Theorem Prover

Developed at University of Karlsruhe

http://www.key-project.org/

Theory specialized for Java(Card).

Can generate proof-obligations from JML specification.

Underlying theory: Sequent Calculus + Dynamic Logic
Dynamic Logic

Dynamic logic extends predicate logic by

- $[\alpha]\phi$
- $\langle\alpha\rangle\phi$

where $\alpha$ is a program and $\phi$ a sub-formula.

The meaning is as follows:

- $[\alpha]\phi$: after all terminating runs of program $\alpha$ formula $\phi$ holds.
- $\langle\alpha\rangle\phi$: after some terminating run of program $\alpha$ formula $\phi$ holds.
The sequent $\phi \Rightarrow [\alpha] \psi$ corresponds to partial correctness of the Hoare formula:

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

If $\alpha$ is deterministic, $\phi \Rightarrow \langle \alpha \rangle \psi$ corresponds to total correctness.
Examples

- $[\{}\phi \equiv \phi$
- $\langle \{} \rangle \phi \equiv \phi$
- $[\text{while(true)}\{\}] \phi \equiv \text{true}$
- $\langle \text{while(true)}\{\} \rangle \phi \equiv \text{false}$
- $[x = x + 1; ]x \geq 4 \equiv x + 1 \geq 4$
- $[x = t; ] \phi \equiv \phi[t/x]$
- $[\alpha_1 \alpha_2] \phi \equiv [\alpha_1][\alpha_2] \phi$

How can we use equivalences in Sequent Calculus?

Add the rule $\frac{\Gamma[\psi/\phi] \Longrightarrow \Delta[\psi/\phi]}{\Gamma \Longrightarrow \Delta}$, where $\phi \equiv \psi$.

This is similar to applyEq.
Dynamic Logic is Modal Logic

- $\langle \alpha \rangle \phi \equiv \neg [\alpha] \neg \phi$
- $[\alpha] \phi \equiv \neg \langle \alpha \rangle \neg \phi$

Furthermore:
- if $\phi$ is a tautology, so is $[\alpha] \phi$
- $[\alpha] (\phi \to \psi) \to ([\alpha] \phi \to [\alpha] \psi)$

Remark: For deterministic programs also the reverse holds

$([\alpha] \phi \to [\alpha] \psi) \to [\alpha] (\phi \to \psi)$
How can we express that program $\alpha$ must terminate?

$\langle \alpha \rangle \text{true}$

This can be used to relate $[\alpha]$ and $\langle \alpha \rangle$:

$\langle \alpha \rangle \phi \equiv [\alpha] \phi \land \langle \alpha \rangle \text{true}$
The formula \( \langle i = t; \alpha \rangle \phi \) is rewritten to

\[
\{ i := t \} \langle \alpha \rangle \phi
\]

Formula \( \{ i := t \} \phi \) is true, iff \( \phi \) holds in a state, where the program variable \( i \) has the value denoted by the term \( t \).

Here:

- \( i \) is a program variable (non-rigid function).
- \( t \) is a term (may contain logical variables).
- \( \phi \) a formula
Simplifying Updates

If $\phi$ contains no modalities, then $\{x := t\} \phi$ is the substitution $\phi[t/x]$ (every occurrence of $x$ is changed to $t$).

A double update $\{x_1 := t_1, x_2 := t_2\} \{x_1 := t'_1, x_3 := t'_3\} \phi$ is automatically rewritten to

$$\{x_1 := t'_1[t_1/x_1, t_2/x_2], x_2 := t_2, x_3 := t'_3[t_1/x_1, t_2/x_2]\} \phi$$
Example: \( \langle \{ i = j; j = i + 1 \} \rangle i = j \)

\[
\langle \{ i = j; j = i + 1 \} \rangle i = j
\equiv\{ i := j \}\{ j := i + 1 \} i = j
\equiv\{ i := j, j := j + 1 \} i = j
\equiv j = j + 1
\equiv false
\]

or alternatively

\[
\langle \{ i = j; j = i + 1 \} \rangle i = j
\equiv\{ i := j \}\{ j := i + 1 \} i = j
\equiv\{ i := j \} i = i + 1
\equiv j = j + 1
\equiv false
\]
Rules for Java Dynamic Logic

- \(\langle\{i = j; \ldots\}\rangle\phi\) is rewritten to:
  \(\{i := j\}\langle\{\ldots\}\rangle\phi\).

- \(\langle\{i = j + k; \ldots\}\rangle\phi\) is rewritten to:
  \(\{i := j + k\}\langle\{\ldots\}\rangle\phi\).

- \(\langle\{i = j + +; \ldots\}\rangle\phi\) is rewritten to:
  \(\langle\{\textbf{int } j_0; j_0 = j; j = j + 1; i = j_0; \ldots\}\rangle\phi\).

- \(\langle\{\textbf{int } k; \ldots\}\rangle\phi\) is rewritten to:
  \(\langle\{\ldots\}\rangle\phi\) and \(k\) is added as new program variable.
Given a simple loop:

\[
\langle \{ \textbf{while} (n > 0) n--; \} \rangle n = 0
\]

How can we prove that the loop terminates for all \( n \geq 0 \) and that \( n = 0 \) holds in the final state?
Method (1): Induction

To prove a property $\phi(x)$ for all $x \geq 0$ we can use induction:

- Show $\phi(0)$.
- Show $\phi(x) \implies \phi(x + 1)$ for all $x \geq 0$.

This proves that $\forall x (x \geq 0 \rightarrow \phi(x))$ holds.
The KeY-System has the rule `int_induction`

\[
\begin{align*}
\Gamma & \Rightarrow \Delta, \phi(0) & \Gamma & \Rightarrow \Delta, \forall X (X \geq 0 \land \phi(X) \rightarrow \phi(X + 1)) \\
\Gamma, \forall X (X \geq 0 \rightarrow \phi(X)) & \Rightarrow \Delta \\
\hline \\
\Gamma & \Rightarrow \Delta
\end{align*}
\]

The three goals are:

- **Base Case:** \( \Rightarrow \phi(0) \)
- **Step Case:** \( \Rightarrow \forall X (X \geq 0 \land \phi(X) \rightarrow \phi(X + 1)) \)
- **Use Case:** \( \forall X (X \geq 0 \rightarrow \phi(X)) \Rightarrow \)
Induction proofs are very difficult to perform for a loop

\[ \langle \{ \textbf{while}(COND) BODY; \ldots \}\rangle \phi \]

The KeY-system supports special rules for while loops using invariants and variants.
The rule `while_invariant_with_variant_dec`

The rule `while_invariant_with_variant_dec` takes an invariant `inv`, a modifies set `{m₁, . . ., mₖ}` and a variant `v`. The following cases must be proven.

- Initially Valid: \( \implies inv \land v \geq 0 \)
- Body Preserves Invariant:
  \[ \implies \{ m₁ := x₁ \| \ldots \| mₖ := xₖ \}(inv \land [\{ b = COND; \}])b = \text{true} \]
  \[ \rightarrow \langle BODY \rangle inv \]

- Use Case:
  \[ \implies \{ m₁ := x₁ \| \ldots \| mₖ := xₖ \}(inv \land [\{ b = COND; \}])b = \text{false} \]
  \[ \rightarrow \langle \ldots \rangle \phi \]

- Termination:
  \[ \implies \{ m₁ := x₁ \| \ldots \| mₖ := xₖ \}(inv \land v \geq 0 \land [\{ b = COND; \}])b = \text{true} \]
  \[ \rightarrow \{ old := v \}\langle BODY \rangle v \leq old \land v \geq 0 \]
Rigid vs. Non-Rigid Functions vs. Variables

KeY distinguishes the following symbols:

- **Rigid Functions**: These are functions that do not depend on the current state of the program.
  - $+, -, \times: \text{integer} \times \text{integers} \rightarrow \text{integer}$ (mathematical operations)
  - $0, 1, \ldots: \text{integer}, \text{TRUE, FALSE} : \text{boolean}$ (mathematical constants)

- **Non-Rigid Functions**: These are functions that depend on current state.
  - $\cdot[\cdot] : \top \times \text{int} \rightarrow \top$ (array access)
  - $\cdot\text{next} : \top \rightarrow \top$ if $\text{next}$ is a field of a class.
  - $i, j : \top$ if $i, j$ are program variables.

- **Variables**: These are logical variables that can be quantified. Variables may not appear in programs.
  - $x, y, z$
\[ \forall x. i = x \rightarrow \langle \{ \text{while}(i > 0)\{i = i - 1; \}\}\rangle i = 0 \]

- 0, 1, - are rigid functions.
- > is a rigid relation.
- i is a non-rigid function.
- x is a logical variable.

Quantification over i is not allowed and x must not appear in a program.
Builtin Rigid Functions

- $+,-,*,/,%,$div$,jmod$: operations on $integer$.
- $\ldots,-1,0,1,\ldots$, TRUE,FALSE, null: constants.
- $(A)$ for any type $A$: cast function.
- $A :: get$ gives the $n$-th object of type $A$. 