The **The Key-Project**

- Theorem Prover
- Developed at University of Karlsruhe
- [http://www.key-project.org/](http://www.key-project.org/).

- Theory specialized for Java(Card).
- Can generate proof-obligations from JML specification.
- Underlying theory: Sequent Calculus + Dynamic Logic
KeY distinguishes the following symbols:

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

- **Non-Rigid Functions**: These are functions that depend on current state.
  - $\cdot[:] : \top \times int \rightarrow \top$ (array access)
  - $\cdot.next : \top \rightarrow \top$ if 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.
  - $x, y, z$
∀x. i = x → ⟨{while(i > 0) {i = i - 1;}}⟩ 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.
Built-in Rigid Functions

- $\,+,-,\times,\div,\%,j\text{div},j\text{mod}$: operations on integer.
- $\ldots,-1,0,1,\ldots$, \textit{TRUE, FALSE, null}: constants.
- $(A)$ for any type $A$: cast function.
- $A::\text{get}$ gives the $n$-th object of type $A$. 
Updates in KeY

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
If $\phi$ contains no modalities, then $\{x := t\}\phi$ is rewritten to $\phi[t/x]$.

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: $\langle\{i = j; j = i + 1\}\rangle i = j$

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

or alternatively

\[
\langle\{i = j; j = i + 1\}\rangle i = j \\
\equiv\{i := j\}\{j := i+1\} i = j \\
\equiv\{i := j\} i = i + 1 \\
\equiv j = j + 1 \\
\equiv \text{false}
\]
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?
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 rule \texttt{int\_induction}

The KeY-System has the rule \texttt{int\_induction}

\[
\Gamma \implies \Delta, \phi(0) \quad \Gamma \implies \Delta, \forall X (X \geq 0 \land \phi(X) \rightarrow \phi(X + 1)) \\
\Gamma, \forall X (X \geq 0 \rightarrow \phi(X)) \implies \Delta \\
\hline \\
\Gamma \implies \Delta
\]

The three goals are:

- **Base Case:** \( \implies \phi(0) \)
- **Step Case:** \( \implies \forall X (X \geq 0 \land \phi(X) \rightarrow \phi(X + 1)) \)
- **Use Case:** \( \forall X (X \geq 0 \rightarrow \phi(X)) \implies \)
Method(2): Loop Invariants with Variants

Induction proofs are very difficult to perform for a loop

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

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 $\{m_1, \ldots, m_k\}$ and a variant $v$. The following cases must be proven.

- **Initially Valid:** $\implies inv \land v \geq 0$
- **Body Preserves Invariant:**
  \[
  \implies \{m_1 := x_1 \parallel \ldots \parallel m_k := x_k\}(inv \land [\{b = \text{COND};\}]b = \text{true} \\
  \implies \langle BODY \rangle inv
  \]
- **Use Case:**
  \[
  \implies \{m_1 := x_1 \parallel \ldots \parallel m_k := x_k\}(inv \land [\{b = \text{COND};\}]b = \text{false} \\
  \implies \langle \ldots \rangle \phi
  \]
- **Termination:**
  \[
  \implies \{m_1 := x_1 \parallel \ldots \parallel m_k := x_k\}(inv \land v \geq 0 \land [\{b = \text{COND};\}]b = \text{true} \\
  \implies \{old := v\}\langle BODY \rangle v \leq old \land v \geq 0
Java code to compute gcd of non-negative numbers:

```java
public static int gcd(int a, int b) {
    while (a != 0 && b != 0) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return (a > b) ? a : b;
}
```

Lets prove it with KeY-System.
Specification

We first need a specification.

Definition (GCD)

Let a and b be natural numbers. A number d is the greatest common divisor (GCD) of a and b iff

1. \( d \mid a \) and \( d \mid b \)
2. If \( c \mid a \) and \( c \mid b \), then \( c \mid d \).

\( d \mid a \) means \( d \) divides \( a \).

\( d \mid a \) \iff \exists q. d \times q = a \)
JML Specification

The specification can be converted to JML:

```java
/*@ 
@ requires a >= 0 && b >= 0;
@ ensures result >= 0;
@ ensures (exists int q; result*q == a) && (exists int q; result*q == b) && (forall int c; (exists int q; c*q == result));
@*/

public static int gcd(int a, int b)

So lets start proving ...
Loop-Invariant

What is the loop invariant?

The algorithm changes $a$ and $b$, but the gcd of $a$ and $b$ should stay the same.

In fact the set of common divisors of $a$ and $b$ never changes. This suggests the following invariant:

$$\forall d. (d | \text{old}(a) \land d | \text{old}(b) \iff d | a \land d | b)$$

In JML this can be specified as:

```jml
/*@ loop_invariant a \geq 0 \land b \geq 0 */
@ (forall int d; true;
@ (exists int q; \text{old}(a) == q*d)
@ \land (exists int q; \text{old}(b) == q*d)
@ \iff (exists int q; a == q*d) \land (exists int q; b == q*d)
@ )
@ assignable a, b;
@ decreases a+b;
@*/
```