        int currentRecord;
    private /*@ spec_public @*/
        LogRecord[] logArray =
            new LogRecord[logFileSize];
    ...
}

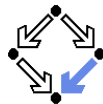
/*@ public normal_behavior
   @ ensures
   @ (\forall int i; 0 <= i && i<logArray.length;
   @   logArray[i].balance <= \result.balance); */
public /*@pure@*/
LogRecord getMaximumRecord(){
    LogRecord max = logArray[0];
    int i=1;
    /*@ loop_invariant
       @   0<=i && i <= logArray.length &&
       @   max!=null &&
       @   (\forall int j; 0 <= j && j<i;
       @     max.balance >= logArray[j].balance);
       @ assignable max, i;
       @ decreases logArray.length - i; */
    while(i<logArray.length){
        LogRecord lr = logArray[i++];
        if (lr.getBalance() > max.getBalance())
            max = lr;
    }
    return max;
}
```


A Loop Example (Contd)



Press button “Start automated proof search”.

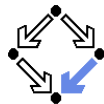
A Loop Example (Contd'2)



The screenshot displays the KeY Prover interface. The 'Tasks' pane shows the current goal: `0 + j_2 >= 0, 1 + -1 * (1 + i_0) + j_2 <= 0, -1 + self_LogFile_1v_0.logArray[1_0].balance * -1 + self_LogFile_1v_0.logArray[1_2].balance >= 0, self_LogFile_1v_0.logArray[1_0].balance >= 1 + acc_0.balance, self_LogFile_1v_0.logArray[1_0].created = TRUE, i_0 <= 2, {i:=1_0 || acc:=acc_0}anon(i, acc), i_0 >= 0, forall j jint j; {j <= -1 | j >= 1_0 | self_LogFile_1v_0.logArray[j].balance <= acc_0.balance}, self_LogFile_1v_0.created = TRUE, self_LogFile_1v_0.logArray.length = 3, self_LogFile_1v_0.currentRecord <= 2, self_LogFile_1v_0.currentRecord >= 0, forall int index_1v_0; {index_1v_0 <= -1 | index_1v_0 >= 3 | self_LogFile_1v_0.logArray[index_1v_0] = null}, inReachableState ==> acc_0 = null, self_LogFile_1v_0.logArray = null, self_LogFile_1v_0 = null, self_LogFile_1v_0.logArray[1_0] = null`. The 'Proof Tree' pane shows a hierarchical structure of proof goals, with the current goal highlighted in yellow. The 'Current Goal' pane shows the goal being worked on, with a '1002:OPEN GOAL' marker. The 'Integrated Deductive Software Design: Ready' status is visible at the bottom.

Press button “Run Simplify”.

A Loop Example (Contd'3)



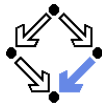
The screenshot shows the KeY-Verifier interface. The 'Current Goal' pane contains the following code:

```
i_2 <= 2,
i_2 >= 0,
forall jint j;
{ i <= -1 | j >= i_2 | self_LogFile_lv_0.logArray[j].balance <= max_2.balance,
self_LogFile_lv_0.<created> = TRUE,
self_LogFile_lv_0.logArray.<created> = TRUE,
self_LogFile_lv_0.logArray.length = 3,
self_LogFile_lv_0.currentRecord <= 2,
self_LogFile_lv_0.currentRecord >= 0,
forall int index_lv0;
(index_lv0 <= -1 | index_lv0 >= 3 | self_LogFile_lv_0.logArray[index_lv0] = null),
inReachableState
==>
max_2
self.
self.
self.
self.
i_2 <
{exc:
{ i: i_1 + 1 |
j: i_3 + i_2 * -1 |
lr: self_LogFile_lv_0.logArray[i_2] |
max: max_2 |
self_LogFile := self_LogFile_lv_0
}
\<[setof-Frame(source=paycard.LogFile, this=self_LogFile)] {
if (lr.getBalance()) max.getBalance() {
max=lr;
}
} catch (java.lang.Throwable e#4) {
exc=true;
throwExc=e#4;
}
}> ( ( exc = TRUE
-> \<[setof-Frame(result=>_j1result10, source=paycard.LogFile, this=self_Log
```

An 'Information' dialog box is open, displaying the message: '2 goals have been closed'. The status bar at the bottom of the window reads: 'SIMPLIFY: 3 goals processed, 2 goals could be closed'.

Verification is successful.

Summary



- Various academic approaches to verifying Java(Card) programs.
 - Jack: <http://www-sop.inria.fr/everest/soft/Jack/jack.html>
 - Jive: <http://www.sct.ethz.ch/research/jive>
 - Mobius: <http://kind.ucd.ie/products/opensource/Mobius>
- Do not yet scale to verification of large Java applications.
 - General language/program model is too complex.
 - Simplifying assumptions about program may be made.
 - Possibly only special properties may be verified.
- Nevertheless helpful for reasoning on Java in the small.
 - Beyond Hoare calculus on programs in toy languages.
- Enforce clearer understanding of language features.
 - Perhaps constructs with complex reasoning are not a good idea...
- Trend: modularization of reasoning.

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