Tutorial:
Integrating Object-oriented Design and Deductive Verification of Software

Wolfgang Ahrendt, Reiner Hähnle
Vladimir Klebanov, Philipp Rümmer

www.key-project.org

20th International Conference on Automated Deduction
July 23, 2005

Part I

Intro, Overview, Architecture

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)	Chalmers University of Technology	University of Koblenz-Landau
Peter H. Schmitt	Reiner Hähnle	Bernhard Beckert
Richard Bubel Andreas Roth Steffen Schlager Isabel Tonin	Wolfgang Ahrendt Tobias Gedell Wojciech Mostowski Philipp Rümmer Angela Wallenburg	Gerd Beuster 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
- **Interactive prover** with advanced UI
- Deep 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

A first 'Kick & Rush' Demo

Intention:

- **First impression, look & feel**
- **Motivate tutorial issues**
- But for now:
  - *No details*
  - *Few explanations*

More Demos to come

First Demo

What Have You (and What Have You NOT) Seen?

- In TogetherCC: UML class diagrams (annotated with OCL)
- In Eclipse: Java code annotated with JML
- Generation of proof obligations (POs) from Eclipse (or TogetherCC) + starting the KeY prover from Eclipse (or TogetherCC)
- Within the KeY prover:
  - POs rendered in JavaDL sequents
  - Construction + presentation of sequent proofs
  - (how to use the prover, really)
  - (design of the calculus)
  - ("taclet" language for defining rules)
  - (automation, implementation, ...)
- (how far does this carry us)

Outline of our Tutorial

- Part I (you are here)
  - Intro
  - First Demo
  - Dynamic Logic intro
  - Specification: JML (+ UML/OCL)
  - Proof obligations
  - Integration in standard tools
  - Second Demo
- Part II
  - JavaCardDL: the *logic*
  - Sequent Calculus
  - Symbolic execution
  - Design of the JavaCardDL *calculus* (demos)

Outline of our Tutorial (contd.)

- Part III
  - The "taclet" language and framework
  - Induction (demo)
  - Arithmetic (demo)
  - Automation
- Part IV
  - Interaction with the Prover (demo)
  - Case studies

The Logic: Dynamic Logic for Java

Dynamic Logic (DL)

- Each FOL formula is a DL formula
- If ϕ a DL formula and α a program:
  - $\langle \alpha \rangle \, \phi$ is a DL formula
  - $[\alpha] \, \phi$ is a DL-Formula
- DL formulas are closed under FOL operators and connectives

Modalities can be arbitrarily nested

Dynamic Logic for Java (JavaDL)

- In $\langle \alpha \rangle \, \phi$, and $[\alpha] \, \phi$, α is a list of **Java statements**
- *No encoding of programs*

Meaning of Dynamic Logic Formulas

For deterministic programs (like single threaded Java):

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

Relation to Hoare Logic

"Partial correctness" assertion

Hoare triple:

$$\{\psi\}\ \alpha\ \{\phi\}$$

"If α is started in a state satisfying ψ and terminates, then its final state satisfies ϕ."

in DL

$$????\psi\ \texttt{->}\ [\alpha]\,\phi$$

JavaDL Examples

Valid formulas

- $\langle$`x = 1; y = 3;`$\rangle$ `x` $<$ `y`
- `x` $< y$ `->` $\langle$`x++;`$\rangle$ `x` $<= y$
- $[$`while(true){x = x;}`$]$ `false`

Non-valid formulas

- `x` $<$ `y` `->` $\langle$`x = y; y = x;`$\rangle$ `y` $<$ `x`
- `x` $< y$ `->` $\langle$`x++;`$\rangle$ `x` $< y$
- $[$`while(x != 0){x = x;}`$]$ `false`

Alternative Formalisms for Correctness Assertions

Correctness Assertions

Can be stated:

1. In the oo specification languages
   - **JML** (Java Modeling Language)
   - **OCL** (Object Constraint Language, part of UML)
2. In **JavaDL** directly

Proof Obligations (POs)

Always in JavaDL,
Either generated from specifications (**1.**) and implementations,
or "hand-crafted" (**2.**)

Architectural Set Ups (1. – 4.)

With "hand-crafted" POs

1. **KeY stand alone prover**, loading POs from `.key` files

With automatic PO generation

- From JML and Java
  2. **JML browser + KeY stand alone prover**
  3. **Eclipse with KeY plug-in**
- From OCL/UML and Java
  4. **TogetherCC with KeY-extensions**

Java Modeling Language (JML)

A **notation** for **formally specifying**

- Behaviour of Java methods
- Admissible states of Java objects

Important features

- Pre/post conditions and invariants
- Notational consistency with Java expressions (Java expressions allowed in JML expressions, including side effect free method calls)
- "Specification only" fields and methods
- Restricting scope of side effects

JML specs appear as comments in `.java` files

JML example 1

```
/*@
  @ public normal_behavior
  @ requires   insertedCard != null;
  @ requires   !customerAuthenticated;
  @ requires   pin == insertedCard.correctPIN;
  @ ensures    customerAuthenticated;
  @ assignable customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ....
```


JML example 2

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


JML example 3

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

JML example 4

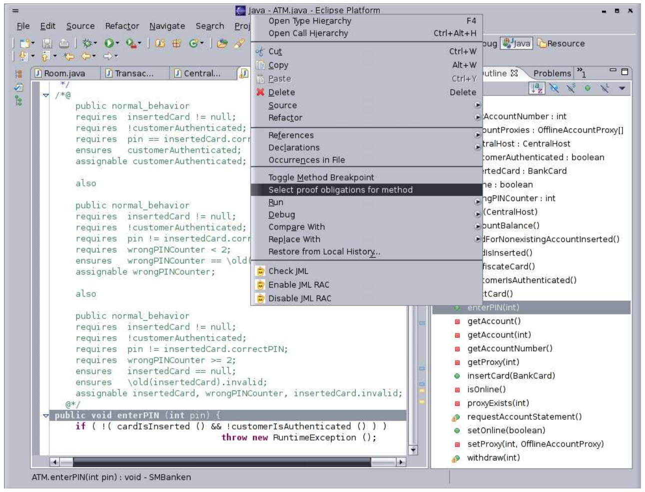
```
public classclass ATM {
/*@
  @ public invariantpublic invariant
  @        accountProxies != null;accountProxies != null;
  @ public invariantpublic invariant
  @        accountProxies.length == maxAccountNumber;accountProxies
  @ public invariant
  @        (\forall int i;
  @             i >= 0 && i < maxAccountNumber;
  @             ( accountProxies[i] == null
  @               ||
  @               accountProxies[i].accountNumber == i ));
  @*/
private /*@ spec_public @*/
    OfflineAccountProxy[] accountProxiesaccountProxies =
                 new OfflineAccountProxy [maxAccountNumber];
```


Eclipse

- *The* modern IDE for Java
- Provides powerful coding support:
  - Code templates, code completion
  - Import management
- *Freely available* via `eclipse.org`
- Very popular and widely distributed
- Well documented plug-in interface

KeY-Eclipse integration

Eclipse context menues, like:



Trigger generation of selected POs + launch prover window

Generating Proof Obligations

- JML expessions e automatically translated into formula $\mathcal{T}(e)$ (in simple cases FOL, in general JavaDL)
- Java is *not* translated, calculus works on unaltered source code
- Both combined in JavaDL

Proof Obligations: Postconditions

Given:

- Implementation α of method *m* of class C
- JML '`requires`' P for *m*
- JML **'`ensures`' Q** for *m*
- JML declares '`invariant`' I for C

Prove:

- $\mathcal{T}(\mathrm{I}) \& \mathcal{T}(\mathrm{P}) \;\text{->}\; \langle\alpha\rangle\, \mathcal{T}(\mathrm{Q})$

$\mathcal{T}(expr)$ = translation of the JML expression *expr* into DL

Proof Obligations: Invariants

Given:

- Implementation α of method *m* of class C
- JML **invariant I** of C
- JML precondition P of *m*

Prove:

- $\mathcal{T}(\mathrm{P}) \& \mathcal{T}(\mathrm{I}) \;\text{->}\; \langle\alpha\rangle\, \mathcal{T}(\mathrm{I})$

Alternative to JML: OCL/UML

Unified Modeling Language — UML
Visual language for OO modelling
Standard of Object Management Group (OMG)
Best-known feature: class diagrams

Object Constraint Language — OCL
Textual specification language
UML sub-standard
Pre/post condition and invariants, attached to class diagrams

KeY-TogetherCC Integration

TogetherCC
Commercial case tool (Borland), supporting UML

KeY extends TogetherCC by:

- Authoring support for OCL constraints
  OCL – natural language translation and co-editing
- PO generation from TogetherCC context menues
- Launching the KeY prover from TogetherCC context menues

JavaCard

Main target application: **smart cards**

- Relative small applications
- Often security/financially/legally critical

KeY system supports a smart card version of Java: **JavaCard**

JavaCard vs. Java

Features omitted in JavaCard

- Multi threading
- Floating point types
- Garbage collection (implementation optional)
- Dynamic class loading

Additional feature of JavaCard

- Transaction mechanism

JavaCard

KeY supports

100% JavaCard

Second Demo

After the Break

- Part I
  - Intro
  - First Demo
  - Java Dymanic Logic intro
  - Specification: JML (+ UML/OCL)
  - Proof obligations
  - Integration in standard tools
  - Second Demo
- Part II
  - JavaCardDL: the *logic*
  - Sequent Calculus
  - Symbolic execution
  - Design of the JavaCardDL *calculus* (demos)

Part II

Logic and Calculus

Dynamic Logic Syntax

A first-order program logic for modeling change of computation states

ProgramFormula ::=
FOFormula |
TotalCorrectnessModality ProgramFormula |
PartialCorrectnessModality ProgramFormula

TotalCorrectnessModality ::= '⟨' CompilableJavaCardStatement '⟩'

PartialCorrectnessModality ::= '[' CompilableJavaCardStatement ']'

- **Modal formulas closed under logical operations (cf. Hoare logic)**
- **JavaCardDL formulas contain unaltered JavaCard source code**

Why Dynamic Logic?

Application-specific	SW Analysis	Universal
Type system	Dynamic Logic	Logical Framework
Static Analysis	Hoare Logic	HOL
Approximation		Encoding
Efficiency		Soundness

Why Dynamic Logic?

Application-specific	SW Analysis	Universal
Type system	Dynamic Logic	Logical Framework
Static Analysis	Hoare Logic	HOL
Approximation		Encoding
Efficiency		Soundness

- Transparency wrt target programming language

- Programs are "first class citizens"
- No encoding of program syntax into logic

Why Dynamic Logic?

Application-specific	SW Analysis	Universal
Type system	Dynamic Logic	Logical Framework
Static Analysis	Hoare Logic	HOL
Approximation		Encoding
Efficiency		Soundness

- Can use reference implementations instead of FOL theories

Class initialization much easier to specify with code

Why Dynamic Logic?

Application-specific	SW Analysis	Universal
Type system	Dynamic Logic	Logical Framework
Static Analysis	Hoare Logic	HOL
Approximation		Encoding
Efficiency		Soundness

- More expressive and flexible than Hoare logic

Not merely partial/total correctness:
- Correctness of program transformations
- Security properties
- Natural temporal extensions (Beckert & Mostowski '03)

Why Dynamic Logic?

Application-specific	SW Analysis	Universal
Type system	Dynamic Logic	Logical Framework
Static Analysis	Hoare Logic	HOL
Approximation		Encoding
Efficiency		Soundness

- Symbolic execution more natural interactive proof paradigma than induction on syntactic structure

Why Dynamic Logic?

Application-specific	SW Analysis	Universal
Type system	Dynamic Logic	Logical Framework
Static Analysis	Hoare Logic	HOL
Approximation		Encoding
Efficiency		Soundness

- Proven technology that scales up
- Used in verification systems KIV, VSE since 1986
- Massive case studies involving imperative programs

Some JavaCardDL Syntax Issues

- FO logical variables disjoint from program variables
  - No quantification over program variables
  - Programs contain no logical variables
- ASCII syntax, key words preceded '\'
- Usual precedence, add brackets where necessary
- If program p appears in a DL formula then the class definitions of all types referenced in p are assumed to be present as well

Dynamic Logic Semantics I

Program formulas evaluated relative to
computation state s and variable assignment β

Example

`\forall` *int* x; (⟨`int i = j++;`⟩ (`i` $= x$))

Definition

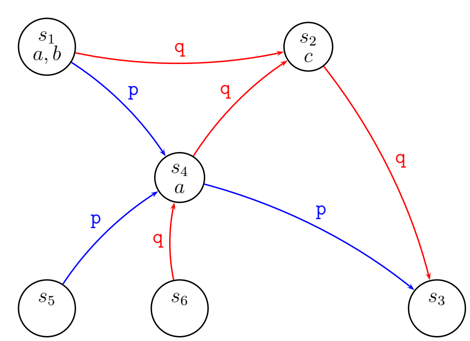
$s, \beta \models \langle \mathtt{p} \rangle \phi$ iff p **totally correct** wrt s and β iff p started in s terminates normally and $s', \beta \models \phi$ in final state s' after execution of p

$s, \beta \models [\mathtt{p}] \phi$ iff p **partially correct** wrt s and β iff whenever started in s p terminates normally then in $s', \beta \models \phi$ final state s' after execution of p

(We rely on Java programs being deterministic)

Dynamic Logic Semantics Example

Kripke structure, where worlds are computation states
Boolean program variables a, b, c, programs p, q



$s_1 \models \langle \mathtt{p} \rangle a$?(ok) $s_1 \models \langle \mathtt{q} \rangle a$?(−) $s_5 \models \langle \mathtt{q} \rangle a$?(−) $s_5 \models [\mathtt{q}] a$?(ok)

First-Order Formula Syntax

FOFormula ::= TernaryOpFormula | BinaryOpFormula |
UnaryOpFormula | NullaryOpFormula |
QuantifiedFormula | AtomicFormula

TernaryOpFormula ::=
'`\if` (' FOFormula ') `\then` (' FOFormula ') `\else` (' FOFormula ')'

BinaryOpFormula ::= FOFormula BinaryOp FOFormula

BinaryOp ::= '&' | '|' | ' -> ' | ' <-> '

UnaryOpFormula ::= '!' FOFormula

NullaryOpFormula ::= '`true`' | '`false`'

QuantifiedFormula ::= Quantifier Type LogVar ';' FOFormula

Quantifier ::= '`\forall`' | '`\exists`'

Dynamic Logic Example Formulas

`\forall` *int* y; $((\langle$`x = 1;`$\rangle\, \mathtt{x} = y)$ <-> $(\langle$`x = 1*1;`$\rangle\, \mathtt{x} = y))$ **Syntax ?**
ok

`\exists` *int* `x`; $([$`x = 1;`$]\,(\mathtt{x} = 1))$ **Syntax ?** bad

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

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

- Program formulas can appear nested

$\langle$`int x;`$\rangle$ `\forall` *int* y; $((\langle \mathtt{p}\rangle\, \mathtt{x} = y)$ <-> $(\langle \mathtt{q}\rangle\, \mathtt{x} = y))$ ok

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

First-Order Term Syntax

Terms are statically typed like in Java

- **Type** is partially ordered finite set of type symbols $\{t_1, \ldots, t_r\}$ closed under $\sqcap$, contains Java types
- Each logical variable $x \in \mathrm{LogVar}$ has **static type** t, declared $t\ x$
- x is term of type t for variable declared as $t\ x$
- Function symbols and predicate symbols declared with **signature**
  - Type FunctionSymbol ['(' Type {',' Type }* ')']
  - PredicateSymbol ['('Type {',' Type }* ')']
- Arguments of complex terms must **conform to** (in the sense of Java) type declared in their signature
- **Equality** symbol ' = ' for most argument types
- Otherwise **no overloading** of variables, functions, predicates

Type System Semantics

Type system semantics accounts for dynamic types of terms

Type System Semantics

Type system semantics accounts for dynamic types of terms

- Universe U disjoint union of subuniverses U^t for each type t

- Type t interpreted in the (possibly empty) subuniverse U^t
- Not all subuniverses U^t are populated (allow abstract classes)

Type System Semantics

Type system semantics accounts for dynamic types of terms

- Each term has static type (declared type of outermost symbol)

Type System Semantics

Type system semantics accounts for dynamic types of terms

- Let $T(t) = \bigcup_{t_0 \prec t} U^{t_0}$ be the universe elements typeable with t

Each $T(t)$ contains typable objects, at least $\texttt{null}^I$

Type System Semantics

Type system semantics accounts for dynamic types of terms

- The dynamic (runtime) type of a term e is the t such that $e^I \in U^t$

Dynamic type of e always conforms to its static type

Type System Semantics

Type system semantics accounts for dynamic types of terms

- Check dynamic type with type function: Type'::instance('Term')'

Rigid and Flexible Terms in Dynamic Logic

Certain FO terms correspond to Java locations:
program variables, **array access**, **attribute access**

Example

$\langle$`int i;`$\rangle$ `\forall` *int* $x;(\mathtt{i}+1=x \;\text{->}\; \langle$`i++;`$\rangle (\mathtt{i}=x))$

- Interpretation of `i` depends on computation state $\Rightarrow$ flexible

Locations are always interpreted flexible

Rigid and Flexible Terms in Dynamic Logic

Certain FO terms correspond to Java locations:
program variables, **array access**, **attribute access**

Example

$\langle$`int i;`$\rangle$ `\forall` *int* $x;(\mathtt{i}+1=x \;\text{->}\; \langle$`i++;`$\rangle (\mathtt{i}=x))$

Rigid and Flexible Terms in Dynamic Logic

Certain FO terms correspond to Java locations:
program variables, **array access**, **attribute access**

Example

$\langle$`int i;`$\rangle$ `\forall` *int* $x;(\mathtt{i}+1=x \;\text{->}\; \langle$`i++;`$\rangle (\mathtt{i}=x))$

- Interpretation of x and $+$ **must not** depend on state $\Rightarrow$ rigid

Logical variables, standard library functions declared rigid

A term containing at least one flexible symbol is flexible, otherwise rigid

Kripke Semantics

Kripke Semantics

- U is fixed: all objects with dynamic type t are in U^t from beginning

Objects have attributes `o.<created>` and `o.<initialized>`

These are set appropriately during object creation

Kripke Semantics

- States $s = (U, I_s) \in S$ have typed universe U, FO interpretation I_s

I_s interprets rigid symbols identically in each state

Kripke Semantics

- Semantics of Java program p is partial function $\rho(\mathrm{p}) : S \to S$

$s, \beta \models \langle \mathrm{p} \rangle\, \phi$ iff $\rho(\mathrm{p})(s) \downarrow$ and $\rho(\mathrm{p})(s), \beta \models \phi$

Kripke Semantics

- A JavaCardDL formula ϕ is valid iff $s, \beta \models \phi$ for all β and all s

Quantification over **all** computation states

State Update Semantics

Need to define ρ for each program p — start with **assignment**

Definition
State update of I at $\mathtt{t}\ \mathtt{x}$ with $u \in T(\mathtt{t})$

$$I_{\mathtt{x}}^{u}(\mathtt{y}) = \begin{cases} I(\mathtt{y}) & \mathtt{x} \neq \mathtt{y} \\ u & \mathtt{x} = \mathtt{y} \end{cases}$$

Assignment semanticsis state update:

$\rho(\mathtt{x=e;})(I) = I_{\mathtt{x}}^{\mathtt{e}^{I,\beta}}$

- e must be side effect-free, no reference type
- Identify states with interpretation since U is fixed

Program Semantics

In general, $\rho(\mathtt{p})$ defines operational semantics for p

- $\rho(\mathtt{if}\ (b)\ \{\alpha\}\ \mathtt{else}\ \{\gamma\};)(I) = \begin{cases} \rho(\alpha)(I) & I, \beta \models b = \mathtt{TRUE} \\ \rho(\gamma)(I) & \text{otherwise} \end{cases}$
- $\rho(\mathtt{while}\ (b)\ \{\alpha\};)(I) = I'$ iff there are $I = I_0, \ldots, I_n = I'$ such that
  - $I_j, \beta \models b = \mathtt{TRUE}$ for $0 \leq j < n$
  - $\rho(\alpha)(I_j) = I_{j+1}$ for $0 \leq j < n$
  - $I_n, \beta \models b = \mathtt{FALSE}$ undefined otherwise

Problems:

- Definitions work only under simplistic assumptions:
  b side-effect free, no exceptions, no breaks, ...
- We need a calculus (syntactic characterization)

Develop a calculus for JavaCard that directly realizes
an operational semantics with adequate syntactic means

Sequents and their Semantics

Sequent	::=	[FormulaList]	'==>'	[FormulaList]
FormulaList	::=	ProgramFormula	{',' ProgramFormula}*	

Notation

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

Schema variables ϕ, ψ match program formulas

Schema variables Γ/Δ match sublists of antecedent/succedent

Semantics

same as **formula** of sequent: $(\psi_1 \& \cdots \& \psi_m) \quad \texttt{->} \quad (\phi_1 | \cdots | \phi_n)$

(No free logical variables occur in program formulas)

Sequent Rules

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

Sound rule (essential):

$$\models (\mathsf{fml}(\Gamma_1 \texttt{==>} \Delta_1)\ \&\cdots\&\ \mathsf{fml}(\Gamma_r \texttt{==>} \Delta_r)) \ \texttt{->}\ \mathsf{fml}(\Gamma \texttt{==>} \Delta)$$

Complete rule (desirable):

$$\models \mathsf{fml}(\Gamma \texttt{==>} \Delta) \ \texttt{->}\ (\mathsf{fml}(\Gamma_1 \texttt{==>} \Delta_1)\ \&\cdots\&\ \mathsf{fml}(\Gamma_r \texttt{==>} \Delta_r))$$

Admissible to have no premisses (iff conclusion is valid: **axiom**)

Some Simple Sequent Rules

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

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

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

$$\text{ALL\_LEFT}\ \frac{\Gamma, \texttt{\textbackslash forall}\ t\ x; \phi, \quad \{x/e^{t'}\}\phi \texttt{ ==> } \Delta}{\Gamma, \texttt{\textbackslash forall}\ t\ x; \phi \texttt{ ==> } \Delta}$$

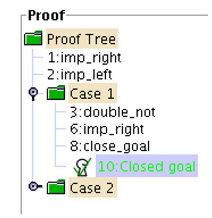
$e^{t'}$ var-free term of type $t' \prec t$

Sequent Calculus Proofs

Goal to prove validity of: $\mathcal{G} = \quad \psi_1, \ldots, \psi_m \texttt{ ==> } \phi_1, \ldots, \phi_n$

- find rule $\mathcal{R}$ whose conclusion matches $\mathcal{G}$
- instantiate $\mathcal{R}$ such that conclusion identical to $\mathcal{G}$
- check that side conditions of $\mathcal{R}$ are satisfied
- mark $\mathcal{G}$ as closed if $\mathcal{R}$ was axiom
- recursively find proofs for resulting premisses $\mathcal{G}_1, \ldots, \mathcal{G}_r$
- tree structure with goal sequent as root
- proof is finished when all goals are closed

Goal-directed proof search
In KeY tool proof displayed as JAVA Swing tree



Proof by Symbolic Program Execution

Which sequent rules for program formulas?
What corresponds to top-level connective in **sequential** program?

First executable statement: follow natural program control flow

Sound and complete rule for conclusions with main formulas:

$$\langle \texttt{p;}\omega \rangle\, \phi, \qquad [\texttt{p;}\omega]\, \phi$$

where $\texttt{p;}$ single legal Java statement, ω the remaining program

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

A Naive Rule for Assignment

$$\text{ASSIGNMENT}\ \frac{\{\mathtt{x}/\mathtt{x}_{old}\}\Gamma,\ \mathtt{x} = \{\mathtt{x}/\mathtt{x}_{old}\}e \ \mathtt{==>}\ \langle\omega\rangle\,\phi,\ \{\mathtt{x}/\mathtt{x}_{old}\}\Delta}{\Gamma \ \mathtt{==>}\ \langle \mathtt{x\ =\ e;}\omega\rangle\,\phi, \Delta}$$

$\mathtt{x}_{old}$ new program variable that "rescues" old value of x

Problems

- **Renaming** makes it difficult to keep track of computation state
- Does not work when e has **side effects** or when x is not variable
- Does not work for **reference types**
- **"Eager"** rule: bad if state change at x is cancelled out by later assignment or is irrelevant for ϕ

Specifying Initial Values

How to express correctness for arbitrary **initial** value of program variable?
Cannot quantify over program variables!

Not allowed: $\backslash\mathtt{forall}\ \mathit{int}\ \mathtt{i}; \langle \mathtt{p(}\cdots\ \mathtt{i}\ \cdots\mathtt{)}\rangle\,\phi$ (program $\neq$ logical variable)

Not intended: $\mathtt{==>}\ \langle \mathtt{p(}\cdots\ \mathtt{i}\ \cdots\mathtt{)}\rangle\,\phi$ (Validity of sequents: quantification over *all* states)

Not allowed: $\backslash\mathtt{forall}\ \mathit{int}\ n; \langle \mathtt{p(}\cdots\ n\ \cdots\mathtt{)}\rangle\,\phi$ (no logical variables in programs)

Solution
Use explicit construct to record state change information

(State) update $\backslash\mathtt{forall}\ \mathit{int}\ n; (\{\mathtt{i} := n\}\langle \mathtt{p(}\cdots\ \mathtt{i}\ \cdots\mathtt{)}\rangle\,\phi)$

Explicit State Updates

Updates record state change

Syntax($\mathtt{v}$, e have value types, e conforms to $\mathtt{v}$)
If $\mathtt{v}$ is program variable, e , e' FO terms, and ϕ any DL formula, then
$\{\mathtt{v} := e\}\phi$ is DL formula and $\{\mathtt{v} := e\}e'$ is FO term

Semantics
$I, \beta \models \{\mathtt{v} := e\}\phi$ iff $I_{\mathtt{v}}^{e^{I,\beta}}, \beta \models \phi$
Semantics identical to that of assignment

Updates work like "lazy" assignments

- Updates are **not assignments**: may contain logical variables
- Updates are **not equations**: change interpretation of PVs

Computing the Effect of Updates

The simplest case: x program variable with **value type**

Computing the Effect of Updates

The simplest case: $\mathtt{x}$ program variable with **value type**

Apply update to **program variable**

$$\{\mathtt{x} := e\}\mathtt{y} \rightsquigarrow \mathtt{y}$$
$$\{\mathtt{x} := e\}\mathtt{x} \rightsquigarrow e$$

Computing the Effect of Updates

The simplest case: $\mathtt{x}$ program variable with **value type**

Apply update to **complex term**

$$\{\mathtt{x} := e\}f(e_1, \ldots, e_n) \rightsquigarrow f(\{\mathtt{x} := e\}e_1, \ldots, \{\mathtt{x} := e\}e_n)$$

Similar for FOL formulas (like substitution)

Update followed by **program formula**

$$\{\mathtt{x} := e\}(\langle \mathtt{p} \rangle\, \phi) \rightsquigarrow \{\mathtt{x} := e\}(\langle \mathtt{p} \rangle\, \phi)$$ **unchanged!**

Update computation delayed until $\mathtt{p}$ symbolically executed

Computing the Effect of Updates

The simplest case: $\mathtt{x}$ program variable with **value type**

Apply update to **logical variable**

$$\{\mathtt{x} := e\}w \rightsquigarrow w$$

Composition of Updates

Updates **lazily** applied (delayed until “final” state), but **eagerly** simplified

Applying updates to updates: **composition** of states

$$\{l_1 := r_1\}\{l_2 := r_2\} = \{l_1 := r_1,\ l_2 := \{l_1 := r_1\}r_2\}$$

Results in **parallel update**: $\{l_1 := v_1,\ \ldots,\ l_n := v_n\}$

Semantics

- All l_i and v_i computed in old state
- All updates done simultaneously
- On conflict $l_i = l_j,\ v_i \neq v_j$ **last update wins**

For example, $\{i := 1 + 2,\ i := 2\} \rightsquigarrow \{i := 2\}$

Assignment Rule Revisited

$$\text{ASSIGN}\ \frac{\Gamma \Longrightarrow \{\mathtt{x} := \mathtt{e}\}\phi, \Delta}{\Gamma \Longrightarrow \langle \mathtt{x\ =\ e;} \rangle\, \phi, \Delta}$$

Rules dealing with programs need to account for updates

Notational convention:

- Updates already present in conclusion not displayed explicitly
- New updates in premise inserted after last present update

Updates simplified eagerly!

Demo: `rh_assign.key`

Some Non-Trivial Java Features

Illustrate main ideas in JavaCardDL calculus

- **Complex** expressions with **side effects**
  `int i = 0; if ((i=2) >= 2) {i++;}` // value of i?
- **Exceptions** (`try-catch-finally`)
- **Aliasing**
  Different navigation expressions may be same object reference

  $$I \models \mathtt{o.age} \doteq 1 \ \mathtt{->}\ \langle \mathtt{u.age\ =\ 2;} \rangle\, \mathtt{o.age} \doteq \mathtt{u.age} \qquad ?$$

  Depends on whether $I \models \mathtt{o} \doteq \mathtt{u}$

The Design Space of a Calculus

All JavaCard language features are fully addressed in KeY

Be aware of the full design space!

The Design Space of a Calculus

All JavaCard language features are fully addressed in KeY

Be aware of the full design space!

- Program transformation, up-front

Pro: Feature needs not be handled in calculus
Contra: Soundness, modified source code
Example in KeY: Only a few rare features, for example, inner classes

The Design Space of a Calculus

All JavaCard language features are fully addressed in KeY

Be aware of the full design space!

- Local transformation, done by a rule on-the-fly

Pro: Flexible, easy to implement, usable, less rules needed
Contra: Not expressive enough for all features
Example in KeY: Complex expressions, method expansion (many others)

The Design Space of a Calculus

All JavaCard language features are fully addressed in KeY

Be aware of the full design space!

- Special purpose constructs in program logic

Pro: Arbitrarily expressive extensions possible
Contra: Increases complexity of all rules
Example in KeY: Abrupt termination, method call, updates, blocks

The Design Space of a Calculus

All JavaCard language features are fully addressed in KeY

Be aware of the full design space!

- Modeling with first-order formulas

Pro: No extension required, enough to express most features
Contra: Creates difficult FO POs, unreadable antecedents, too eager
Example in KeY: Dynamic types, branch predicates

Highlights from JavaCardDL

Expressions with Side Effects

Local program transformation ensures side effect-free expressions

Compute complex subexpressions separately and store in temp. variable

```
i = j++;
```

```
int var = j;
j = (int)(j+1);
i = var;
```

Require guards in all rules to be **simple expressions**

$$\textrm{IF-SPLIT}\ \frac{\Gamma, \mathrm{b} \doteq \mathrm{TRUE} \texttt{ ==> } \langle \pi\ \mathrm{p}\ \omega\rangle\phi, \Delta \quad \Gamma, \mathrm{b} \doteq \mathrm{FALSE} \texttt{ ==> } \langle \pi\ \ \omega\rangle\phi, \Delta}{\Gamma \texttt{ ==> } \langle \pi\ \texttt{if (b) \{p\};}\ \omega\rangle\phi, \Delta}$$

Demo: rh_post_incr.key

Highlights from JavaCardDL
Abrupt Termination

Redirection of control flow via **exceptions**

$$\langle \pi\ \texttt{try \{pq\} catch(T e) \{r\} finally \{s\};}\ \omega\rangle\phi$$

Highlights from JavaCardDL
Try-throw // Symbolic execution

Catching a `throw` statement is controlled by prefix and postfix

TRY-THROW (exc simple)

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

Demo: `rh_exc.key`

Symbolic Execution

Symbolic: Only static information available, proof splitting

Execution: Runtime infrastructure required in calculus

Highlights from JavaCardDL
Abrupt Termination

Redirection of control flow via **exceptions**

$$\langle \pi\ \texttt{try \{pq\} catch(T e) \{r\} finally \{s\};}\ \omega\rangle\phi$$

Solution: symbolic execution rules work on first active statement after **prefix**, followed by postfix

$$\Gamma \texttt{ ==> } \mathcal{U}\langle \underbrace{\pi}_{\text{Prefix}}\quad \underbrace{\xi;}_{\text{Active statement}}\quad \underbrace{\omega}_{\text{Postfix}}\rangle\phi, \Delta$$

Highlights from JavaCardDL
Aliasing

Naive alias resolution causes **proof split** at each reference type access

$$\Gamma, \texttt{o.age} \doteq 1 \texttt{ ==> } \langle \pi\ \texttt{u.age = 2;}\ \omega\rangle \texttt{o.age} \doteq \texttt{u.age}$$

Unnecessary in many cases!

$$\Gamma, \texttt{o.age} \doteq 1 \texttt{ ==> } \langle \pi\ \texttt{u.age = 2; o.age = 2;}\ \omega\rangle \texttt{o.age} \doteq \texttt{u.age}$$

$$\Gamma \texttt{ ==> } \langle \pi\ \texttt{o.age = 1; u.age = 2;}\ \omega\rangle \texttt{u.age} \doteq 2$$

Updates avoid such proof splits:

- **Delay** application of state computation after program execution
- **Eager** simplification of updates, accumulate effect

Simplification and application of updates with reference types not trivial!

Demo: `rh_alias.key`

After the Break

But how does this work in practice?

- How are rules implemented?
- How “automatic” are they applied?
- What about Java integer types?
- And loops? How does induction work?
- How does the prover interface support its user?

Stay tuned to KeY 1.0 !

Part III

The Prover: Concepts, Implementation, Automation

Taclets and Taclet Language

Taclets

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 “validated” w.r.t. more primitive taclets.

Taclet Syntax

Consider a "modus ponens" rule:

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

Here it is as a taclet:

\find (b -> c ==>) \assumes (b ==>) \replacewith(c ==>)
\heuristics(simplify)

- schema variables
- turnstile ($\vdash$)
- find clause
- action clause
- assumes clause
- heuristic declaration

A Branching Rule

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

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


"The Small Print"

Consider the rule for existential quantifiers:

$$\frac{\Gamma,\ \phi(f(x_1,\ldots,x_n))\ \texttt{==>}\ \Delta}{\Gamma,\ \texttt{\textbackslash exists}\ t\ x;\phi(x)\ \texttt{==>}\ \Delta}$$

where $x_1,\ldots,x_n$ are the free variables occurring in $\phi(x)$ and f is a new function symbol with static type t.

```
ex_left {
  \find (\exists u; b ==>)
  \varcond ( \new(sk, \dependingOn(b)) )
  \replacewith ({\subst u; sk}b ==>)
  \heuristics (delta)
};
```

```
\new(v), \notFreeIn(x,y),
\isLocalVariable(v), \static(v), ...
```


Java Card Taclets

Rule if_else_split

$$\frac{\begin{array}{c}\Gamma,\ B \doteq \mathrm{true}\ \texttt{==>}\ \langle\texttt{..}\quad \alpha_1;\ \texttt{...}\rangle F,\ \Delta\\ \Gamma,\ B \doteq \mathrm{false}\ \texttt{==>}\ \langle\texttt{..}\quad \alpha_2;\ \texttt{...}\rangle F,\ \Delta\end{array}}{\Gamma\ \texttt{==>}\ \langle\texttt{..}\quad \texttt{if (B)}\ \alpha_1\ \texttt{else}\ \alpha_2;\ \texttt{...}\rangle F,\ \Delta}$$

with B 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)
};
```

Soundness

“Higher order skolemization”

Modus ponens:

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

Validation proof obligation:

$$\texttt{\textbackslash forall}\ \phi; \texttt{\textbackslash forall}\ \psi; ((\phi \text{ -> } \psi)\ \&\ \phi) \text{ -> } \psi$$

After skolemization:

$$((p \text{ -> } q)\ \&\ p) \text{ -> } q$$

Cross-checking against other Java semantics

- Bali
- Java semantics in Maude

Integer Arithmetics

Summary

Taclets ...

- simple and powerful
- compact and clear notation
- no complicated meta-language
- esay to apply with a GUI
- validation possible

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

Data Type Gap: Integer Semantics

OCL type Integer

- Infinite range, operators have usual mathematical semantics ($\mathbb{Z}$)

Java types `byte`**,** `short`**,** `int`**,** `long`

- Different finite ranges
- Semantics of operators as in $\mathbb{Z}$ except that:

  overflow occurs if result exceeds range,
  i.e., result is calculated modulo size of data type.

- Overflow occurs silently

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

Examples

Valid for Java integer semantics

$\texttt{MAX\_INT+1} = \texttt{MIN\_INT}$

$\texttt{MIN\_INT*}(-1) = \texttt{MIN\_INT}$

$\texttt{\textbackslash exists}\ \mathit{int}\ \texttt{x}, \texttt{y}; !\texttt{x} = 0\ \&\ !\texttt{y} = 0\ \&\ \texttt{x*y} = 0$

Not valid for Java integer semantics

$\texttt{\textbackslash forall}\ \mathit{int}\ \texttt{x}; \texttt{\textbackslash exists}\ \mathit{int}\ \texttt{y}; \texttt{y} > \texttt{x}$

Not a sound rewrite rules for Java integer semantics

$\texttt{x+1} > \texttt{y+1} \quad \rightsquigarrow \quad \texttt{x} > \texttt{y}$

General Problem revisited

- Semantic gap between $\mathbb{Z}$ and Java integers
- Defining a JavaDL semantics for Java integers that. . .
  - is a correct data refinement of $\mathbb{Z}$ Req. (Z)
  - reflects Java integer semantics Req. (J)

3 possible approaches

Semantics	Description	Req. (Z)	Req. (J)
$\mathcal{S}_{OCL}$	corresponds to semantics of $\mathbb{Z}$	$\surd$	X
$\mathcal{S}_{Java}$	corresponds to Java semantics	X	$\surd$
$\mathcal{S}_{KeY}$	hybrid of $\mathcal{S}_{OCL}$ and $\mathcal{S}_{Java}$	$\surd$	$\surd$

Semantics $\mathcal{S}_{OCL}$

$\mathcal{S}_{OCL}$ assigns Java integers the semantics of $\mathbb{Z}$

- Req. (Z) trivially fulfilled
- Req. (J) violated, incorrect programs can be "verified"

Example:
$\models_{\mathcal{S}_{OCL}} \langle \texttt{y=x+1;} \rangle\, \mathtt{y} = \mathtt{x} +_{\mathbb{Z}} 1$

but for $\mathtt{x} = \mathtt{MAX\_INT}$ program not correct

Semantics $\mathcal{S}_{Java}$

$\mathcal{S}_{Java}$ assigns Java integers the semantics defined in the JLS

- Req. (Z) violated
  several abstract states mapped onto one concrete state
- Req. (J) trivially fulfilled

No incorrect programs can be verified, but

- Existence of "incidentally" correct programs
- Difficult to reason about

Our Approach: Semantics $\mathcal{S}_{KeY}$

1. Show the program correct for $\mathbb{Z}$
2. Show that no overflow occurs at every step

Program correct w.r.t. Java semantics

A Sequent Calculus For $\mathcal{S}_{KeY}$

Example: Rule for addition
generates conditions that no overflow occurs
with help of predicate $in_{\mathrm{T}}(\mathtt{x}) \equiv \mathtt{MIN\_}T \leq \mathtt{x} \leq \mathtt{MAX\_}T$

$$\frac{\begin{array}{c}(1)\ \ \Gamma \ \texttt{==>}\ \{\mathtt{z} := \mathtt{x} + \mathtt{y}\}\langle\rangle\, \phi \\ (2)\ \ \Gamma,\ in_T(\mathtt{x}),\ in_T(\mathtt{y}) \ \ \texttt{==>}\ in_T(\mathtt{x}+\mathtt{y}),\ \langle \texttt{z=x+y;} \rangle\, \phi\end{array}}{\Gamma \ \texttt{==>}\ \langle \texttt{z=x+y;} \rangle\, \phi}$$

Summary

The KeY system has 3 pluggable integer semantics,
of which $\mathcal{S}_{KeY}$ has the best properties:

- Safe (though slight loss of completeness)
- Familiar reasoning
- Modularized proofs
- Proof reuse possible when switching from other semantics

Basic Integer Induction Rule

$$\frac{\begin{array}{c}(1)\;\; \Gamma \;{=}{=}{>}\; \mathit{IH}(0),\;\; \Delta \\ (2)\;\; \Gamma \;{=}{=}{>}\; \texttt{\textbackslash forall}\; \mathit{int}\; i; (i \geq 0 \& \mathit{IH}(i) \;{-}{>}\; \mathit{IH}(i+1)),\;\; \Delta \\ (3)\;\; \Gamma,\;\; \texttt{\textbackslash forall}\; \mathit{int}\; i; (i \geq 0 \;{-}{>}\; \mathit{IH}(i)) \;{=}{=}{>}\; \Delta\end{array}}{\Gamma \;{=}{=}{>}\; \Delta}$$

IH = induction hypothesis
i = induction variable

Proving Loops with Induction

An Example

To be proven:

$\texttt{\textbackslash forall}\; \mathit{int}\; \mathit{nl}; (\mathit{nl} > 0 \;\&\; \mathtt{i} = 0 \;{-}{>}\; \{\mathtt{n} := \mathit{nl}\}\langle\texttt{while (i<n) i++;}\rangle\, \mathtt{i} \geq \mathtt{n})$

To be proven (after skolemization):

$$\mathit{nl}_0 > 0 \;\&\; \mathtt{i} = 0 \;{-}{>}\; \{\mathtt{n} := \mathit{nl}_0\}\langle\texttt{while (i<n) i++;}\rangle\, \mathtt{i} \geq \mathtt{n}$$

Induction hypothesis:

$$\{\mathtt{n} := \mathit{nl}_0\}\{\mathtt{i} := \mathtt{n} - k\}\langle\texttt{while (i<n) i++;}\rangle\, \mathtt{i} \geq \mathtt{n}$$

Induction variable: k

Induction Obligations

Base case $(k = 0)$

$$\{\mathtt{n} := nl_0\}\{\mathtt{i} := \mathtt{n} - 0\}\langle\mathtt{while\ (i<n)\ i++;}\rangle\, \mathtt{i} \geq \mathtt{n}$$

Step case $(k \curvearrowright k + 1)$

$$\{\mathtt{n} := nl_0\}\{\mathtt{i} := \mathtt{n} - k_1\}\langle\mathtt{while\ (i<n)\ i++;}\rangle\, \mathtt{i} \geq \mathtt{n} \quad \texttt{->}$$
$$\{\mathtt{n} := nl_0\}\{\mathtt{i} := \mathtt{n} - (k_1 + 1)\}\langle\mathtt{while\ (i<n)\ i++;}\rangle\, \mathtt{i} \geq \mathtt{n}$$

Automation

Summary

Induction ...

- programs can be proved with the “basic” integer induction rule
- lots of human interaction necessary
- quite a viscous task
- research in automation is underway
- invariant rule an alternative

Means of Automation Implemented in KeY

- Global strategies for automatically applying rules in series
- Free-variable calculus for constructing witnesses for quantified formulas (non-destructive, proof-confluent calculus)
- Invocation of external theorem provers, decision procedures
  - Simplify (from ESC/Java)
  - ICS
  - Planned: Export to SMT-LIB format

Strategies

Responsible for selecting next proof expansion step for each goal

1. All possible expansion steps for a goal are computed
   - Steps described by:
     Applied rule/taclet, position, values of schema variables
   - Information is cached in *RuleIndex* and updated when sequent is altered
2. For each possible rule application a *cost* value is computed
   - Integer value: Lower numbers → Preferred steps
   - Cost functions take into account for instance:
     Kind of rule, unifications necessary, depth and context of position
   - Different strategies use different cost functions
3. Step with lowest costs is executed
   - Again caching: Priority queue for sorting expansion steps

Procedure is iterated until no further rules are applicable or chosen maximum number of rule applications is reached

Strategies Currently Present in KeY

Strategies optimized for symbolically executing programs

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

Strategy handling first-order logic

- Implements a complete first-order theorem prover
- But: Weak support for theories (particularly arithmetic)

Implementation of Strategies

- Strategies are written Java, direct part of prover
- Creating new special-purpose strategies is easy
- Cost functions described using a library of *feature* functions and connectives

Free-Variable Calculus

Existential variables used to postpone instantiation

- In KeY called *metavariables*
- Mostly for universally quantified formulas in antecedent

Constraints used to represent unification

- Formula constraints (conjunctions of equations) added when terms have to be substituted for metavariables

$$\frac{\dfrac{\texttt{true} \ll [X_0 \equiv 0],\ \texttt{false} \ll [X_0 \equiv 1],\ \backslash\texttt{forall}\ \mathit{int}\ x; x = 0\ \texttt{==>}}{X_0 = 0,\ \backslash\texttt{forall}\ \mathit{int}\ x; x = 0\ \texttt{==>}}}{\backslash\texttt{forall}\ \mathit{int}\ x; x = 0\ \texttt{==>}}$$

Incremental Closure in Free-Variable Calculus

Closing proofs by simultaneously closing its goals

- When applying taclets with \closegoal, involved constraints are collected for goal
- Proof can be closed if consistent closure constraints exist for all goals
- In KeY: Consistency of closure constraints is checked recursively, closure constraints for all subtrees of proof tree are cached

Color codes in proof tree for status of goals and subtrees

black	no closing constraints exist
blue	closing constraints exist
green	goal is closed with a valid constraint (i.e. no restrictions)

Free-Variable Calculus (2)

Calculus is non-destructive and proof-confluent

- Unifiers are never directly applied to proof
- No backtracking necessary (but: interactive backtracking possible)
- Calculus is mostly useful for pure first-order logic, combination with theories and modal logic ongoing issue

Part IV

The Prover: Interaction and Guidance Case Studies

Interaction and Automation

For realistic programs: Fully-automated verification impossible

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

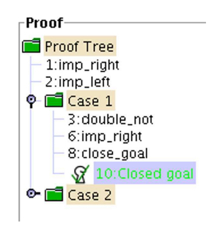
Typical workflow when proving in KeY
(and other interactive provers)

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 Proof Trees

Displayed information

- Inner nodes labelled with rule that was applied
- Colors: Green signals closed subtrees
  Blue subtrees closed for suitable instantiation of metavariables



Navigation

- By selecting inner nodes or leaves in tree
- By selecting leaves in goal list

Working with Proof Trees (2)

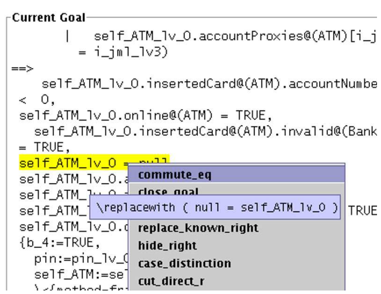
Modifying the proof tree

- Extension: Only through application of rules to goals (as usual in Gentzen-style sequent calculi; next slides)
- Closure: Through taclets with `\closegoal`
- Pruning: Deletion of subtrees (button in toolbar, context menu in tree display)

Working with Sequents: Sequent View

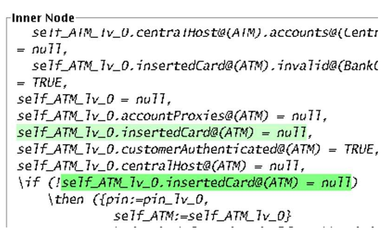
For goals/leaves of tree

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



For inner nodes

- Parts involved in rule application are highlighted



Extension of Proof: Application of Single Taclets

Application of a taclet requires:

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

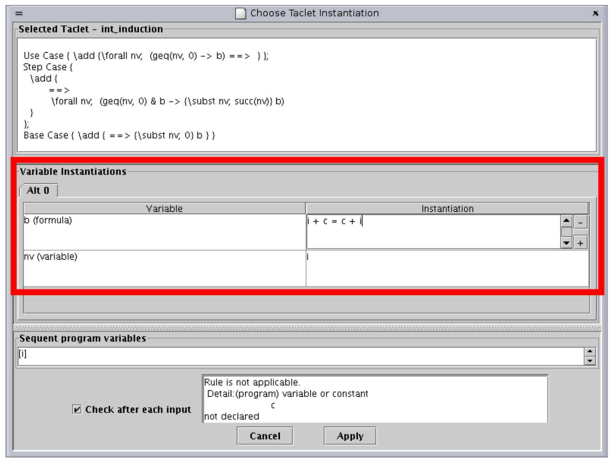
Principal procedure in KeY when applying taclet interactively

1. Selection of application focus using mouse pointer
2. Selection of particular rule from context menu
3. Instantiation of schema variables

Schema Variables: Taclet Instantiation Dialog

Primarily two purposes:

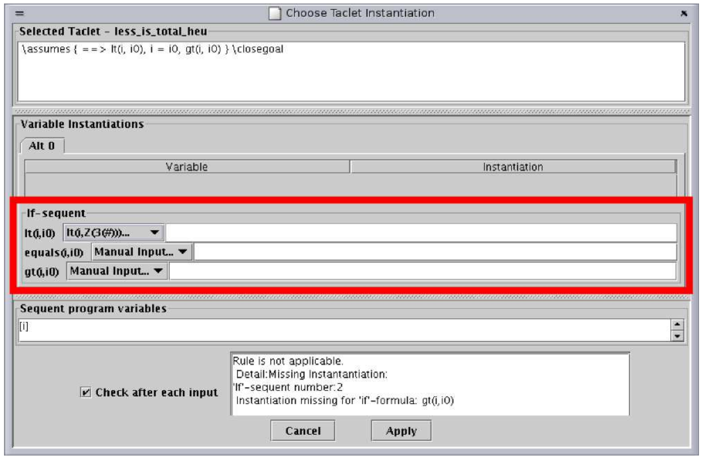
- Enter values of schema variables explicitly



Schema Variables: Taclet Instantiation Dialog

Primarily two purposes:

- Provide assumptions of taclet (assumes clause)



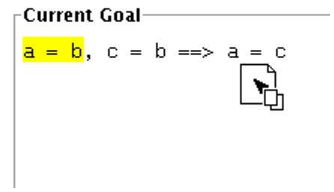
Applying Taclets using Drag'n'Drop

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

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

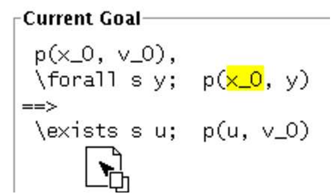
Applying equations

- Hold Ctrl, drag the equation to the term to be rewritten



Instantiating quantified formulas

- Hold Ctrl, drag instantiation term to quantified formula



Schema Variables: Taclet Instantiation Dialog

Primarily two purposes:

Drag'n'drop can be used for copying data from sequent view

Extension of Proof: Automated Application of Rules

Selection of active strategy

- Menu in toolbar

Invocation of strategies

- Explicitly ...
  (button in toolbar, context menus in proof tree and sequent view)
- ... or automatically after each interaction
  (meaningful for strategies *simplifying/normalising* the goals)

Application of strategies possible on

- All goals of a proof
- One particular goal
- Particular subterm or subformula

Extension of Proof: Reusing Existing Proof

To Be Done

"Fundamental" Case Studies: Libraries

Java Collections Framework (JCF)

- Part of JCF (treating sets) was specified using UML/OCL
- Parts of reference implementation were verified
- It was investigated how the consistency of JCF classes with common algebraic datatypes can be shown

JavaCard API

- Most parts of JavaCard API were specified using UML/OCL
- Some parts of reference implementation were verified

Security Case Studies: JavaCard Software

Safety/security properties were treated (specified in dynamic logic)

- No exceptions are thrown, apart from well-specified `ISOExceptions`
- Transactions are properly used
  (do not commit or abort a transaction that was never started, all started exceptions 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 Studies

Computation of Railway Speed Restrictions

- Software by DBSystems for computing schedules for train drivers:
  Speed restrictions, required break powers
- Software was formally specified using UML/OCL (based on existing informal specification), verification planned
- Program translated from Smalltalk to Java
- Program consists of more than 25 classes

Command Parser for Chemical Analysis Devices

- Software by Agilent Technologies
- Ongoing, Goal: specify parser and verify it
- Parser originally written in C++: reimplementation in MeDeLa, then (automatic) conversion to Java

Part V

Wrap-Up

Some Current Directions of Research in KeY

- Integration of deduction and static analysis

Mutual call of analyser/prover, common semantic framework
Implementation of static analysis in theorem proving frame
Replacing loops with generic proof of body
Abstraction of verified program on-the-fly

Some Current Directions of Research in KeY

- Multi-threaded Java

Extension of dynamic logic (fixpoints, global induction)
Granularity of concurrency model
JCSP implementation ready as prototype

Some Current Directions of Research in KeY

- Counter examples

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

Some Current Directions of Research in KeY

- Symbolic error propagation

Symbolic error classes modeled by formulas
Error injection by instrumentation of JavaCardDL rules
Symbolic error propagation via symbolic execution

Some Current Directions of Research in KeY

- Modular verification
- Verification of MISRA C
- Proof visualization, proving as debugging

Generation of proof obligations ensuring "global correctness"
Reduce proof effort by analysing modifiable locations

Some Current Directions of Research in KeY

- Automating of Induction

Simplification of induction claim by code-driven decomposition
"Rippling" applied to updates guides generalization

Acknowledgments

Funding Agencies:

- Deutsche Forschungsgemeinschaft (DFG)
- Deutscher Akademischer Auslandsdienst (DAAD)
- Vetenskapsradet (VR)
- VINNOVA
- Stiftelsen för internationalisering av högre utbildning och forskning (STINT)

The many students who did a thesis or worked as developers

Alumni:
W. Menzel (em.), T. Baar (EPFL), A. Darvas (ETH), M. Giese (RICAM)

Colleagues who collaborated with us:
J. Hunt, K. Johanisson, A. Ranta, D. Sands