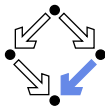


Verifying Java Programs with KeY

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

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



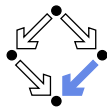
Verifying Java Programs



- **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

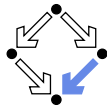


<http://www.key-project.org>

- **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:

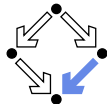
- Rules for predicate logic extended by general rules for modalities.

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:**

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

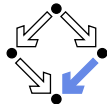
- **Field access rule:**

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

- 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

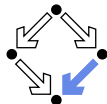


/zvol/formal/bin/startProver &

The screenshot displays the JMLKeY Prover interface. On the left, the 'Proof Search Strategy' pane shows a 'Proof Tree' with a hierarchical structure of nodes, including 'Normal Execution', 'Invariant Initially Valid', and 'Body Preserves Invariant and Decreases V'. The main window shows a code editor with Java code for a method frame. A 'Proof closed' dialog box is open, displaying the message 'Proved. Statistics: Nodes:1516, Branches: 19' and an 'OK' button. The status bar at the bottom indicates 'Strategy: Applied 497 rules (1.9 s), closed 10 goals, 0 remaining'.

```
<{method-frame(source=linsearch.Main): (try { {}  
  if (b_6) {  
    if (_a[i]==x)  
      r=1;  
      i=i+1;  
    }  
  } catch (java.lang.Throwable e#4) {  
    exc_1=true;  
    thrownExc=e#4;  
  }  
}  
= TRUE  
_7=thrownExc==null;  
f (b_7) {  
  try {  
    throw new java.lang.NullPointerException;  
  } catch (java.lang.Throwable e) {  
    exc=e;  
  }  
} else {  
  if (thrownExc instanceof java.lang.Throwable  
    java.lang.Throwable e;  
    e=thrownExc;  
    exc=e;  
  }  
} else {  
  throw thrownExc;  
}  
}> ( ( exc = null  
  -> result = -1  
    & \forall forall jint j;  
      ( )
```

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)

Choose in Menu "File/Load" a package directory or a KeY file.

```
// paycard.key
// This file is part of KeY - Integrated Deductive Software Design
// Copyright (C) 2001-2009 Universitaet Karlsruhe, Germany
//                               Universitaet Koblenz-Landau, Germany
//                               Chalmers University of Technology, Sweden
//
// The KeY system is protected by the GNU General Public License.
// See LICENSE.TXT for details.

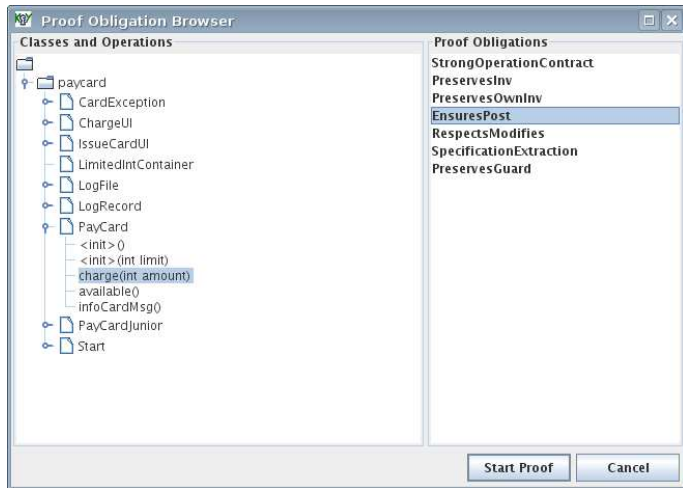
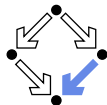
\classpath "classpath";

\javaSource "paycard";

\chooseContract;
```

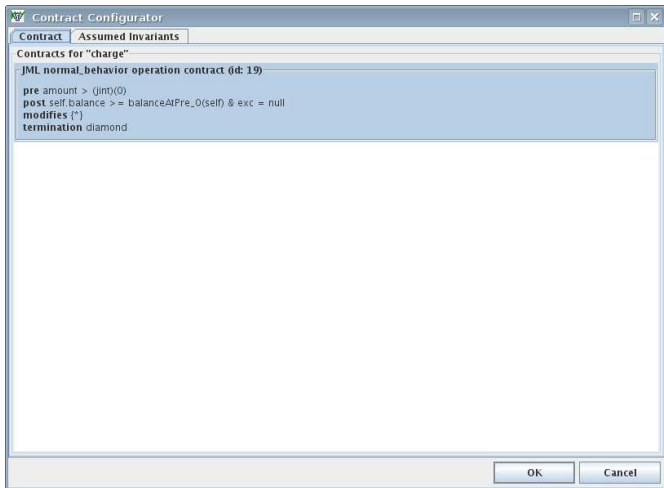
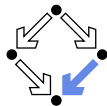
Needed (only) to look up sources of system classes.

A Simple Example (Contd'2)



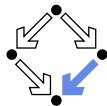
Generate the proof obligations and choose one for verification.

A Simple Example (Contd'3)



Display the chosen proof obligation and start the proof.

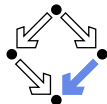
A Simple Example (Contd'4)



The screenshot shows the KeY Prover interface. The 'Current Goal' pane contains the following Dynamic Logic formula:

```
→ p_0.balance = p_0.log.getLogArray(p_0.log.getCurrentRecord().balance
)
& \forallall paycard,PayCard p_0;
( p_0.<created> = TRUE & lp_0 = null
→ p_0.log.currentRecord < p_0.log.getLogArray.Length
& ( p_0.log.currentRecord >= (jint)(0)
& ( lp_0.log.getLogArray >= null
& \forallall jint i;
( 0 <= i & i <= p_0.log.getLogArray.Length
→ lp_0.log.getLogArray[i] = null)
& paycard.LogFile.logFileSize = p_0.log.getLogArray.Length
)))
& \forallall paycard,PayCard p_0;
(p_0.<created> = TRUE & lp_0 = null → p_0.balance >= (jint)(0))
& \forallall paycard,PayCard p_0;
(p_0.<created> = TRUE & lp_0 = null → p_0.limit > (jint)(0))
& \forallall paycard,PayCard p_0;
( p_0.<created> = TRUE & lp_0 = null
→ p_0.available@paycard,PayCard() >= (jint)(0))
& \forallall paycard,PayCard p_0;
( p_0.<created> = TRUE & lp_0 = null
→ p_0.unsuccessfulOperations >= (jint)(0))
& (self.<created> = TRUE & !self = null)
& inInt(amount)
& amount > (jint)(0)
→ {_amount:=amount ||
\for paycard,PayCard x; balanceAtPre_0(x):=x.balance)
\<{
exc=null;try {
self.charge(_amount)@paycard,PayCard;
} catch (java.lang.Throwable e) {
exc=e;
}
}\> (self.balance >= balanceAtPre_0(self) & exc = null)
```

The proof obligation in Dynamic Logic.

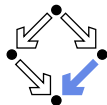


A Simple Example (Contd'5)

```
==>
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" (green arrow).

A Simple Example (Contd'6)



Key-Praver

File View Proof Options Tools About

Run Simplify Prune Proof Reuse

Tasks

Env. with model paycard@10:00:47 AM #1

EnsuresPost (paycard.PayCard.charge, JML normal)

Proof Search Strategy Rules

Proof	Goals	User Constraint
Proof		
Proof Tree		
1: eq_and_2		
2: insert_constant_value		
3: insert_constant_value		
4: inInt		
5: concrete_and_3		
6: impRight		
7: andLeft		
8: andLeft		
9: andLeft		
10: andLeft		
11: andLeft		
12: notLeft		
13: andLeft		
14: andLeft		
15: andLeft		
16: andLeft		
17: andLeft		
18: andLeft		
19: castDel		
20: castDel		
21: castDel		
22: castDel		
23: castDel		
24: castDel		

Inner Node

```
inReachableState
& \forall paycard.PayCard p_0;
  ( p_0.created = TRUE & !p_0 = null
  -> !p_0.log = null)
& \forall paycard.PayCard p_0;
  ( p_0.created = TRUE & !p_0 = null
  -> !p_0.log = null)
& \forall paycard.PayCard p_0;
  ( p_0.created = TRUE & !p_0 = null
  -> !p_0.log.currentRecord.balance
  < p_0.log.logArray.length
  & ( p_0.log.currentRecord >= (jint)0)
  & ( !p_0.log.logArray = null
  & \forall jint i;
    ( 0 <= i
    & i
    <= p_0.log.logArray.length
    -> !p_0.log.logArray[i] = null)
    & paycard.LogFile.logFileSize
    = p_0.log.logArray.length)))
& \forall paycard.PayCard p_0;
  ( p_0.created = TRUE & !p_0 = null
  -> p_0.balance >= (jint)0)
& \forall paycard.PayCard p_0;
  ( p_0.created = TRUE & !p_0 = null
  -> p_0.limit >= (jint)0)
```

Proof closed

Proved.

Statistics:

Nodes: 731

Branches: 12

OK

Strategy: Applied 719 rules (3.6 sec, closed 12 goals, 0 remaining)

Proof runs through automatically.

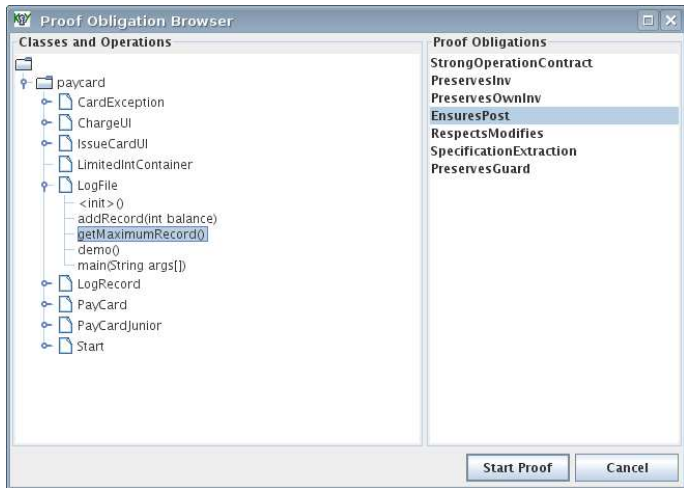
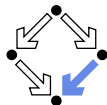
A Loop Example



```
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];
    ...

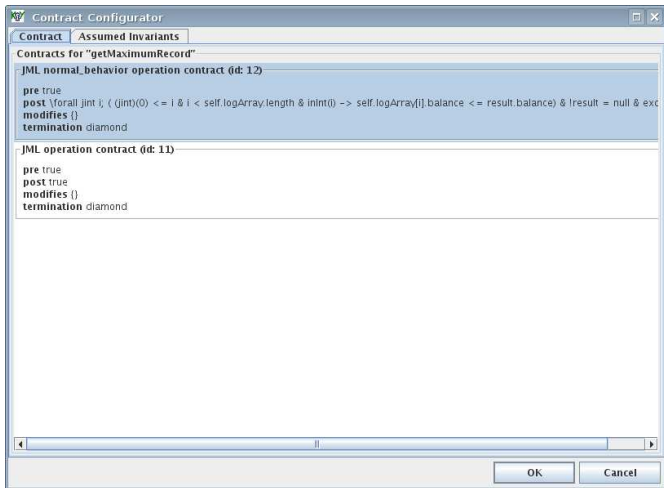
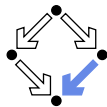
    /*@ public normal_behavior
       @ ensures
       @ (\forall int i; 0 <= i && i<logArray.length;
       @   logArray[i].balance <= \result.balance); */
    public /*@pure@*/
        LogRecord getMaximumRecord(){
        LogRecord max = logArray[0];
        int i=1;
        /*@ loop_invariant
           @   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)



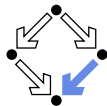
Press button "Start Proof".

A Loop Example (Contd'2)



Press button "OK".

A Loop Example (Contd'3)

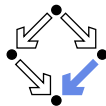


The screenshot shows the KeY Prover interface. The 'Tasks' pane on the left lists several goals, with the first one selected. The 'Proof Search Strategy' pane shows a 'Proof Tree' with a goal labeled '1:OPEN GOAL'. The 'Current Goal' pane displays a complex logical expression involving quantifiers and array operations. The expression is as follows:

```
inReachableState
& \forallall paycard.LogFile p_0;
  (p_0.<created> = TRUE & lp_0 = null -> lp_0.logArray = null)
& \forallall paycard.LogFile p_0;
  (p_0.<created> = TRUE & lp_0 = null -> lp_0.a = null)
& \forallall paycard.LogFile p_0;
  (p_0.<created> = TRUE & lp_0 = null -> lp_0.b = null)
& \forallall paycard.LogFile p_0;
  (
    p_0.<created> = TRUE & lp_0 = null
    -> p_0.logArray.length = paycard.LogFile.logFileSize
    & ( p_0.currentRecord < paycard.LogFile.logFileSize
    & ( p_0.currentRecord >= (jint)0)
    & ( lp_0.logArray = null
    & \forallall jint i;
      ( 0 <= i & i <= p_0.logArray.length
      -> lp_0.logArray[i] = null))))))
& (self.<created> = TRUE & !self = null)
-> \<
  exc=null;try {
    result=self.getMaxiumRecord()@paycard.LogFile;
  } catch (java.lang.Throwable e) {
    exc=e;
  }
}> ( \forallall jint i;
  ( (jint)0 <= i & i < self.logArray.length & inInt(i)
  -> self.logArray[i].balance <= result.balance)
  & !result = null
  & exc = null)
```

Press button "Start" (green arrow).

A Loop Example (Contd'4)



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

```

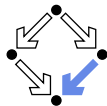
p_0 = null | lp_0.<created> = true | lp_0.0 = null,
\forallall paycard.LogFile p_0;
(p_0 = null | lp_0.<created> = TRUE | p_0.currentRecord >= 0),
\forallall paycard.LogFile p_0;
(p_0 = null | lp_0.<created> = TRUE | p_0.currentRecord <= 2),
self.logArray.length = 3,
\forallall paycard.LogFile p_0;
(p_0 = null | lp_0.<created> = TRUE | p_0.logArray.length = 3),
\forallall paycard.LogFile p_0;
\forallall jint i;
( i <= -1
 | p_0 = null
 | lp_0.<created> = TRUE
 | p_0.logArray.length <= -1 + i
 | lp_0.logArray[i] = null),
self.<created> = TRUE
==>
max_0 = null,
self = null,
self.logArray = null,
self.logArray[i_0] = null,
true
& ( true
-> {i:=i_0 || max:=max_0}anon_0(i, max)
& ( (jint)(1 + i_0) >= 0
& ( self.logArray.length >= (jint)(1 + i_0)
& ( max_0 = null
& ( max_0.<created> = TRUE
& \forallall jint j;
( j <= -1
| j >= (jint)(1 + i_0)
| self.logArray[j].balance <= max_0.balance)))
& inReachableState))

```

The status bar at the bottom indicates: Strategy: Applied 1000 rules (3.3 sec, closed 18 goals, 3 remaining)

Press button "Run Simplify".

A Loop Example (Contd'5)



The screenshot shows the KeY prover interface. The main window displays a goal with a loop structure. An information dialog box is open, stating "1 goal has been closed". The status bar at the bottom indicates "SIMPLIFY: 3 goals processed, 1 goal could be closed!".

Current Goal:

```
forall p:paycard.LogFile p_0:  
  \forall p_0:paycard.LogFile p_0:  
    (p_0 = null | !p_0.<created> = TRUE | p_0.currentRecord >= 0),  
  \forall p_0:paycard.LogFile p_0:  
    (p_0 = null | !p_0.<created> = TRUE | p_0.currentRecord <= 2),  
  self.logArray.length = 3,  
  \forall p_0:paycard.LogFile p_0:  
    (p_0 = null | !p_0.<created> = TRUE | p_0.logArray.length = 3),  
  \forall p_0:paycard.LogFile p_0:  
    \forall jint i:  
      ( i <= -1  
       | p_0 = null  
       | !p_0.<created> = TRUE  
       | self.logArray[i_0].balance <= max_0.balance  
       | self.logArray.<created> = TRUE, i_2 <= 2, !b
```

Information Dialog:

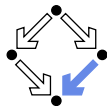
1 goal has been closed

OK

Status Bar: SIMPLIFY: 3 goals processed, 1 goal could be closed!

Press button "Start" (green arrow).

A Loop Example (Contd'6)



The screenshot shows the KEGY Prover interface. The main window displays an inner node with the following code:

```
let p_u = null
  | p_0.<created> = TRUE
  | p_0.logArray.length = 3),
forall paycard.LogFile p_0;
forall jint i;
  ( i <= -1
    | p_0 = null
    | p_0.<created> = TRUE
    | p_0.logArray.length <= -1 + i
    | p_0.logArray[i] = null),
self.<created> = TRUE
```

A dialog box titled "Proof closed" is overlaid on the main window, displaying the following information:

```
Proved.
Statistics:
Nodes:1374
Branches: 26
```

The dialog box has an "OK" button. Below the dialog box, the main window shows the following code snippet:

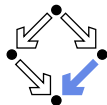
```
non_0(i, max)
...
  & ( self.logArray.length >= (jint)(1 + i_0)
    & ( !max_0 = null
      & ( !max_0.<created> = TRUE
        & forall jint j;
          ( j <= -1
            | j >= (jint)(1 + i_0)
            | self.logArray[j].balance
              <= max_0.balance)))
    & inReachableState))
```

Below the code, it says "Node Nr 1021" and "Upcoming rule application: concrete_and_1 {".

At the bottom of the interface, the status bar reads: "Strategy: Applied 348 rules (1.1 seq, closed 8 goals, 0 remaining)".

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.
 - 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)...