Tutorial

# Integrating Object-oriented Design and Deductive Verification of Software

Bernhard Beckert
Reiner Hähnle
Vladimir Klebanov
Peter H. Schmitt

`www.key-project.org`

Integrated Formal Methods 2007

Oxford, UK
July 2nd, 2007



Part I

# Introduction



## What is this Tutorial about?

- Design
- Formal specification
- Deductive verification

of

- Object-oriented software

This tutorial has been developed in the KeY project.
The demos will use the KeY tool.



## KeY Project Partners

University of Koblenz
Bernhard Beckert

University of Karlsruhe
Peter H. Schmitt

Chalmers University
Reiner Hähnle

**www.key-project.org**

## Integrated Formal Methods

**Specification**

- UML + Object Constraint Language (OCL)
- Java Modeling Language (JML)

**Verification**

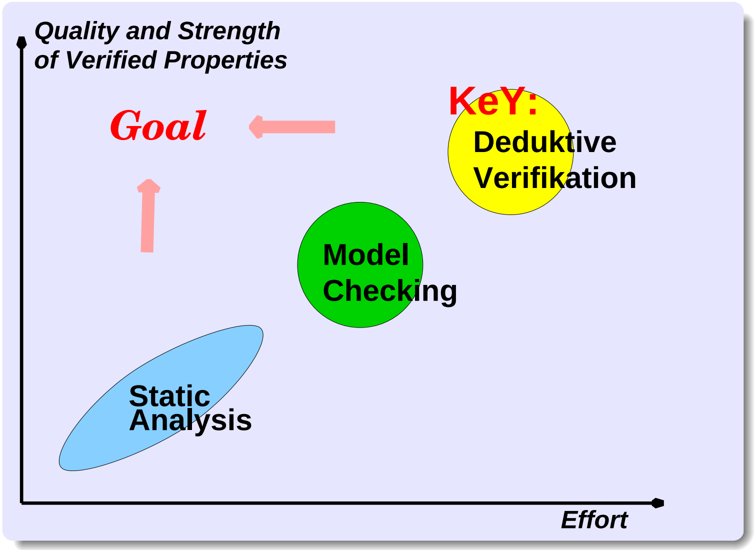
- Dynamic Logic
- Decision procedures

**And ...**

- Static analysis
- Test case generation

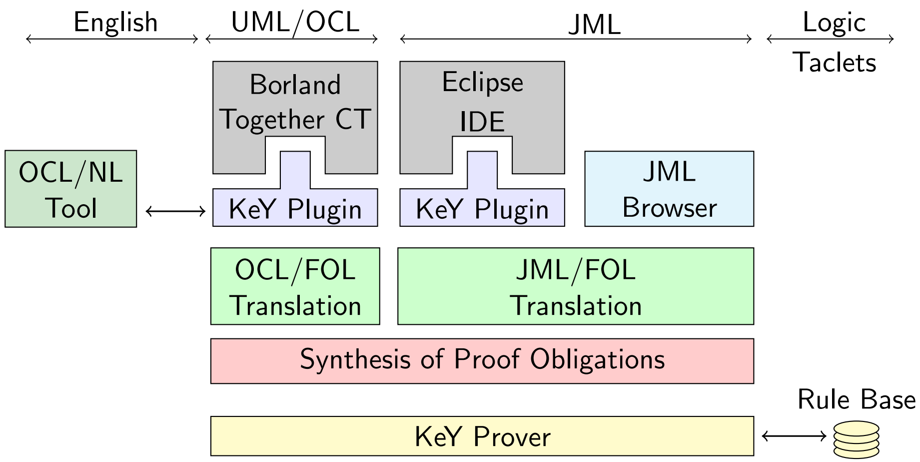


## Different Approaches






## Architecture of the KeY Tool



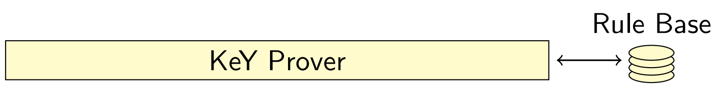



## Choices for the Rule Base

In this tutorial:
100% Java Card

Other rule bases:

- ODL, a minimal abstract object oriented language
- A subset of the C language
- ASM, Abstract State Machines [Stanislas Nachen, ETH Zürich]
- HyKeY, differential dynamic logic for hybrid systems [André Platzer, Univ. of Oldenburg]



## Java Card

### What is Java Card?

- Subset of Java, but with transaction concept
- Sun's official standard for SMART CARDS and embedded devices

### Why Java Card?

Good example for real-world object-oriented language

Java Card has *no*

- garbage collection
- dynamical class loading
- multi-threading
- floating-point arithmetic

Application areas

- security critical
- financial risk
  (e.g. exchanging smart cards is expensive)



Part II

## Specification



Part II

## Specification





## Design by Contract

Class

Invariant

Operation

Precondition
Modifies Clauses
Postcondition
Termination, more precisely: normal or exceptional

Part II

# Specification

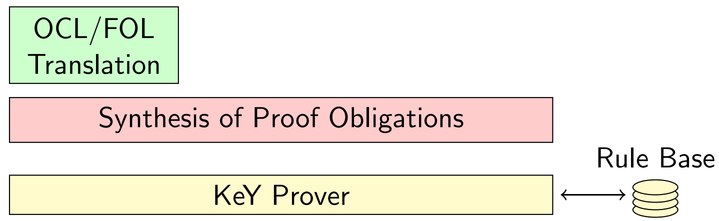




## OCL: Object Constraint Language

**Object Constraint Language**

- Part of the OMG standard UML
- Present Version: 2.0
- Adds formal constraints to UML (class) diagrams
- Accessible to people without a strong mathematical background






## Design by Contract with OCL

Class
Invariant

Operation
Precondition

Postcondition

Modifies Clauses

Termination



## Design by Contract with OCL

```
context ATM
inv:  0 <= self.wrongPinCounter  and
      self.wrongPinCounter <= 2

context  ATM::enterPin(pin: Integer)
pre:  insertedCard <> null and not customerAuthenticated
      and not pin = insertedCard.correctPIN
      and wrongPINCounter < 2
post: wrongPINCounter = wrongPINCounter@pre + 1
      and not customerAuthenticated
```

Modifies Clauses not explicitely supported by OCL

Termination specification not explicitely supported by OCL

## OCL Constraints on the UML Class Ciagram Level




## Proof Obligations

```
context C                     context D extends C
inv:  I                       inv: J
```

**Behavioural Subtyping for classes**

For all instances $o$ of `D` : $o$.`J` implies $o$.`I`.



## Proof Obligations

```
context C::op1              context D::op1
pre:  pre1                  pre:   pre2
post: post1                 post:  post2

D extends C
```

**Behavioural Subtyping for operations**

`pre1` implies `pre2`  and
`post2` implies `post1`



## Proof Obligations

```
context C::op
pre:  pre
post: post
```

Implementation $p$ of op.

**Ensures Postcondition**

If $p$ is started in a state satisfying `pre`
then $p$ terminates and
in the final state `post` is true.

## Proof Obligations

```
context C::op
pre:  pre
post: post
```

```
context C
inv:  I
```

Implementation $p$ of op.

**Preserves Invariant**

If $p$ is started in a state satisfying `pre` and `I`
then $p$ terminates and in the final state `I` is again true.



Part II

## Specification

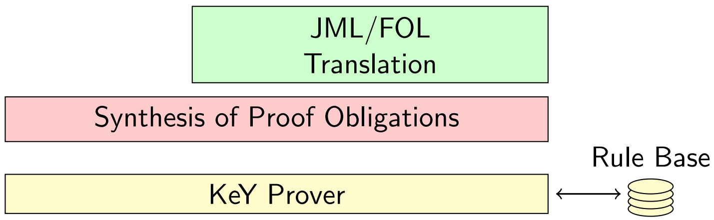




## JML: Java Modeling Language

**Java Modeling Language**

- Behavioral interface specification language for Java
- International community effort
- More and more tools:
  Runtime checkers, static analysis, program verification






## Design by Contract with JML (Invariants)

```
public class ATM {

 /*@ private invariant  wrongPINCounter >= 0 &&
                        wrongPINCounter <= 2
   @*/

  private BankCard insertedCard          = null;
  private boolean  customerAuthenticated = false;
  private int      wrongPINCounter       = 0;

  public void enterPIN (int pin) { ...
  }
}
```

## Design by Contract with JML (Operation Contracts)

```
public class ATM {

/*@ public normal_behavior
  @ requires   insertedCard != null;
  @ requires   !customerAuthenticated;
  @ requires   pin != insertedCard.correctPIN;
  @ requires   wrongPINCounter < 2;
  @ ensures    wrongPINCounter ==
                    \old(wrongPINCounter) + 1;
  @ assignable wrongPINCounter;
  @
  @ also ...
  @*/
  public void enterPIN (int pin) { ...
  }
}
```



## Another Example

```
public class Test {
   private int idx;

   /*@ requires precondition @ */
   /*@ ensures postcondition @ */
   void swapMax(int[] a) {
      int counter = -1; idx = 0;

   /*@ loop_invariant @*/
      while (++counter<a.length) {
         if (a[counter] > a[idx]) idx=counter;
      }
      int tmp=a[idx]; a[idx]=a[0]; a[0]=tmp;
   }
}
```



## JML Specification of swapMax

```
/*@ requires a!=null && a.length > 0;
  @ ensures
  @   (\forall int x; x==idx;
  @    \old(a[0])==a[x] && \old(a[x])==a[0]) &&
  @    (\forall int i; 0 <= i && i<\old(a.length);
  @      a[0] >= a[i] &&
  @      (i!=0 && i!=idx ==> a[i]==\old(a[i])));
  @  diverges false;
  @ */
    void swapMax(int[] a) { ... }
```



## JML Loop Invariant

```
 /*@ loop_invariant
   @   -1<=counter && counter<=a.length &&
   @   0<=idx      && idx<a.length      &&
   @   (\forall int x; x>=0 && x<=counter;
   @       a[idx]>=a[x]);
   @ decreases  (a.length - counter);
   @*/

   while (++counter<a.length) {
        if (a[counter] > a[idx])
          idx=counter;}
```

## Proving Postconditions for *swapMax*

**After termination of the loop, we have . . .**

$$\backslash\text{forall } i \; int; \; ((0 <= i \;\&\; i <= \mathtt{a.length}) \; -> \; \mathtt{a[idx]} >= \mathtt{a}[i])$$

**It is also easy to show that . . .**

`tmp = a[idx]; a[idx] = a[0]; a[0] = tmp;`

has as post-condition

$$\backslash\text{forall } i \; int; \; ((0 <= i \;\&\; i <= \mathtt{a.length} \;\&\; i \neq 0 \;\&\; i \neq \mathtt{idx}) \; -> \; \mathtt{a}[i] = \backslash old\,\mathtt{a}[i])$$

**But . . .**

Loop invariant needs to be strengthened!



## Improved JML Loop Invariant

```
/*@ loop_invariant
  @   -1<=counter && counter<=a.length &&
  @   0<=idx      && idx<a.length      &&
  @   (\forall int x; x>=0 && x<=counter;
  @       a[idx]>=a[x]);
  @ decreases  (a.length - counter);

  @ assignable idx, counter;

  @*/

  while (++counter<a.length) {
       if (a[counter] > a[idx])
         idx=counter;}
```



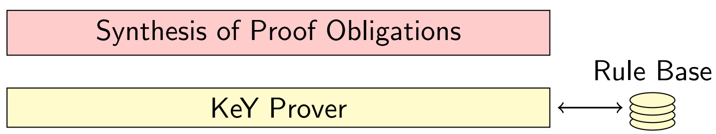
# SECOND DEMO



## Proof Obligations

**Proof Obligations**

- Behavioural Subtyping for classes
- Behavioural Subtyping for operations
- Strong Operation Contract
- Ensures Postcondition
- Preservation of Invariants
- Correctness of Modifies Clauses



## Total Correctness Statement

precondition $\longrightarrow$ $\langle$*program*$\rangle$ postcondition

in state $s_0$

in state $s_0$

in at least one state after termination



## Partial Correctness Statement

precondition $\longrightarrow$ [*program*] postcondition

in state $s_0$

in state $s_0$

in all states after termination



## Specification in Dynamic Logic

```
\programVariables {int pin; ATM self; int _pin; ...}
\problem {
 \forall ATM x0;
     x0.wrongPINCounter = ATM::wrongPINCounter@pre(x0) &
 !self.insertedCard = null &
 !self.customerAuthenticated = TRUE &
 !pin = self.insertedCard.correctPIN &
  self.wrongPINCounter < 2
->
  \< self.enterPIN(_pin)@ATM;\> self.wrongPINCounter =
     ATM::wrongPINCounter@pre(self) + 1
}
```

KeY Prover $\longleftrightarrow$ Rule Base



Part III

# Logic and Calculus

Part III

# Logic and Calculus





## Syntax and Semantics

### Syntax

- Basis: Typed first-order predicate logic
- Modal operators $\langle p \rangle$ and $[p]$ for each (Java Card) program $p$
- Class definitions in background (not shown in formulas)

### Semantics

- Operators refer to the final state of $p$
- $[p]\,F$: If $p$ terminates, then $F$ holds in the final state (partial correctness)
- $\langle p \rangle\,F$: $p$ terminates and $F$ holds in the final state (total correctness)

Java Card DL formulas contain unaltered Java Card source code



## Why Dynamic Logic?

- Transparency wrt target programming language
- More expressive and flexible than Hoare logic
- Can use reference implementations instead of first-order theories
- Symbolic execution is a natural interactive proof paradigm
- Proven technology that scales up



## First-Order Formula Syntax

ASCII syntax, keywords preceded by '\'

### Logical operators

| | |
|---|---|
| `&` | and |
| `\|` | or |
| `->` | implication |
| `<->` | equivalence |
| `!` | negation |

### Logical constants

`true`

`false`

### Conditional terms

`\if(...)\then(...)\else(...)`

### Quantifiers

`\forall`

`\exists`

## Dynamic Logic Example Formulas

$(\texttt{balance} > 1 \ \& \ \texttt{amount} > 1) \ \texttt{->} \ \langle\texttt{charge(amount);}\rangle(\texttt{balance} > 1)$

$\langle\texttt{x = 1;}\rangle([\texttt{while (true) \{\}}]\,\mathit{false})$ Syntax? ok

- Program formulas can appear nested



## Variables

- Logical variables disjoint from program variables
  - No quantification over program variables
  - Programs do not contain logical variables
  - "Program variables" actually non-rigid functions

$\texttt{\textbackslash exists}\ \mathit{int}\ \texttt{x};([\texttt{x = 1;}](\texttt{x} = 1))$ Syntax? bad

- $x$ cannot be a logical variable, because it occurs in the program
- $x$ cannot be a program variable, because it is quantified

$\langle\texttt{int x;}\rangle\texttt{\textbackslash forall}\ \mathit{int}\ \mathit{val};((\langle\texttt{p}\rangle\,\texttt{x} = \mathit{val}) \ \texttt{<->} \ (\langle\texttt{q}\rangle\,\texttt{x} = \mathit{val}))$ Syntax? ok

- p, q equivalent relative to computation state restricted to x



## Rigid and Flexible Terms

**Example**

$\langle\texttt{int i;}\rangle\texttt{\textbackslash forall}\ \mathit{int}\ x;(\texttt{i} + 1 = x \ \texttt{->} \ \langle\texttt{i++;}\rangle(\texttt{i} = x))$

- Interpretation of i depends on computation state $\Rightarrow$ flexible
- Interpretation of $x$ and $+$ must not depend on state $\Rightarrow$ rigid

Locations are always flexible
Logical variables, standard functions are always rigid



## Type System

**Static types**

- Partially ordered finite type hierarchy
- Terms are statically typed (like Java expressions)
- Type casts in logic

**Dynamic types**

- Each term value has a dynamic type
- Dynamic type depends on state
- Dynamic types conform to static types
- Type predicates in logic

## Semantics

### Kripke semantics

- Semantics of a Java program is a partial function from states to states
- $\langle p \rangle\, F$ true in state $s$ iff
  $p$ terminates and $F$ holds in the final state $s'$
  that is reached from $s$ by running $p$
- A Java Card DL formula is valid iff it is true in all states

We need a calculus for checking validity of formulae



Part III

## Logic and Calculus





## Sequents and their Semantics

### Syntax

$$\underbrace{\psi_1, \ldots, \psi_m}_{\textit{Antecedent}} \quad ==> \quad \underbrace{\phi_1, \ldots, \phi_n}_{\textit{Succedent}}$$

where the $\phi_i, \psi_i$ are formulae (without free variables)

### Semantics

Same as the formula

$$(\psi_1 \ \& \ \cdots \ \& \ \psi_m) \quad -> \quad (\phi_1 \mid \cdots \mid \phi_n)$$



## Sequent Rules

### General form

$$\text{RULE NAME}\ \frac{\overbrace{\Gamma_1 ==> \Delta_1 \quad \cdots \quad \Gamma_r ==> \Delta_r}^{\text{Premisses}}}{\underbrace{\Gamma ==> \Delta}_{\text{Conclusion}}}$$

($r = 0$ possible)

### Soundness

If all premisses are valid, then the conclusion is valid

## Some Simple Sequent Rules

$$\text{NOT\_LEFT}\ \frac{\Gamma \texttt{ ==> } A,\ \Delta}{\Gamma,\ !A \texttt{ ==> } \Delta}$$

$$\text{IMP\_LEFT}\ \frac{\Gamma \texttt{ ==> } A,\ \Delta \qquad \Gamma,\ B \texttt{ ==> } \Delta}{\Gamma,\ A \texttt{ -> } B \texttt{ ==> } \Delta}$$

$$\text{CLOSE\_GOAL}\ \frac{}{\Gamma,\ A \texttt{ ==> } A,\ \Delta} \qquad \text{CLOSE\_BY\_TRUE}\ \frac{}{\Gamma \texttt{ ==> } \texttt{true},\ \Delta}$$

$$\text{ALL\_LEFT}\ \frac{\Gamma,\ \texttt{\\forall}\ t\ x; \phi,\ \{x/e\}\phi \texttt{ ==> } \Delta}{\Gamma,\ \texttt{\\forall}\ t\ x; \phi \texttt{ ==> } \Delta}$$

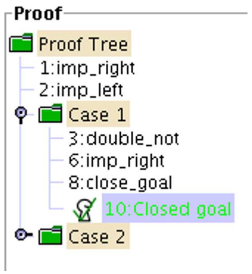
where $e$ var-free term of type $t' \prec t$



## Sequent Calculus Proofs

### Proof tree

- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed






Part III

## Logic and Calculus





## Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

### The Active Statement in a Program

Example

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0; \}\}}}_{\omega}$$

| | |
|---|---|
| active statement | `i=0;` |
| non-active prefix | $\pi$ |
| rest | $\omega$ |

## Proof by Symbolic Program Execution

- Sequent rules execute symbolically the first active statement
- Sequent proof corresponds to symbolic program execution

**Example: The rule for if-then-else (SIMPLIFIED VERSION!)**

$$\frac{\Gamma,\ B \texttt{ ==> } \langle\pi\ \ p\ \ \omega\rangle\,\phi,\ \Delta \qquad \Gamma,\ !B \texttt{ ==> } \langle\pi\ \ q\ \ \omega\rangle\,\phi,\ \Delta}{\Gamma \texttt{ ==> } \langle\pi\ \texttt{if}\ (B)\ \{\ p\ \}\ \texttt{else}\ \{\ q\ \}\ \omega\rangle\,\phi,\ \Delta}$$



Part III

## Logic and Calculus





## Problems to Address

**Object attributes & arrays**

Modelled as non-rigid functions

**Side effects**

Expressions in programs can have side effects

Example

```
if ((y=3) + y < 0) {...} else {...}
```

**Aliasing**

Different names may refer to the same location

Example

After `o.a=17;`, what is `u.a`?



## Other Issues

**Further supported Java Card features**

- method invocation with polymorphism/dynamic binding
- arrays
- abrupt termination
- throwing of NullPointerExceptions, etc.
- object creation and initialisation
- bounded integer data types
- transactions

All Java Card language features are fully addressed in KeY

## Java—A Language of Many Features

**Ways to deal**

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose constructs in program logic

Pro: Feature needs not be handled in calculus
Contra: Modified source code
Example in KeY: Very rare: treating inner classes



## Java—A Language of Many Features

**Ways to deal**

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose constructs in program logic

Pro: Flexible, easy to implement, usable
Contra: Not expressive enough for all features
Example in KeY: Complex expression eval, method inlining, etc., etc.



## Java—A Language of Many Features

**Ways to deal**

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose constructs in program logic

Pro: No logic extensions required, enough to express most features
Contra: Creates difficult first-order POs, unreadable antecedents
Example in KeY: Dynamic types and branch predicates



## Java—A Language of Many Features

**Ways to deal**

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose constructs in program logic

Pro: Arbitrarily expressive extensions possible
Contra: Increases complexity of all rules
Example in KeY: Method frames, updates

## Handling Side Effects

### Problem

- Expressions may have side effects
- Terms in logic have to be side effect free

### Example

`(y=3) + y < 0`

does not only evaluate to a `boolean` value, but also assigns a value to `y`



## Handling Side Effects

### Solution

- Calculus rules realise a stepwise symbolic evaluation (simple transformations)
- Restrict applicability of some rules (e.g., `if-then-else`)

### Example

```
if ((y=3) + y < 0) {...} else {...}
```

rewritten into

```
        y     = 3;
int     val1  = y;
int     val0  = val1 + y;
boolean guard = (val0 < 0);
if (guard) {...} else {...}
```



## Handling Assignment: Explicit State Updates

### Problem

Because of aliasing,
assignment cannot be handled as syntactic substitution

### Solution

State updates as explicit syntactic elements

### Syntax

$$\{loc := val\}\phi$$

where (roughly)

- $loc$ is a program variable $x$, an attribute access $o.a$, or an array access $a[i]$
- $val$ is same as $val$, a literal, or a logical variable



## Assignment Rule in KeY

$$\frac{\Gamma \texttt{ ==> } \{loc := val\}\langle\pi\ \ \omega\rangle\,\phi,\ \Delta}{\Gamma \texttt{ ==> } \langle\pi\ \ loc\texttt{=}val\texttt{;}\ \ \omega\rangle\,\phi,\ \Delta}$$

### Advantages

- no renaming required
- delayed proof branching

### Update simplification in KeY

KeY system has powerful mechanism for simplifying and applying updates

- eager simplification (also: parallel updates)
- lazy application

## Handling Abrupt Termination

**Example: try-throw**

- Abrupt termination handled by "simple" program transformations
- Changing control flow = rearranging program parts

**Example**

TRY-THROW (exc simple)

$$\frac{\Gamma \;\texttt{==>}\; \left\langle \begin{array}{l} \pi \;\texttt{if (exc instanceof T)} \\ \quad \texttt{\{try \{e=exc; r\} finally \{s\}\}} \\ \texttt{else \{s throw exc\};}\; \omega \end{array} \right\rangle \phi}{\Gamma \;\texttt{==>}\; \langle \pi \;\texttt{try\{throw exc; q\} catch(T e)\{r\} finally\{s\};}\; \omega \rangle \phi}$$



## Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - rules for modalities
   - the induction rule
2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by combination of
   - case distinctions (proof branches) and
   - sequences of updates
3. Rules for handling loops
   - rules using loop invariants
   - rules for handling loops by induction
4. Rules for replacing a method invocations by the method's contract
5. Update simplification



Part III

## Logic and Calculus





## Taclets

**Taclets are the "rules" of the KeY system**

Taclets. . .

- have logical content like rules of the calculus
- have pragmatic information for interactive application
- have pragmatic information for automated application
- keep all these concerns separate but close to each other
- can easily be added to the system
- are given in a textual format
- can be verified w.r.t. base taclets

# Taclet Syntax (by Example)

## Modus ponens: Rule

$$\frac{\Gamma,\ \phi,\ \psi \ \texttt{==>}\ \Delta}{\Gamma,\ \phi,\ \phi \texttt{->} \psi \ \texttt{==>}\ \Delta}$$

## Modus ponens: Taclet

```
modus_ponens{
  \find (phi -> psi ==>)
  \assumes (phi ==>)
  \replacewith (psi ==>)
  \heuristics(simplify)
}
```



# An Axiom and a Branching Rule

## Closure rule

```
close_goal {
  \find (==> b)
  \assumes (b ==>)
  \closegoal
  \heuristics(closure)
};
```

## Cut rule

```
cut {
  \add (b ==>);
  \add (==> b)
};
```



# Java Card Taclets

## Rule if_else_split

$$\frac{\begin{array}{c} B = \texttt{TRUE} \ \texttt{==>}\ \langle \pi \ \ p \ \ \omega \rangle F \\ B = \texttt{FALSE} \ \texttt{==>}\ \langle \pi \ \ q \ \ \omega \rangle F \end{array}}{\texttt{==>}\ \langle \pi \ \texttt{if} \ \ (B) \ \ p \ \texttt{else} \ \ q \ \ \omega \rangle F}$$

where $B$ is a Boolean expression without side effects

## Corresponding taclet

```
if_else_split {
  \find (==> <{.. if(#B) #p else #q ...}>post)
  \replacewith  (==> <{.. #p ...}>post) \add (#B = TRUE ==>);
  \replacewith  (==> <{.. #q ...}>post) \add (#B = FALSE ==>)
  \heuristics(if_split)
};
```



# Taclets: Summary

## Taclets are ...

- simple and (sufficiently) powerful
- compact and clear notation
- no complicated meta-language
- easy to apply with a GUI
- validation possible

Part III

# Logic and Calculus





## Verification Calculus Soundness

**A fundamental problem!**

informal language specification

proof rules | formal semantics



## Validating Soundness of Proof Rules

**Bootstrapping**

Validate a core set of rules,
generate and prove verification conditions for additional rules

**Cross-verification**

- against the BALI calculus for Java formalized in Isabelle/HOL [D. von Oheimb, T. Nipkow]
- against the Java semantics in the MAUDE system [J. Meseguer]

**Tests**

Using the compiler test suite Jacks



## From the Java Language Specification

```
PostIncrementExpression:
        PostfixExpression ++
```

*At run time, if evaluation [...] completes abruptly, then the postfix increment expression completes abruptly and no incrementation occurs.*
*Otherwise, the value 1 is added to the value of the variable and the sum is stored back into the variable. Before the addition, binary numeric promotion is performed on the value [...]*
*The value of the postfix increment expression is the value of the variable before the new value is stored.*

## Rule for Postfix Increment

**Intuitive rule (not correct!)**

$$\frac{\texttt{==>}\ \langle \pi\ \texttt{x=y; y=y+1;}\ \omega\rangle\, \phi}{\texttt{==>}\ \langle \pi\ \texttt{x=y++;}\ \omega\rangle\, \phi}$$

**But . . .**

$x = 5$ ==> $\langle\texttt{x=x++;}\rangle\,(x = 6)$ INVALID

**Correct rule**

$$\frac{\texttt{==>}\ \langle \pi\ \texttt{v=y; y=y+1; x=v;}\ \omega\rangle\, \phi}{\texttt{==>}\ \langle \pi\ \texttt{x=y++;}\ \omega\rangle\, \phi}$$



## From the Jacks Conformance Test Suite

```
class T1241r1a {
    final int i=1; static final int j=1;
    static { }
}

class T1241r1b {
     /*@ public normal_behavior
       @ ensures \result == 7;  @ */
    public static int main() {
      int s = 0; T1241r1a a = null;
      s = s + a.j;
      try {s = s + a.i;}
      catch (Exception e) {
        s = s + 2; a = new T1241r1a();
        s = s + a.i + 3;  }
      return s; }
}
```



Part III

# Logic and Calculus





## Interaction and Automation

For realistic programs:
Fully-automated verification impossible

# Interaction and Automation

## Goal in KeY: Integrate automated and interactive proving

- All easy or obvious proof steps should be automated
- Sequents presented to user should be simplified as far as possible
- Primary steps that require interaction: induction, treatment of loops
- Taclets enable interactive rule application mostly using mouse

## Typical workflow when proving in KeY (and other interactive provers)

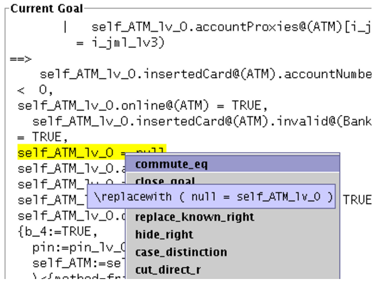
1. Prover runs automatically as far as possible
2. When prover stops user investigates situation and gives hints (makes some interactive steps)
3. Go to 1



# Working with Sequents: Sequent View

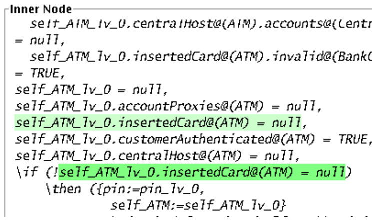
## For goals (leaves of tree)

- Obtaining information about formulas/terms (press Alt key)
- Selecting formulas/terms, applying rules to them




## For inner nodes

- Inspecting parts involved in rule application (highlighted)






# Extension of Proof: Application of Single Taclets

## Taclet application requires

- A proof goal
- Focus of rule application: term/formula in the goal
- Instantiation of schema variables

## Main procedure for applying a taclet interactively

1. Select an application focus using mouse pointer
2. Select a particular rule from the context menu
3. Instantiate schema variables



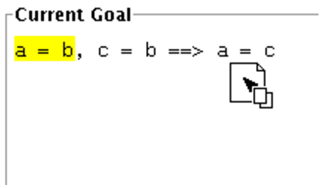
# Applying Taclets using Drag-and-Drop

## Possible for taclets with find-part and one assumption, like . . .

- Rewriting a term using an equation
- Instantiating formulas with universal-type quantifier

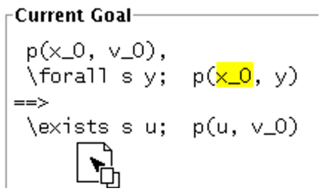
## Applying equations

- Drag the equation onto the term to be rewritten




## Instantiating quantifiers

- Drag instantiation term onto the quantified formula



## Means of Automation Implemented in KeY

- Parameterized strategies for applying rules automatically
- Free-variable first-order calculus
  (non-destructive, proof-confluent)
- Invocation of external theorem provers, decision procedures:
  - Simplify (from ESC/Java)
  - ICS
  - Any other with SMT-LIB interface



## Strategies Currently Present in KeY

Strategies optimized for . . .

**Symbolic execution of programs**

- Come in different flavours: with/without unwinding loops, etc.
- Concentrate on eliminating program and simplifying sequents

**Handling first-order logic**

- Implements a complete first-order theorem prover
- Includes arithmetics solver



# Part IV

## Further Topics



# Part IV

## Further Topics

## Specification of Integer Square Root

### Taken from: Preliminary Design of JML [G. Leavens et al.]

```
/*@ requires y >= 0;
  @ ensures
  @  \result * \result <= y &&
  @ y < (abs(\result)+1) * (abs(\result)+1);
  @ */
  public static int isqrt(int y)
```

### But ...

$\backslash result = 1073741821 = \frac{max\_int-5}{2}$ satisfies spec for $y = 1$.

$1073741821 * 1073741821 = -2147483639 \leq 1$
$1073741822 * 1073741822 = 4 > 1$



## Data Type Gap

### Specification level: Abstract data types

- Integer ($\mathbb{Z}$)
- Set, List

### Implementation level: Concrete programming language data types

- `byte`, `short`, `int`, `long`
- Array



## Examples

### Valid for Java integers

- $\mathtt{MAX\_INT} + 1 = \mathtt{MIN\_INT}$
- $\mathtt{MIN\_INT} * (-1) = \mathtt{MIN\_INT}$
- $\exists x, y.\ (x \neq 0 \land y \neq 0 \land x * y = 0)$

### Not valid for Java integers

- $\forall x. \exists y.\ y > x$

### Not a sound rewrite rule for Java integers

- $x + 1 > y + 1 \quad \rightsquigarrow \quad x > y$



## More Formal Semantics of Java Integer Types

### Range of primitive integer types in Java

| Type | Range | Bits |
|---|---|---|
| `byte` | $[-128, 127]$ | 8 |
| `short` | $[-32768, 32767]$ | 16 |
| `int` | $[-2147483648, 2147483647]$ | 32 |
| `long` | $[-2^{63}, 2^{63}-1]$ | 64 |

# Options for Integer Semantics Rules in KeY

**Java semantics**

- Faithfully axiomatises the overflow semantics of Java integers
- Leads to hard verification problems (lack of intuition)

**Arithmetic semantics**

- Leads to easier verification problems
- Incorrect

**Arithmetic semantics with overflow check**

- Correct
- Leads to moderate verification problems
- Incomplete
  (there are programs that are correct despite overflows)



Part IV

# Further Topics





# Proof Reuse

**Basic Use Case**

1. Verification attempt fails
2. Amend program
3. Recycle unaffected proof parts

**Example: Incremental Verification**

1. Program correct w.r.t. arithmetic semantics? ✔
2. Program correct w.r.t. overflow checking semantics? ✘
3. Fix bug, reuse proof ✔

Successfully used in case studies



# Proof Reuse

**Observations**

- Similar program rule applications focus on similar program parts
- Program rules applicable at a limited number of goals
- Proof structure follows program structure

**Steps**

1. Identify changes in program (program diff)
2. Identify subproofs beginning with unaffected statements
3. Similarity-guided proof replay

Part IV

## Further Topics





# Generating Test Cases

**Testing makes sense, even in cases when a formal proof exists**

- Testing can uncover bugs in environment (hardware, compiler, operating system, virtual machine)
- Testing can uncover specification errors
- Testing can uncover bugs w.r.t. unspecified properties (e.g. timing)
- Tests can be generated from incomplete proofs

**Idea: Use a formal proof to generate test cases**

- KeY provides the path condition for each execution path
- High code coverage (feasible execution paths)
- For infinite number of paths:
  Unwind loops finite number of times, inline method bodies



# Test Case Ingredients

**Generate unit tests**

- Code fragment to be tested
- Test cases
- Test oracle
- Test setup for each execution path



# Example (Finite Number of Execution Paths)

**Compute the middle of three numbers**

```
public static int middle(int x, int y, int z){
    int mid = z;
    if(y<z){
        if(x<y){
            mid = y;
        }else if(x<z){
            mid = x;
        }
    }else{
        if(x>y){
            mid = y;
        }else if(x>z){
            mid = x;
        }
    }
    return mid;
}
```

Part IV

# Further Topics





## Verifying concurrent Java programs

Full reasoning about data

Beyond just safety or race detection

No abstractions



## java.lang.StringBuffer

```
private char value[];
private int count;

public synchronized StringBuffer
                        append(char c) {
    int newcount = count + 1;
    if (newcount > value.length)
        expandCapacity(newcount);
    value[count++] = c;
    return this;
}
```



## Verify That...

$$\mathtt{strb.<lockcount>} = 0 \land \neg\mathtt{strb} = \mathtt{null} \land \mathtt{strb.count} = 0 \rightarrow$$
$$\forall n.\ n > 0 \rightarrow$$
$$\langle^{\{n\}}\mathtt{strb.append(c);}^{\{0\}}\rangle\, \mathtt{strb.count} = n \land$$
$$\forall k.\ 0 \leq k < n \rightarrow \mathtt{strb.value}[k] = \mathtt{c}(p_1(k+1))$$

## Three-Step Programme

1. Unfold
2. Prove atomicity invariant
3. Symbolic execution + induction



## Statistics

- Proof steps: 14622
- Branches: 238 (3 relevant)
- Interactions: 2
- Runtime: ~1 minute
- Result: conjecture false for $n \geq \mathit{MAX\_INT}$



## Concurrency Verification Problems

- Number of threads
  ➥ symmetry reduction (this work)
- Number of interference points
  ➥ exploit locking, data confinement
- Java Memory Model
  ➥ ?



## Alas. . .

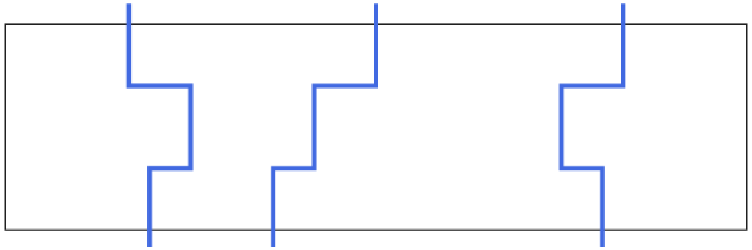
No thread identities **in** programs

No dynamic thread creation (but unbounded concurrency)

Currently only atomic loops

## The Calculus Is Built On. . .

symmetry reduction

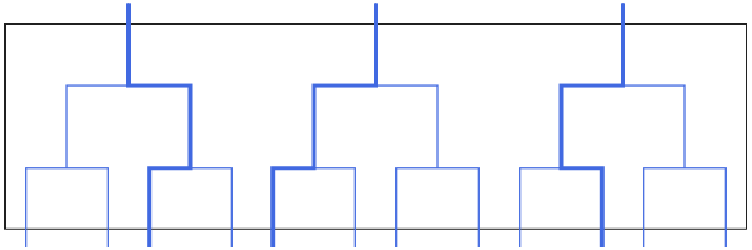


. . . and explicit scheduler formalization



## The Calculus Is Built On. . .

symmetry reduction

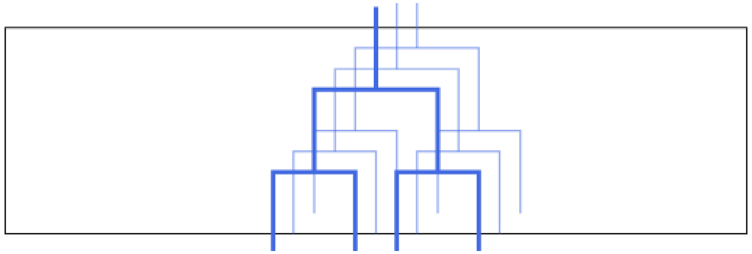


. . . and explicit scheduler formalization



## The Calculus Is Built On. . .

symmetry reduction



. . . and explicit scheduler formalization



Part V

# Wrap Up

Part V

# Wrap Up





## "Fundamental" Case Studies: Libraries

### Java Collections Framework (JCF)

- Part of JCF (treating sets) specified using UML/OCL
- Parts of reference implementation verified

### Java Card API Reference Implementation

- Covers whole of latest API used in practice (2.2.1)
- 60 classes, 4,500 lines of Java code
- Effort: 2–3 (expert) months



## Algorithm Verification

### Schorr-Waite Algorithm

- Graph-marking algorithm (memory-efficient garbage collection)
- Very complicated loop invariant
- One single proof with 17,000 steps



## Security Case Studies: Java Card Software

### Demoney

- Electronic purse application provided by Trusted Logic S.A.

### Mondex Card

- Smart card for electronic financial transactions
- Issued by NatWest in 1996
- Proposed as case study in Grand Challenge
- KeY used to verify a reference implementation in Java Card

## Safety Case Study

**Avionics Software**

- Java implementation of a Flight Manager module at Thales Avionics
- Comprehensive specification using JML, emphasis on class invariants
- Verification of some nested method calls using contracts

**Virtual Machine for Real Time Secury Java**

- Verification of some library functions of the Jamaica VM from Aicas



Part V

# Wrap Up





## Some Current Directions of Research in KeY

- Multi-threaded Java
- Integration of deduction and static analysis
- Integration of verification and testing
- Counter examples
- Symbolic error propagation
- Verification of MISRA C
- Proof visualization, proving as debugging

Extension of dynamic logic for multi-threading
Symbolic execution calculus
Prototype available, `StringBuffer` class verified



## Some Current Directions of Research in KeY

- Multi-threaded Java
- Integration of deduction and static analysis
- Integration of verification and testing
- Counter examples
- Symbolic error propagation
- Verification of MISRA C
- Proof visualization, proving as debugging

Mutual call of analyser/prover, common semantic framework
Implementation of static analysis in theorem proving frame

## Some Current Directions of Research in KeY

- Multi-threaded Java
- Integration of deduction and static analysis
- Integration of verification and testing
- Counter examples
- Symbolic error propagation
- Verification of MISRA C
- Proof visualization, proving as debugging

Generation of test cases from proofs
Symbolic testing
New coverage criteria



## Some Current Directions of Research in KeY

- Multi-threaded Java
- Integration of deduction and static analysis
- Integration of verification and testing
- Counter examples
- Symbolic error propagation
- Verification of MISRA C
- Proof visualization, proving as debugging

Generate counter example from failed proof attempt
Counter example search as proof of uncorrectness



## Some Current Directions of Research in KeY

- Multi-threaded Java
- Integration of deduction and static analysis
- Integration of verification and testing
- Counter examples
- Symbolic error propagation
- Verification of MISRA C
- Proof visualization, proving as debugging

Symbolic error classes modeled by formulas
Error injection by instrumentation of Java Card DL rules
Symbolic error propagation via symbolic execution



## Some Current Directions of Research in KeY

- Multi-threaded Java
- Integration of deduction and static analysis
- Integration of verification and testing
- Counter examples
- Symbolic error propagation
- Verification of MISRA C
- Proof visualization, proving as debugging

## Some Current Directions of Research in KeY

- Multi-threaded Java
- Integration of deduction and static analysis
- Integration of verification and testing
- Counter examples
- Symbolic error propagation
- Verification of MISRA C
- Proof visualization, proving as debugging



## Some Current Directions of Research in KeY

- Multi-threaded Java
- Integration of deduction and static analysis
- Integration of verification and testing
- Counter examples
- Symbolic error propagation
- Verification of MISRA C
- Proof visualization, proving as debugging



Part V

# Wrap Up





## Acknowledgments

**Funding agencies**

- Deutsche Forschungsgemeinschaft (DFG)
- Deutscher Akademischer Auslandsdienst (DAAD)
- Vetenskapsradet (VR)
- VINNOVA
- STINT
- European Union (within the IST framework)

## Acknowledgments

### Students

The many students who did a thesis or worked as developers

### Alumni

W. Menzel (em.), T. Baar (EPFL), A. Darvas (ETH), M. Giese (RICAM), W. Mostowski (U Nijmegen), A. Roth (SAP), S. Schlager

### Colleagues who collaborated with us

J. Hunt, K. Johanisson, A. Ranta, D. Sands



Part V

## Wrap Up





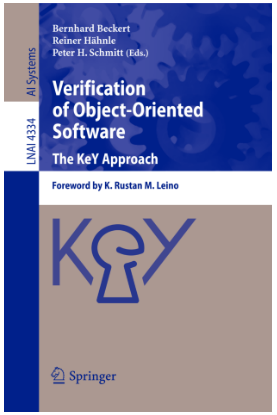
## More Information

### The KeY Book

B. Beckert, R. Hähnle, P. H. Schmitt (eds.)

*Verification of Object-Oriented Software: The KeY Approach*

Springer-Verlag, LNCS 4334, 2007.




### Web site

**www.key-project.org**