Tutorial

Integrating Verification and Testing of Object-Oriented Software

Christian Engel
Christoph Gladisch
Vladimir Klebanov
Philipp Rümmer

www.key-project.org

Tests and Proofs 2008

Prato, Italy
April 9th, 2008

Part I

Introduction

KeY Project Partners

University of Koblenz
Bernhard Beckert

University of Karlsruhe
Peter H. Schmitt

Chalmers University
Reiner Hähnle

www.key-project.org

What is this Tutorial about?

- Design & formal specification
- Deductive verification
- Testing

of

- Object-oriented software

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

Some Buzzwords about KeY

- Java Card as target language
- Integration with two standard SWE tools:
  - Borland Together, a commercial CASE tool
  - Eclipse, an open extensible IDE
- Specification languages
  - JML
  - UML/OCL
- Dynamic logic as program logic
- Verification = symbolic execution + induction
- Sequent style calculus + meta variables + incremental closure
- Interactive/automated prover with advanced UI

KeY Works With

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]

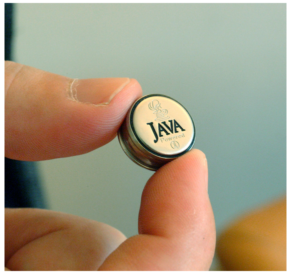
Rule Base

KeY Prover ⟷ Rule Base

Verifying Java Card Programs

What is Java Card?

- Sun's standard for smart cards and embedded devices
- Subset of Java, but with transaction concept



Verifying Java Card Programs

Why Java Card?

Good example for real-world object-oriented language

Java Card *lacks*

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

Application areas are

- security critical
- prone to financial risk

Formal Methods Integrated in KeY

Specification

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

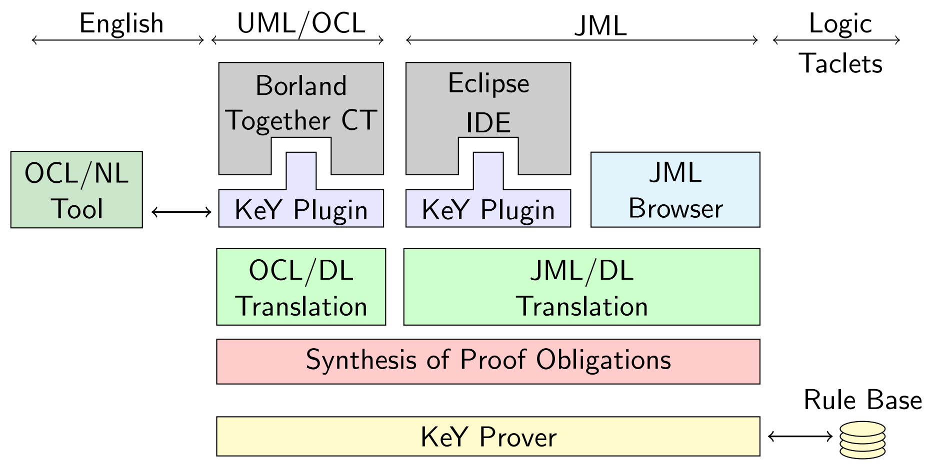
Verification

- Dynamic Logic
- Decision procedures

And . . .

- Static analysis
- Test case generation

Architecture of the KeY Tool



Part II

Specification

Part II

Specification

Design by Contract

Class

Invariant

Operation

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

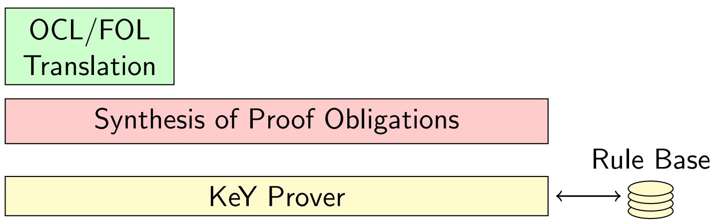
Part II

Specification

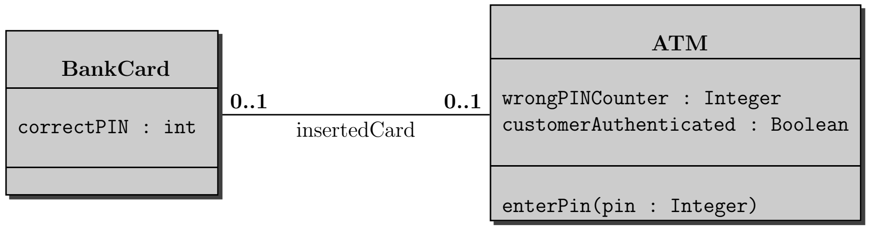
OCL: Object Constraint Language

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



A Typical UML Diagram



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

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.

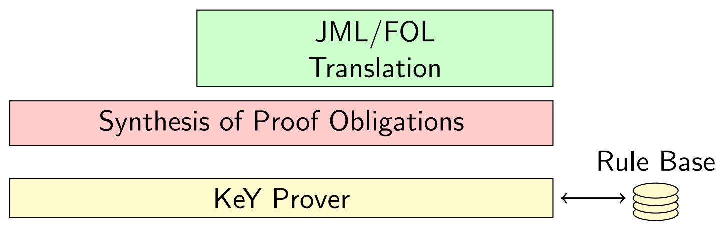
Part II

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)

```java
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)

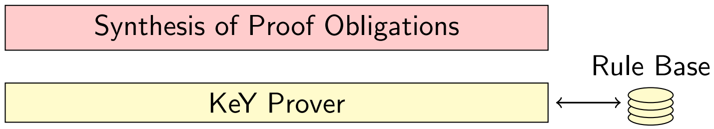
```java
public class ATM {

/*@ 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) { ...
  }
}
```


Proof Obligations

JML Proof Obligations

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



Part II

Specification

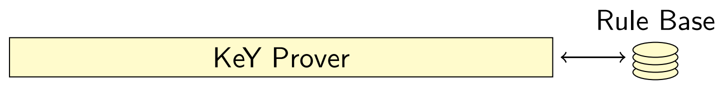
Total Correctness Statement

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

- precondition: in state s_0
- ⟨program⟩: in state s_0
- postcondition: in at least one state after termination

Specification in Dynamic Logic

```
\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 II

Specification

An Example Program `swapMax()`

```
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...

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

But we also need to show that executing...

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

gives us

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

So...

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;}
```

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)
- <p> 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

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

> - Programs are "first-class citizens"
> - No encoding of program syntax nor semantics into logic
> - Rule for each program construct in calculus

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

> Not merely partial/total correctness:
> - Correctness of program transformations
> - Security properties
> - Temporal extensions

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

> Class initialization much easier to specify with code

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

```
(balance > 1 & amount > 1) -> <charge(amount);>(balance > 1)
```

```
<x = 1;>([while (true) {}] false)
```

Syntax? ok

- Program formulas can appear nested

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

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))
```

Syntax? bad

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

```
<int x;>\forall int val;((<p>x = val) <-> (<q>x = val))
```

Syntax? ok

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

Rigid and Flexible Terms

Example

```
<int i;>\forall int x;(i + 1 = x -> <i++;>(i = x))
```

- Interpretation of i depends on computation state ⇒ flexible
- Interpretation of *x* and + must not depend on state ⇒ rigid

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

Semantics

Kripke semantics

- Semantics of a Java program is a partial function from states to states
- <*p*> *F* true in state *s* iff
  p terminates and *F* holds in the final state s'
- 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

$$\underbrace{\psi_1, \dots, \psi_m}_{\textit{Antecedent}} \quad \Longrightarrow \quad \underbrace{\phi_1, \dots, \phi_n}_{\textit{Succedent}}$$

where the ϕ_i, ψ_i are formulae (without free variables)

Semantics

Same as the formula

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

Sequent Rules

General form

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

($r = 0$ possible)

Soundness

If all premisses are valid, then the conclusion is valid

Some Simple Sequent Rules

$$\text{NOT\_LEFT}\ \frac{\Gamma \Longrightarrow A,\ \Delta}{\Gamma,\ !A \Longrightarrow \Delta}$$

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

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

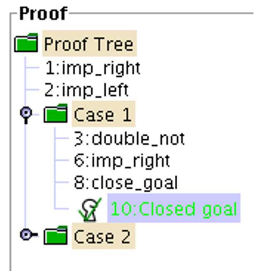
$$\text{ALL\_LEFT}\ \frac{\Gamma,\ \backslash\texttt{forall}\ t\ x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma,\ \backslash\texttt{forall}\ t\ x; \phi \Longrightarrow \Delta}$$

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	π
rest	ω

Proof by Symbolic Program Execution

- Sequent rules execute symbolically the active ($= 1^{\mathrm{st}}$) statement
- Sequent proof corresponds to symbolic program execution

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

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

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.

Integrating Verification and Testing: A Calculus for 100% Java Card

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

Integrating Verification and Testing: A Calculus for 100% Java Card

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

Integrating Verification and Testing: A Calculus for 100% Java Card

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

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

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)

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

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

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

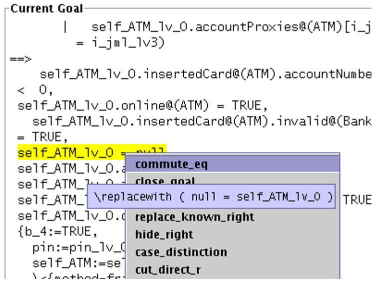
Typical workflow when proving in KeY (and other interactive provers)

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

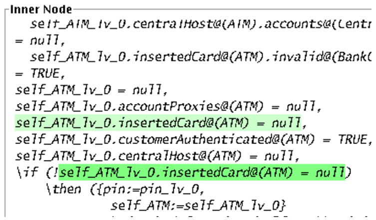
For goals (leaves of proof tree)

- 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

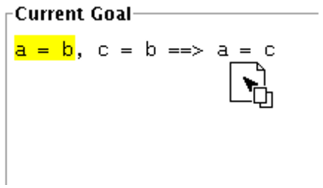
Main procedure for applying a taclet interactively

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

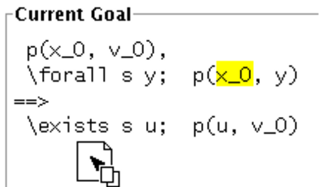
Applying equations

- 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

Integrating Testing and Verification

Part IV

Integrating Testing and Verification

Why Integrate?

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 bugs w.r.t. unspecified properties (e.g. timing)
- Tests are reusable after program changes

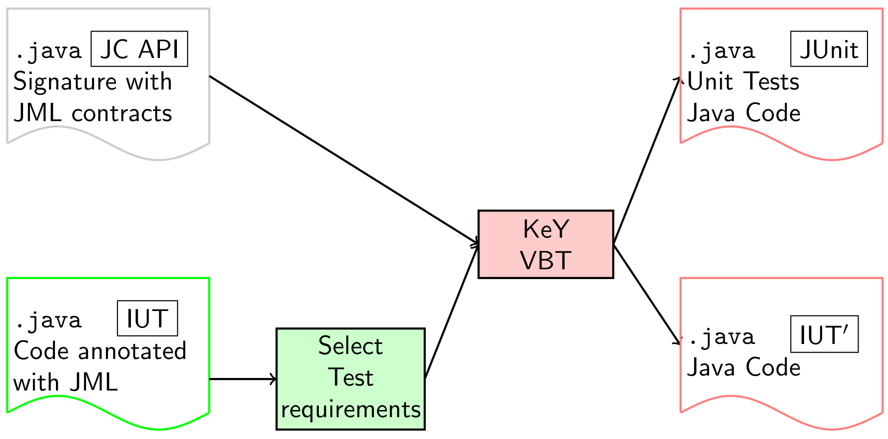
Idea: Use a formal proof to generate test cases

- KeY provides the path condition for each execution path
- High code coverage (feasible execution paths)
- Tests can be generated from incomplete proofs

Part IV

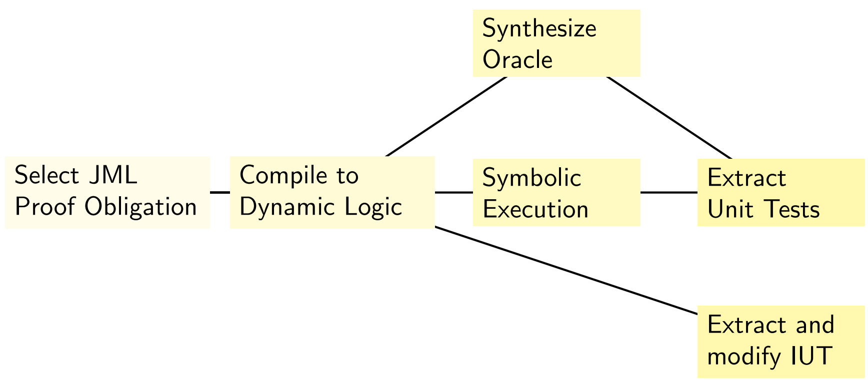
Integrating Testing and Verification

Verification-Based Test Generation: Overview

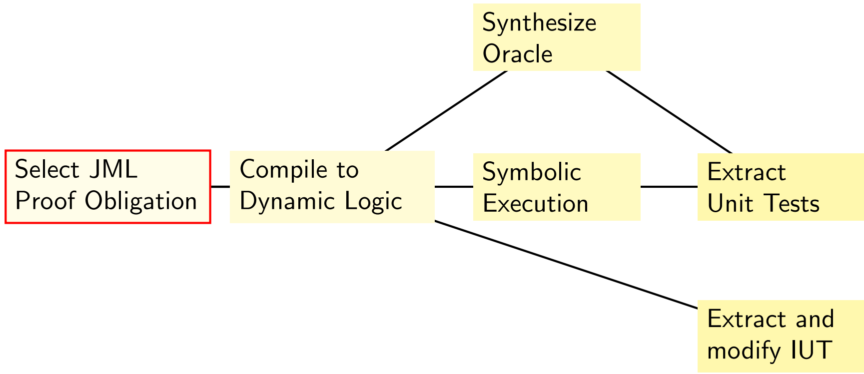


User input — Library — Automatically Generated

Verification-Based Test Generation Process



Verification-Based Test Generation Process

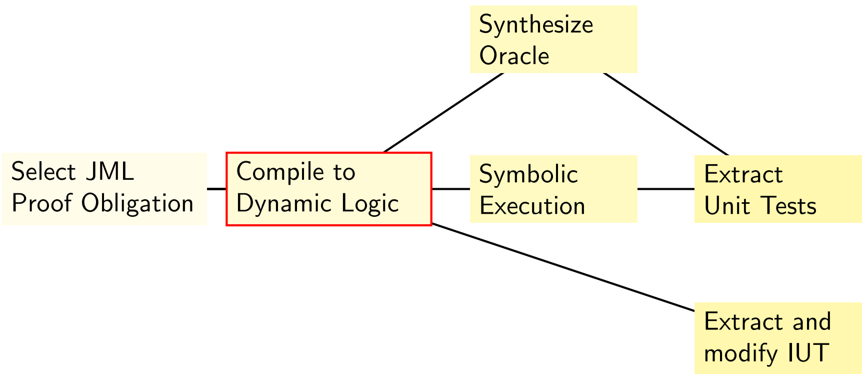


Example: Java Method with JML Specification

```
public class Middle{

    /*@ public normal_behavior
      @  ensures \result==x || \result==y || \result==z;
      @  ensures ...
      @*/
    public static int middle(int x, int y, int z){
        int mid = z;
        ...
    }
}
```


Verification-Based Test Generation Process



From Proof to Test

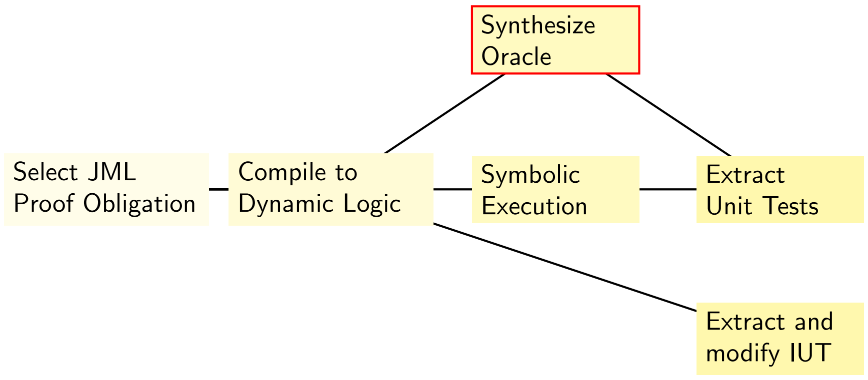
"Normalized" Proof obligation

$$\text{Pre} \Longrightarrow \mathcal{S}\langle\texttt{p}\rangle\,\text{Post}$$

Preparations

- Pre is a set of first-order formulas with preconditions, system invariant
- $\mathcal{S}$ is initial (symbolic) state at start of execution of p
- Extract IUT from $\mathcal{S}$ and p,
- Synthesize test oracle from finitely guarded first-order formula Post

Verification-Based Test Generation Process



Synthesize Oracle

Postcondition obtained from proof tree:

```
((_old0_x = _jmlresult8
        | (_old1_y = _jmlresult8 | _old2_z = _jmlresult8))
    & (  _old1_y >= _jmlresult8 & _old2_z >= _jmlresult8
        | (  _old0_x >= _jmlresult8 & _old2_z >= _jmlresult8
           | _old0_x >= _jmlresult8
          & _old1_y >= _jmlresult8))
    & (  _old1_y <= _jmlresult8 & _old2_z <= _jmlresult8
        | (  _old0_x <= _jmlresult8 & _jmlresult8 >= _old2_z
           | _jmlresult8 >= _old0_x
          & _jmlresult8 >= _old1_y)))
```

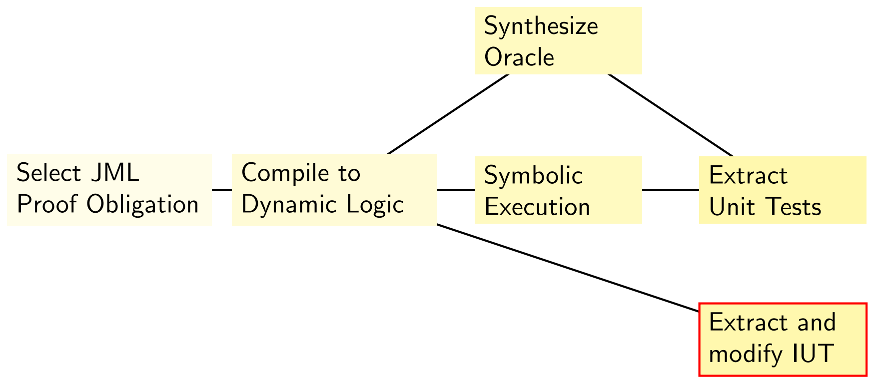
Directly translatable to a boolean expression

Synthesize Oracle

Test oracle:

```
((_old0_x == _jmlresult8
        || (_old1_y == _jmlresult8 || _old2_z == _jmlresult8))
    && (  _old1_y >= _jmlresult8 && _old2_z >= _jmlresult8
        || (  _old0_x >= _jmlresult8 && _old2_z >= _jmlresult8
           || _old0_x >= _jmlresult8
          && _old1_y >= _jmlresult8))
    && (  _old1_y <= _jmlresult8 && _old2_z <= _jmlresult8
        || (  _old0_x <= _jmlresult8 && _jmlresult8 >= _old2_z
           || _jmlresult8 >= _old0_x
          && _jmlresult8 >= _old1_y)))
```


Verification-Based Test Generation Process



Extract IUT – Preparations

State update $\mathcal{S}$

```
{_old0_x:=x_lv_0 ||
   _old1_y:=y_lv_0 ||
   _old2_z:=z_lv_0 ||
   x:=x_lv_0 ||
   y:=y_lv_0 ||
   z:=z_lv_0}
```

Program p

```
_jmlresult8=Middle.middle(x,y,z);
```

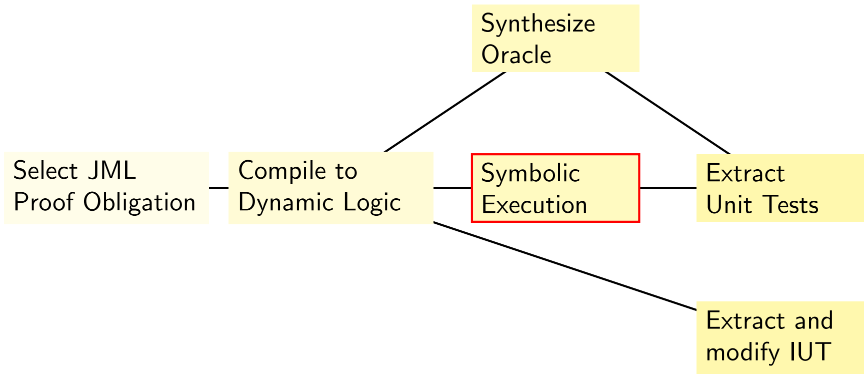

Extract IUT – Resulting Java Code

State update $\mathcal{S}$ is translated into a sequence of assignments

```
_old0_x=x_lv_0;
_old1_y=y_lv_0;
_old2_z=z_lv_0;
x=x_lv_0;
y=y_lv_0;
z=z_lv_0;
_jmlresult8=Middle.middle(x,y,z);
```

Export program context, add getter and setter methods for private fields

Verification-Based Test Generation Process



Symbolic Execution in Logic

Rules of Java Card DL Calculus that axiomatize program formulas implement symbolic execution

$$\textsc{ifElse}\ \frac{\Gamma,\ \mathcal{S}B \Longrightarrow \mathcal{S}\langle\pi\ \ p\ \ \omega\rangle\phi,\ \Delta \qquad \Gamma,\ !\mathcal{S}B \Longrightarrow \mathcal{S}\langle\pi\ \ q\ \ \omega\rangle\phi,\ \Delta}{\Gamma \Longrightarrow \mathcal{S}\langle\pi\ \texttt{if}\ (B)\ \{\ p\ \}\ \texttt{else}\ \{\ q\ \}\ \omega\rangle\phi,\ \Delta}$$

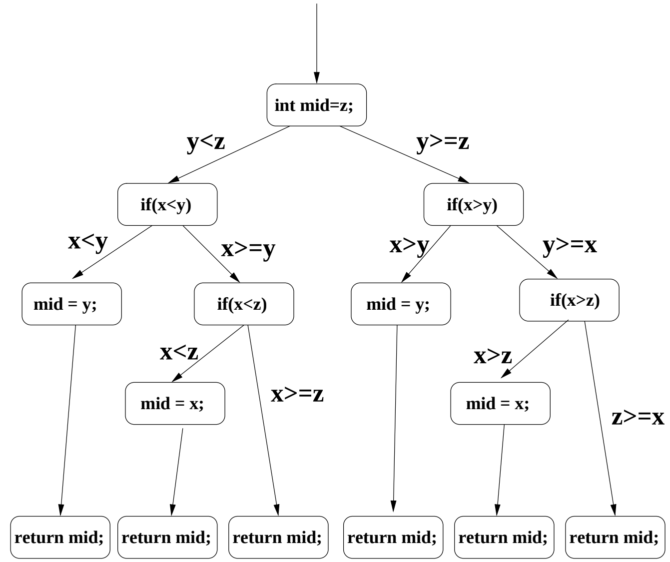
- Branch conditions $\mathcal{S}B$ and $!\mathcal{S}B$ are added to the sequent
- $PC := \bigwedge_{\gamma\in\Gamma}\gamma \wedge \bigwedge_{\delta\in\Delta}\neg\delta$ implies path condition of current path
- If the execution path is infeasable PC is invalid and thus $\Gamma \Longrightarrow \Delta$ valid
- Interleave first-order deduction and symbolic execution

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;
}
```


Symbolic Execution Tree of `middle()`



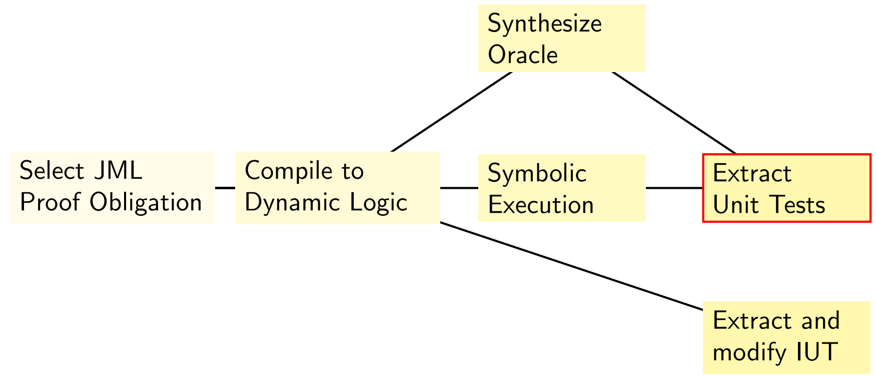
From Proof to Test, Cont'd

Test Generation

$$\mathsf{Pre} \Longrightarrow \mathcal{S}\langle \mathtt{p}\rangle\, \mathsf{Post}$$

1. Attempt verification of PO, construct proof tree
2. Search for exit nodes $\Gamma \Longrightarrow \mathcal{S}\langle\rangle\varphi, \Delta$ in the proof tree
   Search for abnormally terminating paths
   $\Gamma \Longrightarrow \mathcal{S}\langle\pi\ \mathtt{throw\ e;}\omega\rangle\varphi, \Delta$
3. Collect accumulated path conditions at these points; weaken
   $$PC := \bigwedge_{\gamma\in\Gamma}\gamma \wedge \bigwedge_{\delta\in\Delta}\neg\delta$$
4. Find first-order models of PC using, e.g., Simplify or Cogent
   Each model of each path condition yields a set of test data
5. Extract input variable assignment from found models

Verification-Based Test Generation Process



Generated JUnit Test Case

For every "feasible branch" in the proof tree one test method is generated

```
public void testmiddle<i>(){
   <local variable declarations>
   <creation of test data>
   foreach test data tuple:
   {
      <test data assignments>
      <IUT>
      <test oracle>
   }
}
```


Infinite Number of Execution Paths

Code cannot be symbolically executed entirely without using induction or loop invariants

Solution: Use Bounded Symbolic Execution

1. Perform a bounded number of proof steps
2. Unwind loops finite number of times, inline method bodies
3. Compute path conditions also for not yet terminated paths corresponding to leaves $\Gamma \Longrightarrow \mathcal{S}\texttt{<p'>}\varphi, \Delta$ of open branches in the proof tree

Bounded Symbolic Execution – Benefits and Shortcomings

Benefits

1. Test generation remains automatic while a formal proof would need interaction
2. In practice still a high code coverage

Shortcomings

1. No guarantees on code coverage

Example (Infinite Number of Execution Paths)

Determine Maximal Entry of Array

```
/*@ public normal_behavior
  @   ensures (\forall int i;
  @       0<=i && i<arr.length; arr[i]<=\result);
  @*/
public int getMax(int[] arr){
    int max = arr[0];
    for(int i=1; i<arr.length; i++){
        if(arr[i]<max) max = arr[i];
    }
    return max;
}
```

Example (Infinite Number of Execution Paths)

Path conditions for execution paths through the loop body needed

Loops are handled by bounded unwinding

$$\frac{\Gamma \Longrightarrow \mathcal{S}\langle\pi\ \texttt{l:if(c)\{l':\{b'\} while(c)\{b\}\}}\ \omega\rangle\varphi,\ \Delta}{\Gamma \Longrightarrow \mathcal{S}\langle\pi\ \texttt{while(c)\{b\}}\ \omega\rangle\varphi,\ \Delta}$$

Example (Infinite Number of Execution Paths)

Oracles for quantified formulas are needed

Quantified formulas in postcondition are evaluated using loops

```
(\forall int i; 0<=i && i<arr.length; arr[i]<=\result);
```

```
for(int i = 0; i<arr.length; i++){ ... }
```

Restrictions on the admissible quantification domain.

Part IV

Integrating Testing and Verification

Generating Tests from Loop Invariants and Method Specifications

Explicit execution of paths

- Fully automatic
- Limited unwinding of loops and of recursion steps of methods

Abstract/implicit execution of paths

- Requires (user provided) method specifications and loop invariants
- Full feasible branch coverage possible

Example 1: Branch after a Loop

```
void foo1(int n){
  int i=0;
  while(i < n*2){
    n+=2;
    i+=8;
    if(..){..}else{..}
  }
  if(i>=64){
    C();
  }
}
```

Loop invariant rule (simplified)

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{U\}I,\ \Delta \\ \Gamma \Longrightarrow \{U\}\ \{M\}\ (I \wedge lc \rightarrow [\mathtt{b}]\ I),\ \Delta \\ \color{red}{\Gamma \Longrightarrow \{U\}\ \{M\}\ I \wedge \neg lc \rightarrow [\pi\ \ \omega]\ \phi,\ \Delta}\end{array}}{\Gamma \Longrightarrow \{U\}\ [\pi\ \mathtt{while(lc)\{b\}}\ \ \omega]\ \phi,\ \Delta}$$

Result

Using the invariant:
$$\frac{4+i}{1+n-n_{pre}} = 4$$
KeY computed test data with: $n = 33$

Example 2: Branch within a Loop

```
void foo2(int n){
  int i=0;
  while(i < n*2){
    n+=2;
    if(i>=64){
      C();
    }
    i+=8;
  }
}
```

Loop invariant rule (simplified)

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{U\}I,\ \Delta \\ \color{red}{\Gamma \Longrightarrow \{U\}\ \{M\}\ (I \wedge lc \rightarrow [\mathtt{b}]\ I)\Delta} \\ \Gamma \Longrightarrow \{U\}\ \{M\}\ I \wedge \neg lc \rightarrow [\pi\omega]\ \phi,\ \Delta\end{array}}{\Gamma \Longrightarrow \{U\}\ [\pi\ \mathtt{while(lc)\{b\}}\ \ \omega]\ \phi\Delta,}$$

Result

Using the invariant:
$$\frac{4+i}{1+n-n_{pre}} = 4$$
We get: $n = 34$

Example 3: Branch after a Method Call

```
class Foo{
 int i;

 void foo(int n){
   D(n);
   if(i==20){ C(); }
 }

 /*@ requires i<n;
   @ assignable i;
   @ ensures i==n; */
 void D(int n){ while(i<n)...}
}
```

Method contract rule (simplified)

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{U\}\ \{T\}\ Pre,\ \Delta \\ \color{red}{\Gamma \Longrightarrow \{U\}\ \{M \parallel T\}\ Post \rightarrow [\pi\omega]\ \phi,\ \Delta}\end{array}}{\Gamma \Longrightarrow \{U\}\ [\pi\ \mathtt{m(t1,..,tn);}\ \ \omega]\ \phi\Delta,}$$

More Generally

Testing Tasks

- Branch after a loop
- Branch within a loop
- Branch after a method call

(on Friday ...)

- How to compute the precondition more generally
- Required properties of must the specification or invariant

Part IV

Integrating Testing and Verification

13 Why Integrate?

14 Test-Case Generation by Bounded Symbolic Execution

15 Test-Case Generation from Method Specifications and Loop Invariants

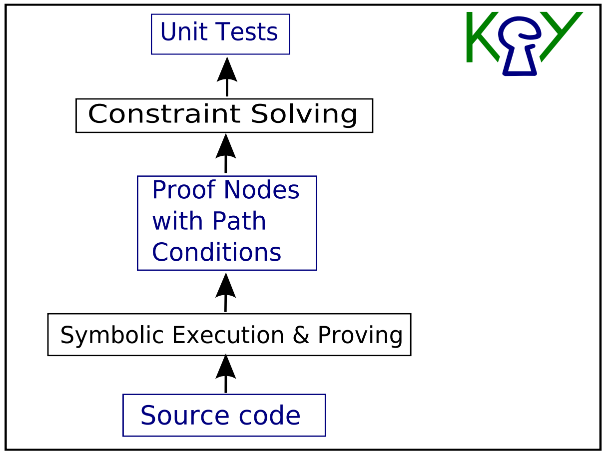
16 White-box testing by Combining Specification Extraction and Black-box testing

17 Proving Incorrectness of Programs

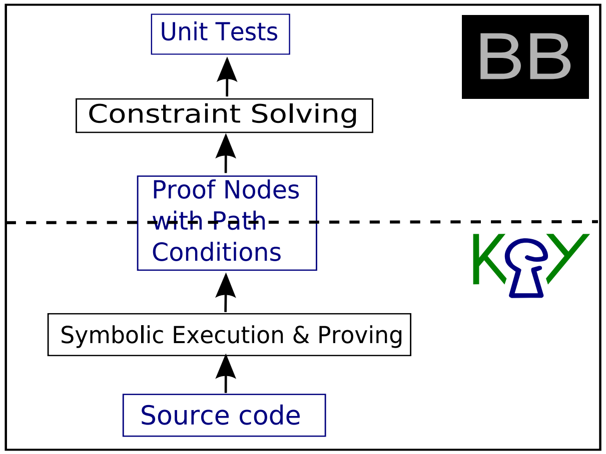
Next: Combining KeY with other Testing Tools

KeY can extend Black-box Testing Tools

Previous Approach



Next Appraoch



Benefits

- Using of existing Black-box Testing Tools for White-box testing
- Separation of concerns - Modulariy
- Combination of Coverage Criteria

Two Kinds of Specifications

Requirement Specificaiton

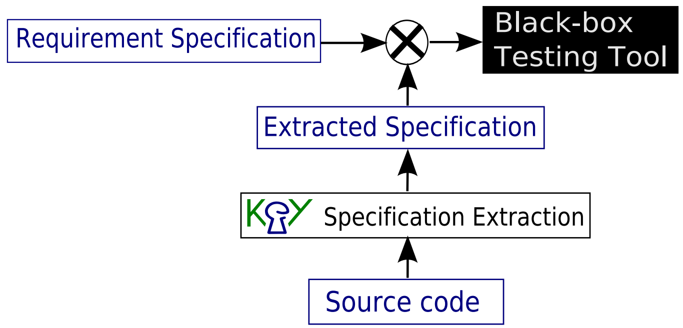
- Given by the user
- Role: To be tested or verified

Extracted Specifictaion

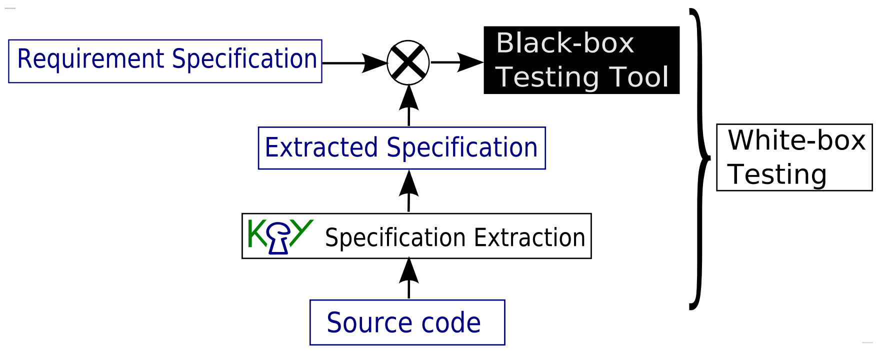
- Is extracted automatically
- Complies with the IUT by construction
- Reflects the structure of the program

Tool Chain

Tool Chain



Tool Chain



Example IUT

```
/*@ public normal_behavior
  @ ensures \result == ((a<b? a : b)<0 ?
  @                    -(a<b? a : b) :
  @                     (a<b? a : b));  @*/

  public static int absMin ( int  a, int  b) {
    int result = b;
    if (a<b) { result=a; }
    if (result<0) { result=-result;}
    return result;
  }
```


Specification Extraction

Proof Obligation

$\Gamma, requires \rightarrow \texttt{<absMin(a,b)>}\ Dummy$

Closed Proof Branches

$$\frac{a = 1, a = 2 \Rightarrow \texttt{<A...>}\ \phi \qquad a = 1, a \neq 2 \Rightarrow \texttt{<B...>}\ \phi}{a = 1 \Rightarrow \texttt{<if(a==2)\{A\}\{B\}...>}\ \phi}$$

Open Proof Branches

$$\overbrace{a <= -1, b <= -1 + a}^{\text{for precondition}} \Rightarrow \{\overbrace{result := -a}^{\text{for postcondition}}\}Dummy$$

$$a >= 0, b <= -1 + a \Rightarrow \{result := a\}Dummy$$

$$b <= -1, a <= -1, b >= a \Rightarrow \{result := -b\}Dummy$$

$$b >= 0, b >= a \Rightarrow \{result := b\}Dummy$$

Combined Specification

```
/*@ public normal_behavior
@ requires true;
@ ensures \result == ((a <  b ? a : b) <  0 ?
@               -((a <  b ? a : b)):
@               (a <  b ? a : b));
@ also
@ requires true && b <= -1 && a <= -1 && b >= a;
@ ensures \result == \old((b * -1));
@ also
@ requires true && b >= 0 && b >= a;
@ ensures \result == \old(b);
@ also
...
@*/
```


Specification Extraction + Black-box Testing = White-box Testing

Properties of this Approach

Black-box Tools

Tool requirements

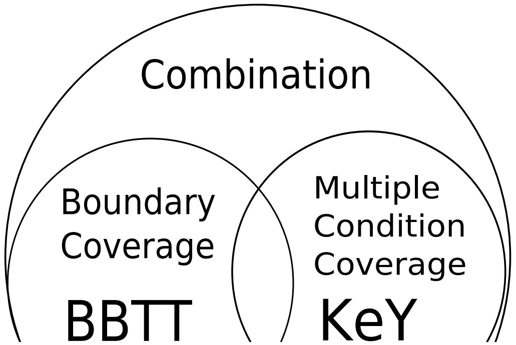
- JML Support
- Derive tests based on method preconditions
- Ensure coverage of specification
- Generate tests automatically

JML supporting Black-box Tools

BB-tools for JML

- JET / UTJML — no coverage guarantees
- Korat — based on class invariants
- JmlAutoTest — implementation is lost
- JmlTT — specification animator, limited test generation support
- jmlunit — generates only the oracle
- jtest — does not generate test data

Combinations of Coverage Criteria



Using the extracted Post Condition

- Requirement Specification

```
/*@  requires true;
  @  ensures  \result!=23;   @*/
  absMin(int a, int b){...}
```

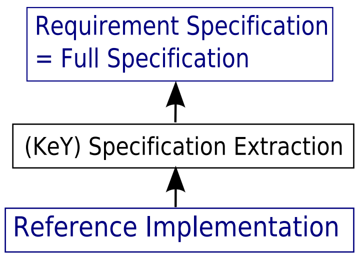
- With Full Specification

```
/*@ requires true;
  @ ensures \result!=23;
  @ also...
  @ requires b >= 0 && b >= a;
  @ ensures \result == \old(b) @*/
```

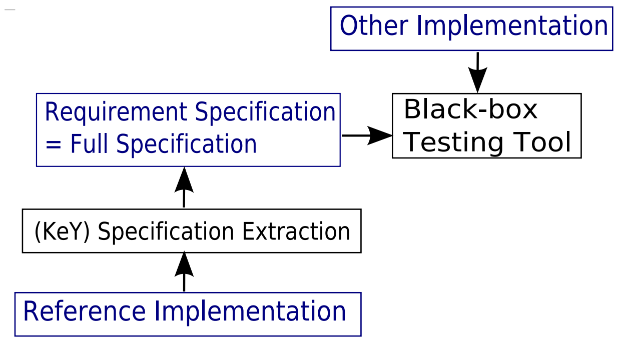
- Simplified

```
/*@ requires b >= 0 && b >= a;
  @ ensures  23 != \old(b);
  @  ...  @*/
```


Requirement Specification from a Reference Implementation



Requirement Specification from a Reference Implementation



Part IV

Integrating Testing and Verification

Proving Incorrectness of Programs

Java DL can also directly express program incorrectness

- Basically: proving with negated input formula + quantification
  ⇒ Symbolic search for inputs that make post-condition fail
- Difference to generating test cases:
  Both path-constraints and post-condition are considered
  (Only "failing test cases" are found)
- Constraint solving by KeY itself
- Symbolic reasoning ⇒ "symbolic test cases" can be found

Proving Incorrectness of Programs (2)

More generally, this covers:

- Reachability analysis:
  Can a program reach certain states?
- Inversion:
  Which pre-states lead to certain post-states?
  (i.e., construct models of the weakest pre-condition)
- Non-termination analysis (more details later)

Example: Bug in Binary Search

```
public int binSearch(int []ar, int target) {
  if (ar.length == 0) return -1;

  int hi = ar.length;
  int lo = 0;
  while (true) {
    int centre = (hi + lo) / 2;
    if (centre == lo) {
      if (ar[centre] == target) return centre;
      else if (ar[centre+1] == target) return centre+1;
      else return -1;
    }

    if (ar[centre] < target) lo = centre;
    else if (target < ar[centre]) hi = centre;
    else return centre;
} }
```

Example: Bug in Binary Search (2)

Pre-condition: array is sorted:

$$ar \neq null \ \wedge \ \forall\, i : nat.\ (i < (ar.length - 1) \rightarrow ar[i] \leq ar[i+1])$$

Post-conditions: result is desired index:

$$result \not\doteq -1 \ \rightarrow \ (0 \leq result < ar.length \ \wedge \ ar[result] \doteq target)$$
$$result \doteq -1 \ \rightarrow \ \forall\, i : nat.\ (i < ar.length \rightarrow ar[i] \neq target)$$

Dis-verification condition:

$$\exists\, pre\text{-}state.\ \neg\big(pre \rightarrow \langle\ \texttt{binSearch}(ar, target)\ \rangle\ \ post\big)$$

Example: Bug in Binary Search (3)

When proving the formula, KeY produces a *constraint* that describes critical inputs (automatically):

$$[\, ar.length \doteq 1 \wedge ar[0] \not\doteq target \,]$$

Result: program behaves wrongly whenever

- the length of the given array is 1, and
- the searched number is not in the array.

Schema for Characterising Incorrectness in DL

$$\exists\, pre\text{-}state.\ \{pre\text{-}state\}$$
$$\neg\big(pre\text{-}conditions \rightarrow \langle\ program\ code\ \rangle\ \ post\text{-}conditions\big)$$

This formula holds if:

- *pre-state* satisfies the pre-conditions, and
- the program does not terminate, or
- terminates but violates the post-conditions.

Pre-state quantification has to cover:

- local variables, class attributes,
- instance attributes, arrays, number of allocated objects.

Technically:

- "Update" is needed for making *pre-state* active

Reasoning about Incorrectness Conditions

$$\exists\, pre\text{-}state.$$
$$\neg\big(pre\text{-}conditions \rightarrow \langle\ program\ code\ \rangle\ \ post\text{-}conditions\big)$$

How to eliminate $\exists\, pre\text{-}state$?

- In KeY: Metavariables + constraint solving
  $\rightarrow$ Backtracking-free proving
  $\rightarrow$ Systematic search for constraints that close a proof

Extension: Non-Termination Detection

Required: KeY + Invariant generator

- As before, incorrectness can be expressed in DL:
  "Program p does not terminate for some pre-state"

$$\exists\, \textit{pre-state}.\ \neg\big(\textit{pre-conditions} \rightarrow \langle\ \textit{program code}\ \rangle\ \textit{true}\big)$$

- In the proof, a non-termination invariant is required
  ⇒ Program cannot reach terminal states
  ⇒ Invariant generator needed as extension to KeY
- Techniques to construct invariants in our approach:
  ⇒ Invariant templates containing metavariables
  ⇒ Refinement based on failed proof attempts

(more information in the talk on Friday)

Example: Gaussian Sum

```
int n = [...];

int sum = 0;

while (n != 0) {
  sum += n;
  n--;
}
```

Problem:

What happens if `n` has a negative value?

A common programming error.

KeY + Invariant generator can prove non-termination automatically:

- Constraint on initial state: $\mathtt{n} < 0$
- Loop invariant (intermediate states): $\mathtt{n} < 0$
  → Found in few iterations (2–4, depending on settings)

Part V

Further Topics

Part V

Further Topics

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

$$\frac{\Gamma,\ \phi,\ \psi \Longrightarrow \Delta}{\Gamma,\ \phi,\ \phi \mathrel{-\!\!>} \psi \Longrightarrow \Delta}$$

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

$$\frac{B = \mathtt{TRUE} \Longrightarrow \langle\pi\ p\ \omega\rangle F \qquad B = \mathtt{FALSE} \Longrightarrow \langle\pi\ q\ \omega\rangle F}{\Longrightarrow \langle\pi\ \mathtt{if}\ \ (B)\ \ p\ \mathtt{else}\ q\ \omega\rangle F}$$

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 V

Further Topics

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!)

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

But ...

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

Correct rule

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

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 V

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

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

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 V

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 V

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...

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

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 $n \geq MAX\_INT$

Concurrency Verification Problems

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

Alas. . .

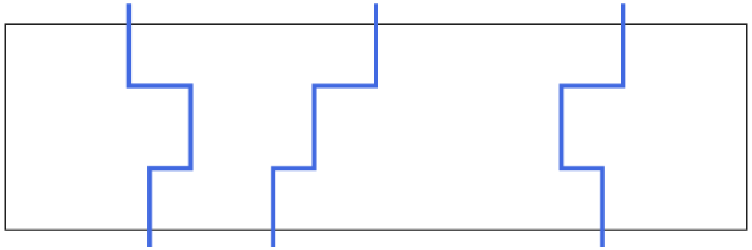
No thread identities **in** programs

No dynamic thread creation (but unbounded concurrency)

Currently only atomic loops

The Calculus Is Built On. . .

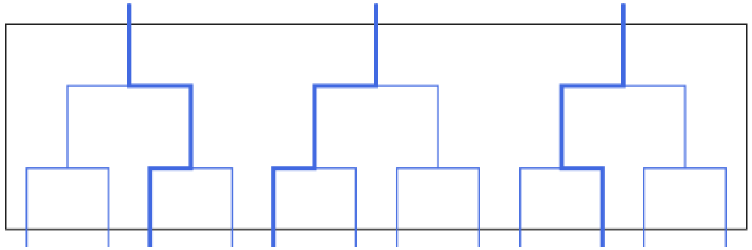
symmetry reduction



. . . and explicit scheduler formalization

The Calculus Is Built On. . .

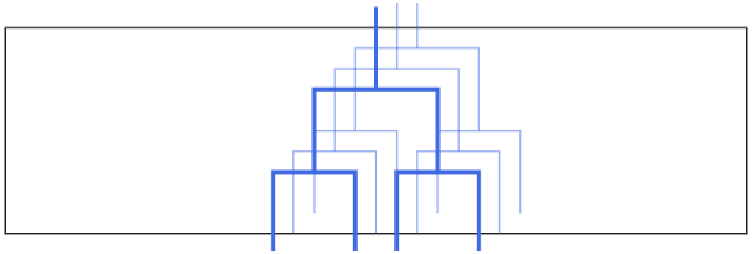
symmetry reduction



. . . and explicit scheduler formalization

The Calculus Is Built On. . .

symmetry reduction



. . . and explicit scheduler formalization

Part VI

Wrap Up

Part VI

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 VI

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 VI

Wrap Up

Acknowledgments

Funding agencies

- Deutsche Forschungsgemeinschaft (DFG)
- Deutscher Akademischer Austauschdienst (DAAD)
- Vetenskapsrådet (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 (U Oslo), W. Mostowski (U Nijmegen), A. Roth (SAP), S. Schlager (SAP)

Colleagues who collaborated with us

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

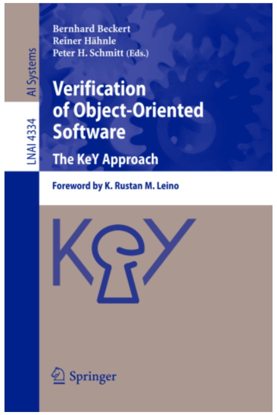
More Information

The KeY Book

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

Verification of Object-Oriented Software: The KeY Approach

Springer-Verlag, LNCS 4334, 2007.



Web site

www.key-project.org

Part VI

Wrap Up