Program Verification with the KeY System
and Deductive Verification of Information-Flow Properties

B. Beckert  V. Klebanov  C. Scheben  P. H. Schmitt | RS3, 10.–13.10.11

Institute for Theoretical Computer Science
Part I

The KeY System – An Overview
Part I

The KeY System – An Overview
KeY Project

www.key-project.org

Project Consortium

- Bernhard Beckert and Peter H. Schmitt, Karlsruhe Institute of Technology
- Reiner Hähnle, TU Darmstadt

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- 100% Java Card
- High degree of automation / usability
KeY Project

www.key-project.org

KeY Tool
- Deductive rules for all Java features
- Symbolic execution
- 100% Java Card
- High degree of automation / usability
Deductive Verification of

- Java programs
- specified and annotated with the Java Modeling Language
- at program level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- 100% Java Card
- High degree of automation / usability

Overview

B. Beckert, V. Klebanov, C. Scheben, P. H. Schmitt – KeY Tutorial
KeY Project

Deductive Verification of
- Java programs
- specified and annotated with the Java Modeling Language
- at program level

KeY Tool
- Deductive rules for all Java features
- Symbolic execution
- 100% Java Card
- High degree of automation / usability
KeY Project

www.key-project.org

Deductive Verification of

- Java programs
- specified and annotated with the Java Modeling Language
- at program level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- 100% Java Card
- High degree of automation / usability
KeY Project

www.key-project.org

Deductive Verification of

- Java programs
- specified and annotated with the Java Modeling Language
- at program level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- 100% Java Card
- High degree of automation / usability
Deductive Verification of
- Java programs
- specified and annotated with the Java Modeling Language
- at program level

KeY Tool
- Deductive rules for all Java features
- Symbolic execution
- 100% Java Card
- High degree of automation / usability
KeY Project

Deductive Verification of
- Java programs
- specified and annotated with the Java Modeling Language
- at program level

KeY Tool
- Deductive rules for all Java features
- Symbolic execution
- 100% Java Card
- High degree of automation / usability
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

- 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

- 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

- 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.
Workflow

1. annotated JAVA program
2. KeY verification browser
3. KeY problem file
4. first-order proof obligations
5. successful or failed proof attempt

- fully automatic translation
- automatic translation triggered by user
- rule-based symbolic execution using KeY
- KeY + SMT solvers

- KeY rule base
- JML*
- Dynamic Logic
- FOL
- Taclet Language
Workflow

1. annotated JAVA program
   - fully automatic translation

2. KeY verification browser
   - automatic translation triggered by user
   - rule-based symbolic execution using KeY
   - KeY + SMT solvers

3. KeY problem file
   - first-order proof obligations

4. successful or failed proof attempt
   - KeY rule base
Workflow

1. annotated JAVA program
2. fully automatic translation
3. KeY verification browser
4. automatic translation triggered by user
5. KeY problem file
6. rule-based symbolic execution using KeY
7. first-order proof obligations
8. KeY + SMT solvers
9. successful or failed proof attempt
10. KeY rule base
Overview

B. Beckert, V. Klebanov, C. Scheben, P. H. Schmitt – KeY Tutorial

RS3, 10.–13.10.11

6/118

Workflow

annotated JAVA program

fully automatic translation

KeY verification browser

automatic translation triggered by user

KeY problem file

rule-based symbolic execution using KeY

first-order proof obligations

KeY + SMT solvers

successful or failed proof attempt

KeY rule base
Workflow

annotated JAVA program

fully automatic translation

KeY verification browser

automatic translation triggered by user

KeY problem file

rule-based symbolic execution using KeY

first-order proof obligations

KeY + SMT solvers

successful or failed proof attempt

KeY rule base
Workflow

annotated JAVA program

fully automatic translation

KeY verification browser

automatic translation triggered by user

KeY problem file

rule-based symbolic execution using KeY

first-order proof obligations

KeY + SMT solvers

successful or failed proof attempt

KeY rule base
Workflow

1. annotated JAVA program
2. fully automatic translation
3. KeY verification browser
4. automatic translation triggered by user
5. KeY problem file
6. rule-based symbolic execution using KeY
7. first-order proof obligations
8. KeY + SMT solvers
9. successful or failed proof attempt
10. KeY rule base
Workflow

JML*

annotated JAVA program

fully automatic translation

KeY verification browser

automatically translated

by user

KeY problem file

rule-based symbolic execution

using KeY

first-order proof obligations

KeY + SMT solvers

successful or failed proof attempt

KeY rule base

Dynamic Logic

FOL

Taclet Language

Overview

B. Beckert, V. Klebanov, C. Scheben, P. H. Schmitt – KeY Tutorial

RS3, 10.–13.10.11
Workflow

1. **JML**
   - annotated JAVA program
   - fully automatic translation
   - KeY verification browser

2. **Dynamic Logic**
   - KeY problem file
   - automatic translation triggered by user
   - rule-based symbolic execution
   - using KeY

3. **FOL**
   - first-order proof obligations
   - KeY + SMT solvers
   - successful or failed proof attempt

4. **Taclet Language**
   - KeY rule base
Workflow

JML*

annotated JAVA program

fully automatic translation

KeY verification browser

KeY problem file

automatic translation triggered by user

Dynamic Logic

first-order proof obligations

rule-based symbolic execution using KeY

FOL

KeY + SMT solvers

successful or failed proof attempt

Taclet Language

KeY rule base

Overview

B. Beckert, V. Klebanov, C. Scheben, P. H. Schmitt – KeY Tutorial
COST Action IC0701 presents **VerifyThus**—
a Linux distribution with 10 program verification tools.

Available as:
- bootable USB stick
- bootable DVD
- virtual machine image

**Included verification tools:**
Boogie, Dafny, ESC/Java2, Jahob, JavaFAN, jStar, KeY, KIV, Krakatoa, Verifast

http://verifythus.cost-ic0701.org
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
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++; }
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++; }
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++; }
}
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++; }
}
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
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
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 (\ldots)
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++; }
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
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

```java
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
```java
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

Specification of `commonEntry` class

```java
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**

```java
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

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

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

 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

Specification of `commonEntry`

```jml
@ ... 
@ 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

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

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

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

### Semantics

<table>
<thead>
<tr>
<th>JML</th>
<th>$\forall T \ x; \ R; \ B;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>predicate logic</td>
<td>$\forall T: x \ (R \rightarrow B)$</td>
</tr>
<tr>
<td>JML</td>
<td>$\exists T \ x; \ R; \ B;$</td>
</tr>
<tr>
<td>predicate logic</td>
<td>$\exists T: x \ (R \land B)$</td>
</tr>
</tbody>
</table>
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 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

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
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

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

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

\[ l \leq k \land k \leq r \land \left( \forall \text{int } i; \ l \leq i \land i < k; \ a1[i] \neq a2[i] \right) \]

plus

\[ \result = k \]

plus reason for termination of the loop

\[ k = r \quad \text{or} \quad k < r \land a1[k] = a2[k] \]

imply the postconditions

\[ (l \leq \result \land \result < r \land a1[\result] = a2[\result]) \land \result = r \]

and

\[ \forall \text{int } j; \ l \leq j \land j < \result; \ a1[j] \neq a2[j] \]
Using a Loop Invariant

On termination of the loop
the invariant

\[
l \leq k \land k \leq r \land
(\forall \text{int } i; l \leq i \land i < k; a1[i] \neq a2[i])
\]

plus

\[
\result = k
\]

plus reason for termination of the loop

\[
k = r \quad \text{or} \quad k < r \land a1[k] = a2[k]
\]

imply the postconditions

\[
(l \leq \result \land \result < r \land a1[\result] = a2[\result])
\]

\[
\lor \result = r
\]

and

\[
\forall \text{int } j; l \leq j \land j < \result; a1[j] \neq a2[j]
\]
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

```java
class CommonEntry {
    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 $\geq 0$ on entering the loop
- strictly decreases in every loop iteration
- but always stays $\geq 0$
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 $\geq 0$ on entering the loop
- strictly decreases in every loop iteration
- but always stays $\geq 0$
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 $\geq 0$ on entering the loop
- strictly decreases in every loop iteration
- but always stays $\geq 0$
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; }
Part II

The Java Modeling Language
– By Example –

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

```java
/*@ 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.
/@ 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
/*@ public normalBehaviour
   @ 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

```java
/*@ public normalBehaviour
  @ 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
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;}}}}
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

class SITA3{
  public int[] a1, a2;

  /**
   * public normal_behavior
   * requires a1.length == a2.length;
   * ensures (
   *   \forall \text{int } i; 0<= i && i < a1.length;
   *   a1[i] == a2[i] ==> 
   *   \forall \text{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
class SITA3{
  public int[] a1, a2;
  /*@
  public normal_behaviour
  @ requires a1.length == a2.length;
  @ ensures (∀int i; 0<= i && i < a1.length;
  @ a1[i] == a2[i] =>
  @ (∀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
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

<table>
<thead>
<tr>
<th>Method Contracts</th>
<th>Quantifiers</th>
<th>Handling Loops</th>
<th>Frame Conditions</th>
<th>Using Contracts</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>☀☀☀</td>
<td>☀</td>
<td>☀</td>
<td>☀</td>
<td>☀</td>
<td>☀</td>
</tr>
</tbody>
</table>

B. Beckert, V. Klebanov, C. Scheben, P. H. Schmitt – KeY Tutorial
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 \seq seq1; model \seq seq2; @*/
/*@ represents seq1 = \dl_array2seq(a1);
   @ represents seq2 = \dl_array2seq(a2);
   @*/

/*@ public normalBehaviour
   @ ensures \dl_seqPerm(seq1,\old(seq1)) &&
         @ \dl_seqPerm(seq2,\old(seq2)) ;
   @*/

public void rearrange(){ ... }
Model Fields

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

/*@ public normalBehaviour
   @ ensures \dl_seqPerm(seq1,\old(seq1)) &&
   @ \dl_seqPerm(seq2,\old(seq2));
   */

public void rearrange(){ ... }
```

model fields are only for specification
Model Fields

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

/*@ public normalBehaviour
   @ ensures \dl_seqPerm(seq1,\old(seq1)) &&
   @ \dl_seqPerm(seq2,\old(seq2)) ;
   */

public void rearrange(){ ... }
```

`\seq` is an abstract data type
```java
/*@ model \seq seq1; model \seq seq2; @*/
/*@ represents seq1 = \dl_array2seq(a1);
    @ represents seq2 = \dl_array2seq(a2);
    @*/

/*@ public normalBehaviour
    @ ensures \dl_seqPerm(seq1,\old(seq1)) &&
    @ \dl_seqPerm(seq2,\old(seq2)) ;
    @*/

public void rearrange(){ ... }
```

represents clauses fix the semantics of model fields
Model Fields

```java
/*@ 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)) ;
 @*/

class
{
 public void rearrange()
{
 ... 
}

array2seq(a) yields the abstract sequence associated with array a
```
/*@ 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 show here
/*@
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

```java
/*@ 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.
Part III

Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
Part III

Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
Syntax and Semantics

Syntax

- Basis: Typed first-order predicate logic
- Modal operators $\langle p \rangle$ and $[p]$ for each (JAVA CARD) program $p$
- Class definitions in background (not shown in formulas)

Semantics (Kripke)

Modal operators allow referring to the final state of $p$:

- $[p] F$: If $p$ terminates, then $F$ holds in the final state  
  (partial correctness)
- $<p> F$: $p$ terminates and $F$ holds in the final state  
  (total correctness)
Syntax and Semantics

Syntax

- Basis: Typed first-order predicate logic
- Modal operators $\langle p \rangle$ and $[p]$ for each (JAVA CARD) program $p$
- Class definitions in background (not shown in formulas)

Semantics (Kripke)

Modal operators allow referring to the final state of $p$:

- $[p] F$: If $p$ terminates, then $F$ holds in the final state
  (partial correctness)
- $\langle p \rangle F$: $p$ terminates and $F$ holds in the final state
  (total correctness)
Syntax and Semantics

Syntax

- Basis: Typed first-order predicate logic
- Modal operators \( \langle p \rangle \) and \([p]\) for each (JAVA CARD) program \(p\)
- Class definitions in background (not shown in formulas)

Semantics (Kripke)

Modal operators allow referring to the final state of \(p\):

- \([p] F\): If \(p\) terminates, then \(F\) holds in the final state
  (partial correctness)
- \(\langle p \rangle F\): \(p\) terminates and \(F\) holds in the final state
  (total correctness)
Syntax

- Basis: Typed first-order predicate logic
- Modal operators \( \langle p \rangle \) and \([p]\) for each (JAVA CARD) program \( p \)
- Class definitions in background (not shown in formulas)

Semantics (Kripke)

Modal operators allow referring to the final state of \( p \):
- \([p] F\): If \( p \) terminates, then \( F \) holds in the final state (partial correctness)
- \( \langle p \rangle F \): \( p \) terminates and \( F \) holds in the final state (total correctness)
Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm

- Programs are “first-class citizens”
- Real Java syntax
Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm

Hoare triple \( \{ \psi \} \alpha \{ \phi \} \) equiv. to DL formula \( \psi \rightarrow [\alpha] \phi \)
Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm

Not merely partial/total correctness:
- can employ programs for specification (e.g., verifying program transformations)
- can express security properties (two runs are indistinguishable)
- extension-friendly (e.g., temporal modalities)
Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm
Dynamic Logic Example Formulas

(balance >= c & amount > 0) --> <charge(amount);> balance > c

<x = 1;>({while (true) {}}, false)

Program formulas can appear nested

\forall int val; ((<p> x = val) <-> (<q> x = val))

p, q equivalent relative to computation state restricted to x
Dynamic Logic Example Formulas

\[(\text{balance } \geq c \land \text{amount} > 0) \rightarrow \langle \text{charge(amount);} \rangle \text{balance} > c\]

\[\langle x = 1; > ([\text{while (true) } {} ] \text{false})\]

- Program formulas can appear nested

\[\forall \text{int val}; ((\langle p \rangle x \doteq val) \leftrightarrow (\langle q \rangle x \doteq val))\]

- \(p, q\) equivalent relative to computation state restricted to \(x\)
Dynamic Logic Example Formulas

(balance ≥ c & amount > 0) → 
<charge(amount);> balance > c

<x = 1;>([while (true) {}] false)

Program formulas can appear nested

\forall int val; (\langle p \rangle x := val \leftrightarrow \langle q \rangle x := val)

p, q equivalent relative to computation state restricted to x
Dynamic Logic Example Formulas

(balance ≥ c & amount > 0) →
<charge(amount);> balance > c

<x = 1;> ([while (true) {}] false)
- Program formulas can appear nested

\forall int val; ((<p>x = val) ↔ (<q>x = val))
- p, q equivalent relative to computation state restricted to x
Dynamic Logic Example Formulas

(balance \geq c \land \text{amount} > 0) \rightarrow
\langle\text{charge(amount)};\rangle \text{balance} > c

\langle x = 1;\rangle([\text{while (true) } \{\}]) \text{false}

- Program formulas can appear nested

\forall \text{int } val; (\langle p x = val \rangle \leftrightarrow \langle q x = val \rangle)

- p, q equivalent relative to computation state restricted to x
Dynamic Logic Example Formulas

\[
\begin{align*}
\text{if } & \ a \neq \text{null} \\
\Rightarrow & \ \\
\text{int } & \ max = 0; \\
\text{if } & \ (a.\text{length} > 0) \ max = a[0]; \\
\text{int } & \ i = 1; \\
\text{while } & \ (i < a.\text{length}) \ {\{} \\
\text{if } & \ (a[i] > max) \ max = a[i]; \\
\text{++} & \ i; \\
{\}} \\
> & \\
( & \\
\forall \text{int } j; (j >= 0 \ & \ j < a.\text{length} \rightarrow max >= a[j]) \\
\& & \\
(a.\text{length} > 0 \rightarrow \\
\exists \text{int } j; (j >= 0 \ & \ j < a.\text{length} \ & \ max = a[j])\})
\end{align*}
\]
Variables

Logical variables disjoint from program variables

- No quantification over program variables
- Programs do not contain logical variables
- “Program variables” actually non-rigid functions
Rigid and Flexible Terms

Example

<int i;> \forall int x; (i + 1 \equiv x \rightarrow <i++;>(i \equiv x))

- Interpretation of $i$ depends on computation state  \Rightarrow \text{flexible}
- Interpretation of $x$ and $+$ do not depend on state \Rightarrow \text{rigid}

Locations are always \text{flexible}

Logical variables, standard functions are always \text{rigid}
Rigid and Flexible Terms

Example

\(<\text{int } i;> \forall \text{int } x; (i + 1 \equiv x \rightarrow \langle i++;\rangle (i \equiv x))\>

- Interpretation of \(i\) depends on computation state \(\Rightarrow\) flexible
- Interpretation of \(x\) and + do not depend on state \(\Rightarrow\) rigid

Locations are always flexible

Logical variables, standard functions are always rigid
Rigid and Flexible Terms

Example

\[\text{Example}\]

\(<\text{int}\ i;>\ \forall\ \text{int}\ x; (i + 1 \equiv x \rightarrow <i++;>(i \equiv x))\]

- Interpretation of \(i\) depends on computation state \(\Rightarrow\) flexible
- Interpretation of \(x\) and + do not depend on state \(\Rightarrow\) rigid

Locations are always flexible
Logical variables, standard functions are always rigid
Rigid and Flexible Terms

Example

\[
\langle \text{int } i; \rangle \forall \text{int } x; (i + 1 \equiv x \rightarrow \langle i++; \rangle (i \equiv x))
\]

- Interpretation of \( i \) depends on computation state \( \Rightarrow \) flexible
- Interpretation of \( x \) and + do not depend on state \( \Rightarrow \) rigid

Locations are always flexible
Logical variables, standard functions are always rigid
Rigid and Flexible Terms

Example

\( <\text{int } i;> \forall \text{int } x; (i + 1 \equiv x \rightarrow <i++;> (i \equiv x)) \)

- Interpretation of \( i \) depends on computation state \( \Rightarrow \text{flexible} \)
- Interpretation of \( x \) and \( + \) do not depend on state \( \Rightarrow \text{rigid} \)

Locations are always flexible
Logical variables, standard functions are always rigid
Validity

A JAVA CARD DL formula is valid iff it is true in all states.

We need a calculus for checking validity of formulas.
Validity

A Java Card DL formula is valid iff it is true in all states.

We need a calculus for checking validity of formulas.
Part III
Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
Part III

Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
**Sequents and their Semantics**

**Syntax**

\[ \psi_1, \ldots, \psi_m \quad \Rightarrow \quad \phi_1, \ldots, \phi_n \]

where the \( \phi_i, \psi_i \) are formulae (without free variables)

**Semantics**

Same as the formula

\[ (\psi_1 \& \ldots \& \psi_m) \quad \Rightarrow \quad (\phi_1 \mid \ldots \mid \phi_n) \]
Sequents and their Semantics

Syntax

\[
\psi_1, \ldots, \psi_m \implies \phi_1, \ldots, \phi_n
\]

where the \( \phi_i, \psi_i \) are formulae (without free variables)

Semantics

Same as the formula

\[
(\psi_1 \land \cdots \land \psi_m) \implies (\phi_1 | \cdots | \phi_n)
\]
Sequent Rules

General form

Premisses

\[ \Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Rightarrow \Delta_r \]

\[ \Gamma \Rightarrow \Delta \]

Conclusion

\((r = 0 \text{ possible: closing rules})\)

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion
Sequent Rules

General form

\[
\frac{\Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Rightarrow \Delta_r}{\Gamma \Rightarrow \Delta}
\]

(rule name) 

Premisses

Conclusion

\(r = 0\) possible: closing rules

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion
Sequent Rules

**General form**

Premisses

\[ \Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Rightarrow \Delta_r \]

\[
\frac{\Gamma \Rightarrow \Delta}{\text{Conclusion}}
\]

\((r = 0 \text{ possible: closing rules})\)

**Soundness**

If all premisses are valid, then the conclusion is valid

**Use in practice**

Goal is matched to conclusion
Sequent Rules

General form

\[
\begin{array}{c}
\Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Rightarrow \Delta_r \\
\Gamma \Rightarrow \Delta
\end{array}
\]

(rule_name)

Premisses

Conclusion

\(r = 0\) possible: closing rules

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion
Some Simple Sequent Rules

**not_left**
\[
\frac{\Gamma \Rightarrow A, \Delta}{\Gamma, \neg A \Rightarrow \Delta}
\]

**imp_left**
\[
\begin{align*}
\frac{\Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}
\end{align*}
\]

**close_goal**
\[
\frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow A, \Delta}
\]

**close_by_true**
\[
\frac{\Gamma \Rightarrow \text{true}, \Delta}{\Gamma \Rightarrow \text{true}, \Delta}
\]

**all_left**
\[
\frac{\Gamma, \forall t x; \phi, \{x/e\} \phi \Rightarrow \Delta}{\Gamma, \forall t x; \phi \Rightarrow \Delta}
\]

where \(e\) var-free term of type \(t' < t\)
Some Simple Sequent Rules

**not_left**

\[
\frac{\Gamma \Rightarrow A, \Delta}{\Gamma, \neg A \Rightarrow \Delta}
\]

**imp_left**

\[
\frac{\Gamma \Rightarrow A, \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}
\]

**close_goal**

\[
\frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow A, \Delta}
\]

**close_by_true**

\[
\frac{\Gamma \Rightarrow true, \Delta}{\Gamma \Rightarrow true, \Delta}
\]

**all_left**

\[
\frac{\Gamma, \forall t \; x; \phi, \{x/e\} \phi \Rightarrow \Delta}{\Gamma, \forall t \; x; \phi \Rightarrow \Delta}
\]

where \(e\) var-free term of type \(t' < t\)
Some Simple Sequent Rules

\textbf{not\_left} \quad \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, !A \Rightarrow \Delta}

\textbf{imp\_left} \quad \frac{\Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}

\textbf{close\_goal} \quad \frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow A, \Delta}

\textbf{close\_by\_true} \quad \frac{\Gamma \Rightarrow true, \Delta}{\Gamma \Rightarrow true, \Delta}

\textbf{all\_left} \quad \frac{\Gamma, \forall t x; \phi, \{x/e\} \phi \Rightarrow \Delta}{\Gamma, \forall t x; \phi \Rightarrow \Delta}

where \( e \) var-free term of type \( t' \prec t \)
Some Simple Sequent Rules

**not_left**

\[
\frac{\Gamma \Rightarrow A, \Delta}{\Gamma, !A \Rightarrow \Delta}
\]

**imp_left**

\[
\frac{\Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}
\]

**close_goal**

\[
\frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow A, \Delta}
\]

**close_by_true**

\[
\frac{\Gamma \Rightarrow true, \Delta}{\Gamma \Rightarrow true, \Delta}
\]

**all_left**

\[
\frac{\Gamma, \forall t x; \phi, \{x/e\}\phi \Rightarrow \Delta}{\Gamma, \forall t x; \phi \Rightarrow \Delta}
\]

where e var-free term of type \( t' < t \)
Some Simple Sequent Rules

**not_left**

\[ \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, ! A \Rightarrow \Delta} \]

**imp_left**

\[ \frac{\Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta} \]

**close_goal**

\[ \frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow A, \Delta} \]

**close_by_true**

\[ \frac{\Gamma \Rightarrow true, \Delta}{\Gamma \Rightarrow true, \Delta} \]

**all_left**

\[ \frac{\Gamma, \forall t \; x; \phi, \{x/e\} \phi \Rightarrow \Delta}{\Gamma, \forall t \; x; \phi \Rightarrow \Delta} \]

where \( e \) var-free term of type \( t' \prec t \)
Sequent Calculus Proofs

**Proof tree**

- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed
Sequent Calculus Proofs

Proof tree

- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed
Sequent Calculus Proofs

Proof tree

- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed
Sequent Calculus Proofs

Proof tree

- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed
Part III

Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
Part III

Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

- Sequent rules execute symbolically the active statement
Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

```java
l:{try{ i=0; j=0; } finally{ k=0; }}
```

- Sequent rules execute symbolically the active statement
Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

\[ l: \{ \text{try} \{ i=0; j=0; \} \ \text{finally} \{ k=0; \} \} \]

- Sequent rules execute symbolically the active statement
Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

```
l:{try{  i=0;  j=0; }  finally{  k=0; }}
```

- passive prefix  \( \pi \)
- active statement  \( i=0; \)
- rest  \( \omega \)

Sequent rules execute symbolically the active statement
Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

\[ l: \{ \text{try} \{ \begin{aligned} i &= 0; \\ j &= 0; \end{aligned} \} \text{ finally} \{ k &= 0; \} \} \]

passive prefix \( \pi \)
active statement \( i = 0; \)
rest \( \omega \)

Sequent rules execute symbolically the active statement
Rules for Symbolic Program Execution

If-then-else rule

\[
\Gamma, B = \text{true} \implies <p \omega> \phi, \Delta \quad \Gamma, B = \text{false} \implies <q \omega> \phi, \Delta
\]
\[
\Gamma \implies <\text{if} (B) \{ p \} \text{ else } \{ q \} \omega> \phi, \Delta
\]

Complicated statements/expressions are simplified first, e.g.

\[
\Gamma \implies <v=y; y=y+1; x=v; \omega> \phi, \Delta
\]
\[
\Gamma \implies <x=y++; \omega> \phi, \Delta
\]

Simple assignment rule

\[
\Gamma \implies \{\text{loc} := \text{val}\}<\omega> \phi, \Delta
\]
\[
\Gamma \implies <\text{loc}=\text{val}; \omega> \phi, \Delta
\]
If-then-else rule

\[
\begin{align*}
\Gamma, B = true & \Rightarrow <p \omega> \phi, \Delta \\
\Gamma, B = false & \Rightarrow <q \omega> \phi, \Delta \\
\Gamma & \Rightarrow <\text{if} (B) \{ p \}\text{ else } \{ q \} \omega> \phi, \Delta
\end{align*}
\]

Complicated statements(expressions are simplified first, e.g.

\[
\begin{align*}
\Gamma & \Rightarrow <v=y; y=y+1; x=v; \omega> \phi, \Delta \\
\Gamma & \Rightarrow <x=y++; \omega> \phi, \Delta
\end{align*}
\]

Simple assignment rule

\[
\begin{align*}
\Gamma & \Rightarrow \{loc := val\} <\omega> \phi, \Delta \\
\Gamma & \Rightarrow <loc=val; \omega> \phi, \Delta
\end{align*}
\]
Rules for Symbolic Program Execution

If-then-else rule

\[ \Gamma, B = true \implies \langle p \omega \rangle \phi, \Delta \quad \Gamma, B = false \implies \langle q \omega \rangle \phi, \Delta \]
\[ \Gamma \implies \langle \text{if} \ (B) \{ \ p \ } \ \text{else} \{ \ q \} \ \omega \rangle \phi, \Delta \]

Complicated statements/expressions are simplified first, e.g.

\[ \Gamma \implies \langle v=y; \ y=y+1; \ x=v; \ \omega \rangle \phi, \Delta \]
\[ \Gamma \implies \langle x=y++; \ \omega \rangle \phi, \Delta \]

Simple assignment rule

\[ \Gamma \implies \{ \text{loc} := \text{val} \} \langle \omega \rangle \phi, \Delta \]
\[ \Gamma \implies \langle \text{loc}=\text{val}; \ \omega \rangle \phi, \Delta \]
Extending DL by Explicit State Updates

Updates
explicit syntactic elements in the logic

Elementary Updates

\{loc := val\} \phi

where (roughly)

- \textit{loc} a program variable \(x\), an attribute access \(o.attr\), or an array access \(a[i]\)
- \textit{val} is same as \textit{loc}, or a literal, or a logical variable

Parallel Updates

\{loc_1 := t_1 \mid \cdots \mid loc_n := t_n\} \phi

no dependency between the \(n\) components (but ‘right wins’ semantics)
Extending DL by Explicit State Updates

Updates

explicit syntactic elements in the logic

Elementary Updates

\[
\{ loc := val \} \phi
\]

where (roughly)
- \( loc \) a program variable \( x \), an attribute access \( o.attrib \), or an array access \( a[i] \)
- \( val \) is same as \( loc \), or a literal, or a logical variable

Parallel Updates

\[
\{ loc_1 := t_1 \parallel \cdots \parallel loc_n := t_n \} \phi
\]

no dependency between the \( n \) components (but ‘right wins’ semantics)
Extending DL by Explicit State Updates

Updates
explicit syntactic elements in the logic

Elementary Updates
\[
\{ \text{loc} := \text{val} \} \phi
\]

where (roughly)
- \textit{loc} a program variable \(x\), an attribute access \(o.attr\), or an array access \(a[i]\)
- \textit{val} is same as \textit{loc}, or a literal, or a logical variable

Parallel Updates
\[
\{ \text{loc}_1 := t_1 \parallel \cdots \parallel \text{loc}_n := t_n \} \phi
\]

no dependency between the \(n\) components (but ‘right wins’ semantics)
Why Updates?

Updates are:
- *lazily applied* (i.e. substituted into postcondition)
- *eagerly parallelised* + simplified

Advantages
- no renaming required
- delayed/minimized proof branching (efficient aliasing treatment)
Why Updates?

Updates are:

- *lazily applied* (i.e. substituted into postcondition)
- *eagerly parallelised* + simplified

Advantages

- no renaming required
- delayed/minimized proof branching (efficient aliasing treatment)
Symbolic Execution with Updates
(by Example)

\[ x < y \implies x < y \]
\[ \rightarrow \]
\[ x < y \implies \{ x := y \mid y := x \} \implies y < x \]
\[ \rightarrow \]
\[ x < y \implies \{ t := x \mid x := y \mid y := x \} \implies y < x \]
\[ \rightarrow \]
\[ x < y \implies \{ t := x \mid x := y \} \{ y := t \} \implies y < x \]
\[ \rightarrow \]
\[ x < y \implies \{ t := x \} \{ x := y \} \{ y := t \} \implies y < x \]
\[ \rightarrow \]
\[ x < y \implies \{ t := x \} \{ x := y \} \{ y := t \} \implies y < x \]
\[ \rightarrow \]
\[ x < y \implies \{ t := x \} \{ x := y \} \{ y := t \} \implies y < x \]
\[ \rightarrow \]
\[ x < y \implies \{ t := x \} \{ x := y \} \{ y := t \} \implies y < x \]
\[ \rightarrow \]
\[ x < y \implies \{ t := x \} \{ x := y \} \{ y := t \} \implies y < x \]
\[ \rightarrow \]
\[ x < y \implies \{ t := x \} \{ x := y \} \{ y := t \} \implies y < x \]
Symbolic Execution with Updates (by Example)

\[ x < y \implies x < y \]

\[ x < y \implies \{ x:=y \mid \{ y:=x \} \} \iff y < x \]

\[ x < y \implies \{ t:=x \mid \{ x:=y \} \} \iff y < x \]

\[ x < y \implies \{ t:=x \} \iff y < x \]

\[ x < y \implies \{ t:=x \} < x=y; y=t; \iff y < x \]

\[ \implies x < y \rightarrow <\text{int } t=x; x=y; y=t;> y < x \]
Symbolic Execution with Updates (by Example)

\[ x < y \implies x < y \]

\[ x < y \implies \{x:=y \parallel y:=x\} \iff y < x \]

\[ x < y \implies \{t:=x \parallel x:=y \parallel y:=x\} \iff y < x \]

\[ x < y \implies \{t:=x \parallel x:=y\}\{y:=t\} \iff y < x \]

\[ x < y \implies \{t:=x\}\{x:=y\}<y=t;> y < x \]

\[ x < y \implies \{t:=x\}<x=y; y=t;> y < x \]

\[ \implies x < y \rightarrow \text{int } t=x; x=y; y=t;> y < x \]
Symbolic Execution with Updates (by Example)

\[ x < y \implies x < y \]
\[ \vdots \]
\[ x < y \implies \{ x:=y \parallel y:=x \} \not\Rightarrow y < x \]
\[ \vdots \]
\[ x < y \implies \{ t:=x \parallel x:=y \parallel y:=x \} \not\Rightarrow y < x \]
\[ \vdots \]
\[ x < y \implies \{ t:=x \parallel x:=y \} \{ y:=t \} \not\Rightarrow y < x \]
\[ \vdots \]
\[ x < y \implies \{ t:=x \} \{ x:=y \} <y=t;> y < x \]
\[ \vdots \]
\[ x < y \implies \{ t:=x \} <x=y; y=t;> y < x \]
\[ \implies x < y \rightarrow <\text{int } t=x; x=y; y=t;> y < x \]
Symbolic Execution with Updates
(by Example)

\[
x < y \implies x < y
\]

\[
\vdots
\]

\[
x < y \implies \{x := y \parallel y := x\} \not\in y < x
\]

\[
\vdots
\]

\[
x < y \implies \{t := x \parallel x := y \parallel y := x\} \not\in y < x
\]

\[
\vdots
\]

\[
x < y \implies \{t := x \parallel x := y\}\{y := t\} \not\in y < x
\]

\[
\vdots
\]

\[
x < y \implies \{t := x\}\{x := y\}\{y := t; > y < x
\]

\[
\vdots
\]

\[
x < y \implies \{t := x\}\{x := y\} y = t; > y < x
\]

\[
\vdots
\]

\[
\implies x < y \implies \text{int } t = x; x = y; y = t; > y < x
\]
Symbolic Execution with Updates
(by Example)

\[
x < y \quad \implies \quad x < y
\]

\[
\vdots
\]

\[
x < y \quad \implies \quad \{x := y \parallel y := x\} \quad \not\!
\]

\[
\vdots
\]

\[
x < y \quad \implies \quad \{t := x \parallel x := y \parallel y := x\} \quad \not\!
\]

\[
\vdots
\]

\[
x < y \quad \implies \quad \{t := x \parallel x := y\} \{y := t\} \quad \not\!
\]

\[
\vdots
\]

\[
x < y \quad \implies \quad \{t := x\} \{x := y\} \{y := t\} \quad \not\!
\]

\[
\vdots
\]

\[
x < y \quad \implies \quad \{t := x\} \quad \not\!
\]

\[
\implies \quad x < y \quad \implies \quad \text{int } t = x; \ x = y; \ y = t; \ y < x
\]
Symbolic Execution with Updates (by Example)

\[
\begin{align*}
x < y & \implies x < y \\
& \vdots \\
x < y & \implies \{x:=y \mid y:=x\} \iff y < x \\
& \vdots \\
x < y & \implies \{t:=x \mid x:=y \mid y:=x\} \iff y < x \\
& \vdots \\
x < y & \implies \{t:=x \mid x:=y\}\{y:=t\} \iff y < x \\
& \vdots \\
x < y & \implies \{t:=x\}\{x:=y\}\langle y=t; \rangle \iff y < x \\
& \vdots \\
x < y & \implies \{t:=x\}\langle x=y; y=t; \rangle \iff y < x \\
& \vdots \\
& \implies x < y \rightarrow \langle \text{int } t=x; x=y; y=t; \rangle \iff y < x
\end{align*}
\]
Program State Representation

Local program variables
Modeled as non-rigid constants

Heap
Modeled with theory of arrays:

\[ \text{heap}: \rightarrow \text{Heap} \text{ (the heap in the current state)} \]
\[ \text{select}: \text{Heap} \times \text{Object} \times \text{Field} \rightarrow \text{Any} \]
\[ \text{store}: \text{Heap} \times \text{Object} \times \text{Field} \times \text{Any} \rightarrow \text{Heap} \]

Heap axioms (excerpt)

\[ \text{select}(\text{store}(h, o, f, x), o, f) = x \]
\[ \text{select}(\text{store}(h, o, f, x), u, f) = \text{select}(h, u, f) \text{ if } o \neq u \]
Program State Representation

Local program variables
Modeled as non-rigid constants

Heap
Modeled with theory of arrays:

\[ \text{heap} : \rightarrow \text{Heap} \text{ (the heap in the current state)} \]

\[ \text{select} : \text{Heap} \times \text{Object} \times \text{Field} \rightarrow \text{Any} \]

\[ \text{store} : \text{Heap} \times \text{Object} \times \text{Field} \times \text{Any} \rightarrow \text{Heap} \]

Heap axioms (excerpt)

\[ \text{select}(\text{store}(h, o, f, x), o, f) = x \]

\[ \text{select}(\text{store}(h, o, f, x), u, f) = \text{select}(h, u, f) \text{ if } o \neq u \]
Program State Representation

Local program variables
Modeled as non-rigid constants

Heap
Modeled with theory of arrays:

\[
\text{heap} : \rightarrow \text{Heap} \quad (\text{the heap in the current state})
\]

\[
\text{select} : \text{Heap} \times \text{Object} \times \text{Field} \rightarrow \text{Any}
\]

\[
\text{store} : \text{Heap} \times \text{Object} \times \text{Field} \times \text{Any} \rightarrow \text{Heap}
\]

Heap axioms (excerpt)

\[
\text{select} (\text{store}(h, o, f, x), o, f) = x
\]

\[
\text{select} (\text{store}(h, o, f, x), u, f) = \text{select}(h, u, f) \text{ if } o \neq u
\]
Handling Abrupt Termination

- Abrupt termination handled by program transformations
- Changing control flow = rearranging program parts

Example
TRY-THROW

\[ \Gamma \Rightarrow \langle \begin{cases} \text{try } \{ e=\text{exc}; \ r \} \text{ finally } \{ s \} \end{cases} \phi, \Delta \\ \text{else} \{ s \text{ throw exc;} \} \ \omega \rangle \]

\[ \Gamma \Rightarrow \langle \text{try} \{ \text{throw exc}; \ q \} \ \text{catch}(T \ e)\{ r \} \ \text{finally}\{ s \} \ \omega \rangle \phi, \Delta \]
Handling Abrupt Termination

- Abrupt termination handled by program transformations
- Changing control flow = rearranging program parts

Example

TRY-THROW

\[ \Gamma \Rightarrow \langle \begin{align*}
& \text{if (exc instanceof T)} \\
& \quad \{ \text{try} \{ e = \text{exc}; \ r \} \text{ finally } \{ s \} \} & \phi, \Delta \\
& \quad \text{else } \{ s \text{ throw exc;} \} & \omega
\end{align*} \rangle \]

\[ \Gamma \Rightarrow \langle \text{try}\{ \text{throw exc; q} \} \text{ catch(T e)\{}r\} \text{ finally}\{s\} \rangle \phi, \Delta \]
Handling Abrupt Termination

- Abrupt termination handled by program transformations
- Changing control flow = rearranging program parts

Example

TRY-THROW

\[ \Gamma \Rightarrow < \pi \text{ if (exc instanceof T)} \]
\[ \{ \text{try } \{ e=\text{exc}; r \} \text{ finally } \{ s \} \} \phi, \Delta \]
\[ \text{else } \{ s \text{ throw exc;} \} \omega \]

\[ \Gamma \Rightarrow < \pi \text{ try}\{ \text{throw exc; q} \} \text{ catch(T e)}\{ r \} \text{ finally}\{ s \} \omega > \phi, \Delta \]
Part III

Program Verification with Dynamic Logic

7 Java Card DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% Java Card

11 Taclets – KeY’s Rule Description Language
Part III

Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
Supported Java Features

- method invocation with polymorphism/dynamic binding
- object creation and initialisation
- arrays
- abrupt termination
- throwing of NullPointerExceptions, etc.
- bounded integer data types
- transactions

All JAVA CARD language features are fully addressed in KeY
Supported Java Features

- method invocation with polymorphism/dynamic binding
- object creation and initialisation
- arrays
- abrupt termination
- throwing of NullPointerExceptions, etc.
- bounded integer data types
- transactions

All JAVA CARD language features are fully addressed in KeY
Java—A Language of Many Features

Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

Pro: Feature needs not be handled in calculus
Contra: Modified source code
Example in KeY: Very rare: treating inner classes
Java—A Language of Many Features

Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

**Pro:** Flexible, easy to implement, usable

**Contra:** Not expressive enough for all features

**Example in KeY:** Complex expression eval, method inlining, etc., etc.
Java—A Language of Many Features

Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

Pro: No logic extensions required, enough to express most features
Contra: Creates difficult first-order POs, unreadable antecedents
Example in KeY: Dynamic types and branch predicates
Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

**Pro:** Arbitrarily expressive extensions possible
**Contra:** Increases complexity of all rules

*Example in KeY:* Method frames, updates
Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules

2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates

3. Rules for handling loops
   - using loop invariants
   - using induction

4. Rules for replacing a method invocations by the method’s contract

5. Update simplification
Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules

2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates

3. Rules for handling loops
   - using loop invariants
   - using induction

4. Rules for replacing a method invocations by the method’s contract

5. Update simplification
Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules

2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates

3. Rules for handling loops
   - using loop invariants
   - using induction

4. Rules for replacing a method invocations by the method’s contract

5. Update simplification
Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules

2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates

3. Rules for handling loops
   - using loop invariants
   - using induction

4. Rules for replacing a method invocations by the method’s contract

5. Update simplification
Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules

2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates

3. Rules for handling loops
   - using loop invariants
   - using induction

4. Rules for replacing a method invocations by the method’s contract

5. Update simplification
Part III

Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
Part III

Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
Taclets:
KeY’s Rule Description Language

Taclets ...  
- represent sequent calculus rules in KeY  
- use a simple text-based format  
- are descriptive, but with operational flavor  
- are *not* a tactic metalanguage
Taclet Syntax

\[ \text{andLeft} \quad \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \& B \Rightarrow \Delta} \]

Taclet

\[
\text{andLeft} \{ \\
\quad \text{\textbackslash find ( A & B ==> )} \\
\quad \text{\textbackslash replacewith ( A, B ==>)} \\
\}
\]

- Unique name
- Find expression:
  - Formula (Term) to be modified
    - Sequent arrow ==> formula must occur top level and on the corresponding side of the sequent
- Goal Description: describes new sequent
Taclet Syntax

\[
\text{andLeft} \quad \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \& B \Rightarrow \Delta}
\]

Taclet

\[
\text{andLeft} \ {\{
   \find ( A \& B \Rightarrow ) \\
   \replacewith ( A, B \Rightarrow )
\}};
\]

- Unique name
- Find expression:
  - Formula (Term) to be modified
  - Sequent arrow \(\Rightarrow\) formula must occur top level \(and\) on the corresponding side of the sequent.
- Goal Description: describes new sequent
**Taclet Syntax**

\[
\text{andLeft} \quad \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \& B \Rightarrow \Delta}
\]

**Taclet**

```latex
\text{andLeft} \{
\textbf{\texttt{\textbackslash find}} \ (A \& B \Rightarrow )
\textbf{\texttt{\textbackslash replacewith}} \ (A, B \Rightarrow )
\}
```

- **Unique name**
- **Find expression:**
  - Formula (Term) to be modified
  - Sequent arrow ==> formula must occur top level and on the corresponding side of the sequent.
- **Goal Description:** describes new sequent
Taclet Syntax

\[
\text{andLeft} \quad \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \& B \Rightarrow \Delta}
\]

Taclet

\text{andLeft} \{ \\
\text{\textbackslash find ( A \& B ==> )} \\
\text{\textbackslash replacewith ( A, B ==>)}} \\
\}

- Unique name
- Find expression:
  - Formula (Term) to be modified
  - Sequent arrow ==> formula must occur top level and on the corresponding side of the sequent.
- Goal Description: describes new sequent
**Taclet Syntax**

\[
\text{andLeft } \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \& B \Rightarrow \Delta}
\]

---

**Taclet**

```plaintext
andLeft { 
  \find ( A \& B ==> ) 
  \replacewith ( A, B ==>) 
};
```

- **Unique name**
- **Find expression:**
  - Formula (Term) to be modified
  - Sequent arrow ==> formula must occur top level *and* on the corresponding side of the sequent.
- **Goal Description:** describes new sequent
Some rules are only sound in a certain context

\[
\text{modusPonens} \quad \frac{\Gamma, A, B \implies \Delta}{\Gamma, A, A \rightarrow B \implies \Delta}
\]

\text{Taclet}

\begin{verbatim}
modusPonens {
  \assumes ( A ==> )
  \find ( A -> B ==> )
  \replacewith( B ==> )
};
\end{verbatim}
Some rules are only sound in a certain context

modusPonens \[
\Gamma, A, B \Rightarrow \Delta \\
\Gamma, A, A \rightarrow B \Rightarrow \Delta
\]

Taclet

modusPonens {
  \assumes ( A ==> )
  \find ( A -> B ==> )
  \replacewith( B ==> )
};
Some rules are only sound in a certain context

**Taclet Syntax**

\[ \Gamma, A, B \Rightarrow \Delta \]
\[ \Gamma, A, A \rightarrow B \Rightarrow \Delta \]

**Taclet**

modusPonens {
  \assumes ( A ==> )
  \find ( A -> B ==> )
  \replacewith( B ==> )
};
Taclet Syntax

Proof Splitting: andRight

\[
\frac{\Gamma \Rightarrow A, \Delta \quad \Gamma \Rightarrow B, \Delta}{\Gamma \Rightarrow A \& B, \Delta}
\]

\[
\text{andRight} \begin{array}{l}
\text{\find ( \Rightarrow A \& B )} \\
\text{\replacewith (\Rightarrow A );} \\
\text{\replacewith (\Rightarrow B )}
\end{array}
\]

};

Variable Conditions: allRight

\[
\frac{\Gamma \Rightarrow \{x/c\} \Phi, \Delta}{\Gamma \Rightarrow \forall T \ x; \Phi, \Delta}
\]

\[
\text{allRight} \begin{array}{l}
\text{\find ( \Rightarrow \forall x;\phi )} \\
\text{\varcond(\new(c,\dependingOn(\phi)))} \\
\text{\replacewith ( \Rightarrow \{\subst x;c}\phi )}
\end{array}
\]

};
Taclet Syntax

Proof Splitting: andRight

\[ \Gamma \Rightarrow A, \Delta \quad \Gamma \Rightarrow B, \Delta \]
\[ \Gamma \Rightarrow A \land B, \Delta \]

andRight {
\find ( \Rightarrow A \land B )
\replacewith (\Rightarrow A )
\replacewith (\Rightarrow B )
}\;

Variable Conditions: allRight

\[ \Gamma \Rightarrow \{ x/c \} \Phi, \Delta \]
\[ \Gamma \Rightarrow \forall T \, x; \Phi, \Delta \]

c new

allRight {
\find ( \Rightarrow \forall x; \phi )
\varcond(\new(c,\dependingOn(\phi)))
\replacewith ( \Rightarrow \{ subst \, x;c\} \phi )
}\;
Taclets for Program Transformations

\[
\Gamma \Rightarrow \begin{cases} 
\pi \text{ if (exc == null) } \{ 
\text{try} \{ \text{throw new NPE()}; \text{catch}(T \ e) \{r\}; \} \text{ else if (exc instanceof T) } \{e=exc; \ r\} \} \text{ else throw exc; } \omega
\end{cases} \phi
\]

\[
\Gamma \Rightarrow <\pi \text{ try}\{\text{throw exc}; \ q\} \text{ catch}(T \ e)\{r\}; \ \omega> \phi
\]

\text{\textbackslash find} ( <.. \text{ try } \{ \text{throw } \#se; \ \#slist \} \\
\quad \text{catch} ( \#t \ \#v0 ) \{ \#slist1 \} \ldots \text{ post } )

\text{\textbackslash replacewith} ( \\
\quad <.. \text{ if (} \#se == \text{null}) \{ 
\quad \text{try} \{ \text{throw new NullPointerException(); } \} \n\quad \text{catch}(\#t \ \#v0) \{ \#slist1 \}
\quad \} \text{ else if } (\#se \text{ instanceof } \#t) \{ 
\quad \#t \ \#v0 = (\#t) \ #se;
\quad \#slist1
\quad \} \text{ else throw } \#se; \ \ldots > \text{ post } )
Part IV

Verifying Information Flow Properties

12 Non-Interference
- Definition
- Reformulation and Formalisation – Alternating Quantifiers
- Reformulation and Formalisation – Self Composition
- Declassification

13 JML Non-Interference Specifications
- Views and Security Policies
- Non-Interference in JML
Part IV

Verifying Information Flow Properties

12 Non-Interference
- Definition
- Reformulation and Formalisation – Alternating Quantifiers
- Reformulation and Formalisation – Self Composition
- Declassification

13 JML Non-Interference Specifications
- Views and Security Policies
- Non-Interference in JML
Non-Interference

Prominent information flow property: non-interference

Simple case:
- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$
  - high security variables $high$

Definition (Non-interference – Version 1)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ \iff when starting $P$ with arbitrary values for $low$, then the values of $low$ after executing $P$, are independent of the choices of $high$. 
Non-Interference

Prominent information flow property: **non-interference**

Simple case:
- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$ and
  - high security variables $high$

**Definition (Non-interference – Version 1)**

For program $P$ the high variables $high$ do not interfere with the low variables $low$ ⇔ when starting $P$ with arbitrary values for $low$, then the values of $low$ after executing $P$, are independent of the choices of $high$. 
Non-Interference

Prominent information flow property: non-interference

Simple case:
- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$
  - high security variables $high$

Definition (Non-interference – Version 1)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ ⇔
when starting $P$ with arbitrary values for $low$, then the values of $low$ after executing $P$, are independent of the choices of $high$. 
Non-Interference

Prominent information flow property: non-interference

Simple case:
- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$
  - high security variables $high$

Definition (Non-interference – Version 1)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ $\iff$
when starting $P$ with arbitrary values for $low$, then the values of $low$ after executing $P$, are independent of the choices of $high$. 
Non-Interference

Prominent information flow property: **non-interference**

Simple case:
- program $P$
- partition of the program variables of $P$ in
  - low security variables $low$
  - high security variables $high$

**Definition (Non-interference – Version 1)**

For program $P$ the high variables $high$ do not interfere with the low variables $low$ $\iff$

when starting $P$ with arbitrary values for $low$, then the values of $low$ after executing $P$, are independent of the choices of $high$. 
Non-Interference

Prominent information flow property: non-interference

Simple case:

- program \( P \)
- partition of the program variables of \( P \) in
  - low security variables \( low \) and
  - high security variables \( high \)

Definition (Non-interference – Version 1)

For program \( P \) the high variables \( high \) do not interfere with the low variables \( low \)

\[ \iff \]

when starting \( P \) with arbitrary values for \( low \), then the values of \( low \) after executing \( P \), are independent of the choices of \( high \).
Examples

Which methods are secure?

class MiniExamples {
    public int l;
    private int h;

    void m_1() {
        l = h;
    }

    void m_2() {
        if (l > 0) {h = 1;}
        else {h = 2;};
    }

    void m_3() {
        if (h > 0) {l = 1;}
        else {l = 2;};
    }

    void m_4() {
        h = 0; l = h;
    }
}
Examples

Which methods are secure?

class MiniExamples {
    public int l;
    private int h;

    void m_1() {
        l = h;
    }

    void m_2() {
        if (l>0) {h=1;}
        else {h=2;};
    }

    void m_3() {
        if (h>0) {l=1;}
        else {l=2;};
    }

    void m_4() {
        h=0; l=h;
    }
}
Examples

Which methods are secure?

class MiniExamples {
    public int l;
    private int h;

    void m_1() {
        l = h;
    }

    void m_2() {
        if (l > 0) {h = 1;}
        else {h = 2;};
    }

    void m_3() {
        if (h > 0) {l = 1;}
        else {l = 2;};
    }

    void m_4() {
        h = 0; l = h;
    }
}
Examples

Which methods are secure?

class MiniExamples {
    public int l;
    private int h;

    void m_1() {
        l = h;
    }

    void m_2() {
        if (l > 0) {h = 1;}
        else {h = 2;};
    }

    void m_3() {
        if (h > 0) {l = 1;}
        else {l = 2;};
    }

    void m_4() {
        h = 0; l = h;
    }
}
Examples

Which methods are secure?

class MiniExamples {
    public int l;
    private int h;

    void m_1() {
        l = h;
    }

    void m_2() {
        if (l>0) {h=1;}
        else {h=2;};
    }

    void m_3() {
        if (h>0) {l=1;}
        else {l=2;};
    }

    void m_4() {
        h=0; l=h;
    }
}
Examples

Which methods are secure?

```java
void m_5() {
    l = h; l = l - h;
}

void m_6() {
    if (false) l = h;
}
```
Examples

Which methods are secure?

```java
void m_5() {
    l=h; l=l-h;
}

void m_6() {
    if (false) l=h;
}
```
Examples

Which methods are secure?

```java
void m_5() {
    l=h; l=l-h;
}
```

```java
void m_6() {
    if (false) l=h;
}
```

```java
}
```
Formalisation in JavaDL – Alternating Quantifiers

**Definition (Non-interference – Version 2)**

For program $P$ the high variables $\textit{high}$ do not interfere with the low variables $\textit{low}$ if

$\forall \textit{in}_l \exists \textit{r} \forall \textit{in}_h (\{ \textit{low} := \textit{in}_l \mid \textit{high} := \textit{in}_h \} [P] \textit{low} = \textit{r})$

- **Problem:** not suitable for automatic verification $\rightsquigarrow$ instantiation of existential quantifier difficult.
Formalisation in JavaDL – Alternating Quantifiers

Definition (Non-interference – Version 2)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ if

$$\forall in_l \exists r \forall in_h (\{low := in_l || high := in_h\}[P]low = r)$$

for all low input values $in_l$ there exist low output values $r$ such that for all high input values $in_h$ if we assign the values $in_l$ to the program variables $low$ and $in_h$ to the program variables $high$ then after execution of $P$ the values of $low$ are $r$.

Problem: not suitable for automatic verification $\leadsto$ instantiation of existential quantifier difficult.
Definition (Non-interference – Version 2)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ if

$$\forall in_l \exists r \forall in_h (\{low := in_l \mid high := in_h\}[P]low = r)$$

Problem: not suitable for automatic verification $\sim$ instantiation of existential quantifier difficult.
Formalisation in JavaDL – Alternating Quantifiers

Definition (Non-interference – Version 2)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ if

$\forall in_l \exists r \forall in_h (\{low := in_l \mid high := in_h\}[P]low = r)$

- **Problem**: not suitable for automatic verification $\leadsto$ instantiation of existential quantifier difficult.
Definition (Non-interference – Version 2)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ if for all low input values $in_l$ there exist low output values $r$ such that for all high input values $in_h$ if we assign the values $in_l$ to the program variables $low$ and $in_h$ to the program variables $high$ then after execution of $P$ the values of $low$ are $r$. 

$$\forall in_l \exists r \forall in_h (\{low := in_l \mid high := in_h\}[P]low = r)$$

Problem: not suitable for automatic verification $\Rightarrow$ instantiation of existential quantifier difficult.
Definition (Non-interference – Version 2)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ if for all low input values $in_l$ there exist low output values $r$ such that for all high input values $in_h$ if we assign the values $in_l$ to the program variables $low$ and $in_h$ to the program variables $high$ then after execution of $P$ the values of $low$ are $r$.

$$\forall in_l \exists r \forall in_h (\{low := in_l \ | \ high := in_h\}[P]low = r)$$

- **Problem**: not suitable for automatic verification $\rightsquigarrow$ instantiation of existential quantifier difficult.
Formalisation in JavaDL – Alternating Quantifiers

Definition (Non-interference – Version 2)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ $\iff$

for all low input values $in_l$ there exist low output values $r$ such that for all

high input values $in_h$ if we assign the values $in_l$ to the program variables $low$ and $in_h$ to the program variables $high$ then after execution of $P$ the values of $low$ are $r$.

$$\forall in_l \exists r \forall in_h (\{low := in_l \parallel high := in_h\}[P]low = r)$$

- **Problem:** not suitable for automatic verification $\leadsto$ instantiation of existential quantifier difficult.
Formalisation in JavaDL – Self Composition

Definition (Non-interference – Version 3)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ if running two instances of $P$ on the same low values but on arbitrary high values result in low variables which have the same values.

\[
\forall in_l \forall in^1_h \forall in^2_h \forall out^1_l \forall out^2_l \ ( \\
\{ low := in_l \ || \ high := in^1_h \} [P] out^1_l = low \\
\land \ { low := in_l \ || \ high := in^2_h \} [P] out^2_l = low \\
\to out^1_l = out^2_l 
\)
Formalisation in JavaDL – Self Composition

Definition (Non-interference – Version 3)

For program $P$ the high variables $\text{high}$ do not interfere with the low variables $\text{low}$ if running two instances of $P$ on the same low values but on arbitrary high values result in low variables which have the same values.

$$\forall \text{in}_1 \forall \text{in}^1_h \forall \text{in}^2_h \forall \text{out}_1^1 \forall \text{out}_1^2 \left( \begin{array}{l}
\{\text{low} := \text{in}_1 \parallel \text{high} := \text{in}^1_h \} [P] \text{out}_1^1 = \text{low} \\
\land \{\text{low} := \text{in}_1 \parallel \text{high} := \text{in}^2_h \} [P] \text{out}_1^2 = \text{low} \\
\rightarrow \text{out}_1^1 = \text{out}_1^2
\end{array} \right)$$
Definition (Non-interference – Version 3)

For program $P$ the high variables $high$ do not interfere with the low variables $low$ if running two instances of $P$ on the same low values but on arbitrary high values result in low variables which have the same values.

$$
\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \ ( \\
\{ low := in_l \ || \ high := in_h^1 \} [P] out_l^1 = low \\
\land \{ low := in_l \ || \ high := in_h^2 \} [P] out_l^2 = low \\
\rightarrow out_l^1 = out_l^2 
)$$
Definition (Non-interference – Version 3)

For program \( P \) the high variables \( \text{high} \) do not interfere with the low variables \( \text{low} \) \iff running two instances of \( P \) on the same low values but on arbitrary high values result in low variables which have the same values.

\[
\forall \text{in}_l \forall \text{in}_h^1 \forall \text{in}_h^2 \forall \text{out}_l^1 \forall \text{out}_l^2 \left( \\
\{ \text{low} := \text{in}_l \parallel \text{high} := \text{in}_h^1 \} [P] \text{out}_l^1 = \text{low} \\
\land \{ \text{low} := \text{in}_l \parallel \text{high} := \text{in}_h^2 \} [P] \text{out}_l^2 = \text{low} \\
\rightarrow \text{out}_l^1 = \text{out}_l^2 \right)
\]
Program Variables vs. Heaps

■ **So far:** definitions and formalisations talk about program variables.

\[
\forall \text{in}_l \forall \text{in}^1_h \forall \text{in}^2_h \forall \text{out}^1_l \forall \text{out}^2_l \ ( \\
\{ \text{low} := \text{in}_l || \text{high} := \text{in}^1_h \}[P] \text{out}^1_l = \text{low} \\
\land \{ \text{low} := \text{in}_l || \text{high} := \text{in}^2_h \}[P] \text{out}^2_l = \text{low} \\
\rightarrow \text{out}^1_l = \text{out}^2_l 
\]

■ **Problem:** formulas contain for every program variable a quantifier and an update.

■ **Better:** talking about heaps and sets of heap locations \(\leadsto\) all heap locations can be updated at once.
Self Composition with Heaps

Definition (Non-interference for Heaps)

A program $P$ satisfies non-interference for a partition of the heap locations in low security locations $\text{low}$ and high security locations $\text{high}$

$$
\forall \text{Heap } h^1_{\text{in}}, h^2_{\text{in}}, h^1_{\text{out}}, h^2_{\text{out}} ( \\
\{\text{heap} := h^1_{\text{in}}\}[P]h^1_{\text{out}} = \text{heap} \\
\land \{\text{heap} := h^2_{\text{in}}\}[P]h^2_{\text{out}} = \text{heap} \\
\rightarrow (h^1_{\text{in}} \sim_{\text{low}} h^2_{\text{in}} \rightarrow h^1_{\text{out}} \sim_{\text{low}} h^2_{\text{out}}) )
$$

$$
h^1 \sim_{\text{low}} h^2 \iff \forall \text{Object } o \forall \text{Field } f ( \\
(o, f) \in \text{low} \rightarrow \{\text{heap} := h^1\}o.f = \{\text{heap} := h^2\}o.f )
$$
Self Composition with Heaps

Definition (Non-interference for Heaps)

A program $P$ satisfies non-interference for a partition of the heap locations in low security locations $\text{low}$ and high security locations $\text{high}$

$\iff$

running two instances of $P$ on arbitrary heaps result in heaps which agree on the low-locations $\text{low}$ if the input heaps agree on $\text{low}$.

$$\forall \text{Heap } h^1_{in}, h^2_{in}, h^1_{out}, h^2_{out} \ ($$

$$\{\text{heap} := h^1_{in}\}[P]h^1_{out} = \text{heap}$$

$$\land \ {\{\text{heap} := h^2_{in}\}[P]h^2_{out} = \text{heap}}$$

$$\rightarrow (h^1_{in} \sim_{\text{low}} h^2_{in} \rightarrow h^1_{out} \sim_{\text{low}} h^2_{out}) \ )$$

$$h^1 \sim_{\text{low}} h^2 \iff \forall \text{Object } o \forall \text{Field } f \ ($$

$$(o, f) \in \text{low} \rightarrow \{\text{heap} := h^1\}o.f = \{\text{heap} := h^2\}o.f \ )$$
Self Composition with Heaps

Definition (Non-interference for Heaps)

A program $P$ satisfies non-interference for a partition of the heap locations in low security locations $low$ and high security locations $high$ if the input heaps agree on $low$.

$$\forall \text{Heap } h_1^{\text{in}}, h_2^{\text{in}}, h_1^{\text{out}}, h_2^{\text{out}} \left( \right.$$  

$$\{\text{heap} := h_1^{\text{in}}\}[P]h_1^{\text{out}} = \text{heap} \wedge \{\text{heap} := h_2^{\text{in}}\}[P]h_2^{\text{out}} = \text{heap} \rightarrow (h_1^{\text{in}} \sim_{low} h_2^{\text{in}} \rightarrow h_1^{\text{out}} \sim_{low} h_2^{\text{out}} ) \right)$$

$$h_1 \sim_{low} h_2 \leftrightarrow \forall \text{Object } o \forall \text{ Field } f \left( \right.$$  

$$(o, f) \in low \rightarrow \{\text{heap} := h_1\}o.f = \{\text{heap} := h_2\}o.f \right)$$
Self Composition with Heaps

Definition (Non-interference for Heaps)

A program $P$ satisfies non-interference for a partition of the heap locations in low security locations $low$ and high security locations $high$ if:

$$\iff$$

running two instances of $P$ on arbitrary heaps result in heaps which agree on the low-locations $low$ if the input heaps agree on $low$.

$$
\forall \text{Heap } h_1^{in}, h_2^{in}, h_1^{out}, h_2^{out} ( \\
\{heap := h_1^{in}\}[P]h_1^{out} = heap \\
\land \{heap := h_2^{in}\}[P]h_2^{out} = heap \\
\rightarrow (h_1^{in} \sim_{low} h_2^{in} \rightarrow h_1^{out} \sim_{low} h_2^{out}) )
$$

$$h_1^{\sim_{low}} h_2^{\sim_{low}} \iff \forall \text{Object } o \forall \text{Field } f ( \\
(o, f) \in low \rightarrow \{heap := h_1^{\sim\sim}\}o.f = \{heap := h_2^{\sim\sim}\}o.f )$$
Self Composition with Heaps

Definition (Non-interference for Heaps)

A program $P$ satisfies non-interference for a partition of the heap locations in low security locations $\text{low}$ and high security locations $\text{high}$

$\Leftrightarrow$

running two instances of $P$ on arbitrary heaps result in heaps which agree on the low-locations $\text{low}$ if the input heaps agree on $\text{low}$.

\[
\forall \text{Heap } h_1^{\text{in}}, h_2^{\text{in}}, h_1^{\text{out}}, h_2^{\text{out}} ( \\
\{ \text{heap} := h_1^{\text{in}} \} [P] h_1^{\text{out}} = \text{heap} \\
\land \{ \text{heap} := h_2^{\text{in}} \} [P] h_2^{\text{out}} = \text{heap} \\
\rightarrow (h_1^{\text{in}} \sim_{\text{low}} h_2^{\text{in}} \rightarrow h_1^{\text{out}} \sim_{\text{low}} h_2^{\text{out}}) )
\]

$h_1^{\text{low}} \sim_{\text{low}} h_2^{\text{high}} \Leftrightarrow \forall \text{Object } o \forall \text{Field } f ( \\
(o, f) \in \text{low} \rightarrow \{ \text{heap} := h_1^{\text{high}} \} o.f = \{ \text{heap} := h_2^{\text{high}} \} o.f )
Declassification

- Declassification of terms.
- **Semantics:** the evaluation of the term in the pre-state of the program invocation is allowed to leak.

\[
\forall \text{Heap } h_1^{in}, h_2^{in}, h_1^{out}, h_2^{out} ( \\
\{heap := h_1^{in}\}[P]h_1^{out} = heap \\
\land \{heap := h_2^{in}\}[P]h_2^{out} = heap \\
\rightarrow (h_1^{in} \sim_{low} h_2^{in} \land \{heap := h_1^{in}\}term = \{heap := h_2^{in}\}term \\
\rightarrow h_1^{out} \sim_{low} h_2^{out}) )
\]
Termination-Sensitivity

The formalisation can be made termination sensitive:

\[ \forall \text{Heap } h^1_{in}, h^2_{in}, h^1_{out}, h^2_{out} ( \]
\[ \{ \text{heap} := h^1_{in} \}[P]h^1_{out} = \text{heap} \]
\[ \wedge \{ \text{heap} := h^2_{in} \}[P]h^2_{out} = \text{heap} \]
\[ \rightarrow (h^1_{in} \sim_{low} h^2_{in} \rightarrow h^1_{out} \sim_{low} h^2_{out}) \]
\[ \vee \{ \text{heap} := h^1_{in} \}[P]\text{false} \wedge \{ \text{heap} := h^2_{in} \}[P]\text{false} ) \]
Part IV

Verifying Information Flow Properties

Non-Interference
- Definition
- Reformulation and Formalisation – Alternating Quantifiers
- Reformulation and Formalisation – Self Composition
- Declassification

JML Non-Interference Specifications
- Views and Security Policies
- Non-Interference in JML
Goal

Modular program-level specification of information flow properties for Java programs.

- Extension of JML for information flow specification.
- Seamless integration of functional and information flow specifications.
- Suitable for the specification of interfaces.
- Should not transgress the principle of information hiding.
- Should not transgress the separation of specification and implementation.
- Precise (semantic) declassification.
Views

Users → Views → Heap Locations:

- **User** has some **view** on a system.
- A **view** observes a set of **heap locations**.

**Not central:** individual user ⊸ concentration on views.
Views

- **Security Requirement:** No (user of a) view may learn more about the initial program-state through the execution of a program than the values of the heap locations belonging to the view.

- Requirement implies security policy (and security lattice).

- **Information flow property:** language-based non-interference (with declassification)
Implicit Security Policy

Views define an implicit security policy:

- Information may flow freely between $x \leftrightarrow y$.
- Information may flow $y \rightarrow z$, but $y \not\leftrightarrow z$.
Implicit Security Policy

Views define an implicit security policy:

- Information may flow freely between $x \leftrightarrow y$.
- Information may flow $y \rightarrow z$, but $y \nleftrightarrow z$. 
Implicit Security Policy

Views define an implicit security policy:

- Information may flow freely between $x \leftrightarrow y$.
- Information may flow $y \rightarrow z$, but $y \not\leftrightarrow z$. 
Implicit Security Policy

Views define an implicit security policy:

- Information may flow $y \rightarrow x$, but $y \leftrightarrow x$.
- Information may flow $y \rightarrow z$, but $y \leftrightarrow z$.
- Information may not flow $x \leftrightarrow z$.
Implicit Security Policy

Views define an implicit security policy:

- Information may flow $y \rightarrow x$, but $y \leftarrow x$.
- Information may flow $y \rightarrow z$, but $y \leftarrow z$.
- Information may not flow $x \leftrightarrow z$.
Implicit Security Policy

Views define an implicit security policy:

- Information may flow $y \rightarrow x$, but $y \leftrightarrow x$.
- Information may flow $y \rightarrow z$, but $y \leftrightarrow z$.
- Information may not flow $x \leftrightarrow z$. 
Dynamic Views

Feature:

- During program execution the set of heap locations belonging to a view may change.
- Example: view → last element of a linked list
Views in the pre-state:

Heap

Views in the post-state:

Heap

The value of $x$ in the post-state has to be a constant.
Security Policy for Dynamic Views

Views in the pre-state:

Heap

$x$

Views in the post-state:

Heap

View 1

$x$

The value of $x$ in the post-state has to be a constant.
Security Policy for Dynamic Views

Views in the pre-state:

Heap

View 1

Views in the post-state:

Heap

Nothing has to be checked.
Security Policy for Dynamic Views

Views in the pre-state:

Views in the post-state:

Heap

View 1

Heap

y

Nothing has to be checked.
Security Policy for Dynamic Views

Views in the pre-state:

- Heap
- View 1

Information may flow:
- $y \rightarrow y$
- $y \rightarrow x$

Views in the post-state:

- Heap
- View 1

Information may not flow:
- $x \not\rightarrow y$
- $x \not\rightarrow x$
Security Policy for Dynamic Views

Views in the pre-state:

Heap

View 1

Information may flow

- \( y \rightarrow y \)
- \( y \rightarrow x \)

Views in the post-state:

Heap

View 1

Information may not flow

- \( x \rightarrow y \)
- \( x \rightarrow x \)
Security Policy for Dynamic Views

Views in the pre-state:

Heap

View 1

View 2

Views in the post-state:

Heap

View 2

View 1

Information may flow:

- \( y \rightarrow y \)
- \( y \rightarrow x \)
- \( z \rightarrow z \)
- \( y \rightarrow z \)
- \( x \rightarrow z \)

Information may not flow:

- \( x \rightarrow y \)
- \( x \rightarrow x \)
- \( z \rightarrow y \)
- \( z \rightarrow x \)

Non-Interference
Security Policy for Dynamic Views

Views in the pre-state:

Heap

View 1

View 2

y

x

Views in the post-state:

Heap

View 2

View 1

y

x

Information may flow

- y → y
- y → x
- z → z
- y → z
- x → z

Information may not flow

- x → y
- x → x
- z → y
- z → x
Security Policy for Dynamic Views

Views in the pre-state:

Heap

View 1

View 2

y

x

z

Views in the post-state:

Heap

View 2

View 1

y

x

z

Information may flow

- y → y
- y → x
- z → z
- y → z
- x → z

Information may not flow

- x → y
- x → x
- z → y
- z → x
Example: Banking System – Use-Case Diagram

BankSystem

- viewBalance
- drawMoney
- createAccount

Bank-Customer

Non-Interference

B. Beckert, V. Klebanov, C. Scheben, P. H. Schmitt – KeY Tutorial
Example: Banking System – Class Diagram

```java
Bank
- instance
- currentNumAccounts : int
+ Bank(int maxAccounts)
+ getInstance() : Bank
+ login(String user, String password) : boolean

UserAccount
- user : String
- password : String
- currentNumBankAccounts : int

BankAccount
- balance : int
- id : int
```

Non-Interference Specifications

B. Beckert, V. Klebanov, C. Scheben, P. H. Schmitt – KeY Tutorial
Example: Banking System – Object Diagram

Instance: Bank

UserAccount-1: UserAccount
- int currentNumBankAccounts = 2
- String password = abc
- String user = Herbert

BankAccount-1-1: BankAccount
- balance = 147
- id = 1

BankAccount-1-2: BankAccount
- balance = 23
- id = 1

UserAccount-2: UserAccount
- int currentNumBankAccounts = 1
- String password = huhu
- String user = Bert

BankAccount-2-1: BankAccount
- balance = 5678
- id = 3

UserAccount-3: UserAccount
- int currentNumBankAccounts = 0
- String password = cool
- String user = Anton
Example: Banking System – Views

Bank
-\begin{itemize}
  \item \text{instance} \\
  \item \text{-currentNumAccounts} : \text{int}
\end{itemize}

+\text{Bank}(\text{int maxAccounts}) \\
+\text{getInstance}() : \text{Bank} \\
+\text{login}(\text{String user, String password}) : \text{boolean}

\text{Bank-Employee-View} = \text{-CN}

UserAccount
-\begin{itemize}
  \item \text{user} : \text{String} \\
  \item \text{-password} : \text{String} \\
  \item \text{-currentNumBankAccounts} : \text{int}
\end{itemize}

\begin{align*}
\text{Bank-Customer-View} = & \text{userAccounts}[*] \\
\cup & \text{userAccounts}[*].\text{user} \\
\cup & \text{userAccounts}[*].\text{currentNumBankAccounts} \\
\cup & \text{userAccounts}[*].\text{bankAccounts}[*] \\
\cup & \text{userAccounts}[*].\text{bankAccounts}[*].\text{balance} \\
\cup & \text{userAccounts}[*].\text{bankAccounts}[*].\text{id}
\end{align*}

Bank-Employee-View = userAccounts[*]
Example: Banking System – Views

Bank

- instance
- currentNumAccounts : int

+ Bank(int maxAccounts)
+ getInstance() : Bank
+ login(String user, String password) : boolean

Bank-Customer-View = user
∪ password
∪ currentNumAccounts
∪ bankAccounts[∗].balance
∪ bankAccounts[∗].id

Bank-Employee-View =

UserAccount

- user : String
- password : String
- currentNumBankAccounts : int

Bank-Customer-View =

Bank-Employee-View =

BankAccount

- balance : int
- id : int

Non-Interference
Non-Interference Specifications in JML

Specification as method contracts:

- Specification of the set of views which define the implicit security policy for the method (*respects-clause*).
- Specification of the security level of the parameters (*parameter_dep-clause*).

```java
public int low;
private int high;

void m(int param) {
    low = param;
}
```

- Specification of intentional information leakage (*declassify-clause*).
Non-Interference Specifications in JML

```java
public int low;
private int high;

/*@ respects {low};
@*/

public void m(int param) {
    low = param;
}
```

- **Views in JML:** expressions of type `\locset`.
- **Views can be named:** definition of model fields.
- Approach complies to the principle of information hiding!
Non-Interference Specifications in JML

```java
public int low;
private int high;

/*@ respects {low};
  @ parameter_dep {low};
  @*/
public void m(int param) {
  low = param;
}
```

- **Views in JML**: expressions of type `locset`.
- **Views can be named**: definition of model fields.
- **Approach complies to the principle of information hiding!**
Example: Password Checker

class PasswordFile {
    private int[] names, passwords;
    //@ invariant names.length == passwords.length;

    public boolean check(int user, int password) {
        for (int i = 0; i < names.length; i++) {
            if (names[i] == user && passwords[i] == password) {
                return true;
            }
        }
        return false;
    }
}
Example: Password Checker

class PasswordFile {
    private int[] names, passwords;
    //@ invariant names.length == passwords.length;

    public boolean check(int user, int password) {
        for (int i = 0; i < names.length; i++) {
            if (names[i] == user && passwords[i] == password) {
                return true;
            }
        }
        return false;
    }
}

Non-Interference

B. Beckert, V. Klebanov, C. Scheben, P. H. Schmitt – KeY Tutorial
Example: Password Checker

class PasswordFile {
    private int[] names, passwords;
    //@ invariant names.length == passwords.length;

    public boolean check(int user, int password) {
        for (int i = 0; i < names.length; i++) {
            if (names[i] == user && passwords[i] == password) {
                return true;
            }
        }
        return false;
    }
}

Heap
names
passwords
Example: Password Checker

class PasswordFile {
    private int[] names, passwords;
    // @ invariant names.length == passwords.length;

    public boolean check(int user, int password) {
        for (int i = 0; i < names.length; i++) {
            if (names[i] == user && passwords[i] == password) {
                return true;
            }
        }
        return false;
    }
}

Heap
unknown locations

names
passwords
/*@ respects
@  {names [0]},
@  {names [0], passwords [0]},
@  {names [1]},
@  {names [1], passwords [1]},
@  ...
@*/

public boolean check(int user, int password) { ...
/*@ model int userIndex;
    @ represents userIndex \ such \ that 
    @    0 <= userIndex 
    @    && userIndex < names.length;
    @*/

/*@ respects
    @    {names[userIndex]},
    @    {names[userIndex], passwords[userIndex]};
    @*/

public boolean check(int user, int password) { ...
/*@ model int userIndex; 
@ represents userIndex \ such\_that 
@ 0 <= userIndex 
@ && userIndex < names.length; 
@
@ model \locset nameUser; 
@ represents nameUser = {names[userIndex]}; 
@
@ model \locset loginUser; 
@ represents loginUser = 
@ {nameUser, passwords[userIndex]}; 
@*/
//@ respects nameUser, loginUser;
public boolean check(int user, int password) { ...
Example: Password Checker

class PasswordFile {
    private int[] names, passwords;
    //@ invariant names.length == passwords.length;

    public boolean check(int user, int password) {
        for (int i = 0; i < names.length; i++) {
            if (names[i] == user &&
            passwords[i] == password) {
                return true;
            }
        }
        return false;
    }
}
Example: Password Checker

class PasswordFile {
    private int[] names, passwords;
    //@ invariant names.length == passwords.length;

    public boolean check(int user, int password) {
        for (int i = 0; i < names.length; i++) {
            if (names[i] == user && passwords[i] == password) {
                return true;
            }
        }
        return false;
    }
}
JML Specification Example

/*@ ... 
  @ model \locset anyUser;
  @*/

/*@ respects nameUser, loginUser;
  @ parameter_dep anyUser, anyUser: anyUser;
  @*/

public boolean check(int user, int password) { ...
Example: Password Checker

class PasswordFile {
    private int[] names, passwords;
    //@ invariant names.length == passwords.length;

    public boolean check(int user, int password) {
        for (int i = 0; i < names.length; i++) {
            if (names[i] == user &&
                passwords[i] == password) {
                return true;
            }
        }
        return false;
    }
    return false;
}
Declassification

Information is declassified in form of a term:

- Evaluation of the term in the pre-state of the method invocation is allowed to leak. *(what-axes)*

Restrictions:

- Leakage should be authorised by some view: leak only in case information can be computed by the authorising view. *(who-axes)*
- Flow should be restricted to some view: leak only to a specified view. *(who-axes)*
- Declassification bound to some condition: leak only if the condition evaluates to true in the pre-state of the method invocation. *(when-axes)*
/*@ normal_behavior
@ respects nameUser, loginUser;
@ parameter_dep anyUser;
@ declassify ( \exists int i;
@ 0 <= i && i < names.length;
@ names[i] == user
@ && passwords[i] == password
@ )
@ \from {names[\*], passwords[\*]}\n@ \to anyUser;
@*/

public boolean check(int user, int password) {
Declassification is part of the method contract → separation of specification and implementation.

Precise (semantic) declassification.

Low references on objects with high fields need to be allowed.

Static verification of data structures with elements of different security levels possible.
Part V

Wrap Up

14 Further Usage of Verification Technology

15 Directions of Current Research in KeY

16 Different Approaches
Part V

Wrap Up

14 Further Usage of Verification Technology

15 Directions of Current Research in KeY

16 Different Approaches
Verification performs deep *Program Analysis*

Information in (partial) proofs usable for other purposes
Verification Driven Test Generation

- Specification- and code-based approach
- Achieve strong hybrid coverage criteria
- Exploit strong correspondence: proof branches ↔ program execution paths
- Each leaf of (partial) proof branch contains constraint on inputs resulting in corresponding execution path
Verification Driven Test Generation

- Specification- and code-based approach
- Achieve strong hybrid coverage criteria
- Exploit strong correspondence:
  proof branches ↔ program execution paths
- Each leaf of (partial) proof branch contains constraint on inputs resulting in corresponding execution path
Verification Driven Test Generation

- Specification- and code-based approach
- Achieve strong hybrid coverage criteria
- Exploit strong correspondence: proof branches ↔ program execution paths
- Each leaf of (partial) proof branch contains constraint on inputs resulting in corresponding execution path
Verification Driven Test Generation

- Specification- and code-based approach
- Achieve strong hybrid coverage criteria
- Exploit strong correspondence: proof branches ↔ program execution paths
- Each leaf of (partial) proof branch contains constraint on inputs resulting in corresponding execution path
Part V
Wrap Up

14 Further Usage of Verification Technology

15 Directions of Current Research in KeY

16 Different Approaches
Part V
Wrap Up

14 Further Usage of Verification Technology

15 Directions of Current Research in KeY

16 Different Approaches
Directions of Current Research in KeY

Extending the scope of verification

- Concurrency and distribution
- Information-flow properties
- Floating-point arithmetic
- Safety-Critical Java (different memory model)
- Resource bounds (memory, time)
- Product lines
- Compiling verifier
Extending the scope of verification

- Concurrency and distribution
- Information-flow properties
- Floating-point arithmetic
- Safety-Critical Java (different memory model)
- Resource bounds (memory, time)
- Product lines
- Compiling verifier
Directions of Current Research in KeY

Extending the scope of verification

- Concurrency and distribution
- Information-flow properties
- Floating-point arithmetic
- Safety-Critical Java (different memory model)
- Resource bounds (memory, time)
- Product lines
- Compiling verifier
Directions of Current Research in KeY

Extending the scope of verification

- Concurrency and distribution
- Information-flow properties
- Floating-point arithmetic
- Safety-Critical Java (different memory model)
- Resource bounds (memory, time)
- Product lines
- Compiling verifier
Directions of Current Research in KeY

Extending the scope of verification

- Concurrency and distribution
- Information-flow properties
- Floating-point arithmetic
- Safety-Critical Java (different memory model)
- Resource bounds (memory, time)
- Product lines
- Compiling verifier
Extending the scope of verification

- Concurrency and distribution
- Information-flow properties
- Floating-point arithmetic
- Safety-Critical Java (different memory model)
- Resource bounds (memory, time)
- Product lines
- Compiling verifier
Extending the scope of verification

- Concurrency and distribution
- Information-flow properties
- Floating-point arithmetic
- Safety-Critical Java (different memory model)
- Resource bounds (memory, time)
- Product lines
- Compiling verifier
Modelling and specification

- Modular specification of heap structures
dynamic frames, abstract data types
- Compositional models of concurrency and distribution
- Refinement
- Support for the specification process
Modelling and specification

- Modular specification of heap structures
dynamic frames, abstract data types

- Compositional models of concurrency and distribution

- Refinement

- Support for the specification process
Modelling and specification

- Modular specification of heap structures
dynamic frames, abstract data types
- Compositional models of concurrency and distribution
- Refinement
- Support for the specification process
Directions of Current Research in KeY

Modelling and specification

- Modular specification of heap structures
dynamic frames, abstract data types
- Compositional models of concurrency and distribution
- Refinement
- Support for the specification process
Part V
Wrap Up

14 Further Usage of Verification Technology

15 Directions of Current Research in KeY

16 Different Approaches
Part V

Wrap Up

14 Further Usage of Verification Technology

15 Directions of Current Research in KeY

16 Different Approaches
Different Approaches to Software Verification

General Purpose Systems
- General purpose
- Elaborate support for theories, abstract data types
- Target object level and meta level

Verification systems for OO languages
- Special purpose, tuned for that
- Close to programming language
- Integration into software development process/tools

Combining these advantages remains a challenge
Different Approaches to Software Verification

General Purpose Systems

- General purpose
- Elaborate support for theories, abstract data types
- Target object level and meta level

Verification systems for OO languages

- Special purpose, tuned for that
- Close to programming language
- Integration into software development process/tools

Combining these advantages remains a challenge
Different Approaches to Software Verification

General Purpose Systems

- General purpose
- Elaborate support for theories, abstract data types
- Target object level and meta level

Verification systems for OO languages

- Special purpose, tuned for that
- Close to programming language
- Integration into software development process/tools

Combining these advantages remains a challenge
Different Approaches to Software Verification

General Purpose Systems
- General purpose
- Elaborate support for theories, abstract data types
- Target object level and meta level

Verification systems for OO languages
- Special purpose, tuned for that
- Close to programming language
- Integration into software development process/tools

Combining these advantages remains a challenge
Different Approaches to Software Verification

General Purpose Systems

- General purpose
- Elaborate support for theories, abstract data types
- Target object level *and* meta level

Verification systems for OO languages

- Special purpose, tuned for that
- Close to programming language
- Integration into software development process/tools

Combining these advantages remains a challenge
Different Approaches to Software Verification

General Purpose Systems

- General purpose
- Elaborate support for theories, abstract data types
- Target object level and meta level

Verification systems for OO languages

- Special purpose, tuned for that
- Close to programming language
- Integration into software development process/tools

Combining these advantages remains a challenge
Different Approaches to Software Verification

General Purpose Systems

- General purpose
- Elaborate support for theories, abstract data types
- Target object level \textit{and} meta level

Verification systems for OO languages

- Special purpose, tuned for that
- Close to programming language
- Integration into software development process/tools

Combining these advantages remains a challenge
Different Approaches to Software Verification

General Purpose Systems
- General purpose
- Elaborate support for theories, abstract data types
- Target object level \textit{and} meta level

Verification systems for OO languages
- Special purpose, tuned for that
- Close to programming language
- Integration into software development process/tools

Combining these advantages remains a challenge
### Different Approaches to Software Verification

#### General Purpose Systems
- General purpose
- Elaborate support for theories, abstract data types
- Target object level *and* meta level

#### Verification systems for OO languages
- Special purpose, tuned for that
- Close to programming language
- Integration into software development process/tools

**Combining these advantages remains a challenge**
THE END

(for now)