# 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.

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: $\forall a : x = a \Rightarrow [\texttt{y:=x*x}]\ x = a \wedge y = a^2$
  - 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:
  - $$\frac{\Gamma \vdash F[T/X]}{\Gamma \vdash [X := T]F}$$
  - $$\frac{\Gamma \vdash [C_1][C_2]F}{\Gamma \vdash [C_1; C_2]F}$$
  - $$\frac{\Gamma \vdash [C_1]F \qquad \Gamma \vdash [C_2]F}{\Gamma \vdash [C_1 \cup C_2]F}$$
  - $$\frac{\Gamma \vdash F \qquad \Gamma \vdash [C^*](F \Rightarrow [C]F)}{\Gamma \vdash [C^*]F}$$
  - $$\frac{\Gamma \vdash F \Rightarrow G}{\Gamma \vdash [F?]G}$$

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:

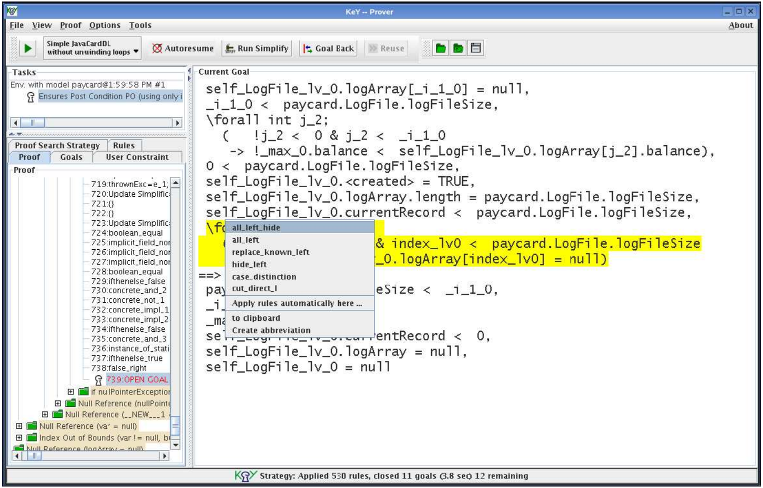
$$\frac{\Gamma, o = o' \vdash F(T) \qquad \Gamma, o \neq o' \vdash F(o'.a)}{\Gamma \vdash \{o.a \leftarrow T\}F(o'.a)}$$

  - 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 &



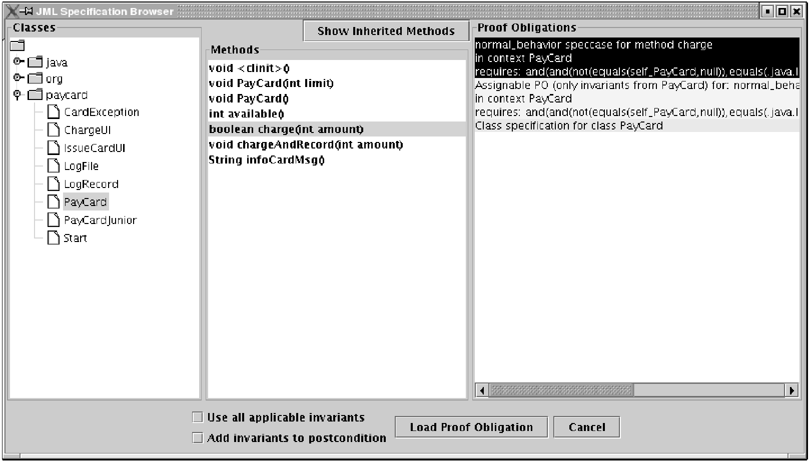
# A Simple Example

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;

...
```

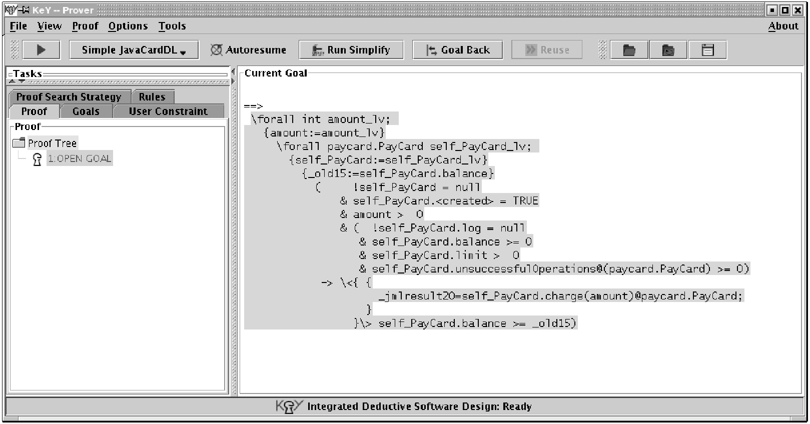
```
/*@
  @ public normal_behavior
  @ requires amount>0 ;
  @ assignable
  @   unsuccessfulOperations, balance;
  @ ensures balance >= \old(balance);
  @*/
public boolean charge(int amount) {
  if (this.balance+amount>=this.limit) {
    this.unsuccessfulOperations++;
      return false;
  } else {
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

...
}
```

# A Simple Example (Contd)




Generate and load the proof obligations.

# A Simple Example (Contd'2)



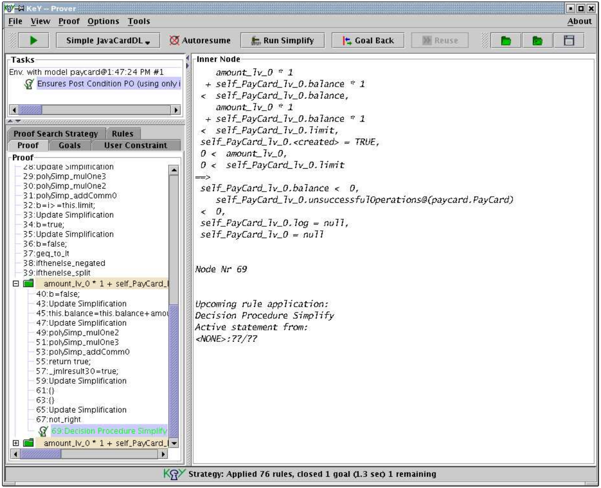

Select the automatic proof strategy "Simple JavaCardDL".

# A Simple Example (Contd'3)

```
==>
 \forall int amount_lv;
   {amount:=amount_lv}
     \forall paycard.PayCard self_PayCard_lv;
       {self_PayCard:=self_PayCard_lv}
         {_old16:=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.unsuccessfulOperations@(paycard.PayCard) >= 0)
            -> \<{ {
                     _jmlresult30=self_PayCard.charge(amount)@paycard.PayCard;
                   }
                 }\> self_PayCard.balance >= _old16)
```

Press the "Run" button and then "Run Simplify".

# A Simple Example (Contd'4)




Proof runs through (almost) automatically.

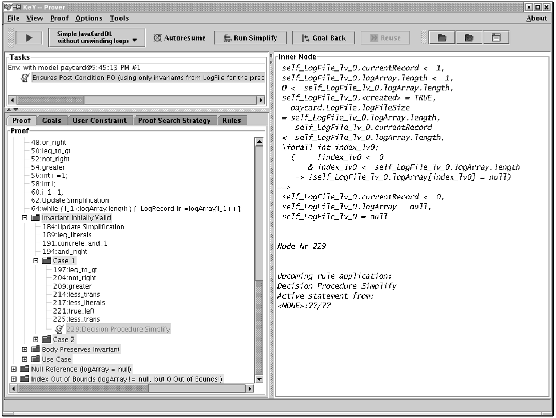
# A Loop Example

```
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);
  @ diverges true; */
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;
    @*/
  while(i<logArray.length){
    LogRecord lr = logArray[i++];
    if (lr.getBalance() > max.getBalance())
      max = lr;
  }
  return max;
}
```

# A Loop Example (Contd)

Proof strategy: “Simple JavaCardDL without unwinding loops”.




Various human interactions required (see demo).

# Summary

- Various academic approaches to verifying Java(Card) programs.
  - Loop: http://www.sos.cs.ru.nl/research/loop/main.html
  - Jack: http://www-sop.inria.fr/everest/soft/Jack/core.html
  - Jive: http://www.sct.ethz.ch/research/jive
- 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)...