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.
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

KeY: Deduktive Verifikation

Quality and Strength of Verified Properties

Goal

Model Checking

Static Analysis

Effort

Architecture of the KeY Tool

- English, UML/OCL, JML, Logic Taclets
- OCL/NL Tool
- UML/OCL
- JML
- OCL/FOL Translation
- JML/FOL Translation
- Synthesis of Proof Obligations
- Rule Base
- KeY Prover
- KeY Plugin
- Eclipse IDE
- JML Browser
- OCL/FOL Translation
- JML/FOL Translation

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
- HyKeY, differential dynamic logic for hybrid systems

[Stanislas Nachen, ETH Zürich]
[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

Design by Contract

Class
Invariant

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

Specification

Design by Contract

OCL Specification

JML Specification

OCL: 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

OCL/FOL Translation

Rule Base

Synthesis of Proof Obligations

KeY Prover

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 Diagram Level

Proof Obligations

context C
inv: I

context D extends C
inv: J

Behavioural Subtyping for classes
For all instances $o$ of $D$ : $o$.J implies $o$.I.

Proof Obligations

context C::op1
pre: pre1
post: post1

context D::op1
pre: pre2
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

Implementation \( p \) of \( op \).

Preserves Invariant

If \( p \) is started in a state satisfying \( \text{pre} \) and \( I \) then \( p \) terminates and in the final state \( I \) is again true.

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)

```java
public class ATM {
    private BankCard insertedCard = null;
    private boolean customerAuthenticated = false;
    private int wrongPINCounter = 0;

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

    public void enterPIN (int pin) {
        ...}
}
```
Design by Contract with JML (Operation Contracts)

```java
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

```java
public class Test {
    private int idx;

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

        /*@ loopInvariant @*/
        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

```java
/*@ 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])));
 @ decreases (a.length - counter);
 @*/

void swapMax(int[] a) { ... }
```

JML Loop Invariant

```java
/*@ loopInvariant
 @ -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 ...**

\[\forall i \in \mathbb{Z}; ((0 \leq i \land i < a.length) \implies a[idx] \geq a[i])\]

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

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

has as post-condition

\[\forall i \in \mathbb{Z}; ((0 \leq i \land i < a.length \land i \neq 0 \land i \neq idx) \implies a[i] = olda[i])\]

**But ...**

Loop invariant needs to be strengthened!

Improved JML Loop Invariant

```java
/*@ 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;
}
```

**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 → ϕ → postcondition

in state $s_0$ in at least one state after termination

**Partial Correctness Statement**

precondition → $[\text{program}]$ postcondition

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
}

---

**Part III**

Logic and Calculus
Part III

Logic and Calculus

6 Java Card DL
7 Sequent Calculus
8 Rules for Programs: Symbolic Execution
9 A Calculus for 100% Java Card
10 Taclets and Taclet Language
11 Correctness of Proof Rules
12 Interactive and Automated Proof Construction

Syntax and Semantics

Syntax

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

Semantics

- Operators refer to the final state of $p$
- $\lbrack p \rbrack F$: 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)

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
- $\rightarrow$ implication
- $\leftrightarrow$ equivalence
- $\neg$ negation

Logical constants

- true
- false

Conditional terms

|if(...)|then(...)|else(...)|

Quantifiers

|\forall\forall|
|\exists\exists|
Dynamic Logic Example Formulas

(balance > 1 & amount > 1) \rightarrow \langle \text{charge(amount);} \rangle (balance > 1)

\langle x = 1; \rangle \{[\text{while (true) } \{\} \} \text{false}\}

Variables

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

\exists int x; ([x = 1;] (x = 1))

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

\langle \text{int } x; \rangle \forall \text{int } val; ([\langle \langle p \rangle \rangle x = val \rangle) \leftrightarrow ([\langle q \rangle x = val \rangle)

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

Rigid and Flexible Terms

Example

\langle \text{int } i; \rangle \forall \text{int } x; ([i + 1 = x \rightarrow \{\text{i++;}\} (i = x)]

- Interpretation of i depends on computation state \Rightarrow \text{flexible}
- Interpretation of x and + must not depend on state \Rightarrow \text{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

Sequents and their Semantics

Syntax

\[
\psi_1, \ldots, \psi_m \implies \phi_1, \ldots, \phi_n
\]

Antecedent
Succedent

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

Semantics

Same as the formula

\[
(\psi_1 \& \cdots \& \psi_m) \implies (\phi_1 | \cdots | \phi_n)
\]

Part III

Logic and Calculus

- Java Card DL
- Sequent Calculus
- Rules for Programs: Symbolic Execution
- A Calculus for 100% Java Card
- Taclets and Taclet Language
- Correctness of Proof Rules
- Interactive and Automated Proof Construction

Sequent Rules

General form

\[
\text{RULE NAME} \quad \Gamma_1 \implies \Delta_1 \quad \cdots \quad \Gamma_r \implies \Delta_r \\
\Gamma \implies \Delta
\]

Premisses
Conclusion

$(r = 0$ possible$)$

Soundness

If all premisses are valid, then the conclusion is valid
Some Simple Sequent Rules

**NOT_LEFT** \[ \Gamma ==> A, \Delta \]
\[ \Gamma, \lnot A ==> \Delta \]

**IMP_LEFT** \[ \Gamma ==> A, \Delta \]
\[ \Gamma, B ==> \Delta \]
\[ \Gamma, A \rightarrow B ==> \Delta \]

**CLOSE_GOAL** \[ \Gamma, A ==> A, \Delta \]

**CLOSE_BY_TRUE** \[ \Gamma ==> \text{true}, \Delta \]

**ALL_LEFT** \[ \Gamma, \forall t x; \phi, \{x/e\} \phi ==> \Delta \]
\[ \Gamma, \forall t x; \phi ==> \Delta \]

where e var-free term of type t' < t

Part III

Logic and Calculus

- Java Card DL
- Sequent Calculus
- Rules for Programs: Symbolic Execution
  - A Calculus for 100% Java Card
  - Taclets and Taclet Language
  - Correctness of Proof Rules
  - Interactive and Automated Proof Construction

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

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

\[ 1:\{\text{try}\{ i=0; j=0; \} \text{finally}\{ k=0; \}} \]

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!)

\[
\begin{align*}
\Gamma, B &= \Rightarrow (\pi p \omega) \phi, \Delta \\
\Gamma &\Rightarrow (\pi (if (B) \{p\} else \{q\}) \omega) \phi, \Delta \\
\Gamma, B &= \Rightarrow (\pi q \omega) \phi, \Delta
\end{align*}
\]

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?
```
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

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

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

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

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

**Example**
\[
\begin{align*}
\text{if } ((y=3) + y < 0) \{ & \ldots \} \text{ else } \{ & \ldots \} \\
\text{rewritten into} \\
y &= 3; \\
\text{int } \text{val1} = y; \\
\text{int } \text{val0} = \text{val1} + y; \\
\text{boolean } \text{guard} = (\text{val0} < 0); \\
\text{if } (\text{guard}) \{ & \ldots \} \text{ else } \{ & \ldots \}
\end{align*}
\]

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**
\[
\begin{align*}
\Gamma & \implies > \{ loc := val \} \langle \pi \omega \rangle \phi, \Delta \\
\Gamma & \implies > \langle \pi \text{loc=}val; \omega \rangle \phi, \Delta
\end{align*}
\]

**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)

\[ \Gamma \implies \begin{cases}
\pi \text{ if (exc instanceof T)} \\
\{ \text{try } \{ e=\text{exc}; r \} \text{ finally } \{ s \} \} \phi \\
\text{else } \{ \text{throw exc; } s \} \phi
\end{cases} \]

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

1. Java Card DL
2. Sequent Calculus
3. Rules for Programs: Symbolic Execution
4. A Calculus for 100% Java Card
5. Taclets and Taclet Language
6. Correctness of Proof Rules
7. Interactive and Automated Proof Construction

Taclets

Taclets are the “rules” of the KeY system

- 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**

\[ \Gamma, \phi, \psi \implies \Delta \]
\[ \Gamma, \phi, \phi \implies \psi \implies \Delta \]

**Modus ponens: Taclet**

```latex
modus_ponens{
  \find (phi \implies psi \implies)
  \assumes (phi \implies)
  \replacewith (psi \implies)
  \heuristics(simplify)
}
```

**Java Card Taclets**

**Rule if else split**

\[ B = \text{TRUE} \implies \langle \pi \ p \ \omega \rangle F \]
\[ B = \text{FALSE} \implies \langle \pi \ q \ \omega \rangle F \]
\[ \implies \langle \pi \ \text{if} \ (B) \ p \ \text{else} \ q \ \omega \rangle F \]

where \( B \) is a Boolean expression without side effects

**Corresponding taclet**

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

An Axiom and a Branching Rule

**Closure rule**

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

**Cut rule**

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

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

Java Card DL
Sequent Calculus
Rules for Programs: Symbolic Execution
A Calculus for 100% Java Card
Taclets and Taclet Language
Correctness of Proof Rules
Interactive and Automated Proof Construction

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!)

$$\Rightarrow \left\langle \pi \ x=y; \ y=y+1; \ \omega \right\rangle \phi$$

$$\Rightarrow \left\langle \pi \ x=y++; \ \omega \right\rangle \phi$$

But ...

$$x = 5 \Rightarrow (x = x++;) (x = 6)$$

INVALID

Correct rule

$$\Rightarrow \left\langle \pi \ v=y; \ y=y+1; \ x=v; \ \omega \right\rangle \phi$$

$$\Rightarrow \left\langle \pi \ x=y++; \ \omega \right\rangle \phi$$

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; }
    }

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

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
- Select an application focus using mouse pointer
- Select a particular rule from the context menu
- Instantiate schema variables

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)

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

Integrating Object-oriented Design and Deductive Verification of Software: Proof Construction

Part IV

Further Topics

- Dealing with Integers
- Proof Reuse
- Generating Test Cases
- Concurrency
Specification of Integer Square Root

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

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

But...

\(\result = 1073741821 = \frac{\text{max int} - 5}{2}\) satisfies spec for \(y = 1\).

1073741821 * 1073741821 = -2147483639 \(\leq 1\)

1073741822 * 1073741822 = 4 > 1

Examples

Valid for Java integers

- \(\text{MAX_INT} + 1 = \text{MIN_INT}\)
- \(\text{MIN_INT} * (-1) = \text{MIN_INT}\)
- \(\exists x, y. (x \neq 0 \land y \neq 0 \land x \cdot y = 0)\)

Not valid for Java integers

- \(\forall x. \exists y. y > x\)

Not a sound rewrite rule for Java integers

- \(x + 1 > y + 1 \Rightarrow x > y\)

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

More Formal Semantics of Java Integer Types

Range of primitive integer types in Java

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>[-128, 127]</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>[-32768, 32767]</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>[-2147483648, 2147483647]</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>[-2^63, 2^63 - 1]</td>
<td>64</td>
</tr>
</tbody>
</table>
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)

Proof Reuse

Basic Use Case
- Verification attempt fails
- Amend program
- Recycle unaffected proof parts

Example: Incremental Verification
- Program correct w.r.t. arithmetic semantics? ✓
- Program correct w.r.t. overflow checking semantics? ❌
- Fix bug, reuse proof ✓

Successfully used in case studies

Part IV

Further Topics

1 Dealing with Integers
2 Proof Reuse
3 Generating Test Cases
4 Concurrency

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

Example (Finite Number of Execution Paths)

```java
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

13. Dealing with Integers
14. Proof Reuse
15. Generating Test Cases
16. Concurrency

Verifying concurrent Java programs

- Full reasoning about data
- Beyond just safety or race detection
- No abstractions

java.lang.StringBuffer

```java
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...

\[
\text{strb.<lockcount> = 0} \land \neg \text{strb = null} \land \text{strb.count = 0} \rightarrow \\
\forall n. n > 0 \rightarrow \\
\langle \{n\}\text{strb.append(c)};\{0\}\rangle \text{strb.count} = n \land \\
\forall k. 0 \leq k < n \rightarrow \text{strb.value}[k] = c(p_1(k+1))
\]
Three-Step Programme

- Unfold
- Prove atomicity invariant
- Symbolic execution + induction

Statistics

- Proof steps: 14622
- Branches: 238 (3 relevant)
- Interactions: 2
- Runtime: \(\sim 1\) minute
- Result: conjecture false for \(n \geq \text{MAX}_\text{INT}\)

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...

... and explicit scheduler formalization

symmetry reduction

Part V

Wrap Up

symmetry reduction

... and explicit scheduler formalization
Part V

Wrap Up

Case Studies

Current Directions of Work

Acknowledgments

Algorithm Verification

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

“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

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

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

Part V

Wrap Up

Case Studies

Current Directions of Work

Acknowledgments

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

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

---

**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

Part V

Wrap Up

Case Studies

Current Directions of Work

Acknowledgments

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

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

More Information

The KeY Book
B. Beckert, R. Hähnle, P. H. Schmitt (eds.)
Verification of Object-Oriented Software: The KeY Approach

Web site
www.key-project.org