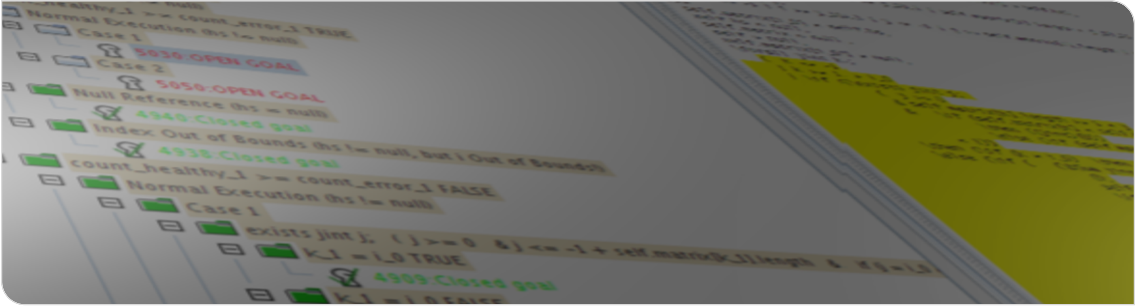
Karlsruher Institut für Technologie

KeY Tutorial

Verify This 2021

Mattias Ulbrich | 26 March 2021



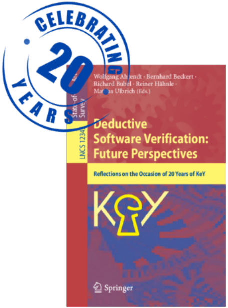
KeY

more than 20 years experience

KeY

Karlsruher Institut für Technologie

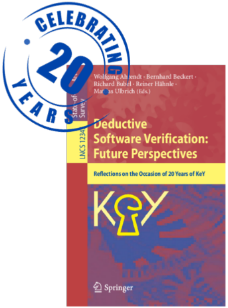
**more than 20
years experience**



LNCS 12345

more than 20
years experience

many, many, many
contributors

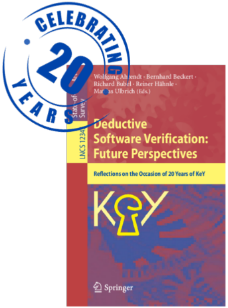


LNCS 12345

Karlsruher Institut für Technologie

more than 20 years experience

many, many, many contributors



LNCS 12345

Karlsruhe Institute of Technology

TECHNISCHE UNIVERSITÄT DARMSTADT

KeY

www.key-project.org

Java

www.key-project.org

Java

Deductive Verification

www.key-project.org

Java

Deductive Verification

Java Modeling Language JML

www.key-project.org

Java

Deductive Verification

Java Modeling Language JML

Source Code Analysis

`www.key-project.org`

Karlsruher Institut für Technologie

Java

Deductive Verification

Java Modeling Language JML

Dynamic Logic / Symbolic Execution

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

Java

Java 1.4
(+ a bit)

Deductive
Verification

Sequential
Code

Java Modeling
Language JML

Dynamic Logic /
Symbolic Execution

Source Code
Analysis

www.key-project.org

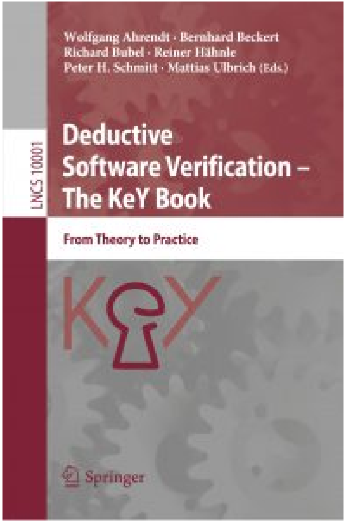
The KeY Book

The reference for the system:

Deductive Software Verification – The KeY Book

LNCS volume 10001

Springer 2016.



KeY Workflow

annotated Java program

↓

KeY proof management

↓

contract proof obligation

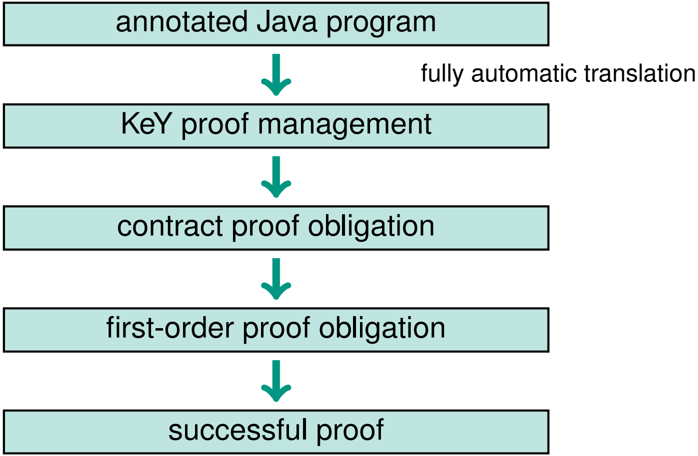
↓

first-order proof obligation

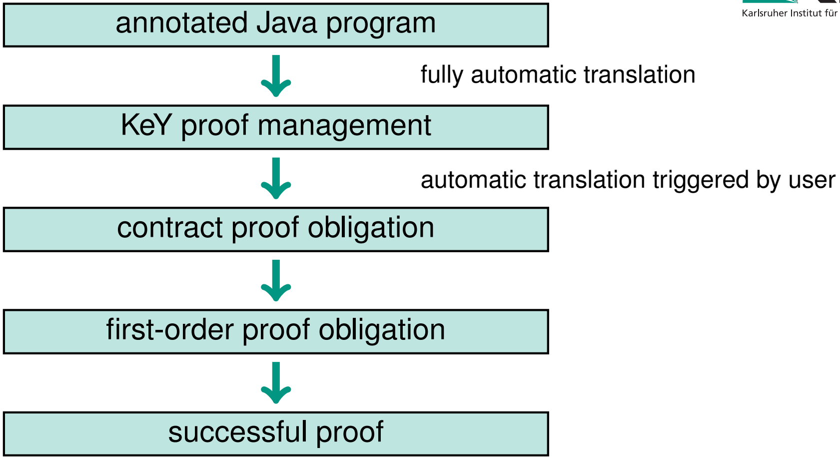
↓

successful proof

KeY Workflow



KeY Workflow



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

↓

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

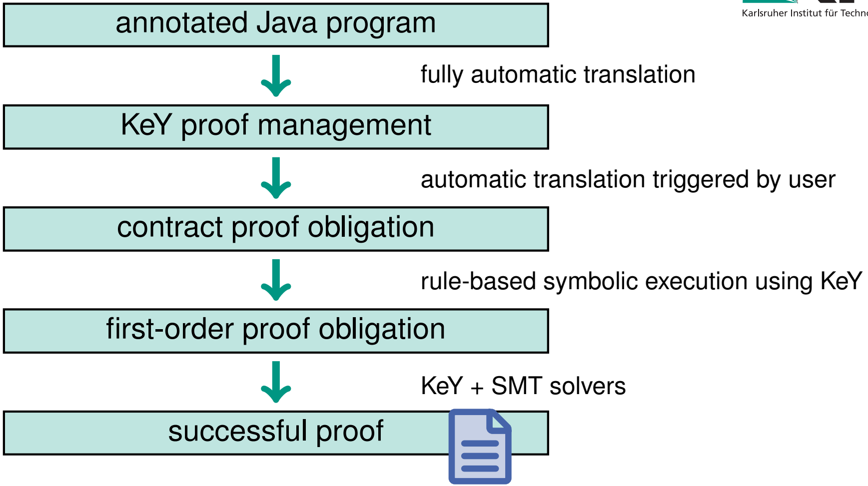
first-order proof obligation

↓ KeY + SMT solvers

successful proof

KeY Workflow

Karlsruher Institut für Technologie



KeY Workflow

JML*

annotated Java program

↓ fully automatic translation

KeY proof management

↓ automatic translation triggered by user

contract proof obligation

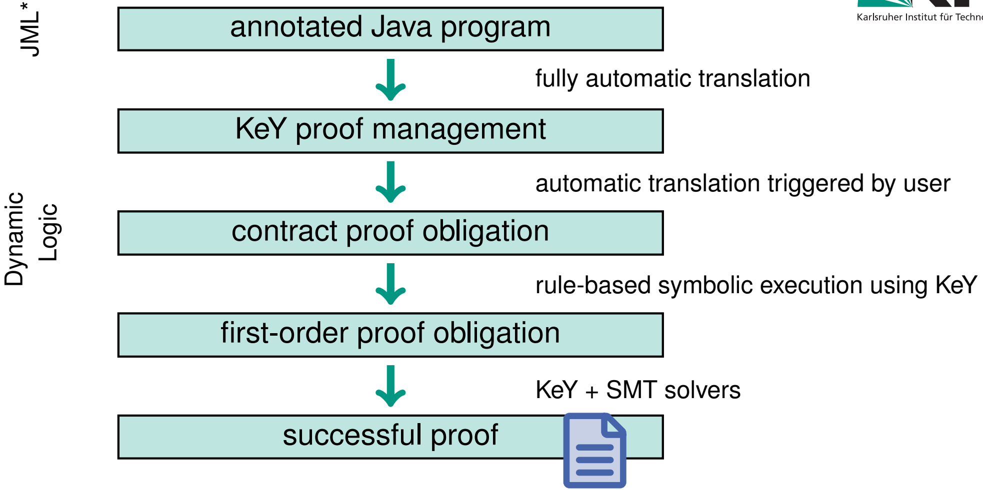
↓ rule-based symbolic execution using KeY

first-order proof obligation

↓ KeY + SMT solvers

successful proof

KeY Workflow



KeY Workflow

JML*

annotated Java program

↓ fully automatic translation

KeY proof management

↓ automatic translation triggered by user

Dynamic Logic

contract proof obligation

↓ rule-based symbolic execution using KeY

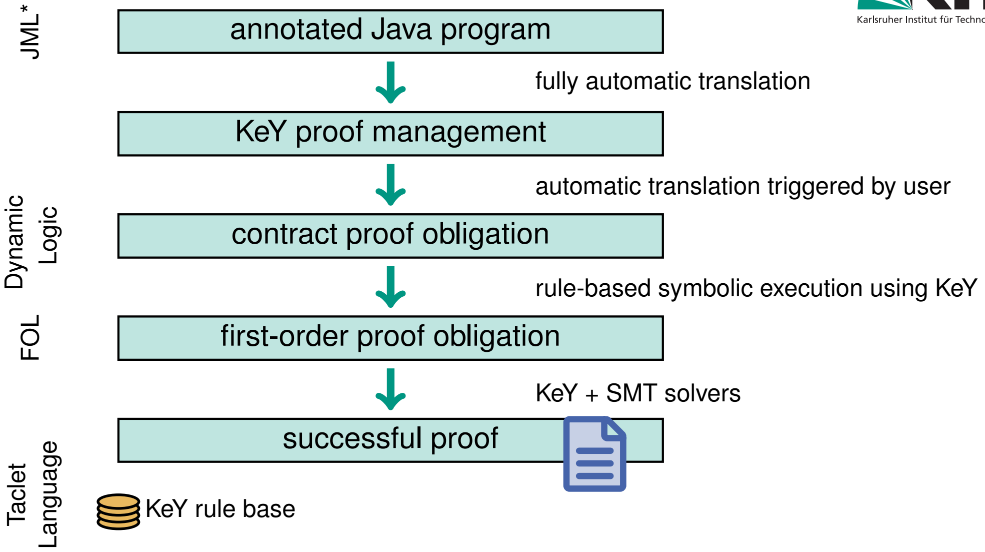
FOL

first-order proof obligation

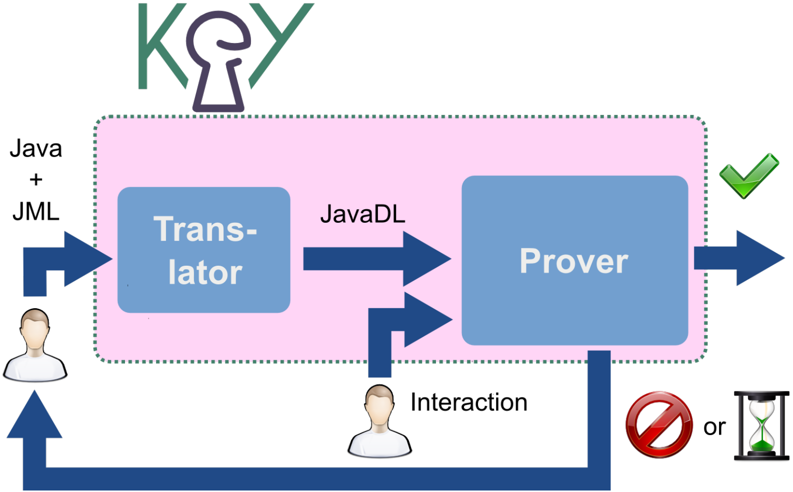
↓ KeY + SMT solvers

successful proof

KeY Workflow



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

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 amonst others, in particular OpenJML (David Cok)
- Influenced other specification languages, e.g., ACSL

Method Contracts

Karlsruher Institut für Technologie

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

Karlsruher Institut für Technologie

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

Method Contracts

Karlsruher Institut für Technologie

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
  1. a precondition `requires`

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
  1. a precondition `requires`
  2. a postcondition `ensures`

Method Contracts

Karlsruher Institut für Technologie

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
  1. a precondition `requires`
  2. a postcondition `ensures`
  3. a frame `assignable`

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
  1. a precondition `requires`
  2. a postcondition `ensures`
  3. 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
  1. a precondition `requires`
  2. a postcondition `ensures`
  3. a frame `assignable`
- JML keywords start with backslash (`\old`, `\forall`, ...)
- Exceptional cases specified separately (`signals`)

Method Contracts

Karlsruher Institut für Technologie

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
  1. a precondition `requires`
  2. a postcondition `ensures`
  3. 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

Karlsruher Institut für Technologie

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

Object Invariants

Karlsruher Institut für Technologie

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`

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

```
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

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

Karlsruher Institut für Technologie

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);`

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);`
  - `ensures \invariant_for(this);`

Loop Specifications

Loop Specifications

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

Three specification items

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

Loop Specifications

Loop Specifi

```
/*@ loop_inva
  @ loop_inva
  @
  @ decreases
  @ assignabl
  @*/
for(int i = 0
  a[i] = i;
}
```

riant

every loop

Java allows more than we'd whish for.

Find invariants for these loops:

```
do {
  [...]
} while (a[++left] >= a[left - 1]);

for (int k = left; ++left <= right; k = ++left) {
  [...]
}
```

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

Proving JDK's Dual Pivot Quicksort Correct [Beckert et al. VSTEE 2017]

Expressions

Karlsruher Institut für Technologie

JML extends Java

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

JML expressions

- *A* `==>` *B* ... logical implication
- *A* `<==>` *B* ... logical equivalence
- `\old(·)` ... 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`, `\count`, etc.

```
(\exists int x; x/2 == x/4)
(\forall int i,j; 0<=j && i<j && j<a.length; a[i] < a[
(\sum int i; 0<=i && 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

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

Other Program Logics

$\psi \to [p]\varphi$	$\Longleftrightarrow$	$\{\psi\}\ p\ \{\varphi\}$	$\Longleftrightarrow$	$\psi \to wlp(p, \varphi)$
DL		Hoare calculus		weakest (liberal) precondition

(mostly)

The Calculus

Karlsruher Institut für Technologie

- Sequent calculus (Gentzen-style calculus)
- Sequent is of the shape

$$\gamma_1, \ldots, \gamma_n \Longrightarrow \delta_1, \ldots, \delta_m$$

(meaning $\bigwedge \gamma_i \to \bigvee \delta_i$)

- Rules are of the form

$$\frac{\Gamma_1 \Longrightarrow \Delta_1 \quad \ldots \quad \Gamma_n \Longrightarrow \Delta_n}{\Gamma \Longrightarrow \Delta}$$

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

Sample FOL rules

$$\frac{a, b \Longrightarrow}{a \wedge b \Longrightarrow}$$

$$\frac{\Longrightarrow a \qquad \Longrightarrow b}{\Longrightarrow a \wedge b}$$

$$\frac{\Longrightarrow \varphi[x/c]}{\Longrightarrow \forall x.\varphi}$$

for a fresh constant c

The Calculus: Symbolic Execution

Local variable assignment

$$\frac{\Longrightarrow \quad \{x := v\}\varphi}{\Longrightarrow \quad [\mathtt{x\ =\ v;}]\varphi}$$

Think of $\{x := v\}\varphi$ as "let $x = v$ in φ"

The Calculus: Symbolic Execution

Local variable assignment

$$\frac{\Longrightarrow \quad \{x := v\}\varphi}{\Longrightarrow \quad [\texttt{x = v;}]\varphi}$$

Think of $\{x := v\}\varphi$ as "let $x = v$ in φ"

Field assignment

$$\frac{\begin{array}{rll} [\text{NULL}] & o = \texttt{null} \Longrightarrow & [\texttt{throw new NullPointerException();}]\varphi \\ [\text{NORMAL}] & o \neq \texttt{null} \Longrightarrow & \{\mathit{heap} := \mathit{store}(\mathit{heap}, o, \texttt{C::f}, v)\}\varphi \end{array}}{\Longrightarrow \quad [\texttt{o.f = v;}]\varphi}$$

The Calculus: Symbolic Execution

Local variable assignment

$$\frac{\Longrightarrow \quad \{x := v\}\varphi}{\Longrightarrow \quad [\texttt{x = v;}]\varphi}$$

Think of $\{x := v\}\varphi$ as "let $x = v$ in φ"

Heaps are 2-dimensional McCarthy arrays

Field assignment

$$\frac{\begin{array}{lll}[\text{NULL}] & o = \texttt{null} \Longrightarrow & [\texttt{throw new NullPointerException();}]\varphi \\ [\text{NORMAL}] & o \neq \texttt{null} \Longrightarrow & \{heap := store(heap, o, \texttt{C::f}, v)\}\varphi\end{array}}{\Longrightarrow \quad [\texttt{o.f = v;}]\varphi}$$

The Calculus: Symbolic Execution

Karlsruher Institut für Technologie

Local variable assignment

$$\frac{\Longrightarrow \{x := v\}\varphi}{\Longrightarrow [\texttt{x = v;}]\varphi}$$

Think of $\{x := v\}\varphi$ as "let $x = v$ in φ"

Field assignment

Heaps are 2-dimensional McCarthy arrays

$$\frac{\begin{array}{lll}[\text{NULL}] & o = \texttt{null} & \Longrightarrow [\texttt{throw new NullPointerException();}]\varphi \\ [\text{NORMAL}] & o \neq \texttt{null} & \Longrightarrow \{heap := store(heap, o, \texttt{C::f}, v)\}\varphi\end{array}}{\Longrightarrow [\texttt{o.f = v;}]\varphi}$$

and many, many more rules

about 200 rules for symbolic execution
over 1500 rules in total

Loops

KIT
Karlsruher Institut für Technologie

$$\text{simpleInv}\ \frac{\begin{array}{l}\Longrightarrow inv \\ \Longrightarrow \mathcal{A}_{heap}\mathcal{A}_{local}\big((inv \land se \doteq TRUE) \rightarrow [p_{norm}]inv\big) \\ \Longrightarrow \mathcal{A}_{heap}\mathcal{A}_{local}\big((inv \land se \doteq FALSE) \rightarrow [\pi\ \omega]\varphi\big)\end{array}}{\Longrightarrow [\pi\ \texttt{while}(se)\ \{\ p_{norm}\ \}\ \omega]\varphi}$$

where

- se is a simple expression and p_{norm} cannot terminate abruptly;
- $(inv, mod, term)$ is a loop specification for the loop to which the rule is applied;
- $\mathcal{A}_{heap} = \{\texttt{heap} := c_h\}$ anonymizes the heap; $c_h{:}Heap$ is a fresh constant;
- $\mathcal{A}_{local} = \{l_1 := c_1 \parallel \cdots \parallel l_n := c_n\}$ 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_{norm}; each c_i is a fresh constant of the same type as l_i.

Loops

$$\text{abruptTermInv}\;\frac{\begin{array}{l}\Longrightarrow inv\\ \Longrightarrow \mathscr{A}_{heap}\mathscr{A}_{local}\big((inv \wedge [\mathsf{b}{=}nse;]\mathsf{b} \doteq TRUE) \;\rightarrow\; [\widehat{\mathsf{b}{=}nse;\,p}]post\big)\\ \Longrightarrow \mathscr{A}_{heap}\mathscr{A}_{local}\big((inv \wedge [\mathsf{b}{=}nse;]\mathsf{b} \doteq FALSE) \;\rightarrow\; [\pi\;\mathsf{b}{=}nse;\;\omega]\varphi\big)\end{array}}{\Longrightarrow [\pi\;\mathtt{while}(nse)\;\{\;p\;\}\;\omega]\varphi}$$

$$\begin{array}{lll} post & & (\textsc{Exception} \not\doteq null \rightarrow [\pi\;\mathtt{throw\;Exception;}\;\omega]\varphi)\\ & \wedge & (\textsc{Break} \doteq TRUE \rightarrow [\pi\;\omega]\varphi)\\ & \wedge & (\textsc{Return} \doteq TRUE \rightarrow [\pi\;\mathtt{return\;res;}\;\omega]\varphi)\\ & \wedge & (\textsc{Normal} \rightarrow inv)\end{array}$$

Dynamic Frames

Karlsruher Institut für Technologie

Observation: Framing Problem

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

Dynamic Frames

Karlsruher Institut für Technologie

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, ...

Dynamic Frames

Karlsruher Institut für Technologie

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, ...

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

Dynamic Frames

Karlsruher Institut für Technologie

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, ...

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

- `\locset` $=$ `java.lang.Object` $\times$ *FieldName*

Dynamic Frames

Karlsruher Institut für Technologie

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, ...

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

- `\locset` $=$ `java.lang.Object` $\times$ *FieldName*
- `o.f` $\rightsquigarrow$ $(o, \texttt{C::f})$, `o.*` $\rightsquigarrow$ $\bigcup_f (o, f)$

Dynamic Frames

Karlsruher Institut für Technologie

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, ...

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

- `\locset` $=$ `java.lang.Object` $\times$ *FieldName*
- `o.f` $\rightsquigarrow (o, \texttt{C::f})$, `o.*` $\rightsquigarrow \bigcup_f (o, f)$
- `a[i..j]` $\rightsquigarrow \bigcup_{k=i}^{j} (a, [k])$, `a[*]` $\rightsquigarrow \bigcup_{k \geq 0} (a, [k])$

Dynamic Frames

Karlsruher Institut für Technologie

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, ...

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

- `\locset` $=$ `java.lang.Object` $\times$ *FieldName*
- `o.f` $\rightsquigarrow (o, \texttt{C::f})$, `o.*` $\rightsquigarrow \bigcup_f (o, f)$
- `a[i..j]` $\rightsquigarrow \bigcup_{k=i}^{j} (a, [k])$, `a[*]` $\rightsquigarrow \bigcup_{k \geq 0} (a, [k])$
- `\nothing` $= \emptyset$

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, ...

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

- `\locset` $=$ `java.lang.Object` $\times$ *FieldName*
- `o.f` $\rightsquigarrow (o, \texttt{C::f})$, `o.*` $\rightsquigarrow \bigcup_f (o, f)$
- `a[i..j]` $\rightsquigarrow \bigcup_{k=i}^{j} (a, [k])$, `a[*]` $\rightsquigarrow \bigcup_{k \geq 0} (a, [k])$
- `\nothing` $= \emptyset$
- ghost/model fields.

Dynamic Frames

Karlsruher Institut für Technologie

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, ...

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

- `\locset` $=$ `java.lang.Object` $\times$ *FieldName*
- `o.f` $\rightsquigarrow (o, \texttt{C::f})$, `o.*` $\rightsquigarrow \bigcup_f (o, f)$
- `a[i..j]` $\rightsquigarrow \bigcup_{k=i}^{j} (a, [k])$, `a[*]` $\rightsquigarrow \bigcup_{k \geq 0} (a, [k])$
- `\nothing` $= \emptyset$
- ghost/model fields.

Typical pattern (+ a grain of salt)

```
//@ ghost \locset footprint;

//@ invariant footprint =
//@   this.* \cup next.footprint;

//@ ...
//@ ensures \new_elems_fresh(footprint);
//@ assignable footprint;
void m();

//@ ...
//@ accessible footprint;
int /*@pure*/ query();
```

Dynamic Frames – Dependency Analysis

Karlsruher Institut für Technologie

Typical problem

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

Pure method

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

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

Dynamic Frames – Dependency Analysis

Karlsruher Institut für Technologie

Typical problem

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

Pure method

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

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

Dynamic Frames – Dependency Analysis

Typical problem

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

Pure method

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

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

$$\frac{\begin{array}{l}\texttt{o.footprint@}h_1 = \texttt{o.footprint@}h_2, \\ \{heap := h_1\}[\texttt{r} = \texttt{o.query();}]q_1 = r, \\ \{heap := h_2\}[\texttt{r} = \texttt{o.query();}]q_2 = r\end{array}}{\Longrightarrow \quad q_1 = q_2}$$

Dynamic Frames – Dependency Analysis

Karlsruher Institut für Technologie

Typical problem

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

Pure method

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

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

$$\frac{\begin{array}{l}\texttt{o.footprint}@h_1 = \texttt{o.footprint}@h_2, \\ \{heap := h_1\}[\texttt{r} = \texttt{o.query();}]q_1 = r, \\ \{heap := h_2\}[\texttt{r} = \texttt{o.query();}]q_2 = r\end{array}}{\Longrightarrow \quad q_1 = q_2}$$

- Non-interference proof wrt. $\complement$`o.footprint`

Dynamic Frames – Dependency Analysis

Karlsruher Institut für Technologie

Typical problem

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

Pure method

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

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

$$\begin{array}{ll} & \texttt{o.footprint}@h_1 = \texttt{o.footprint}@h_2, \\ & \{heap := h_1\}[\texttt{r = o.query();}]q_1 = r, \\ & \{heap := h_2\}[\texttt{r = o.query();}]q_2 = r \\ \hline \Longrightarrow & q_1 = q_2 \end{array}$$

- Non-interference proof wrt. $\complement$`o.footprint`
- Then axiom in logic: $\texttt{o.footprint}@h_1 = \texttt{o.footprint}@h_2 \rightarrow o.query()@h_1 = o.query@h_2$

Dynamic Frames – Dependency Analysis

Typical problem

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

Pure method

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

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

$$\frac{\begin{array}{l}\texttt{o.footprint}@h_1 = \texttt{o.footprint}@h_2,\\ \{heap := h_1\}[\texttt{r = o.query();}]q_1 = r,\\ \{heap := h_2\}[\texttt{r = o.query();}]q_2 = r\end{array}}{\Longrightarrow \quad q_1 = q_2}$$

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

Dynamic Frames – Dependency Analysis

Karlsruher Institut für Technologie

Typical problem

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

Pure method

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

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

$$\begin{array}{ll} & \texttt{o.footprint}@h_1 = \texttt{o.footprint}@h_2, \\ & \{heap := h_1\}[\texttt{r} = \texttt{o.query();}]q_1 = r, \\ & \{heap := h_2\}[\texttt{r} = \texttt{o.query();}]q_2 = r \\ \hline \Longrightarrow & q_1 = q_2 \end{array}$$

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

KeY as a Platform

Karlsruher Institut für Technologie

`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)

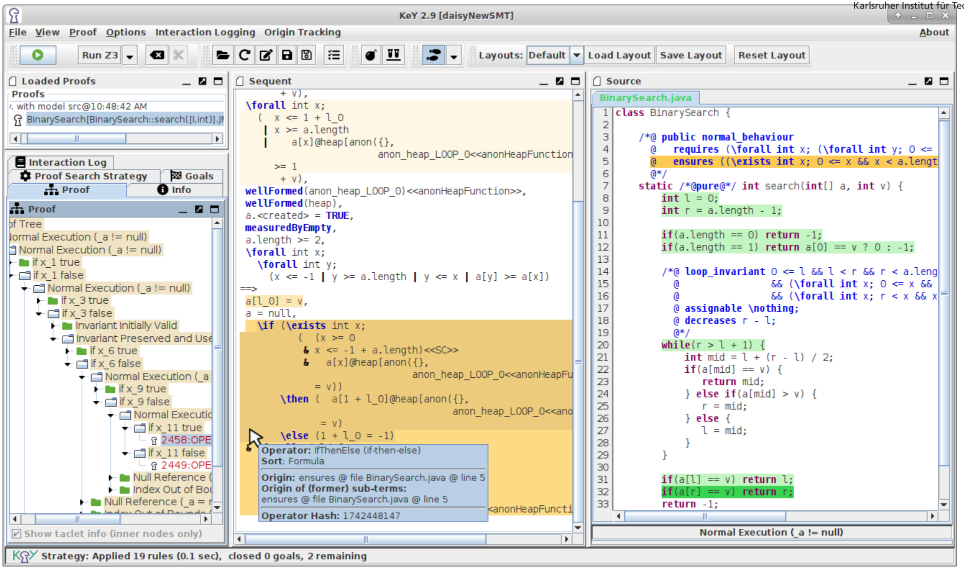
The tool

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

“Single Click Jar”

```
java -jar key-2.8.0-exe.jar
```

Demo

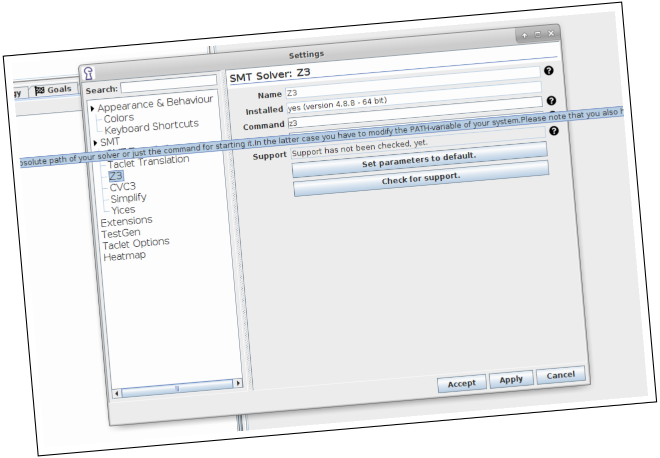


KeY 2.9 [daisyNewSMT]
File View Proof Options Interaction Logging Origin Tracking About
Run Z3
Layouts: Default Load Layout Save Layout Reset Layout
Loaded Proofs
Sequent
Source
BinarySearch.java
Interaction Log
Proof Search Strategy
Goals
Proof
Info
Operator: IfThenElse (if-then-else)
Sort: Formula
Origin: ensures @ file BinarySearch.java @ line 5
Origin of (former) sub-terms:
ensures @ file BinarySearch.java @ line 5
Operator Hash: 1742448147
Normal Execution (_a != null)
Strategy: Applied 19 rules (0.1 sec), closed 0 goals, 2 remaining

Cheat Sheet

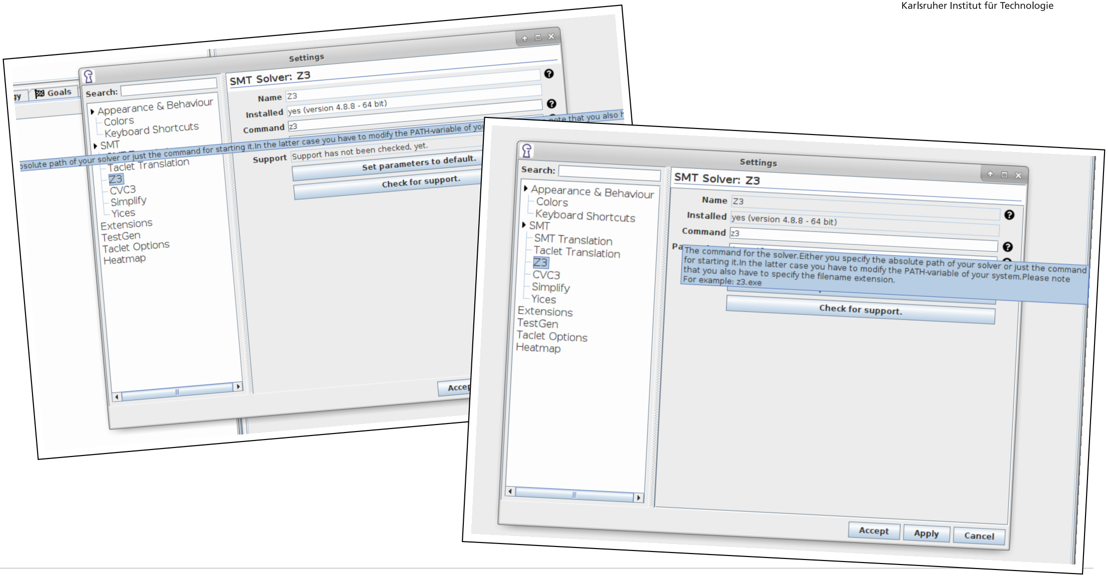
1. KeY usually loads **all** `.java` files in a directory and all subdirectories.
2. Good workflow:
   - Load proof obligation
   - Right click on $\Longrightarrow$.
   - Choose the "Full Auto Pilot" Macro
   - Inspect unclosed goals.
   - When hoping for a closed goal apply macro "Try close goals below".
3. "Hide Intermediate ProofSteps", "Expand Goals Only" from context menu in proof make that a lot more readible.
4. Prefer `\strictly_nothing` over `\nothing`
5. The origin of a formula is highlighted when hovering (orange)
6. Symbolic execution trace is highlighted (green)
7. The challenges should work without help of an smt solver

Micro Challenge



Micro Challenge

Karlsruher Institut für Technologie



Micro Challenge