

Tutorial

The KeY Approach to Deductive Verification of Object-Oriented Programs

Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle,
Philipp Rümmer, Peter H. Schmitt

www.key-project.org

5th International Symposium on
Formal Methods for Components and Objects

Amsterdam, NL
November 9, 2006


What is this Tutorial all About?

It is about an *approach* and *tool* for the

- Design
- Formal specification
- Deductive verification

of

- OO software

The *approach*, *tool*, and *project* is named 

in the following: 'KeY'

KeY Project Partners



University of
Karlsruhe (TH)

Peter H. Schmitt

Richard Bubel
Steffen Schlager
Isabel Tonin

CHALMERS

Chalmers UT
Göteborg

Reiner Hähnle

Wolfgang Ahrendt
Daniel Larsson
Philipp Rümmer
Angela Wallenburg



University of
Koblenz-Landau

Bernhard Beckert

Gerd Beuster
Christoph Gladisch
Vladimir Klebanov

Some Buzzwords Early On

- **Java** as target language
- **Dynamic logic** as program logic
- Verification = **symbolic execution** + **induction**
- **Sequent style** calculus + meta variables + incremental closure
- Prover is **interactive** + **automated**
- Integration with two standard SWE tools:
 - **TogetherCC**, a commercial CASE tool
 - **Eclipse**, an open extensible IDE
- Specification languages
 - **JML**
 - **OCL/UML**
- **Smart cards** as main target application

Part I

Overview of the KeY system

First Demo

Supported Specification Languages: OCL

Object Constraint Language

Part of the OMG standard UML

Scope:

Add formal constraints to UML (class) diagrams

Supported Specification Languages:

JML

Java Modeling Language

Behavioral interface specification language for Java

International community effort lead by Gary T. Leavens, Iowa State
building on the Larch approach

Comes with assertion and runtime checkers

OCL and JML

both

- specify method behaviour: pre/post conditions
- specify admissible states: class invariants
- essentially full first order
- support inter-object navigation

differences

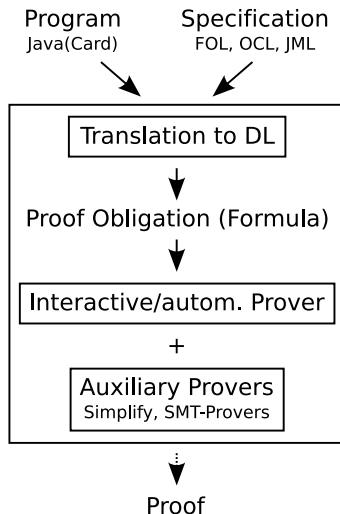
- OCL model oriented:
 - attached to class diagrams
 - 'talks' UML
- JML implementation oriented:
 - attached to Java programs
 - 'talks' Java
 - specifies exceptional behaviour also
- JML only: restricting scope of side effects

JML example

```
/*@ public normal_behavior
   @ requires a != null;
   @ ensures (\forall int j; j >= 0 && j < a.length;
             @           \result >= a[j]);
   @ ensures a.length > 0 ==>
   @       (\exists int j; j >= 0 && j < a.length;
             @           \result == a[j]);
   @*/

public static /*@ pure @*/ int max(int[] a) {
    if ( a.length == 0 ) return 0;
    int max = a[0], i = 1;
    while ( i < a.length ) {
        if ( a[i] > max ) max = a[i];
        ++i;
    }
    return max;
}
```

KeY Architecture



Part II

Logic and Calculus

Java DL: A Dynamic Logic for Java

Syntax

- Basis: sorted first-order logic
- Modal operators $\langle p \rangle$ and $[p]$ for each sequence p of Java statements

Semantics

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

Java DL formulas contain Java source code

Java DL: A Dynamic Logic for Java

Syntax

- Basis: sorted first-order logic
- Modal operators $\langle p \rangle$ and $[p]$ for each sequence p of Java statements

Semantics

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

Java DL formulas contain Java source code

First-Order Formula Syntax

Logical operators

$\&$ and

$|$ or

\rightarrow implication

\leftrightarrow equivalence

$!$ negation

Logical constants

true

false

Quantifiers

\forall forall

\exists exists

Dynamic Logic Example Formulas

$(\text{balance} \geq c \ \& \ \text{amount} > 0) \rightarrow \langle \text{charge}(\text{amount}); \rangle \text{balance} > c$

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

- p, q **equivalent** relative to x

Dynamic Logic Example Formulas

```
a != null & a.length = 3
->
  \<
    int max = a[0];
    for ( int i = 1; i < a.length; ++i ) {
      if ( a[i] > max ) max = a[i];
    }
  \>
  (
    max >= a[0] & max >= a[1] & max >= a[2]
    &
    ( max = a[0] | max = a[1] | max = a[2] )
  )
```

Features of Dynamic Logic

- Transparency wrt target programming language

- Enables 'symbolic execution' style proving (natural for *interactive* proof)
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Employ programs for specification
- Fits nicely into Gentzen-style sequent calculi

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

Features of Dynamic Logic

- Transparency wrt target programming language
- Enables 'symbolic execution' style proving
(natural for *interactive proof*)
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Employ programs for specification
- Fits nicely into Gentzen-style sequent calculi

Features of Dynamic Logic

- Transparency wrt target programming language
- Enables 'symbolic execution' style proving (natural for *interactive* proof)
- **Encompasses Hoare Logic**
- More expressive and flexible than Hoare logic
- Employ programs for specification
- Fits nicely into Gentzen-style sequent calculi

Hoare triple $\{\psi\} \alpha \{\phi\}$ equiv. to DL formula $\psi \rightarrow [\alpha] \phi$

Features of Dynamic Logic

- Transparency wrt target programming language
- Enables 'symbolic execution' style proving (natural for *interactive* proof)
- Encompasses Hoare Logic
- **More expressive and flexible than Hoare logic**
- Employ programs for specification
- Fits nicely into Gentzen-style sequent calculi

On top of partial/total correctness:

- Correctness of program transformations
- Security properties (two runs are indistinguishable)
- Extension friendly (e.g. temporal modalities)

Features of Dynamic Logic

- Transparency wrt target programming language
- Enables 'symbolic execution' style proving (natural for *interactive* proof)
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- **Employ programs for specification**
- Fits nicely into Gentzen-style sequent calculi

e.g. assert 'non-cyclic' using a program

Features of Dynamic Logic

- Transparency wrt target programming language
- Enables 'symbolic execution' style proving (natural for *interactive* proof)
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Employ programs for specification
- Fits nicely into Gentzen-style sequent calculi

Suitable for interactive/automatic proving

DL Sequents

Syntax

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

where the ϕ_i, ψ_i are closed DL formulae

Semantics

the above sequent *means*:

$$\psi_1 \& \dots \& \psi_m \rightarrow \phi_1 \mid \dots \mid \phi_n$$

Sequent Rules

General form

$$\text{RULE NAME} \frac{\overbrace{\Gamma_1 \implies \Delta_1 \quad \dots \quad \Gamma_r \implies \Delta_r}^{\text{Premises}}}{\underbrace{\Gamma \implies \Delta}_{\text{Conclusion}}}$$

($r = 0$ possible)

Soundness

If all premisses are valid, then the conclusion is valid

Application

In order to prove a goal matching the conclusion, prove the (instantiated) premisses

Some Simple Sequent Rules

$$\text{NOT_LEFT} \frac{\Gamma \implies \phi, \Delta}{\Gamma, !\phi \implies \Delta}$$

$$\text{IMP_LEFT} \frac{\Gamma \implies \phi, \Delta \quad \Gamma, \psi \implies \Delta}{\Gamma, \phi \rightarrow \psi \implies \Delta}$$

$$\text{CLOSE_GOAL} \frac{}{\Gamma, \phi \implies \phi, \Delta}$$

$$\text{CLOSE_BY_TRUE} \frac{}{\Gamma \implies \text{true}, \Delta}$$

$$\text{ALL_LEFT} \frac{\Gamma, \backslash \text{forall } t \ x; \phi, \{x/e\}\phi \implies \Delta}{\Gamma, \backslash \text{forall } t \ x; \phi \implies \Delta}$$

where e ground term of type $t' \preceq t$

Proof by Symbolic Program Execution

If-then-else rule (simplified)

$$\frac{\Gamma, B \implies \langle p \ \omega \rangle \phi, \Delta \quad \Gamma, !B \implies \langle q \ \omega \rangle \phi, \Delta}{\Gamma \implies \langle \text{if } (B) \{ p \} \text{ else } \{ q \} \ \omega \rangle \phi, \Delta}$$

Complicated statements are simplified first, e.g.

$$\frac{\Gamma \implies \langle v=y; y=y+1; x=v; \omega \rangle \phi, \Delta}{\Gamma \implies \langle x=y++; \omega \rangle \phi, \Delta}$$

Simple assignment rule

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

Extending DL by Explicit State Updates

Updates

explicit syntactic elements in the logic

Elementary Updates

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

where

- loc a program variable x , an attribute access $o.attr$, or an array access $a[i]$
- val a logical term (no side effects)
- ϕ is a DL formula

Parallel composition of updates

$$\{loc_1 := val_1 \parallel \dots \parallel loc_n := val_n\}\phi$$

Program states represented as FO structures

Java DL considers program symbols as first-order vocabulary

| | | |
|-----------------------------|---|------------------|
| Variables, class attributes | → | Constants |
| Instance attributes | → | Unary functions |
| Arrays | → | Binary functions |

Example

```
class C { int a; C[] b; }
```

- Class C represented by the function symbols:

$$\begin{aligned} a : C &\rightarrow \text{int} & ar_C : C[] &\rightarrow \text{int} \rightarrow C \\ b : C &\rightarrow C[] \end{aligned}$$

Here: no control information (program counter, thrown exceptions)

Update Demo

Update Handling in the Calculus

- Updates are collected and **parallelised**
- **Eager** simplification among updates
- **Lazy** application (i.e. substitution) to postcondition

Further update constructors in Java DL

Guards, quantification

Handling Abrupt Termination

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

Example

TRY-THROW (exc simple)

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

Handling Abrupt Termination

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

Example

TRY-THROW (exc simple)

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

Proof by Symbolic Program Execution

Active Statements

Example

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

| | |
|-------------------|-------------------|
| active statement | <code>i=0;</code> |
| non-active prefix | π |
| rest | ω |

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

Proof by Symbolic Program Execution

If-then-else rule (simplified)

$$\frac{\Gamma, B \implies \langle \pi \ p \ \omega \rangle \phi, \Delta \quad \Gamma, !B \implies \langle \pi \ q \ \omega \rangle \phi, \Delta}{\Gamma \implies \langle \pi \ \text{if } (B) \{ p \} \ \text{else } \{ q \} \ \omega \rangle \phi, \Delta}$$

Complicated statements are simplified first, e.g.

$$\frac{\Gamma \implies \langle \pi \ v=y; y=y+1; x=v; \omega \rangle \phi, \Delta}{\Gamma \implies \langle \pi \ x=y++; \omega \rangle \phi, \Delta}$$

Simple assignment rule

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

Invariant Rule

Classical Invariant Rule

$$\text{INV} \frac{\Gamma \implies I, \Delta \quad I, B \vdash [p]I \quad I, !B \vdash \phi}{\Gamma \implies [\text{if } (B) \{ p \}] \phi, \Delta}$$

Differences in Key

- Invariant consists of 2 parts:
Formula I + Locations Mod that can be modified by loop
- Possible abrupt termination of loop handled

Total Correctness (Diamond Formulas) handled with Variants

- Each iteration decreases the value of a non-negative expression t

JML example

```
/*@ public normal_behavior
   @ requires a != null;
   @ ensures (\forall int j; j >= 0 && j < a.length;
   @           \result >= a[j]);
   @ ensures a.length > 0 ==>
   @         (\exists int j; j >= 0 && j < a.length;
   @           \result == a[j]);
   @*/

public static /*@ pure @*/ int max(int[] a) {
    if ( a.length == 0 ) return 0;
    int max = a[0], i = 1;
    while ( i < a.length ) {
        if ( a[i] > max ) max = a[i];
        ++i;
    }
    return max;
}
```

JML example: Invariant

```
...
public static /*@ pure @*/ int max(int[] a) {
    if ( a.length == 0 ) return 0;
    int max = a[0], i = 1;
    /*@
        @ loop_invariant
        @     i <= a.length
        @     &&
        @     (\forall int j; j >= 0 && j < i; max >= a[j])
        @     &&
        @     (\exists int j; j >= 0 && j < i; max == a[j]);
        @ modifies i, max;
        @ decreases a.length - i;
    @*/
    while ( i < a.length ) {
        ...
    }
}
```

Third Demo

Components of the Calculus

- 1 Non-program rules
 - first-order rules
 - rules for data-types (primarily: arithmetic)
 - rules for modalities
- 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
 - unwinding + induction
- 4 Rules for replacing a method invocations by the method's contract
- 5 Update simplification

Coverage of Java features

The calculus covers:

- method invocation, dynamic binding
- polymorphism
- abrupt termination
- checking for nullpointer exceptions
- object creation and initialisation
- arrays
- finiteness of integer data types
- transactions (Java Card)

By that, KeY covers the full 'Java Card' language.

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)

Implementing Rules: Taclets

Uniform language for different classes of rules

- First-order calculus
- Specific to Java DL: symbolic execution for Java
- Axioms of theories: arithmetic, lists, etc.
- Lemmas

Simple, high-level language

- Adding, modifying, and removing formulas
- Conditions restricting applicability of rules
- No complex features like loops
- Suitable both for interactive and automated systems
- Lemmas are validated wrt. base taclets

Taclet Syntax (by Example)

Modus ponens: Rule

$$\frac{\Gamma, \phi, \psi \implies \Delta}{\Gamma, \phi, \phi \rightarrow \psi \implies \Delta}$$

Modus ponens: Taclet

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

Rule if_else_split

$$\frac{\begin{array}{l} \Gamma, B = \text{TRUE} \implies \langle \pi \ \alpha_1; \omega \rangle F, \Delta \\ \Gamma, B = \text{FALSE} \implies \langle \pi \ \alpha_2; \omega \rangle F, \Delta \end{array}}{\Gamma \implies \langle \pi \ \text{if } (B) \ \alpha_1 \ \text{else } \alpha_2; \omega \rangle F, \Delta}$$

where B is a Boolean expression without side effects

Corresponding taclet

```
if_else_split {
  \find (==> <{.. if(#se) #s0 else #s1 ...}>post)
  \replacewith (==> <{.. #s0 ...}>post) \add (#se = TRUE ==>);
  \replacewith (==> <{.. #s1 ...}>post) \add (#se = FALSE ==>);
  \heuristics(if_split)
};
```

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
- Tactlets 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

Modifying the proof tree

- Extension:
Through application of rules to goals
- Closure:
Through application of closure rules
- Pruning:
Deletion of subtrees

Extension of Proof: Application of Single Taclets

Application of a taclet requires

- A proof goal
- Focus of rule application: term/formula
(part of sequent that can be modified by rule)
- Instantiation of schema variables of taclet

Ways to provide this information

- Selection of application focus and taclet with mouse pointer
- Instantiation dialog for taclets
(e.g., specify an induction hypothesis)
- Drag'n'drop
(e.g., applying equations, instantiating quantified formulas)

Means of Automation Implemented in KeY

Global “Strategies” for automatically applying rules in series

- Symbolic execution
- Complete first-order reasoning (uses free-variable calculus)
- Different arithmetic procedures (e.g., Omega, Buchberger)

Invocation of external theorem provers, decision procedures

- Simplify (from ESC/Java)
- ICS
- Export to SMT-LIB format

Part III

Further Topics

User can choose among 3 Integer Semantics

Java Integers

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

Arithmetic Integers

- Leads to easier verification problems
- Incorrect

Arithmetic Integers with overflow check

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

Proving Programs Incorrect

Java DL can express incorrectness of programs

\exists *pre-state*.

$\neg(\textit{pre-conditions} \rightarrow \langle \textit{program code} \rangle \textit{post-conditions})$

This formula holds if the program (for valid input)

- does not terminate, or
- terminates but violates the post-conditions.

Proving incorrectness

- Metavariables + Java DL calculus
used to systematically construct pre-state (classes of pre-states)
- Similarities to symbolic testing
- Constraint solving as part of FOL proving

Part IV

Wrap Up

Library Case Studies

Java Collections Framework (JCF)

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

Java Card API

- Most parts of Java Card API specified using UML/OCL
- Some parts of reference implementation verified

Schorr-Waite Algorithm

- Standard benchmark for verification systems
- Graph marking algorithm for garbage collection
- Java implementation: 2 classes, core algorithm 25 lines of code
- Heavy aliasing, frame problem
- Specified and verified

Security Case Studies: Java Card Software

Safety/security properties specified in dynamic logic

- ‘Only certain exceptions can be thrown’
- Transactions are properly used
(do not commit or abort a transaction that was never started, all started Transactions are also closed)
- Data consistency
(also if a smartcard is “ripped out” during operation)
- Absence of overflows for integer operations

Two studies in this area (for which some critical parts were verified)

- Demoney (about 3000 lines):
Electronic purse application provided by Trusted Logic S.A.
- SafeApplet (about 600 lines): RSA based authentication applet

Safety Case Study

Computation of Railway Speed Restrictions

- Software by DBSystems for computing schedules for train drivers: Speed restrictions, required break powers
- Software formally specified using UML/OCL (based on existing informal specification)
- Program translated from Smalltalk to Java

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
- Symbolic error propagation
- Automating Induction Proofs
- Modular verification
- Verification of MISRA C
- Proof visualization, proving as debugging

Extension of dynamic logic (fixpoints, global induction)

Granularity of concurrency model

JCSP implementation ready as prototype

Some Current Directions of Research in KeY

- Multi-threaded Java
- **Integration of deduction and static analysis**
- Integration of verification and testing
- Symbolic error propagation
- Automating Induction Proofs
- Modular verification
- 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**
- Symbolic error propagation
- Automating Induction Proofs
- Modular verification
- 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
- **Symbolic error propagation**
- Automating Induction Proofs
- Modular verification
- Verification of MISRA C
- Proof visualization, proving as debugging

Symbolic error classes modeled by formulas

Error injection by instrumentation of Java 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
- Symbolic error propagation
- Automating Induction Proofs
- Modular verification
- Verification of MISRA C
- Proof visualization, proving as debugging

Synthesise induction schemata from failed proof attempts

Some Current Directions of Research in KeY

- Multi-threaded Java
- Integration of deduction and static analysis
- Integration of verification and testing
- Symbolic error propagation
- Automating Induction Proofs
- **Modular verification**
- Verification of MISRA C
- Proof visualization, proving as debugging

Generation of proof obligations ensuring “global correctness”
Reduce proof effort by analysing ‘observable state’

Some Current Directions of Research in KeY

- Multi-threaded Java
- Integration of deduction and static analysis
- Integration of verification and testing
- Symbolic error propagation
- Automating Induction Proofs
- Modular verification
- **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
- Symbolic error propagation
- Automating Induction Proofs
- Modular verification
- Verification of MISRA C
- Proof visualization, proving as debugging

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), T. Gedell (CTH), M. Giese (RICAM), W. Mostowski (RUN), A. Roth (SAP)

Colleagues who collaborated with us

J. Hunt, K. Johannisson, 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.

To appear in Dec. 2006.

Web site

www.key-project.de

Extra Slides



Other Transformations: Postincrement

Intuitive rule

$$\frac{\Gamma \implies \langle x=y; y=y+1; \omega \rangle \phi, \Delta}{\Gamma \implies \langle x=y++; \omega \rangle \phi, \Delta}$$

But ...

$$x = 5 \implies \langle x=x++ \rangle (x = 6) \quad \text{INVALID}$$

Correct rule

$$\frac{\Gamma \implies \langle v=y; y=y+1; x=v; \omega \rangle \phi, \Delta}{\Gamma \implies \langle x=y++; \omega \rangle \phi, \Delta}$$

Other Transformations: Postincrement

Intuitive rule

$$\frac{\Gamma \implies \langle x=y; y=y+1; \omega \rangle \phi, \Delta}{\Gamma \implies \langle x=y++; \omega \rangle \phi, \Delta}$$

But ...

$$x = 5 \implies \langle x=x++ \rangle (x = 6) \quad \text{INVALID}$$

Correct rule

$$\frac{\Gamma \implies \langle v=y; y=y+1; x=v; \omega \rangle \phi, \Delta}{\Gamma \implies \langle x=y++; \omega \rangle \phi, \Delta}$$

Other Transformations: Postincrement

Intuitive rule (not correct!)

$$\frac{\Gamma \implies \langle x=y; y=y+1; \omega \rangle \phi, \Delta}{\Gamma \implies \langle x=y++; \omega \rangle \phi, \Delta}$$

But ...

$$x = 5 \implies \langle x=x++ \rangle (x = 6) \quad \text{INVALID}$$

Correct rule

$$\frac{\Gamma \implies \langle v=y; y=y+1; x=v; \omega \rangle \phi, \Delta}{\Gamma \implies \langle x=y++; \omega \rangle \phi, \Delta}$$