KeY Tutorial

Verify This 2021
Mattias Ulbrich | 26 March 2021
more than 20 years experience
more than 20 years experience
more than 20 years experience

many, many, many contributors
KeY

more than 20 years experience

many, many, many contributors
KeY

Java

Deductive Verification

Java Modeling Language JML

www.key-project.org
KeY

Java

Deductive Verification

Java Modeling Language JML

Source Code Analysis

www.key-project.org
KeY

Java

Deductive Verification

Java Modeling Language JML

Source Code Analysis

Sequential Code

Dynamic Logic / Symbolic Execution

www.key-project.org
KeY

Java

Deductive Verification

Java Modeling Language JML

Source Code Analysis

Java 1.4 (+ a bit)

Sequential Code

Dynamic Logic / Symbolic Execution

www.key-project.org
The reference for the system:

*Deductive Software Verification – The KeY Book*

LNCS volume 10001

Springer 2016.
KeY Workflow

1. annotated Java program
2. KeY proof management
3. contract proof obligation
4. first-order proof obligation
5. successful proof
KeY Workflow

1. annotated Java program
2. KeY proof management
3. contract proof obligation
4. first-order proof obligation
5. successful proof

Fully automatic translation
KeY Workflow

1. annotated Java program
   - fully automatic translation
2. KeY proof management
   - automatic translation triggered by user
3. contract proof obligation
4. first-order proof obligation
5. successful proof
KeY Workflow

1. annotated Java program

   - fully automatic translation

2. KeY proof management

   - automatic translation triggered by user

3. contract proof obligation

   - rule-based symbolic execution using KeY

4. first-order proof obligation

   - successful proof
KeY Workflow

1. annotated Java program
   - fully automatic translation

2. KeY proof management
   - automatic translation triggered by user

3. contract proof obligation
   - rule-based symbolic execution using KeY

4. first-order proof obligation
   - KeY + SMT solvers

5. successful proof
KeY Workflow

annotated Java program

fully automatic translation

KeY proof management

automatic translation triggered by user

contract proof obligation

rule-based symbolic execution using KeY

first-order proof obligation

KeY + SMT solvers

successful proof
KeY Workflow

1. annotated Java program
2. fully automatic translation
3. KeY proof management
4. automatic translation triggered by user
5. contract proof obligation
6. rule-based symbolic execution using KeY
7. first-order proof obligation
8. KeY + SMT solvers
9. successful proof
KeY Workflow

JML*
- annotated Java program
  - fully automatic translation
  - KeY proof management

Dynamic Logic
- contract proof obligation
  - automatic translation triggered by user
  - rule-based symbolic execution using KeY
  - first-order proof obligation
  - KeY + SMT solvers
- successful proof
KeY Workflow

- JML*: annotated Java program
  - fully automatic translation
  - KeY proof management

- Dynamic Logic: contract proof obligation
  - automatic translation triggered by user

- FOL: first-order proof obligation
  - rule-based symbolic execution using KeY
  - KeY + SMT solvers

- successful proof
KeY Workflow

- JML*: annotated Java program
- KeY proof management: fully automatic translation
- Dynamic Logic: contract proof obligation
  - automatic translation triggered by user
- FOL: first-order proof obligation
  - rule-based symbolic execution using KeY
- Taclet Language: KeY rule base
- Successful proof: KeY + SMT solvers
KeY Verification Process
A Case Study: The TimSort Bug

[De Gouw et al., CAV 2015]

**TimSort**

- Standard algorithm: Open JDK, Android, Apache, Haskell, Python
- Clever combination of merge sort and insertion sort

Bug found during (failed) verification attempt with KeY

Throws uncaught `ArrayIndexOutOfBoundsException` for certain inputs

Symbolic counter example generation & analysis lead to witness

Interaction (understanding intermediate proof state) crucial

Verification of fixed version with KeY

Proof: JDK code with bug fix does not throw an exception

2,200,000 rule applications – 99.8 % automatic
A Case Study: The TimSort Bug

[De Gouw et al., CAV 2015]

TimSort

- Standard algorithm: Open JDK, Android, Apache, Haskell, Python
- Clever combination of merge sort and insertion sort

Bug found during (failed) verification attempt with KeY

- Throws uncaught ArrayIndexOutOfBoundsException for certain inputs
- Symbolic counter example generation & analysis lead to witness
- Interaction (understanding intermediate proof state) crucial
A Case Study: The TimSort Bug

[De Gouw et al., CAV 2015]

TimSort
- Standard algorithm: Open JDK, Android, Apache, Haskell, Python
- Clever combination of merge sort and insertion sort

Bug found during (failed) verification attempt with KeY
- Throws uncaught ArrayIndexOutOfBoundsException for certain inputs
- Symbolic counter example generation & analysis lead to witness
- Interaction (understanding intermediate proof state) crucial

Verification of fixed version with KeY
- Proof: JDK code with bug fix does not throw an exception
- 2,200,000 rule applications – 99.8 % automatic
The Java Modeling Language

JML

- JML independent of KeY.
- There is a JML community on its own.
- Main person: Gary Leavens (UCF Orlando)
- KeY is one tool amongst others, in particular OpenJML (David Cok)
- Influenced other specification languages, e.g., ACSL
Method Contracts

Post increment

class Increment {
    int x, y;

    //@ behavior
    @ requires true;
    @ ensures x == \old(y);
    @ ensures y == \old(y)+1;
    @ assignable this.x, this.y;
    @ signals (Exception e) false;
    @*
    
    public void increment() {
        x = y++;
    }
}

Method contracts in JML:

- B. Meyer’s Design by contract
Method Contracts

Post increment

```java
class Increment {
    int x, y;

    /*@ behavior
        @ requires true;
        @ ensures x == \old(y);
        @ ensures y == \old(y)+1;
        @ assignable this.x, this.y;
        @ signals (Exception e) false;
    @*/
    public void increment() {
        x = y++;
    }
}
```

Method contracts in JML:

- B. Meyer’s *Design by contract*
- A contract is a triple of
class Increment {
    int x, y;

    /*@ behavior
     * @ requires true;
     * @ ensures x == \old(y);
     * @ ensures y == \old(y)+1;
     * @ assignable this.x, this.y;
     * @ signals (Exception e) false;
     * @*/
    public void increment() {
        x = y++;
    }
}
Method Contracts

Post increment

class Increment {
    int x, y;

    /*@ behavior
        @ requires true;
        @ ensures x == \old(y);
        @ ensures y == \old(y)+1;
        @ assignable this.x, this.y;
        @ signals (Exception e) false;
        @*/
    public void increment() {
        x = y++;
    }
}

Method contracts in JML:
- B. Meyer's *Design by contract*
- A contract is a triple of
  - a precondition requires
  - a postcondition ensures
Method Contracts

Post increment

class Increment {
    int x, y;

    /**@ behavior
    @ requires true;
    @ ensures x == \old(y);
    @ ensures y == \old(y)+1;
    @ assignable this.x, this.y;
    @ signals (Exception e) false;
    @*/
    public void increment() {
        x = y++;
    }
}

Method contracts in JML:

- B. Meyer’s *Design by contract*
- A contract is a triple of
  - a precondition *requires*
  - a postcondition *ensures*
  - a frame *assignable*
Post increment

```java
class Increment {
    int x, y;

    /*@ behavior
    @ requires true;
    @ ensures x == \old(y);
    @ ensures y == \old(y)+1;
    @ assignable this.x, this.y;
    @ signals (Exception e) false;
    @*/
    public void increment() {
        x = y++;
    }
}
```

Method contracts in JML:

- B. Meyer's *Design by contract*
- A contract is a triple of
  - a precondition `requires`
  - a postcondition `ensures`
  - a frame `assignable`
- JML keywords start with backslash (`\old, \forall, ...`)
Method Contracts

Post increment

class Increment {
    int x, y;

    /*@ behavior */
    @ requires true;
    @ ensures x == \old(y);
    @ ensures y == \old(y)+1;
    @ assignable this.x, this.y;
    @ signals (Exception e) false;
    @*/

    public void increment() {
        x = y++;
    }
}

Method contracts in JML:

- B. Meyer's *Design by contract*
- A contract is a triple of
  - a precondition \texttt{requires}
  - a postcondition \texttt{ensures}
  - a frame \texttt{assignable}
- JML keywords start with backslash (\texttt{old}, \texttt{forall}, ...)
- Exceptional cases specified separately (\texttt{signals})
Method Contracts

### Post increment

```java
class Increment {
    int x, y;

    /**@ behavior
     * @ requires true;
     * @ ensures x == \old(y);
     * @ ensures y == \old(y)+1;
     * @ assignable this.x, this.y;
     * @ signals (Exception e) false;
     */
    public void increment() {
        x = y++;
    }
}
```

### Method contracts in JML:
- B. Meyer’s *Design by contract*
- A contract is a triple of
  - a precondition `requires`
  - a postcondition `ensures`
  - a frame `assignable`
- JML keywords start with backslash (\old, \forall, ...)
- Exceptional cases specified separately (signals)
- mostly interested in the `normal_behavior`
Object Invariants

Array access

class SomeClass {
    int[] array;
    int index;
    // ...

    //@ invariant

    //@ normal_behavior
    @ requires true;
    @ ensures true;
    //@
    public int getAtIndex() {
        return array[index];
    }
}

Object invariants in JML:
Object Invariants

Array access

class SomeClass {
    int[] array;
    int index;
    // ...

    //@ invariant 0 <= index &&
    //@ index < array.length;

    /**@ normal_behavior
        @ requires true;
        @ ensures true;
        @*/
    public int getAtIndex() {
        return array[index];
    }
}

Object invariants in JML:
Object Invariants

Array access

class SomeClass {
    int[] array;
    int index;
    // ...

    //@ invariant 0 <= index &&
    //@     index < array.length;

   /*@ normal_behavior
        @ requires true;
        @ ensures true;
        @*/
    public int getAtIndex() {
        return array[index];
    }
}
Object Invariants

Array access

class SomeClass {
    int[] array;
    int index;
    // ...

    //@ invariant 0 <= index &&
    //@      index < array.length;

    /*@ normal_behavior
       @ requires true;
       @ ensures true;
       @*/
    public int getAtIndex() {
        return array[index];
    }
}

Object invariants in JML:
- also called instance invariants
- start with \texttt{invariant}
Object Invariants

Array access

class SomeClass {
    int[] array;
    int index;
    // ...

    //@ invariant 0 <= index &&
    //@     index < array.length;

    //@ normal_behavior
    @ requires true;
    @ ensures true;
    @*/
    public int getAtIndex() {
        return array[index];
    }
}

Object invariants in JML:

- also called instance invariants
- start with `invariant`
- “visible state semantics”
Object Invariants

Array access

class SomeClass {
    int[] array;
    int index;
    // ...
    //@ invariant 0 <= index &&
    //@ index < array.length;
    
    /*@ normal_behavior
        @ requires true;
        @ ensures true;
        @*/
    public int getAtIndex() {
        return array[index];
    }
}

Object invariants in JML:

- also called instance invariants
- start with `invariant`
- “visible state semantics”
- There are defaults:
Object Invariants

Array access

```java
class SomeClass {
    int[] array;
    int index;
    // ...

    //@ invariant 0 <= index &&
    //@     index < array.length;

    //@ normal_behavior
    @ requires true;
    @ ensures true;
    @*/
    public int getAtIndex() {
        return array[index];
    }
}
```

Object invariants in JML:
- also called instance invariants
- start with `invariant`
- “visible state semantics”
- There are defaults:
  - all fields are non-null: `invariant array != null`

Object Invariants
Object Invariants

Array access

class SomeClass {
    int[] array;
    int index;
    // ...

    //@ invariant 0 <= index &&
    //@ index < array.length;

    //@ normal_behavior
    @ requires true;
    @ ensures true;
    @*/
    public int getAtIndex() {
        return array[index];
    }
}

Object invariants in JML:

- also called instance invariants
- start with invariant
- “visible state semantics”
- There are defaults:
  - all fields are non-null: invariant array != null

Object invariants in KeY (JML*):

Object Invariants

Array access

class SomeClass {
    int[] array;
    int index;
    // ...

    //@ invariant 0 <= index &&
    //@ index < array.length;

   /*@ normal_behavior
       @ requires true;
       @ ensures true;
       @*/
    public int getAtIndex() {
        return array[index];
    }
}

Object invariants in JML:

- also called instance invariants
- start with `invariant`
- “visible state semantics”
- There are defaults:
  - all fields are non-null: `invariant array != null`

Object invariants in KeY (JML*):

- Explicit predicate: `\invariant_for(·)`
Object Invariants

Array access

class SomeClass {
    int[] array;
    int index;
    // ...

    //@ invariant 0 <= index &&
    //@     index < array.length;

   /*@ normal_behavior
    @ requires true;
    @ ensures true;
    @*/
    public int getAtIndex() {
        return array[index];
    }
}

Object invariants in JML:

- also called instance invariants
- start with `invariant`
- “visible state semantics”
- There are defaults:
  - all fields are non-null: `invariant array != null`

Object invariants in KeY (JML*):

- Explicit predicate: `\invariant_for(·)`
- There are defaults:
Object Invariants

Array access

class SomeClass {
    int[] array;
    int index;
    // ...

    //@ invariant 0 <= index &&
    //@ index < array.length;

    /** normal_behavior
     * @ requires true;
     * @ ensures true;
     */
    public int getAtIndex() {
        return array[index];
    }
}

Object invariants in JML:
- also called instance invariants
- start with `{invariant}
- ”visible state semantics”
- There are defaults:
  - all fields are non-null: `{invariant array != null}

Object invariants in KeY (JML*):
- Explicit predicate: `{\invariant_for(\).
- There are defaults:
  - `{requires \invariant_for(this);`
Array access

class SomeClass {
    int[] array;
    int index;
    // ...

    //@ invariant 0 <= index &&
    //@ index < array.length;

    /** normal_behavior
        @ requires true;
        @ ensures true;
        @@*/
    public int getAtIndex() {
        return array[index];
    }
}

Object invariants in JML:

- also called instance invariants
- start with `invariant`
- “visible state semantics”
- There are defaults:
  - all fields are non-null: `invariant array != null`

Object invariants in KeY (JML*):

- Explicit predicate: `\invariant_for(·)`
- There are defaults:
  - `requires \invariant_for(this);`
  - `ensures \invariant_for(this);`
Loop Specifications

Three specification items
- Loop invariant(s) `loop_invariant`
  (must hold before and after every loop iteration)
- Loop variant `decreases`
- A loop frame `assignable`

```java
for(int i = 0; i < a.length; i++) {
    a[i] = i;
}
```
Java allows more than we’d wish for.

Find invariants for these loops:

```java
/**
 * @loop_invariant
 * @loop_invariant
 * @decreases
 * @assignable
 */
for(int i = 0; i < a.length; i++) {
a[i] = i;
}
```

(taken verbatim from java.util.DualPivotQuicksort, openjdk-7)

Proving JDK’s Dual Pivot Quicksort Correct [Beckert et al. VSTEE 2017]
JML extends Java

Any side-effect-free Java expression is also a JML expression.

JML expressions

- $A \Rightarrow B$ ... logical implication
- $A \iff B$ ... logical equivalence
- $\text{old}(\cdot)$ ... value at method start

Quantifiers

- **always** in parentheses
- mostly used with integers
- allowed to quantify over all types, not only integer (which raises questions on the domain of all objects ...)
- generalisations exist: $\sum$, $\text{count}$, etc.

\[
\exists\ int \ x; \ x/2 = x/4
\]
\[
\forall\ int \ i,j; \ 0\leq j \land i<j \land j<a.length; \ a[i] < a[j]
\]
\[
\sum\ int \ i; \ 0\leq i \land i<a.length; \ a[i]
\]
Java Dynamic Logic

JavaDL

- Dynamic Logic proposed late 70s/early 80s
- Pratt, Vaughan, Fisher, Ladner
- Harel has good theory
Dynamic Logic

- Basis: Typed first-order logic
- Modal logic
- Programs constitute the modalities.
- Class declarations remain in background

\[ [p] \varphi : \text{If } p \text{ terminates, then } \varphi \text{ holds in the final state (partial) } \]
\[ \langle p \rangle \varphi : p \text{ terminates and } \varphi \text{ holds in the final state (total) } \]

Other Program Logics

\[
\psi \rightarrow [p] \varphi \iff \{\psi\} p \{\varphi\} \iff \psi \rightarrow \text{wlp}(p, \varphi)
\]

weakest (liberal) precondition

(mostly)
Sequent calculus (Gentzen-style calculus)

Sequent is of the shape

\[ \gamma_1, \ldots, \gamma_n \Rightarrow \delta_1, \ldots, \delta_m \]

(meaning \( \land \gamma_i \rightarrow \lor \delta_i \))

Rules are of the form

\[
\frac{\Gamma_1 \Rightarrow \Delta_1 \ldots \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta}
\]

Rules are applied from bottom to top:
“If I have to show the conclusion, I can instead show the premisses.”

Sample FOL rules

\[
\frac{a, b \Rightarrow}{a \land b \Rightarrow} \\
\frac{\Rightarrow a \quad \Rightarrow b}{\Rightarrow a \land b} \\
\frac{\Rightarrow \varphi[x/c]}{\Rightarrow \forall x.\varphi}
\]

for a fresh constant \( c \)
Local variable assignment

\[ \implies \{ x := v \} \varphi \]

\[ \implies [x = v] \varphi \]

Think of \( \{ x := v \} \varphi \) as “let \( x = v \) in \( \varphi \)”
The Calculus: Symbolic Execution

Local variable assignment

\[ \Rightarrow \{ x := v \} \varphi \Rightarrow [x = v;] \varphi \]

Think of \( \{ x := v \} \varphi \) as “let \( x = v \) in \( \varphi \)”

Field assignment

\[
\begin{align*}
\text{[NULL]} & \quad o = \text{null} \quad \Rightarrow \quad [\text{throw new NullPointerException();}] \varphi \\
\text{[NORMAL]} & \quad o \neq \text{null} \quad \Rightarrow \quad \{ \text{heap := store(heap, o, C::f, v)} \} \varphi \\
& \Rightarrow \quad [o.f = v;] \varphi
\end{align*}
\]
The Calculus: Symbolic Execution

Local variable assignment

\[
\{ x := v \} \varphi \quad \Rightarrow \quad [ x = v ; ] \varphi
\]

Think of \( \{ x := v \} \varphi \) as "let \( x = v \) in \( \varphi \)"

Field assignment

- [NULL] \( o = \text{null} \) \( \Rightarrow \) \( [\text{throw new NullPointerException();}] \varphi \)
- [NORMAL] \( o \neq \text{null} \) \( \Rightarrow \) \( \{ \text{heap := store(heap, o, C::f, v)} \} \varphi \)

\( \Rightarrow \) \( [ o.f = v ; ] \varphi \)

Heaps are 2-dimensional McCarthy arrays

Over 1500 rules in total

Many, many more rules

16/25 26 March 2021 Mattias Ulbrich: KeY Tutorial Institute of Information Security and Dependable Systems (KASTEL)
The Calculus: Symbolic Execution

Local variable assignment

\[ \Rightarrow \{x := v\}\varphi \]
\[ \Rightarrow [x = v;]\varphi \]

Think of \( \{x := v\}\varphi \) as “let \( x = v \) in \( \varphi \)”

Field assignment

\begin{align*}
\text{[NULL]} & \quad o = \text{null} & \Rightarrow & \quad [\text{throw new NullPointerException();}]\varphi \\
\text{[NORMAL]} & \quad o \neq \text{null} & \Rightarrow & \quad \{\text{heap := store(heap, o, C::f, v)}\}\varphi \\
& & \Rightarrow & \quad [o.f = v;]\varphi
\end{align*}

Heaps are 2-dimensional McCarthy arrays

and many, many more rules

about 200 rules for symbolic execution
over 1500 rules in total
Loops

\[
\begin{align*}
\text{simpleInv} & \implies inv \\
\implies \mathcal{A}_{\text{heap}} \mathcal{A}_{\text{local}} \left( (inv \land se \doteq \text{TRUE}) \rightarrow [p_{\text{norm}}]inv \right) \\
\implies \mathcal{A}_{\text{heap}} \mathcal{A}_{\text{local}} \left( (inv \land se \doteq \text{FALSE}) \rightarrow [\pi \omega]\varphi \right) \\
\implies [\pi \text{while}(se) \{ p_{\text{norm}} \} \omega]\varphi
\end{align*}
\]

where

- \( se \) is a simple expression and \( p_{\text{norm}} \) cannot terminate abruptly;
- \((inv, mod, term)\) is a loop specification for the loop to which the rule is applied;
- \( \mathcal{A}_{\text{heap}} = \{ \text{heap} := c_h \} \) anonymizes the heap; \( c_h : \text{Heap} \) is a fresh constant;
- \( \mathcal{A}_{\text{local}} = \{ l_1 := c_1 || \cdots || l_n := c_n \} \) anonymizes all local variables \( l_1, \ldots, l_n \) that are the target of an assignment (left-hand side of an assignment statement) in \( p_{\text{norm}} \); each \( c_i \) is a fresh constant of the same type as \( l_i \).
Loops

\[
\begin{align*}
\text{abruptTermInv} & \quad \Rightarrow inv \\
& \quad \Rightarrow A_{\text{heap}} A_{\text{local}} \left( (\text{inv} \land [b=\text{nse}; b \triangleright \text{TRUE}) \rightarrow [b=\text{nse}; p]\text{post} \right) \\
& \quad \Rightarrow A_{\text{heap}} A_{\text{local}} \left( (\text{inv} \land [b=\text{nse}; b \triangleright \text{FALSE}) \rightarrow [\pi b=\text{nse}; \omega]\phi \right) \\
& \quad \Rightarrow [\pi \text{ while}(\text{nse}) \{ p \} \omega]\phi
\end{align*}
\]

\[
\begin{align*}
\text{post} & \quad (\text{EXCEPTION} \neq \text{null} \rightarrow [\pi \text{ throw EXCEPTION}; \omega]\phi) \\
& \quad \land (\text{BREAK} \triangleright \text{TRUE} \rightarrow [\pi \omega]\phi) \\
& \quad \land (\text{RETURN} \triangleright \text{TRUE} \rightarrow [\pi \text{ return res}; \omega]\phi) \\
& \quad \land (\text{NORMAL} \rightarrow \text{inv})
\end{align*}
\]
Observation: Framing Problem

It may be more challenging to prove that things do not happen than that they happen.
Dynamic Frames

Observation: Framing Problem

It may be more challenging to prove that things do not happen than that they happen.

Solutions include separation logic, ownership types, dynamic frames, . . . .
Dynamic Frames

Observation: Framing Problem

It may be more challenging to prove that things do *not* happen than that they happen.

**Solutions** include separation logic, ownership types, *dynamic frames*, . . .
Observation: Framing Problem

It may be more challenging to prove that things do *not* happen than that they happen.

Solutions include separation logic, ownership types, **dynamic frames**, . . .

\[ \text{locset} \rightarrow \text{data type for dynamic frames (JML*)} \]
Observation: Framing Problem

It may be more challenging to prove that things do not happen than that they happen.

Solutions include separation logic, ownership types, dynamic frames, . . . .

\texttt{\textbackslash locset} – data type for dynamic frames (JML*)

\texttt{\textbackslash locset = java.lang.Object \times FieldName}
Observation: Framing Problem

It may be more challenging to prove that things do not happen than that they happen.

Solutions include separation logic, ownership types, dynamic frames, . . . .

\texttt{\textbackslash locset} – data type for dynamic frames (JML*)

- \texttt{\textbackslash locset} = \texttt{java.lang.Object} \times \textit{FieldName}
- \texttt{o.f} \rightsquigarrow (\texttt{o}, \texttt{C::f}), \quad \texttt{o.*} \rightsquigarrow \bigcup_f (\texttt{o}, \texttt{f})
Observation: Framing Problem

It may be more challenging to prove that things do not happen than that they happen.

Solutions include separation logic, ownership types, dynamic frames, . . . .

\texttt{\textbackslash locset} – data type for dynamic frames (JML*)

- \texttt{\textbackslash locset} = java.lang.Object × \textit{FieldName}
- \texttt{o.f} \rightsquigarrow (\textit{o}, \texttt{C::f}), \quad \texttt{o.*} \rightsquigarrow \bigcup \texttt{f}(\textit{o}, f)
- \texttt{a[i..j]} \rightsquigarrow \bigcup_{k=i}^{j} (\textit{a}, [k]), \quad \texttt{a[*]} \rightsquigarrow \bigcup_{k\geq 0} (\textit{a}, [k])
Observation: Framing Problem

It may be more challenging to prove that things do not happen than that they happen.

Solutions include separation logic, ownership types, dynamic frames, . . . .

\begin{itemize}
  \item \texttt{\locset} – data type for dynamic frames (JML*)
  \item \texttt{\locset} = \texttt{java.lang.Object} × \textit{FieldName}
  \item o.f \rightsquigarrow (o, C::f), \quad o.* \rightsquigarrow ∪_f (o, f)
  \item a[i..j] \rightsquigarrow ∪_{k=i}^j (a, [k]), \quad a[*] \rightsquigarrow ∪_{k≥0} (a, [k])
  \item \texttt{\nothing} = ∅
\end{itemize}
Dynamic Frames

Observation: Framing Problem

It may be more challenging to prove that things do not happen than that they happen.

Solutions include separation logic, ownership types, dynamic frames, . . . .

\textbackslash locset – data type for dynamic frames (JML*)

- \textbackslash locset = \text{java.lang.Object} \times \text{FieldName}
- o.f \leadsto (o, C::f), \quad o.* \leadsto \bigcup_{f}(o, f)
- a[i..j] \leadsto \bigcup_{k=i}^{j}(a, [k]), \quad a[*] \leadsto \bigcup_{k\geq 0}(a, [k])
- \textbackslash nothing = \emptyset
- ghost/model fields.
Observation: Framing Problem

It may be more challenging to prove that things do not happen than that they happen.

Solutions include separation logic, ownership types, dynamic frames, ....

Dynamic Frames

Typical pattern (+ a grain of salt)

```java
//@ ghost \locset footprint;
//@ invariant footprint =
//@ this.* \cup next.footprint;
//@ ... 
//@ ensures \new_elems_fresh(footprint);
//@ assignable footprint;
void m();
//@ ...
//@ accessible footprint;
int /*@pure*/ query();
```

\texttt{\locset} – data type for dynamic frames (JML*)

- \texttt{\locset} = java.lang.Object × FieldName
- o.f \rightsquigarrow (o, C::f), o.* \rightsquigarrow ∪_f(o, f)
- a[i..j] \rightsquigarrow ∪_{k=i}^j(a, [k]), a[*] \rightsquigarrow ∪_{k≥0}(a, [k])
- \texttt{\nothing} = \emptyset
- ghost/model fields.
Typical problem

```java
o.query() == o.query()@heap[p.g:=5]
```

- With `accessible` no need to look inside the definitions

Pure method

```java
//@ accessible footprint;
int /*@pure*/ query();
```
Typical problem

```java
o.query() == o.query()@heap[p.g:=5]
```

- With `accessible` no need to look inside the definitions
- if two heaps are equal on footprint, then queries evaluate to same value.

Pure method

```java
//@ accessible footprint;
int /*@pure*/ query();
```
Dynamic Frames – Dependency Analysis

Typical problem

```
o.query() == o.query()@heap[p.g:=5]
```

- With `accessible` no need to look inside the definitions
- if two heaps are equal on footprint, then queries evaluate to same value.
- Proof obligation:

  \[
  o.footprint@h_1 = o.footprint@h_2,
  \{heap := h_1\}[r = o.query();]q_1 = r,
  \{heap := h_2\}[r = o.query();]q_2 = r
  \]

  \[\implies q_1 = q_2\]

Pure method

```java
//@ accessible footprint;
int /*@pure*/ query();
```
Typical problem

```
o.query() == o.query()@heap[p.g:=5]
```

- With **accessible** no need to look inside the definitions
- if two heaps are equal on footprint, then queries evaluate to same value.
- Proof obligation:

  $o.\text{footprint}@h_1 = o.\text{footprint}@h_2,$
  \[
  \{ heap := h_1 \}[r = o.\text{query}();]q_1 = r,
  \{ heap := h_2 \}[r = o.\text{query}();]q_2 = r
  \]

  $\implies q_1 = q_2$

- Non-interference proof wrt. $\mathcal{C}_0.\text{footprint}$
Typical problem

\[ o.\text{query}() == o.\text{query}()@\text{heap}[p.g:=5] \]

- With **accessible** no need to look inside the definitions
- if two heaps are equal on footprint, then queries evaluate to same value.

Proof obligation:

- \( o.\text{footprint}@h_1 = o.\text{footprint}@h_2, \)
- \( \{ heap := h_1 \}[r = o.\text{query}();]q_1 = r, \)
- \( \{ heap := h_2 \}[r = o.\text{query}();]q_2 = r \)

\[ \implies q_1 = q_2 \]

- Non-interference proof wrt. \( C_o.\text{footprint} \)
- Then axiom in logic: \( o.\text{footprint}@h_1 = o.\text{footprint}@h_2 \rightarrow o.\text{query}()@h_1 = o.\text{query}@h_2 \)
**Typical problem**

\[ o.\text{query}() == o.\text{query}()@\text{heap}[p.g:=5] \]

- With **accessible** no need to look inside the definitions
- if two heaps are equal on footprint, then queries evaluate to same value.
- Proof obligation:

  \[
  \begin{align*}
  & o.\text{footprint}@h_1 = o.\text{footprint}@h_2, \\
  & \{ \text{heap} := h_1 \}[r = o.\text{query}(); q_1 = r, \\
  & \{ \text{heap} := h_2 \}[r = o.\text{query}(); q_2 = r] \\
  \end{align*}
  \]

  \[ \implies q_1 = q_2 \]

- Non-interference proof wrt. \( \text{Co.footprint} \)
- Then axiom in logic: \( o.\text{footprint}@h_1 = o.\text{footprint}@h_2 \to o.\text{query}()@h_1 = o.\text{query}@h_2 \)
- Caution with recursive queries ...

**Pure method**

```java
//@ accessible footprint;
int /*@pure*/ query();
```
Typical problem

\[ o.\text{query()} == o.\text{query()}@\text{heap}[p.g:5] \]

- With \textit{accessible} no need to look inside the definitions
- if two heaps are equal on footprint, then queries evaluate to same value.
- Proof obligation:

\[
\begin{align*}
\text{o.footprint}@h_1 &= \text{o.footprint}@h_2, \\
\{heap := h_1\}[r = o.\text{query}();]q_1 &= r, \\
\{heap := h_2\}[r = o.\text{query}();]q_2 &= r \\
\implies q_1 &= q_2
\end{align*}
\]

- Non-interference proof wrt. \(\text{Co.o.footprint}\)
- Then axiom in logic: \(\text{o.footprint}@h_1 = \text{o.footprint}@h_2 \rightarrow o.\text{query()}@h_1 = o.\text{query}@h_2\)
- Caution with recursive queries ...
- Automation ...

Pure method

```java
//@ accessible footprint;
int /*@pure*/ query();
```
KeY as a Platform

https://www.key-project.org/download/  “Single Click Jar”

- deductive Java verification
- also for concurrent code (permissions)
- support for the full JavaCard language (incl. transactions)
- test case generation
- counterexample generation
- symbolic execution engine for Java
- symbolic execution debugger
- deductive information flow analysis (with two DL-operators)
- floating point support (brand new)
- open source (GPL / EPL)
The tool

https://www.key-project.org/download/  “Single Click Jar”

“Single Click Jar”

java -jar key-2.8.0-exe.jar
Demo
KeY usually loads all .java files in a directory and all subdirectories.

Good workflow:
- Load proof obligation
- Right click on ⇒.
- Choose the “Full Auto Pilot” Macro
- Inspect unclosed goals.
- When hoping for a closed goal apply macro “Try close goals below”.

“Hide Intermediate ProofSteps”, “Expand Goals Only” from context menu in proof make that a lot more readible.

Prefer \strictly_nothing over \nothing

The origin of a formula is highlighted when hovering (orange)

Symbolic execution trace is highlighted (green)

The challenges should work without help of an smt solver
Micro Challenge
Micro Challenge