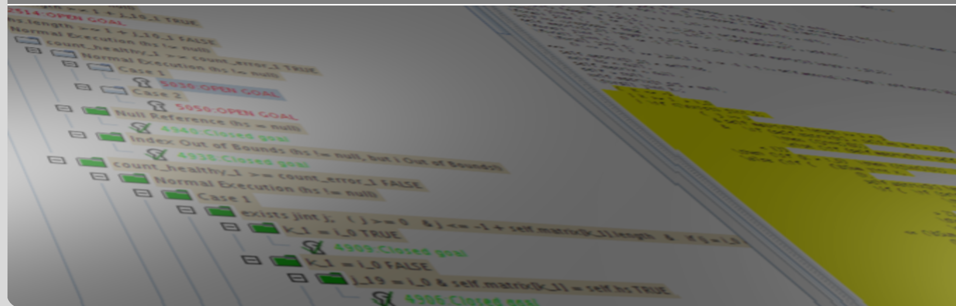
KIT
Karlsruhe Institute of Technology

Deductive Verification of Object-Oriented Software

Part A

Bernhard Beckert | VTSA, 24.–28.08.2015

KIT – Institute for Theoretical Computer Science



Part I

The KeY System – An Overview

Part I

The KeY System – An Overview

KeY Project

Karlsruhe Institute of Technology

www.key-project.org

Project Consortium

- Bernhard Beckert and
  Peter H. Schmitt,
  Karlsruhe Institute of Technology
- Reiner Hähnle
  TU Darmstadt
- Wolfgang Ahrendt
  Chalmers Univ., Gothenburg

KeY Project

www.key-project.org

KeY Project

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Project

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- Java Card (Java 1.4)
- Semi-automated
  (automation and usability both important)

KeY Project

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- Java Card (Java 1.4)
- Semi-automated
  (automation and usability both important)

KeY Project

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- Java Card (Java 1.4)
- Semi-automated
  (automation and usability both important)

KeY Project

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- Java Card (Java 1.4)
- Semi-automated
  (automation and usability both important)

KeY Project

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- Java Card (Java 1.4)
- Semi-automated
  (automation and usability both important)

Specific Features of the KeY Approach

Karlsruhe Institute of Technology

- Part II: The Java Modeling Language
  - Program-level specification and annotation
- Part III: Program Verification with Dynamic Logic
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation
- Part IV: Verifying Information Flow Properties
  - JML extended with information-flow concepts
  - Non-interference expressed in Dynamic Logic
- Not covered in this tutorial
  - Additional benefits: test case generation, symbolic debugging.

Specific Features of the KeY Approach

- Part II: The Java Modeling Language
  - Program-level specification and annotation
- Part III: Program Verification with Dynamic Logic
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation
- Part IV: Verifying Information Flow Properties
  - JML extended with information-flow concepts
  - Non-interference expressed in Dynamic Logic
- Not covered in this tutorial
  - Additional benefits: test case generation, symbolic debugging.

Specific Features of the KeY Approach

- Part II: The Java Modeling Language
  - Program-level specification and annotation
- Part III: Program Verification with Dynamic Logic
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation
- Part IV: Verifying Information Flow Properties
  - JML extended with information-flow concepts
  - Non-interference expressed in Dynamic Logic
- Not covered in this tutorial
  - Additional benefits: test case generation, symbolic debugging.

Specific Features of the KeY Approach

Karlsruhe Institute of Technology

- Part II: The Java Modeling Language
  - Program-level specification and annotation
- Part III: Program Verification with Dynamic Logic
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation
- Part IV: Verifying Information Flow Properties
  - JML extended with information-flow concepts
  - Non-interference expressed in Dynamic Logic
- Not covered in this tutorial
  - Additional benefits: test case generation, symbolic debugging.

Specific Features of the KeY Approach

Karlsruhe Institute of Technology

- Part II: The Java Modeling Language
  - Program-level specification and annotation
- Part III: Program Verification with Dynamic Logic
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation
- Part IV: Verifying Information Flow Properties
  - JML extended with information-flow concepts
  - Non-interference expressed in Dynamic Logic
- Not covered in this tutorial
  - Additional benefits: test case generation, symbolic debugging.

Specific Features of the KeY Approach

Karlsruhe Institute of Technology

- Part II: The Java Modeling Language
  - Program-level specification and annotation
- Part III: Program Verification with Dynamic Logic
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation
- Part IV: Verifying Information Flow Properties
  - JML extended with information-flow concepts
  - Non-interference expressed in Dynamic Logic
- Not covered in this tutorial
  - Additional benefits: test case generation, symbolic debugging.

Specific Features of the KeY Approach

Karlsruhe Institute of Technology

- Part II: The Java Modeling Language
  - Program-level specification and annotation
- Part III: Program Verification with Dynamic Logic
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation
- Part IV: Verifying Information Flow Properties
  - JML extended with information-flow concepts
  - Non-interference expressed in Dynamic Logic
- Not covered in this tutorial
  - Additional benefits: test case generation, symbolic debugging.

Specific Features of the KeY Approach

Karlsruhe Institute of Technology

- Part II: The Java Modeling Language
  - Program-level specification and annotation
- Part III: Program Verification with Dynamic Logic
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation
- Part IV: Verifying Information Flow Properties
  - JML extended with information-flow concepts
  - Non-interference expressed in Dynamic Logic
- Not covered in this tutorial
  - Additional benefits: test case generation, symbolic debugging.

Specific Features of the KeY Approach

KIT Karlsruhe Institute of Technology

- Part II: The Java Modeling Language
  - Program-level specification and annotation
- Part III: Program Verification with Dynamic Logic
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation
- Part IV: Verifying Information Flow Properties
  - JML extended with information-flow concepts
  - Non-interference expressed in Dynamic Logic
- Not covered in this tutorial
  - Additional benefits: test case generation, symbolic debugging.

Specific Features of the KeY Approach

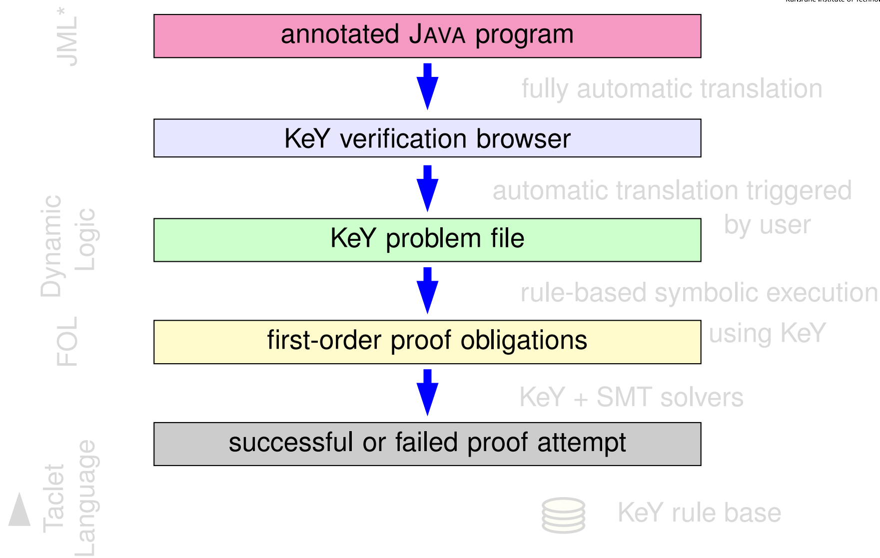
Karlsruhe Institute of Technology

- Part II: The Java Modeling Language
  - Program-level specification and annotation
- Part III: Program Verification with Dynamic Logic
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation
- Part IV: Verifying Information Flow Properties
  - JML extended with information-flow concepts
  - Non-interference expressed in Dynamic Logic
- Not covered in this tutorial
  - Additional benefits: test case generation, symbolic debugging.

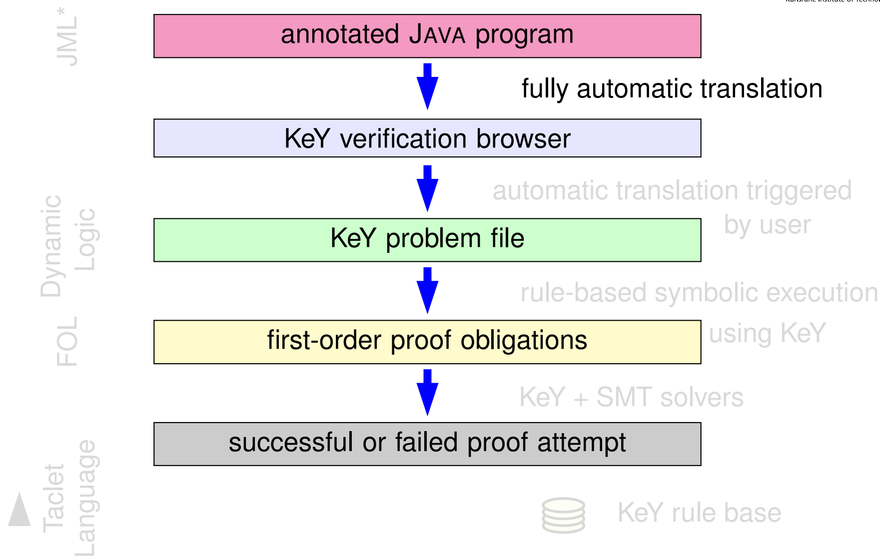
Specific Features of the KeY Approach

- Part II: The Java Modeling Language
  - Program-level specification and annotation
- Part III: Program Verification with Dynamic Logic
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation
- Part IV: Verifying Information Flow Properties
  - JML extended with information-flow concepts
  - Non-interference expressed in Dynamic Logic
- Not covered in this tutorial
  - Additional benefits: test case generation, symbolic debugging.

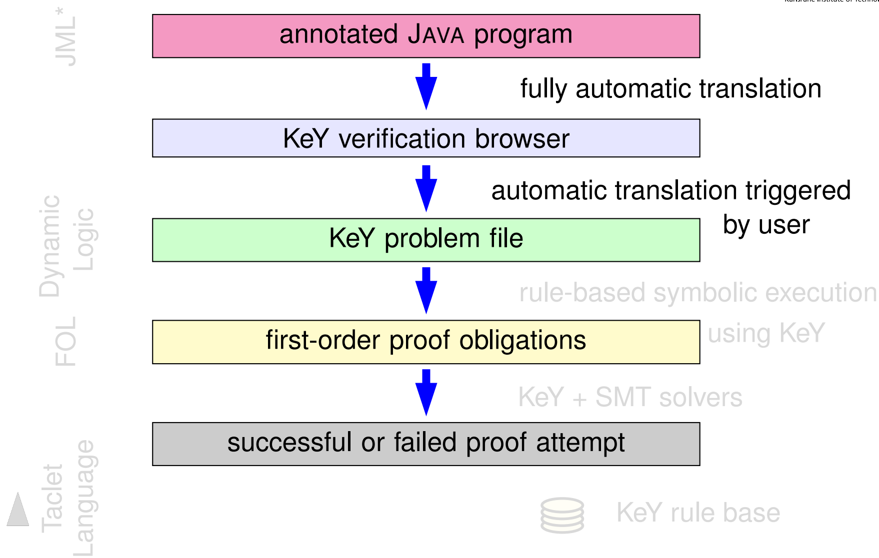
Workflow



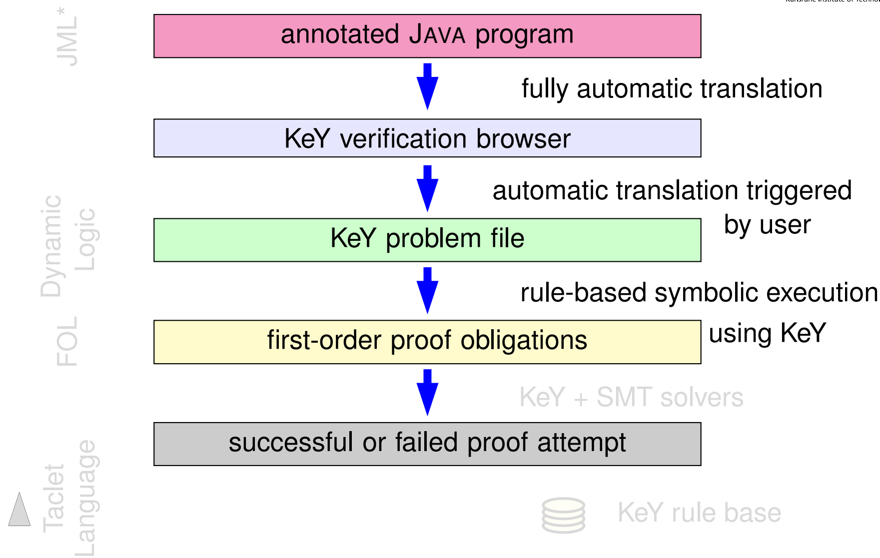
Workflow



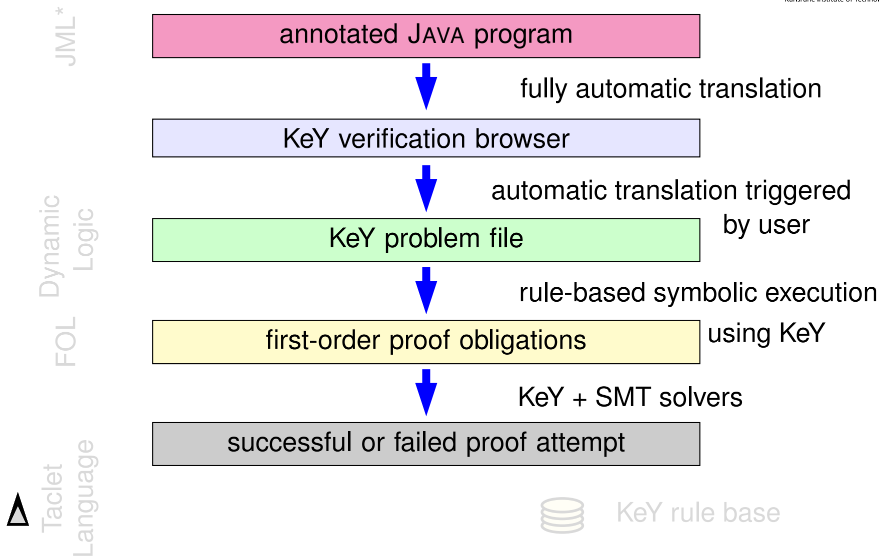
Workflow



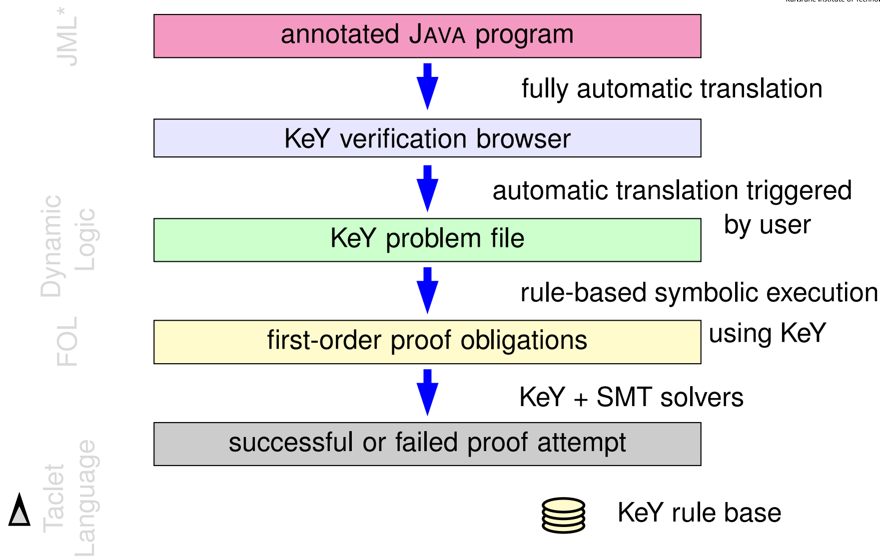
Workflow



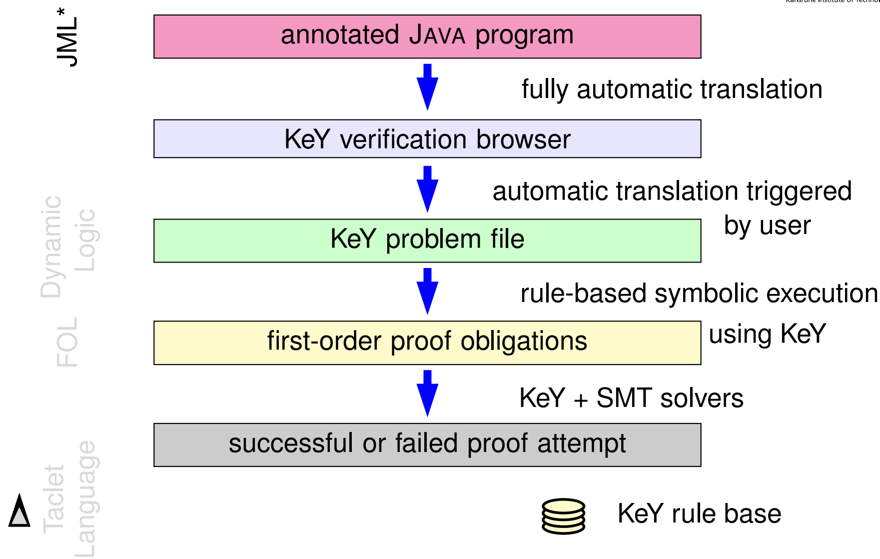
Workflow



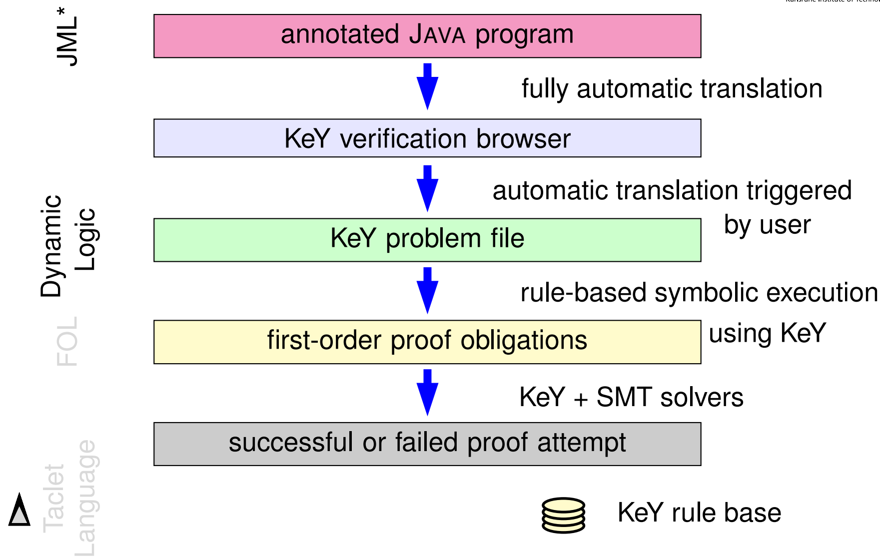
Workflow



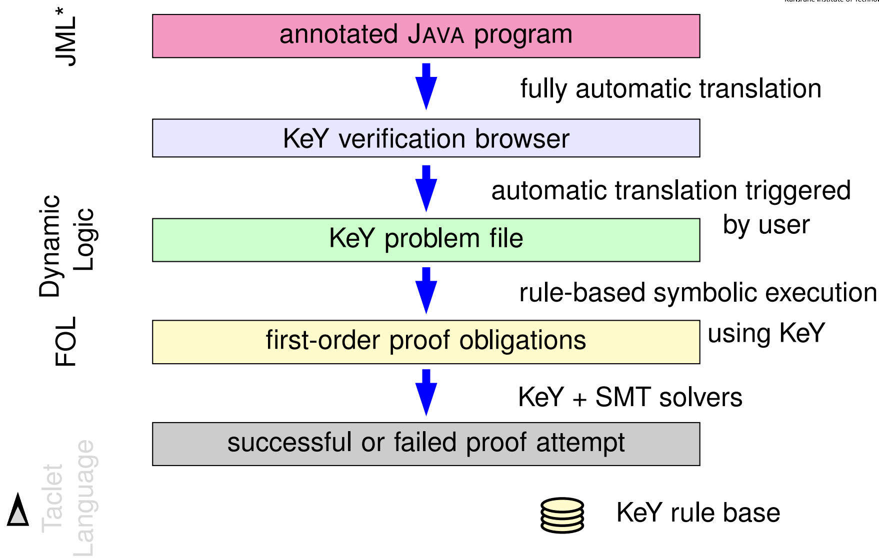
Workflow



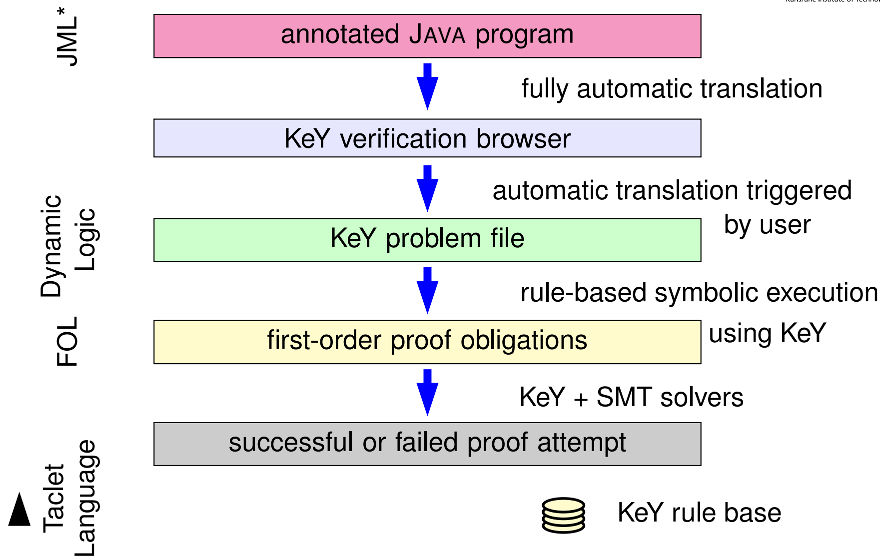
Workflow



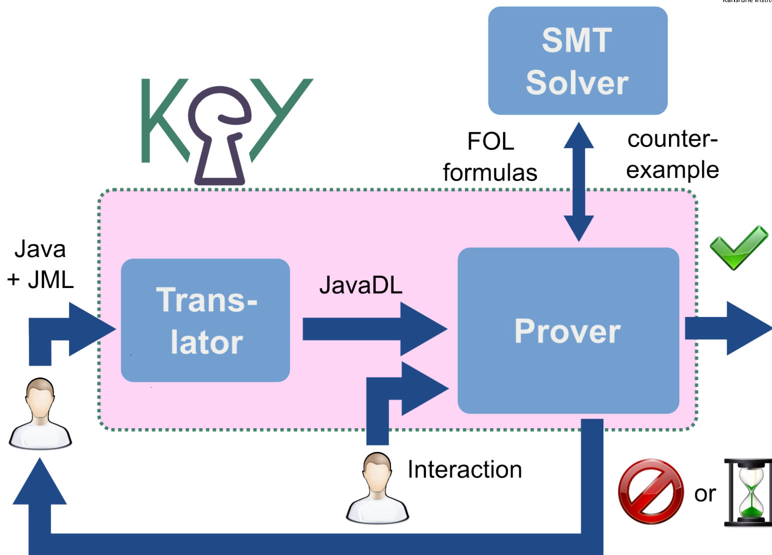
Workflow



Workflow



KeY Verification Process



Example: JML Specification of `max()`

```
/*@ requires a != null
  @ ensures (\forall int i; 0<=i && i<a.length ;
  @           \result >= a[i]);
  @ ensures a.length > 0 ==>
  @          (\exists int i; 0<=i && i<a.length ;
  @           \result == a[i]); @*/
public static int max(int[] a) {
  if ( a.length == 0 ) return 0;
  int max = a[0], i = 1;
/*@ loop_invariant
  @      i <= a.length  &&
  @      (\forall int j; j>=0 && j<i; max>=a[j])  &&
  @      (\exists int j; j>=0 && j<i; max==a[j]);
  @ modifies i, max;
  @ decreases a.length - i;  @*/
  while ( i < a.length ) {
    if ( a[i] > max ) max = a[i];
    ++i;
  }
```

DEMO

The KeY Tool

A Case Study: The TimSort Bug

Karlsruhe Institute of Technology

[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

KIT Karlsruhe Institute of Technology

[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

KIT Karlsruhe Institute of Technology

[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

Part II

The Java Modeling Language
– By Example –

1. Method Contracts
2. Quantifiers
3. Handling Loops
4. Frame Conditions
5. Using Contracts
6. Abstraction

Part II

The Java Modeling Language – By Example –

1. Method Contracts
2. Quantifiers
3. Handling Loops
4. Frame Conditions
5. Using Contracts
6. Abstraction

Method Contracts

Karlsruhe Institute of Technology

JML Spec of Postincrement

```
public class PostInc{
    public PostInc act;
    public int x,y;

/*@ public normal_behavior
  @  requires true;
  @  ensures  act.x == \old(act.y) &&
  @           act.y == \old(act.y) + 1;
  @*/
  public void postinc() { act.x = act.y++; }}
```

Method Contracts

Karlsruhe Institute of Technology

JML Spec of Postincrement

```
public class PostInc{
    public PostInc act;
    public int x,y;

/*@ public normal_behavior
  @  requires true;
  @  ensures  act.x == \old(act.y) &&
  @           act.y == \old(act.y) + 1;
  @*/
  public void postinc() { act.x = act.y++; }}
```

JML annotation occur as special comments in source programm

Method Contracts

Karlsruhe Institute of Technology

JML Spec of Postincrement

```
public class PostInc{
    public PostInc act;
    public int x,y;

/*@ public normal_behavior
  @  requires true;
  @  ensures   act.x == \old(act.y) &&
  @            act.y == \old(act.y) + 1;
  @*/
  public void postinc() { act.x = act.y++; }}
```

precondition

Method Contracts

Karlsruhe Institute of Technology

JML Spec of Postincrement

```
public class PostInc{
    public PostInc act;
    public int x,y;

/*@ public normal_behavior
  @  requires true;
  @  ensures  act.x == \old(act.y) &&
  @           act.y == \old(act.y) + 1;
  @*/
  public void postinc() { act.x = act.y++; }}
```

postcondition

Method Contracts

Karlsruhe Institute of Technology

JML Spec of Postincrement

```
public class PostInc{
    public PostInc act;
    public int x,y;

/*@ public normal_behavior
  @  requires true;
  @  ensures   act.x == \old(act.y) &&
  @            act.y == \old(act.y) + 1;
  @*/
  public void postinc() { act.x = act.y++; }}
```

JML operator `\old(e)` refers to value of `e` in prestate

Method Contracts

Karlsruhe Institute of Technology

JML Spec of Postincrement

```
public class PostInc{
    public PostInc act;
    public int x,y;

/*@ public normal_behavior
  @  requires true;
  @  ensures   act.x == \old(act.y) &&
  @            act.y == \old(act.y) + 1;
  @*/
  public void postinc() { act.x = act.y++; }}
```

All side-effect-free Java expressions allowed

Method Contracts

Karlsruhe Institute of Technology

JML Spec of Postincrement

```
public class PostInc{
    public PostInc act;
    public int x,y;

/*@ public normal_behavior
  @  requires true;
  @  ensures   act.x == \old(act.y) &&
  @            act.y == \old(act.y) + 1;
  @*/
  public void postinc() { act.x = act.y++; }}
```

Plus special operators (\ . . .)

Method Contracts

Non-null default

```
public class PostInc{
    public PostInc /*@ nullable @*/ act;
    public int x,y;

/*@ public normal_behavior
  @  requires act != null;
  @  ensures  act.x == \old(act.y) &&
  @           act.y == \old(act.y) + 1;
  @*/
  public void postinc() { act.x = act.y++; }}
```

Method Contracts

Non-null default

```
public class PostInc{
    public PostInc /*@ nullable @*/ act;
    public int x,y;

/*@ public normal_behavior
  @  requires act != null;
  @  ensures  act.x == \old(act.y) &&
  @           act.y == \old(act.y) + 1;
  @*/
  public void postinc() { act.x = act.y++; }}
```

By default JML assumes all fields and parameters to be non null

Method Contracts

Karlsruhe Institute of Technology

Non-null default

```
public class PostInc{
    public PostInc /*@ nullable @*/ act;
    public int x,y;

/*@ public normal_behavior
  @  requires act != null;
  @  ensures  act.x == \old(act.y) &&
  @           act.y == \old(act.y) + 1;
  @*/
  public void postinc() { act.x = act.y++; }}
```

The default is overwritten by the keyword `nullable`

Method Contracts

Non-null default

```
public class PostInc{
    public PostInc /*@ nullable @*/ act;
    public int x,y;

/*@ public normal_behavior
  @  requires act != null;
  @  ensures  act.x == \old(act.y) &&
  @           act.y == \old(act.y) + 1;
  @*/
  public void postinc() { act.x = act.y++; }}
```

In this case the precondition has to be adapted accordingly

Method Contracts

Karlsruhe Institute of Technology

Specification of commonEntry

```
class SITAPar{  public int[] a1,a2;
 /*@ public normal_behaviour
   @   requires 0<=l && l<r && r<=a1.length && r<=a2.length;
   @   assignable \nothing;
   @   ensures ( l <= \result && \result < r &&
   @              a1[\result] == a2[\result] )
                || \result == r ;
   @   ensures (\forall int j; l <= j && j < \result;
   @                                 a1[j] != a2[j]);
   @*/
  public int  commonEntry(int l, int r) { ... }
}
```

Method Contracts

Karlsruhe Institute of Technology

Specification of commonEntry

```
class SITAPar{  public int[] a1,a2;
 /*@ public normal_behaviour
   @   requires 0<=l && l<r && r<=a1.length && r<=a2.length;
   @   assignable \nothing;
   @   ensures ( l <= \result && \result < r &&
   @               a1[\result] == a2[\result] )
                || \result == r ;
   @   ensures (\forall int j; l <= j && j < \result;
   @                               a1[j] != a2[j]);
   @*/
  public int  commonEntry(int l, int r) { ... }
}
```

JML uses `\result` to refer to the return value of a method

Method Contracts

Specification of `commonEntry`

```
class SITAPar{  public int[] a1,a2;
 /*@ public normal_behaviour
   @   requires 0<=l && l<r && r<=a1.length && r<=a2.length;
   @   assignable \nothing;
   @   ensures ( l <= \result && \result < r &&
   @             a1[\result] == a2[\result] )
               || \result == r ;
   @   ensures (\forall int j; l <= j && j < \result;
   @                              a1[j] != a2[j]);
   @*/
  public int  commonEntry(int l, int r) { ... }
}
```

Method `commonEntry` looks for an index within the bounds

Method Contracts

Karlsruhe Institute of Technology

Specification of commonEntry

```
class SITAPar{  public int[] a1,a2;
 /*@ public normal_behaviour
   @   requires 0<=l && l<r && r<=a1.length && r<=a2.length;
   @   assignable \nothing;
   @   ensures ( l <= \result && \result < r &&
   @             a1[\result] == a2[\result] )
               || \result == r ;
   @   ensures (\forall int j; l <= j && j < \result;
   @                                a1[j] != a2[j]);
   @*/
  public int  commonEntry(int l, int r) { ... }
}
```

such that the two arrays have the same entry

Method Contracts

Karlsruhe Institute of Technology

Specification of `commonEntry`

```
class SITAPar{  public int[] a1,a2;
 /*@ public normal_behaviour
   @   requires 0<=l && l<r && r<=a1.length && r<=a2.length;
   @   assignable \nothing;
   @   ensures ( l <= \result && \result < r &&
   @                a1[\result] == a2[\result] )
               || \result == r ;
   @   ensures (\forall int j; l <= j && j < \result;
   @                               a1[j] != a2[j]);
   @*/
  public int  commonEntry(int l, int r) { ... }
}
```

If no such index exists the return value is the upper bound

Method Contracts

Karlsruhe Institute of Technology

Specification of `commonEntry`

```
class SITAPar{  public int[] a1,a2;
 /*@ public normal_behaviour
   @   requires 0<=l && l<r && r<=a1.length && r<=a2.length;
   @   assignable \nothing;
   @   ensures ( l <= \result && \result < r &&
   @               a1[\result] == a2[\result] )
               || \result == r ;
   @   ensures (\forall int j; l <= j && j < \result;
   @                               a1[j] != a2[j]);
   @*/
  public int  commonEntry(int l, int r) { ... }
}
```

Furthermore, `\result` should be the first index of this kind

Part II

The Java Modeling Language – By Example –

1. Method Contracts
2. Quantifiers
3. Handling Loops
4. Frame Conditions
5. Using Contracts
6. Abstraction

Part II

The Java Modeling Language – By Example –

1. Method Contracts
2. Quantifiers
3. Handling Loops
4. Frame Conditions
5. Using Contracts
6. Abstraction

Quantifiers in JML

Karlsruhe Institute of Technology

Specification of `commonEntry`

```
@ ...
@   ensures (\forall int j; l <= j && j < \result;
@                             a1[j] != a2[j] );
@ ...
```

Quantified formulas in JML consist of

- the quantifier
- the range restriction
- the body

Quantifiers in JML

Karlsruhe Institute of Technology

Specification of commonEntry

```
@ ...
@   ensures (\forall int j; l <= j && j < \result;
@                                a1[j] != a2[j] );
@ ...
```

Quantified formulas in JML consist of

- the quantifier
- the range restriction
- the body

Quantifiers in JML

Karlsruhe Institute of Technology

Specification of commonEntry

```
@ ...
@   ensures (\forall int j; l <= j && j < \result;
@                           a1[j] != a2[j] );
@ ...
```

Quantified formulas in JML consist of

- the quantifier
- the range restriction
- the body

Quantifiers in JML

Karlsruhe Institute of Technology

Specification of `commonEntry`

```
@ ...
@   ensures (\forall int j; l <= j && j < \result;
@                           a1[j] != a2[j] );
@ ...
```

Quantified formulas in JML consist of

- the quantifier
- the range restriction
- the body

Quantifiers in JML

Karlsruhe Institute of Technology

Semantics

JML	`\forall T x; R; B;`
predicate logic	$\forall\, T{:}x\ (R \rightarrow B)$
JML	`\exists T x; R; B;`
predicate logic	$\exists\, T{:}x\ (R \wedge B)$

Part II

The Java Modeling Language – By Example –

1 Method Contracts

2 Quantifiers

3 Handling Loops

4 Frame Conditions

5 Using Contracts

6 Abstraction

Part II

The Java Modeling Language – By Example –

1. Method Contracts
2. Quantifiers
3. Handling Loops
4. Frame Conditions
5. Using Contracts
6. Abstraction

Loop Invariants

Loop Invariant for `commonEntry`

```
class SITAPar{  public int[] a1,a2;   ...
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @    (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

Loop Invariants

Karlsruhe Institute of Technology

Loop Invariant for commonEntry

```
class SITAPar{  public int[] a1,a2;   ...
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @    (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

The loop invariant

Loop Invariants

Loop Invariant for `commonEntry`

```
class SITAPar{  public int[] a1,a2;   ...
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @    (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

The loop invariant is valid before entering the loop since

Loop Invariants

Karlsruhe Institute of Technology

Loop Invariant for `commonEntry`

```
class SITAPar{  public int[] a1,a2;   ...
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @    (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

`l <= k && k <= r` follows from `k==l` and precondition `l<r`

Loop Invariants

KIT — Karlsruhe Institute of Technology

Loop Invariant for `commonEntry`

```
class SITAPar{  public int[] a1,a2;    ...
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant    l <= k && k <= r &&
  @     (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

`l <= k && k <= r` follows from `k==l` and precondition `l<r` and quantification is empty

Loop Invariants

KIT
Karlsruhe Institute of Technology

Loop Invariant for `commonEntry`

```
class SITAPar{  public int[] a1,a2;    ...
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @     (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

If the loop body is started in a state satisfying the invariant,
it terminates in a state satisfying the invariant

Loop Invariants

Loop Invariant for commonEntry

```
class SITAPar{  public int[] a1,a2;   ...
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @    (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

Distinguish break and non-break case!

Using a Loop Invariant

On termination of the loop

the invariant

```
l <= k && k <= r &&
(\forall int i; l <= i && i < k; a1[i] != a2[i])
```

plus

```
\result = k
```

plus reason for termination of the loop

`k == r` or `k < r && a1[k] == a2[k]`

imply the postconditions

```
(l <= \result && \result < r && a1[\result] == a2[\result])
  || \result == r
```

and

```
\forall int j; l <= j && j < \result; a1[j] != a2[j]
```

Using a Loop Invariant

On termination of the loop
the invariant

```
  l <= k && k <= r &&
  (\forall int i; l <= i && i < k; a1[i] != a2[i])
```

plus

```
\result = k
```

plus reason for termination of the loop

`k == r` or `k < r && a1[k] == a2[k]`

imply the postconditions

```
(l <= \result && \result < r && a1[\result] == a2[\result])
  || \result == r
```

and

```
\forall int j; l <= j && j < \result; a1[j] != a2[j]
```

Loop Termination

```
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @ (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

- is ≥ 0 on entering the loop
- strictly decreases in every loop iteration
- but always stays ≥ 0

Loop Termination

```
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @ (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

The loop variant

- is ≥ 0 on entering the loop
- strictly decreases in every loop iteration
- but always stays ≥ 0

Loop Termination

```
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @ (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

The loop variant

- is ≥ 0 on entering the loop
- strictly decreases in every loop iteration
- but always stays ≥ 0

Loop Termination


```
 public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @ (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

The loop variant

- is ≥ 0 on entering the loop
- strictly decreases in every loop iteration
- but always stays ≥ 0

Loop Termination

```
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @ (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

The loop variant

- is ≥ 0 on entering the loop
- strictly decreases in every loop iteration
- but always stays ≥ 0

Part II

The Java Modeling Language
– By Example –

1. Method Contracts
2. Quantifiers
3. Handling Loops
4. Frame Conditions
5. Using Contracts
6. Abstraction

Part II

The Java Modeling Language
– By Example –

1 Method Contracts

2 Quantifiers

3 Handling Loops

4 Frame Conditions

5 Using Contracts

6 Abstraction

Assignable Clauses

KIT — Karlsruhe Institute of Technology

```
/*@ public normal_behaviour
  @  requires  0 <= pos1 && 0 <= pos2 &&
  @            pos1 < a.length && pos2 < a.length;
  @  ensures  a[pos1]  == \old(a[pos2]) &&
  @           a[pos2]  == \old(a[pos1]);
  @  assignable a[pos1], a[pos2];
  @*/
 public void  swap(int[] a, int pos1, int pos2) {
  int temp;
  temp = a[pos1]; a[pos1] = a[pos2]; a[pos2] = temp;}
```

Assignable Clauses

Karlsruhe Institute of Technology

```
/*@ public normal_behaviour
  @  requires  0 <= pos1 && 0 <= pos2 &&
  @            pos1 < a.length && pos2 < a.length;
  @  ensures  a[pos1]  == \old(a[pos2]) &&
  @           a[pos2]  == \old(a[pos1]);
  @  assignable a[pos1], a[pos2];
  @*/
 public void  swap(int[] a, int pos1, int pos2) {
  int temp;
  temp = a[pos1]; a[pos1] = a[pos2]; a[pos2] = temp;}
```

At most the locations in the assignable clause may be changed

Assignable Clauses

```
/*@ public normal_behaviour
  @  requires  0 <= pos1 && 0 <= pos2 &&
  @            pos1 < a.length && pos2 < a.length;
  @  ensures  a[pos1]  == \old(a[pos2]) &&
  @           a[pos2]  == \old(a[pos1]);
  @  assignable a[pos1], a[pos2];
  @*/
 public void  swap(int[] a, int pos1, int pos2) {
  int temp;
  temp = a[pos1]; a[pos1] = a[pos2]; a[pos2] = temp;}
```

Everything else must remain unchanged

Assignable Clauses

Karlsruhe Institute of Technology

```
/*@ public normal_behaviour
  @  requires  0 <= pos1 && 0 <= pos2 &&
  @            pos1 < a.length && pos2 < a.length;
  @  ensures  a[pos1]  == \old(a[pos2]) &&
  @           a[pos2]  == \old(a[pos1]);
  @  assignable a[pos1], a[pos2];
  @*/
 public void  swap(int[] a, int pos1, int pos2) {
  int temp;
  temp = a[pos1]; a[pos1] = a[pos2]; a[pos2] = temp;}
```

Local variables need not be included

Assignable Clauses

KIT Karlsruhe Institute of Technology

```
/*@ public normal_behaviour
  @  requires  0 <= pos1 && 0 <= pos2 &&
  @            pos1 < a.length && pos2 < a.length;
  @  ensures  a[pos1]  == \old(a[pos2]) &&
  @           a[pos2]  == \old(a[pos1]);
  @  assignable a[pos1], a[pos2];
  @*/
 public void  swap(int[] a, int pos1, int pos2) {
  int temp;
  temp = a[pos1]; a[pos1] = a[pos2]; a[pos2] = temp;}
```

Assignable clauses are evaluated in the prestate

Part II

The Java Modeling Language – By Example –

1 Method Contracts

2 Quantifiers

3 Handling Loops

4 Frame Conditions

5 Using Contracts

6 Abstraction

Part II

The Java Modeling Language – By Example –

1. Method Contracts
2. Quantifiers
3. Handling Loops
4. Frame Conditions
5. Using Contracts
6. Abstraction

Using Contracts


```
class SITA3{ public int[] a1, a2;
  /*@ public normal_behaviour
   @  requires  a1.length == a2.length;
   @  ensures  (\forall int i; 0<= i && i < a1.length;
   @   a1[i] == a2[i] ==>
   @   (\forall int j; 0<= j && j < i; a1[j] == a2[j]));
   @  assignable a1[*],a2[*];
   @*/
  public void  rearrange(){ int m = 0 ; int k = 0;
   while (m < a1.length) { m = commonEntry(m,a1.length);
    if (m < a1.length) {swap(a1,m,k);
     if (a1 != a2) { swap(a2,m,k);} k = k+1 ; m = m+1;}}}}
```

Using Contracts

Karlsruhe Institute of Technology

```
class SITA3{ public int[] a1, a2;
  /*@ public normal_behaviour
   @  requires  a1.length == a2.length;
   @  ensures  (\forall int i; 0<= i && i < a1.length;
   @   a1[i] == a2[i] ==>
   @   (\forall int j; 0<= j && j < i; a1[j] == a2[j]));
   @  assignable a1[*],a2[*];
   @*/
  public void  rearrange(){ int m = 0 ; int k = 0;
   while (m < a1.length) { m = commonEntry(m,a1.length);
    if (m < a1.length) {swap(a1,m,k);
     if (a1 != a2) { swap(a2,m,k);} k = k+1 ; m = m+1;}}}}
```

Method `rearrange`

Using Contracts

```
class SITA3{ public int[] a1, a2;
  /*@ public normal_behaviour
   @  requires  a1.length == a2.length;
   @  ensures  (\forall int i; 0<= i && i < a1.length;
   @   a1[i] == a2[i] ==>
   @   (\forall int j; 0<= j && j < i; a1[j] == a2[j]));
   @  assignable a1[*],a2[*];
   @*/
  public void  rearrange(){ int m = 0 ; int k = 0;
   while (m < a1.length) { m = commonEntry(m,a1.length);
    if (m < a1.length) {swap(a1,m,k);
     if (a1 != a2) { swap(a2,m,k);} k = k+1 ; m = m+1;}}}}
```

Method `rearrange` uses methods `commonEntry`

Using Contracts

```
class SITA3{ public int[] a1, a2;
  /*@ public normal_behaviour
   @  requires  a1.length == a2.length;
   @  ensures  (\forall int i; 0<= i && i < a1.length;
   @   a1[i] == a2[i] ==>
   @   (\forall int j; 0<= j && j < i; a1[j] == a2[j]));
   @  assignable a1[*],a2[*];
   @*/
  public void  rearrange(){ int m = 0 ; int k = 0;
   while (m < a1.length) { m = commonEntry(m,a1.length);
    if (m < a1.length) {swap(a1,m,k);
     if (a1 != a2) { swap(a2,m,k);} k = k+1 ; m = m+1;}}}}
```

Method `rearrange` uses methods `commonEntry` and `swap`

Using Contracts

```
class SITA3{ public int[] a1, a2;
  /*@ public normal_behaviour
   @  requires  a1.length == a2.length;
   @  ensures  (\forall int i; 0<= i && i < a1.length;
   @   a1[i] == a2[i] ==>
   @   (\forall int j; 0<= j && j < i; a1[j] == a2[j]));
   @  assignable a1[*],a2[*];
   @*/
  public void  rearrange(){ int m = 0 ; int k = 0;
   while (m < a1.length) { m = commonEntry(m,a1.length);
    if (m < a1.length) {swap(a1,m,k);
     if (a1 != a2) { swap(a2,m,k);} k = k+1 ; m = m+1;}}}}
```

Verification of `rearrange` uses their contracts, not their implementation

Using Contracts

```
class SITA3{ public int[] a1, a2;
  /*@ public normal_behaviour
   @  requires  a1.length == a2.length;
   @  ensures  (\forall int i; 0<= i && i < a1.length;
   @   a1[i] == a2[i] ==>
   @    (\forall int j; 0<= j && j < i; a1[j] == a2[j]));
   @  assignable a1[*],a2[*];
   @*/
  public void  rearrange(){ int m = 0 ; int k = 0;
   while (m < a1.length) { m = commonEntry(m,a1.length);
    if (m < a1.length) {swap(a1,m,k);
     if (a1 != a2) { swap(a2,m,k);} k = k+1 ; m = m+1;}}}}
```

Key to scalability

Part II

The Java Modeling Language – By Example –

1. Method Contracts
2. Quantifiers
3. Handling Loops
4. Frame Conditions
5. **Using Contracts**
6. Abstraction

Part II

The Java Modeling Language
– By Example –

1. Method Contracts
2. Quantifiers
3. Handling Loops
4. Frame Conditions
5. Using Contracts
6. Abstraction

Model Fields

```
/*@ model \seq seq1; model \seq seq2; @*/
/*@ represents seq1 = \dl_array2seq(a1);
  @ represents seq2 = \dl_array2seq(a2);
  @*/

/*@ public normal_behaviour
  @  ensures  \dl_seqPerm(seq1,\old(seq1)) &&
  @           \dl_seqPerm(seq2,\old(seq2)) ;
  @*/
 public void  rearrange(){ ... }
```

Model Fields

```
/*@ model \seq seq1; model \seq seq2; @*/
/*@ represents seq1 = \dl_array2seq(a1);
  @ represents seq2 = \dl_array2seq(a2);
  @*/

/*@ public normal_behaviour
  @  ensures  \dl_seqPerm(seq1,\old(seq1)) &&
  @           \dl_seqPerm(seq2,\old(seq2)) ;
  @*/
 public void  rearrange(){ ... }
```

model fields are only for specification

Model Fields

```
/*@ model \seq seq1; model \seq seq2; @*/
/*@ represents seq1 = \dl_array2seq(a1);
  @ represents seq2 = \dl_array2seq(a2);
  @*/

/*@ public normal_behaviour
  @  ensures  \dl_seqPerm(seq1,\old(seq1)) &&
  @           \dl_seqPerm(seq2,\old(seq2)) ;
  @*/
 public void  rearrange(){ ... }
```

\seq is an abstract data type

Model Fields

```
/*@ model \seq seq1; model \seq seq2; @*/
/*@ represents seq1 = \dl_array2seq(a1);
  @ represents seq2 = \dl_array2seq(a2);
  @*/

/*@ public normal_behaviour
  @  ensures  \dl_seqPerm(seq1,\old(seq1)) &&
  @           \dl_seqPerm(seq2,\old(seq2)) ;
  @*/
 public void  rearrange(){ ... }
```

`represents` clauses fix the semantics of model fields

Model Fields

Karlsruhe Institute of Technology

```
/*@ model \seq seq1; model \seq seq2; @*/
/*@ represents seq1 = \dl_array2seq(a1);
  @ represents seq2 = \dl_array2seq(a2);
  @*/

/*@ public normal_behaviour
  @  ensures  \dl_seqPerm(seq1,\old(seq1)) &&
  @           \dl_seqPerm(seq2,\old(seq2)) ;
  @*/
 public void  rearrange(){ ... }
```

`array2seq(a)` yields the abstract sequence associated with array a

Model Fields

```
/*@ model \seq seq1; model \seq seq2; @*/
/*@ represents seq1 = \dl_array2seq(a1);
  @ represents seq2 = \dl_array2seq(a2);
  @*/

/*@ public normal_behaviour
  @  ensures  \dl_seqPerm(seq1,\old(seq1)) &&
  @           \dl_seqPerm(seq2,\old(seq2)) ;
  @*/
 public void  rearrange(){ ... }
```

Only additional postcondition shown here

Model Fields

```
/*@ model \seq seq1; model \seq seq2; @*/
/*@ represents seq1 = \dl_array2seq(a1);
  @ represents seq2 = \dl_array2seq(a2);
  @*/

/*@ public normal_behaviour
  @  ensures  \dl_seqPerm(seq1,\old(seq1)) &&
  @           \dl_seqPerm(seq2,\old(seq2)) ;
  @*/
 public void  rearrange(){ ... }
```

`seqPerm(s1,s2)` is a predicate in the data type `\seq`,
true if `s1` is a permutation of `s2`

Model Fields

```
/*@ model \seq seq1; model \seq seq2; @*/
/*@ represents seq1 = \dl_array2seq(a1);
  @ represents seq2 = \dl_array2seq(a2);
  @*/

/*@ public normal_behaviour
  @  ensures  \dl_seqPerm(seq1,\old(seq1)) &&
  @           \dl_seqPerm(seq2,\old(seq2)) ;
  @*/
 public void  rearrange(){ ... }
```

The `\dl` prefix is a technical detail necessary since `\seq` is not (yet) part of official JML

Model Fields

Karlsruhe Institute of Technology

```
/*@ model \seq seq1; model \seq seq2; @*/
/*@ represents seq1 = \dl_array2seq(a1);
  @ represents seq2 = \dl_array2seq(a2);
  @*/

/*@ public normal_behaviour
  @  ensures  \dl_seqPerm(seq1,\old(seq1)) &&
  @           \dl_seqPerm(seq2,\old(seq2)) ;
  @*/
 public void  rearrange(){ ... }
```

Model fields allow abstraction and information hiding.
They can be defined and used in interfaces.