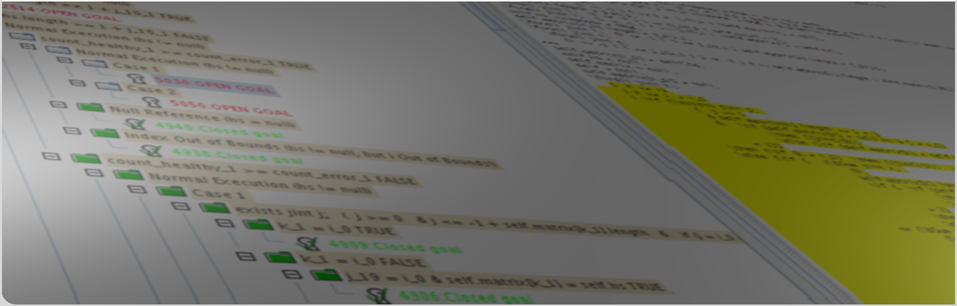
Program Verification with the KeY System

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

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

Karlsruhe Institute of Technology

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

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

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

AI Systems

LNAI 4334

Bernhard Beckert
Reiner Hähnle
Peter H. Schmitt (Eds.)

Verification of Object-Oriented Software

The KeY Approach

Foreword by K. Rustan M. Leino

Springer

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.

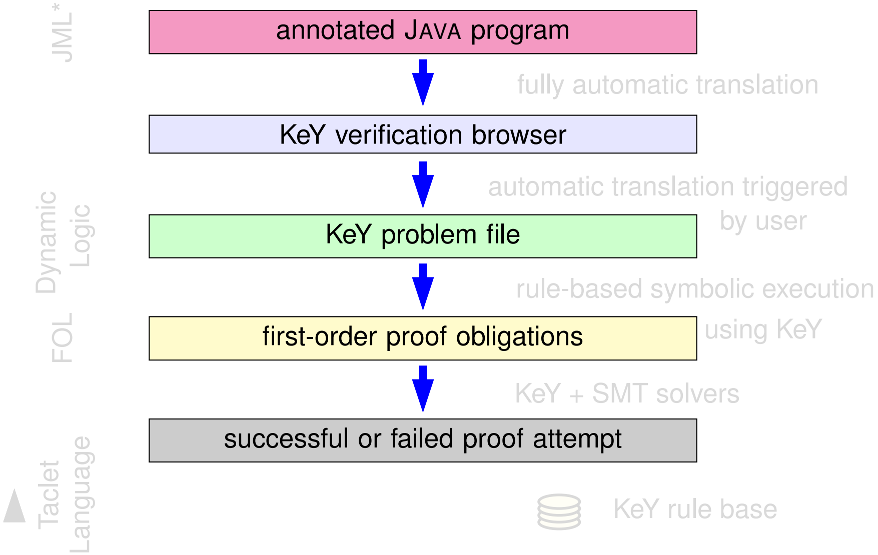
Specific Features of the KeY Approach

- 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



JML*

annotated JAVA program

fully automatic translation

KeY verification browser

automatic translation triggered by user

Dynamic Logic

KeY problem file

rule-based symbolic execution using KeY

FOL

first-order proof obligations

KeY + SMT solvers

successful or failed proof attempt

Taclet Language

KeY rule base

Workflow

JML*

annotated JAVA program

fully automatic translation

KeY verification browser

automatic translation triggered by user

Dynamic Logic

KeY problem file

rule-based symbolic execution using KeY

FOL

first-order proof obligations

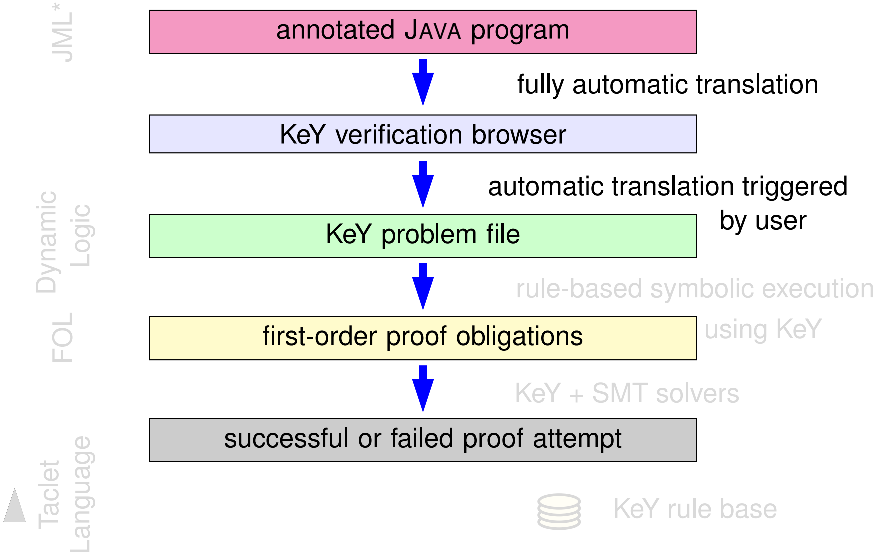
KeY + SMT solvers

successful or failed proof attempt

Taclet Language

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

JML*

annotated JAVA program

↓ fully automatic translation

KeY verification browser

↓ automatic translation triggered by user

Dynamic Logic

KeY problem file

↓ rule-based symbolic execution using KeY

FOL

first-order proof obligations

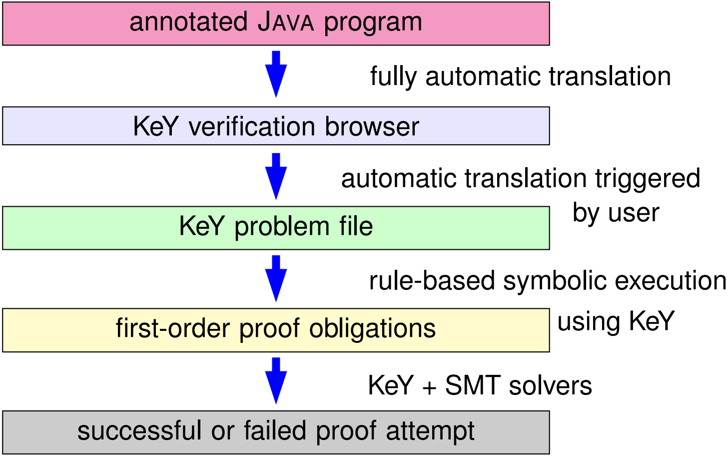
↓ KeY + SMT solvers

successful or failed proof attempt

Taclet Language

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

JML*

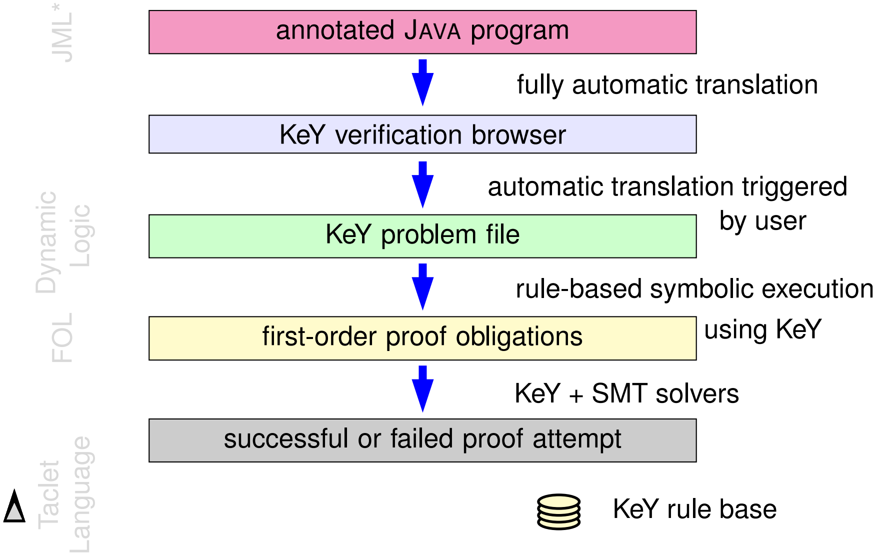
Dynamic Logic

FOL

Taclet Language

KeY rule base

Workflow



JML *

annotated JAVA program

fully automatic translation

KeY verification browser

automatic translation triggered by user

Dynamic Logic

KeY problem file

rule-based symbolic execution using KeY

FOL

first-order proof obligations

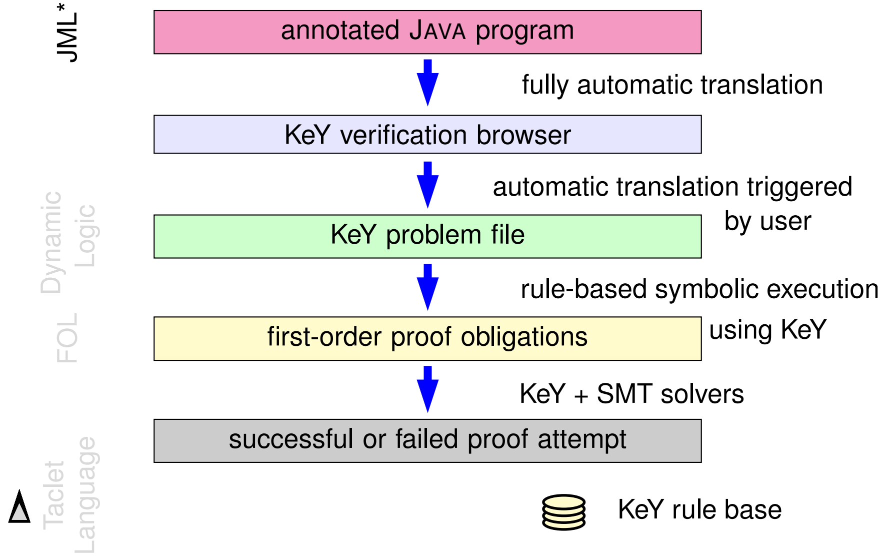
KeY + SMT solvers

successful or failed proof attempt

Taclet Language

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

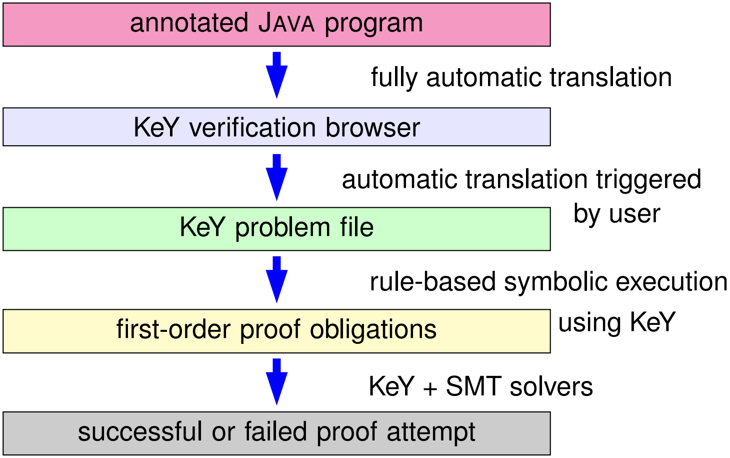
Workflow

JML*

Dynamic Logic

FOL

Taclet Language



annotated JAVA program

fully automatic translation

KeY verification browser

automatic translation triggered by user

KeY problem file

rule-based symbolic execution using KeY

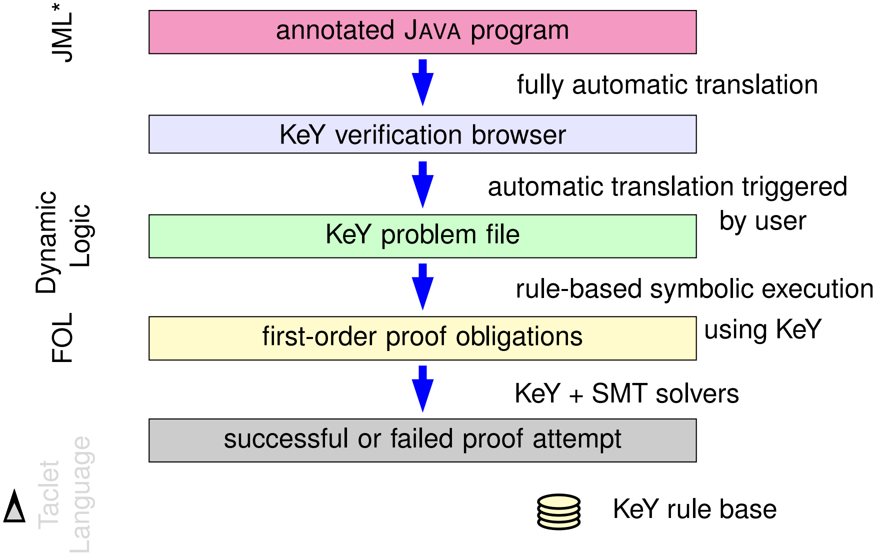
first-order proof obligations

KeY + SMT solvers

successful or failed proof attempt

KeY rule base

Workflow



JML*

annotated JAVA program

fully automatic translation

KeY verification browser

automatic translation triggered by user

Dynamic Logic

KeY problem file

rule-based symbolic execution using KeY

FOL

first-order proof obligations

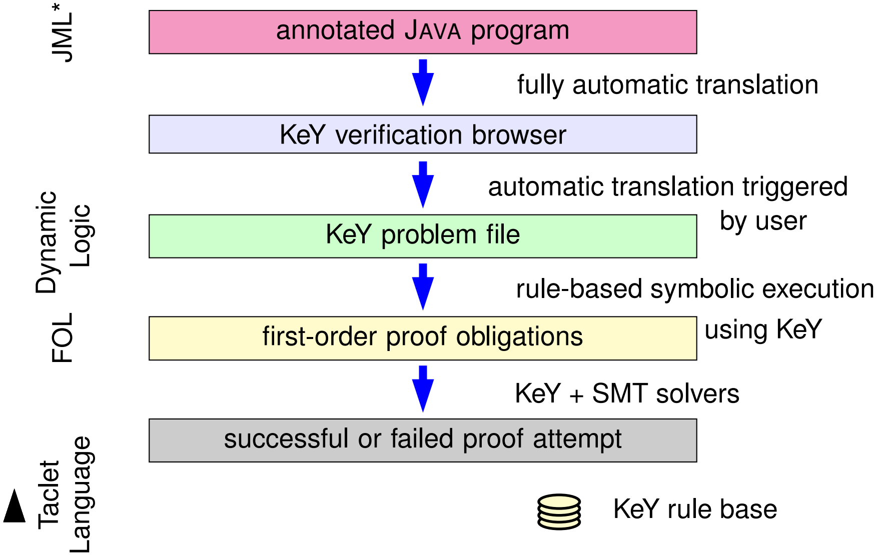
KeY + SMT solvers

successful or failed proof attempt

△ Taclet Language

KeY rule base

Workflow



JML*

annotated Java program

fully automatic translation

KeY verification browser

automatic translation triggered by user

Dynamic Logic

KeY problem file

rule-based symbolic execution using KeY

FOL

first-order proof obligations

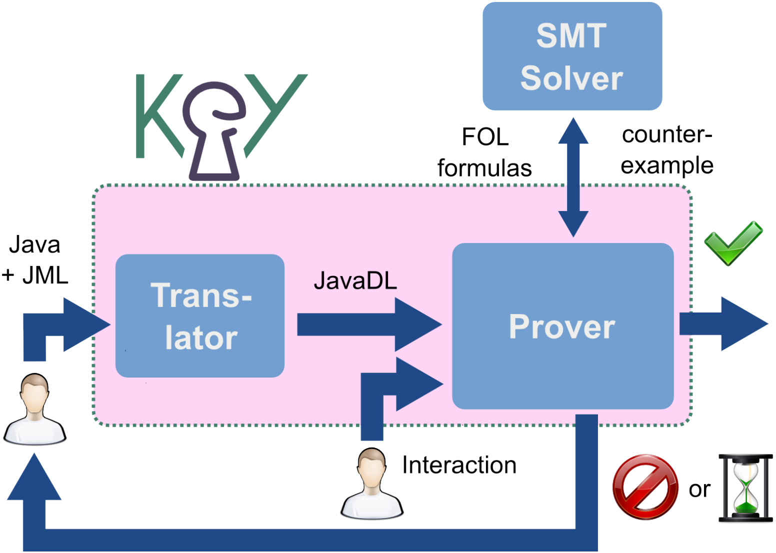
KeY + SMT solvers

successful or failed proof attempt

Taclet Language

KeY rule base

KeY Verification Process



Advertisment

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 –

Part II

The Java Modeling Language
– By Example –

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

Method Contracts

JML Spec of Postincrement

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

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

Method Contracts

JML Spec of Postincrement

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

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

JML annotation occur as special comments in source programm

Method Contracts

JML Spec of Postincrement

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

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

precondition

Method Contracts

JML Spec of Postincrement

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

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

postcondition

Method Contracts

JML Spec of Postincrement

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

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

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

Method Contracts

JML Spec of Postincrement

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

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

All side-effect-free Java expressions allowed

Method Contracts

JML Spec of Postincrement

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

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

Plus special operators (\ . . .)

Method Contracts

Non-null default

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

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

Method Contracts

Non-null default

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

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

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

Method Contracts

Non-null default

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

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

The default is overwritten by the keyword `nullable`

Method Contracts

Non-null default

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

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

In this case the precondition has to be adapted accordingly

Method Contracts

Specification of `commonEntry`

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

Method Contracts

Specification of `commonEntry`

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

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

Method Contracts

Specification of `commonEntry`

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

Method `commonEntry` looks for an index within the bounds

Method Contracts

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`

```
@ ...
@   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

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

Part II

The Java Modeling Language
– By Example –

Part II

The Java Modeling Language
– By Example –

Loop Invariants

Loop Invariant for `commonEntry`

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

Loop Invariants

Loop Invariant for `commonEntry`

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

The loop invariant

Loop Invariants

Loop Invariant for `commonEntry`

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

The loop invariant is valid before entering the loop since

Loop Invariants

Loop Invariant for `commonEntry`

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

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

Loop Invariants

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

plus

```
\result = k
```

plus reason for termination of the loop

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

imply the postconditions

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

and

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

Using a Loop Invariant

On termination of the loop
the invariant

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

plus

```
\result = k
```

plus reason for termination of the loop

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

imply the postconditions

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

and

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

Loop Termination

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

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

Loop Termination

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

The loop variant

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

Loop Termination

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

The loop variant

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

Loop Termination

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

The loop variant

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

Loop Termination

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

The loop variant

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

Part II

The Java Modeling Language
– By Example –

Part II

The Java Modeling Language
– By Example –

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

Assignable Clauses

```
/*@ 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

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

At most the locations in the assignable clause may be changed

Assignable Clauses

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

Everything else must remain unchanged

Assignable Clauses

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

Local variables need not be included

Assignable Clauses

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

Assignable clauses are evaluated in the prestate

Part II

The Java Modeling Language
– By Example –

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

Part II

The Java Modeling Language
– By Example –

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

Using Contracts

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

Using Contracts

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

Method `rearrange`

Using Contracts

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

Method `rearrange` uses methods `commonEntry`

Using Contracts

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

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

Using Contracts

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

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

Using Contracts

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

Key to scalability

Part II

The Java Modeling Language
– By Example –

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

Part II

The Java Modeling Language
– By Example –

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

Model Fields

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

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

Model Fields

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

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

model fields are only for specification

Model Fields

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

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

`\seq` is an abstract data type

Model Fields

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

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

`represents` clauses fix the semantics of model fields

Model Fields

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

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

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

Model Fields

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

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

Only additional postcondition show here

Model Fields

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

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

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

Model Fields

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

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

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

Model Fields

```
/*@ 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

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)
- $<p>\,F$: 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)
- `<`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)
- `<`p`>` 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 ψ `->` `[`α`]` ϕ

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

$(\texttt{balance} >= c \ \& \ \texttt{amount} > 0) \ -\!\!>$
$\texttt{<charge(amount);>} \ \texttt{balance} > c$

$\texttt{<x = 1;>}([\texttt{while (true) \{\}}]\, \mathit{false})$

- Program formulas can appear nested

$\texttt{\textbackslash forall}\ \mathit{int}\ \mathit{val};\ ((\texttt{<p>}\,\texttt{x} \doteq \mathit{val}) \ <\!\!-\!\!> \ (\texttt{<q>}\,\texttt{x} \doteq \mathit{val}))$

- p, q equivalent relative to computation state restricted to `x`

Dynamic Logic Example Formulas

$(\texttt{balance} >= c \ \& \ \texttt{amount} > 0) \ {->}$
$\texttt{<charge(amount);>} \ \texttt{balance} > c$

$\texttt{<x = 1;>}([\texttt{while (true) \{\}}]\, \mathit{false})$

- Program formulas can appear nested

$\backslash\texttt{forall}\ \mathit{int}\ \mathit{val};\ ((\texttt{<p>}\,\mathrm{x} \doteq \mathit{val})\ \ {<->}\ \ (\texttt{<q>}\,\mathrm{x} \doteq \mathit{val}))$

- p, q equivalent relative to computation state restricted to x

Dynamic Logic Example Formulas

$(\texttt{balance} >= c\ \&\ \texttt{amount} > 0)$ -> `<charge(amount);>` $\texttt{balance} > c$

`<x = 1;>` $($`[while (true) {}]` $\mathit{false})$

- Program formulas can appear nested

$\backslash\texttt{forall}\ \mathit{int}\ \mathit{val};\ ((\texttt{<p>}\,\texttt{x} \doteq \mathit{val})\ \texttt{<->}\ (\texttt{<q>}\,\texttt{x} \doteq \mathit{val}))$

- p, q equivalent relative to computation state restricted to x

Dynamic Logic Example Formulas

$(\texttt{balance} >= c \;\&\; \texttt{amount} > 0) \;{->}\; \texttt{<charge(amount);>}\; \texttt{balance} > c$

$\texttt{<x = 1;>}([\texttt{while (true) \{\}}]\,\mathit{false})$

- Program formulas can appear nested

$\backslash\texttt{forall}\; \mathit{int}\; \mathit{val}; ((\texttt{<p>}\,\texttt{x} \doteq \mathit{val}) \;{<->}\; (\texttt{<q>}\,\texttt{x} \doteq \mathit{val}))$

- p, q equivalent relative to computation state restricted to x

Dynamic Logic Example Formulas

$(\texttt{balance} >= c \ \& \ \texttt{amount} > 0) \ \texttt{->}$
$\texttt{<charge(amount);>} \ \texttt{balance} > c$

$\texttt{<x = 1;>}([\texttt{while (true) \{\}}]\, \mathit{false})$

- Program formulas can appear nested

$\texttt{\textbackslash forall}\ \mathit{int}\ \mathit{val};\ ((\texttt{<p>}\,\texttt{x} \doteq \mathit{val}) \ \texttt{<->} \ (\texttt{<q>}\,\texttt{x} \doteq \mathit{val}))$

- p, q equivalent relative to computation state restricted to `x`

Dynamic Logic Example Formulas

```
  a != null
->
  <
    int max = 0;
    if ( a.length > 0 ) max = a[0];
    int i = 1;
    while ( i < a.length ) {
      if ( a[i] > max ) max = a[i];
      ++i;
    }
  >
    (
      \forall int j; (j >= 0 & j < a.length -> max >= a[j])
      &
      (a.length > 0 ->
        \exists int j; (j >= 0 & j < a.length & max = a[j]))
    )
```

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

$\texttt{<int i;>}\ \texttt{\textbackslash forall}\ \mathit{int}\ x;(\texttt{i} + 1 \doteq x \texttt{ -> } \texttt{<i++;>}(\texttt{i} \doteq x))$

- Interpretation of `i` depends on computation state ⇒ flexible
- Interpretation of x and $+$ do not depend on state ⇒ rigid

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

Rigid and Flexible Terms

Example

$\texttt{<int i;>}\backslash\texttt{forall}\ \mathit{int}\ x; (\texttt{i} + 1 \doteq x\ \texttt{->}\ \texttt{<i++;>}(\texttt{i} \doteq x))$

- Interpretation of `i` depends on computation state ⇒ flexible
- Interpretation of x and $+$ do not depend on state ⇒ rigid

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

Rigid and Flexible Terms

Example

$\texttt{<int i;>}\,\backslash\texttt{forall}\ \mathit{int}\ x; (\texttt{i} + 1 \doteq x \ \texttt{->}\ \texttt{<i++;>}(\texttt{i} \doteq x))$

- Interpretation of `i` depends on computation state ⇒ flexible
- Interpretation of x and $+$ do not depend on state ⇒ rigid

Locations are always flexible

Logical variables, standard functions are always rigid

Rigid and Flexible Terms

Example

`<int i;>` $\backslash$`forall` $int\ x;$ $($`i` $+ 1 \doteq x$ `->` `<i++;>` $($`i` $\doteq x))$

- Interpretation of `i` depends on computation state ⇒ flexible
- Interpretation of x and $+$ do not depend on state ⇒ rigid

Locations are always flexible

Logical variables, standard functions are always rigid

Rigid and Flexible Terms

Example

$\texttt{<int i;>}\,\backslash\texttt{forall}\ \mathit{int}\ x; (\texttt{i} + 1 \doteq x \ \texttt{->}\ \texttt{<i++;>}(\texttt{i} \doteq x))$

- Interpretation of `i` depends on computation state ⇒ flexible
- Interpretation of x and $+$ do not depend on state ⇒ 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

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

$$\underbrace{\psi_1, \ldots, \psi_m}_{Antecedent} \quad \Longrightarrow \quad \underbrace{\phi_1, \ldots, \phi_n}_{Succedent}$$

where the ϕ_i, ψ_i are formulae (without free variables)

Semantics

Same as the formula

$$(\psi_1 \ \ \& \ \cdots \ \& \ \ \psi_m) \quad \texttt{->} \quad (\phi_1 \ | \ \cdots \ | \ \phi_n)$$

Sequents and their Semantics

Syntax

$$\underbrace{\psi_1, \ldots, \psi_m}_{Antecedent} \quad \Longrightarrow \quad \underbrace{\phi_1, \ldots, \phi_n}_{Succedent}$$

where the ϕ_i, ψ_i are formulae (without free variables)

Semantics

Same as the formula

$$(\psi_1 \ \& \ \cdots \ \& \ \psi_m) \quad \text{->} \quad (\phi_1 \ | \ \cdots \ | \ \phi_n)$$

Sequent Rules

General form

$$\text{rule\_name}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}^{\text{Premisses}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}}$$

($r = 0$ 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

$$\text{rule\_name}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}^{\text{Premisses}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}}$$

($r = 0$ 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

$$\text{rule\_name}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}^{\text{Premisses}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}}$$

($r = 0$ 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

$$\text{rule\_name}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}^{\text{Premisses}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}}$$

($r = 0$ 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 $$\frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\,A \Longrightarrow \Delta}$$

imp_left $$\frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \mathrel{-\!\!>} B \Longrightarrow \Delta}$$

close_goal $$\frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

close_by_true $$\frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

all_left $$\frac{\Gamma, \texttt{\textbackslash forall}\ t\ x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \texttt{\textbackslash forall}\ t\ x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

$$\text{not\_left}\quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\,A \Longrightarrow \Delta}$$

$$\text{imp\_left}\quad \frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \mathrel{-\!\!>} B \Longrightarrow \Delta}$$

$$\text{close\_goal}\quad \frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

$$\text{close\_by\_true}\quad \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

$$\text{all\_left}\quad \frac{\Gamma, \backslash\mathtt{forall}\; t\; x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\mathtt{forall}\; t\; x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

$$\text{not\_left}\ \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\,A \Longrightarrow \Delta}$$

$$\text{imp\_left}\ \frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \;\text{->}\; B \Longrightarrow \Delta}$$

$$\text{close\_goal}\ \frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

$$\text{close\_by\_true}\ \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

$$\text{all\_left}\ \frac{\Gamma, \backslash\texttt{forall}\ t\ x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\texttt{forall}\ t\ x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

$$\text{not\_left}\ \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

$$\text{imp\_left}\ \frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \mathrel{{-}{>}} B \Longrightarrow \Delta}$$

$$\text{close\_goal}\ \frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

$$\text{close\_by\_true}\ \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

$$\text{all\_left}\ \frac{\Gamma, \backslash\mathtt{forall}\ t\ x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\mathtt{forall}\ t\ x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

$$\mathsf{not\_left}\ \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

$$\mathsf{imp\_left}\ \frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \mathrel{-\!\!>} B \Longrightarrow \Delta}$$

$$\mathsf{close\_goal}\ \frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

$$\mathsf{close\_by\_true}\ \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

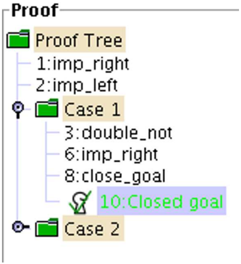
$$\mathsf{all\_left}\ \frac{\Gamma, \backslash\mathtt{forall}\ t\ x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\mathtt{forall}\ t\ x; \phi \Longrightarrow \Delta}$$

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

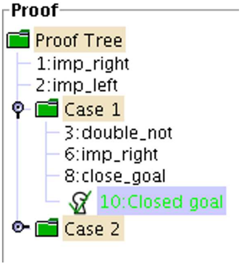


```
Proof
Proof Tree
  1:imp_right
  2:imp_left
  Case 1
    3:double_not
    6:imp_right
    8:close_goal
      10:Closed goal
  Case 2
```

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

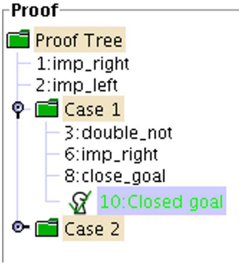


Proof
Proof Tree
1:imp_right
2:imp_left
Case 1
3:double_not
6:imp_right
8:close_goal
10:Closed goal
Case 2

Sequent Calculus Proofs

Proof tree

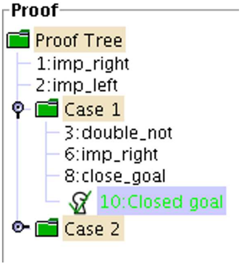
- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed



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



Proof

- Proof Tree
  - 1:imp_right
  - 2:imp_left
  - Case 1
    - 3:double_not
    - 6:imp_right
    - 8:close_goal
    - 10:Closed goal
  - Case 2

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

```
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:{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

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0; \}\}}}_{\omega}$$

passive prefix	π
active statement	`i=0;`
rest	ω

- 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

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0; \}\}}}_{\omega}$$

passive prefix	π
active statement	`i=0;`
rest	ω

- Sequent rules execute symbolically the active statement

Rules for Symbolic Program Execution

If-then-else rule

$$\frac{\Gamma, B = \mathit{true} \Longrightarrow \texttt{<}p\ \ \omega\texttt{>}\,\phi, \Delta \qquad \Gamma, B = \mathit{false} \Longrightarrow \texttt{<}q\ \ \omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<if (}B\texttt{) \{ }p\texttt{ \} else \{ }q\texttt{ \} }\omega\texttt{>}\,\phi, \Delta}$$

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

$$\frac{\Gamma \Longrightarrow \texttt{<v=y; y=y+1; x=v; }\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<x=y++; }\omega\texttt{>}\,\phi, \Delta}$$

Simple assignment rule

$$\frac{\Gamma \Longrightarrow \{loc := val\}\texttt{<}\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<}loc\texttt{=}val\texttt{; }\omega\texttt{>}\,\phi, \Delta}$$

Rules for Symbolic Program Execution

If-then-else rule

$$\frac{\Gamma, B = \textit{true} \Longrightarrow \texttt{<}p\ \ \omega\texttt{>}\,\phi, \Delta \qquad \Gamma, B = \textit{false} \Longrightarrow \texttt{<}q\ \ \omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<if (}B\texttt{) \{ }p\texttt{ \} else \{ }q\texttt{ \} }\omega\texttt{>}\,\phi, \Delta}$$

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

$$\frac{\Gamma \Longrightarrow \texttt{<v=y; y=y+1; x=v; }\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<x=y++; }\omega\texttt{>}\,\phi, \Delta}$$

Simple assignment rule

$$\frac{\Gamma \Longrightarrow \{loc := val\}\texttt{<}\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<}loc\texttt{=}val\texttt{; }\omega\texttt{>}\,\phi, \Delta}$$

Rules for Symbolic Program Execution

If-then-else rule

$$\frac{\Gamma, B = \textsf{\textit{true}} \Longrightarrow \texttt{<}p\ \ \omega\texttt{>}\,\phi, \Delta \qquad \Gamma, B = \textsf{\textit{false}} \Longrightarrow \texttt{<}q\ \ \omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<if (}B\texttt{) \{ }p\texttt{ \} else \{ }q\texttt{ \} }\omega\texttt{>}\,\phi, \Delta}$$

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

$$\frac{\Gamma \Longrightarrow \texttt{<v=y; y=y+1; x=v; }\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<x=y++; }\omega\texttt{>}\,\phi, \Delta}$$

Simple assignment rule

$$\frac{\Gamma \Longrightarrow \{loc := val\}\texttt{<}\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<}loc\texttt{=}val\texttt{; }\omega\texttt{>}\,\phi, \Delta}$$

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 $o.attr$, 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

$$\{loc := val\}\,\phi$$

where (roughly)

- loc a program variable x, an attribute access $o.attr$, or an array access $a[i]$
- val is same as loc, or a literal, or a logical variable

Parallel Updates

$$\{loc_1 := t_1 \,||\, \cdots \,||\, loc_n := t_n\}\,\phi$$

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 $o.attr$, or an array access $a[i]$
- val is same as loc, or a literal, or a logical variable

Parallel Updates

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

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)

$$\mathtt{x} < \mathtt{y} \implies \mathtt{x} < \mathtt{y}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{x:=y} \parallel \mathtt{y:=x}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y} \parallel \mathtt{y:=x}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y}\}\{\mathtt{y:=t}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\mathtt{<y=t;>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\mathtt{<x=y;\ y=t;>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\implies \mathtt{x} < \mathtt{y} \mathrel{-\!\!>} \mathtt{<int\ t=x;\ x=y;\ y=t;>}\ \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates

(by Example)

$$\mathtt{x} < \mathtt{y} \;\Longrightarrow\; \mathtt{x} < \mathtt{y}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \;\Longrightarrow\; \{\mathtt{x:=y} \parallel \mathtt{y:=x}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \;\Longrightarrow\; \{\mathtt{t:=x} \parallel \mathtt{x:=y} \parallel \mathtt{y:=x}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \;\Longrightarrow\; \{\mathtt{t:=x} \parallel \mathtt{x:=y}\}\{\mathtt{y:=t}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \;\Longrightarrow\; \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\mathtt{<y=t;>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \;\Longrightarrow\; \{\mathtt{t:=x}\}\mathtt{<x=y;\ y=t;>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\Longrightarrow\; \mathtt{x} < \mathtt{y} \mathrel{->} \mathtt{<int\ t=x;\ x=y;\ y=t;>}\ \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates

(by Example)

$$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \mathtt{x} < \mathtt{y}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \{\mathtt{x:=y} \parallel \mathtt{y:=x}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \{\mathtt{t:=x} \parallel \mathtt{x:=y} \parallel \mathtt{y:=x}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \{\mathtt{t:=x} \parallel \mathtt{x:=y}\}\{\mathtt{y:=t}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\mathtt{<y=t;>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \{\mathtt{t:=x}\}\mathtt{<x=y;\ y=t;>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\Longrightarrow \quad \mathtt{x} < \mathtt{y} \ \mathtt{->}\ \mathtt{<int\ t=x;\ x=y;\ y=t;>}\ \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates

(by Example)

$$\texttt{x} < \texttt{y} \quad \Longrightarrow \quad \texttt{x} < \texttt{y}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \quad \Longrightarrow \quad \{\texttt{x:=y} \parallel \texttt{y:=x}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \quad \Longrightarrow \quad \{\texttt{t:=x} \parallel \texttt{x:=y} \parallel \texttt{y:=x}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \quad \Longrightarrow \quad \{\texttt{t:=x} \parallel \texttt{x:=y}\}\{\texttt{y:=t}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \quad \Longrightarrow \quad \{\texttt{t:=x}\}\{\texttt{x:=y}\}\texttt{<y=t;>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \quad \Longrightarrow \quad \{\texttt{t:=x}\}\texttt{<x=y; y=t;>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\Longrightarrow \quad \texttt{x} < \texttt{y} \texttt{ -> <int t=x; x=y; y=t;>}\ \texttt{y} < \texttt{x}$$

Symbolic Execution with Updates

(by Example)

$$\mathtt{x} < \mathtt{y} \quad\Longrightarrow\quad \mathtt{x} < \mathtt{y}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \quad\Longrightarrow\quad \{\mathtt{x{:=}y} \parallel \mathtt{y{:=}x}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \quad\Longrightarrow\quad \{\mathtt{t{:=}x} \parallel \mathtt{x{:=}y} \parallel \mathtt{y{:=}x}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \quad\Longrightarrow\quad \{\mathtt{t{:=}x} \parallel \mathtt{x{:=}y}\}\{\mathtt{y{:=}t}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \quad\Longrightarrow\quad \{\mathtt{t{:=}x}\}\{\mathtt{x{:=}y}\}\mathtt{<y=t;>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \quad\Longrightarrow\quad \{\mathtt{t{:=}x}\}\mathtt{<x=y;\ y=t;>}\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\Longrightarrow\quad \mathtt{x} < \mathtt{y}\ \mathtt{->}\ \mathtt{<int\ t=x;\ x=y;\ y=t;>}\ \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates
(by Example)

$$
\begin{array}{lcl}
\mathtt{x} < \mathtt{y} & \Longrightarrow & \mathtt{x} < \mathtt{y} \\
 & \vdots & \\
\mathtt{x} < \mathtt{y} & \Longrightarrow & \{\mathtt{x:=y} \parallel \mathtt{y:=x}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x} \\
 & \vdots & \\
\mathtt{x} < \mathtt{y} & \Longrightarrow & \{\mathtt{t:=x} \parallel \mathtt{x:=y} \parallel \mathtt{y:=x}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x} \\
 & \vdots & \\
\mathtt{x} < \mathtt{y} & \Longrightarrow & \{\mathtt{t:=x} \parallel \mathtt{x:=y}\}\{\mathtt{y:=t}\}\mathtt{<>}\ \mathtt{y} < \mathtt{x} \\
 & \vdots & \\
\mathtt{x} < \mathtt{y} & \Longrightarrow & \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\mathtt{<y=t;>}\ \mathtt{y} < \mathtt{x} \\
 & \vdots & \\
\mathtt{x} < \mathtt{y} & \Longrightarrow & \{\mathtt{t:=x}\}\mathtt{<x=y;\ y=t;>}\ \mathtt{y} < \mathtt{x} \\
 & \vdots & \\
 & \Longrightarrow & \mathtt{x} < \mathtt{y} \mathrel{-\!>} \mathtt{<int\ t=x;\ x=y;\ y=t;>}\ \mathtt{y} < \mathtt{x}
\end{array}
$$

Symbolic Execution with Updates

(by Example)

$$
\begin{array}{lcl}
\texttt{x} < \texttt{y} & \Longrightarrow & \texttt{x} < \texttt{y} \\
 & \vdots & \\
\texttt{x} < \texttt{y} & \Longrightarrow & \{\texttt{x:=y} \parallel \texttt{y:=x}\}\texttt{<>}\ \texttt{y} < \texttt{x} \\
 & \vdots & \\
\texttt{x} < \texttt{y} & \Longrightarrow & \{\texttt{t:=x} \parallel \texttt{x:=y} \parallel \texttt{y:=x}\}\texttt{<>}\ \texttt{y} < \texttt{x} \\
 & \vdots & \\
\texttt{x} < \texttt{y} & \Longrightarrow & \{\texttt{t:=x} \parallel \texttt{x:=y}\}\{\texttt{y:=t}\}\texttt{<>}\ \texttt{y} < \texttt{x} \\
 & \vdots & \\
\texttt{x} < \texttt{y} & \Longrightarrow & \{\texttt{t:=x}\}\{\texttt{x:=y}\}\texttt{<y=t;>}\ \texttt{y} < \texttt{x} \\
 & \vdots & \\
\texttt{x} < \texttt{y} & \Longrightarrow & \{\texttt{t:=x}\}\texttt{<x=y; y=t;>}\ \texttt{y} < \texttt{x} \\
 & \vdots & \\
 & \Longrightarrow & \texttt{x} < \texttt{y} \ \texttt{->}\ \texttt{<int t=x; x=y; y=t;>}\ \texttt{y} < \texttt{x}
\end{array}
$$

Program State Representation

Local program variables

Modeled as non-rigid constants

Heap

Modeled with theory of arrays:

$$\begin{array}{ll} heap: & \to Heap \text{ (the heap in the current state)} \\ select: & Heap \times Object \times Field \to Any \\ store: & Heap \times Object \times Field \times Any \to Heap \end{array}$$

Heap axioms (excerpt)

$$select(store(h, o, f, x), o, f) = x$$
$$select(store(h, o, f, x), u, f) = select(h, u, f) \text{ if } o \neq u$$

Program State Representation

Local program variables

Modeled as non-rigid constants

Heap

Modeled with theory of arrays:

$$\begin{array}{ll} \mathit{heap}: & \to \mathit{Heap} \text{ (the heap in the current state)} \\ \mathit{select}: & \mathit{Heap} \times \mathit{Object} \times \mathit{Field} \to \mathit{Any} \\ \mathit{store}: & \mathit{Heap} \times \mathit{Object} \times \mathit{Field} \times \mathit{Any} \to \mathit{Heap} \end{array}$$

Heap axioms (excerpt)

$$\mathit{select}(\mathit{store}(h, o, f, x), o, f) = x$$
$$\mathit{select}(\mathit{store}(h, o, f, x), u, f) = \mathit{select}(h, u, f) \text{ if } o \neq u$$

Program State Representation

Local program variables

Modeled as non-rigid constants

Heap

Modeled with theory of arrays:

$$\begin{array}{ll} heap: & \rightarrow Heap \text{ (the heap in the current state)} \\ select: & Heap \times Object \times Field \rightarrow Any \\ store: & Heap \times Object \times Field \times Any \rightarrow Heap \end{array}$$

Heap axioms (excerpt)

$$select(store(h, o, f, x), o, f) = x$$
$$select(store(h, o, f, x), u, f) = select(h, u, f) \text{ if } o \neq u$$

Handling Abrupt Termination

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

Example

TRY-THROW

$$\frac{\Gamma \Longrightarrow \left\langle \begin{array}{l} \texttt{if (exc instanceof T)} \\ \quad \texttt{\{try \{e=exc; r\} finally \{s\}\}} \\ \quad \texttt{else \{s throw exc;\}}\ \ \omega \end{array} \right\rangle \phi, \Delta}{\Gamma \Longrightarrow \texttt{<try\{throw exc; q\} catch(T e)\{r\} finally\{s\}}\ \omega\texttt{>}\ \phi, \Delta}$$

Handling Abrupt Termination

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

Example

TRY-THROW

$$\frac{\Gamma \Longrightarrow \left\langle \begin{array}{l} \texttt{if (exc instanceof T)} \\ \quad \texttt{\{try \{e=exc; r\} finally \{s\}\}} \\ \quad \texttt{else \{s throw exc;\}} \quad \omega \end{array} \right\rangle \phi,\ \Delta}{\Gamma \Longrightarrow \texttt{<try\{throw exc; q\} catch(T e)\{r\} finally\{s\}}\ \omega\texttt{>}\, \phi,\ \Delta}$$

Handling Abrupt Termination

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

Example

TRY-THROW

$$\frac{\Gamma \Longrightarrow \left\langle \begin{array}{l} \pi\ \texttt{if (exc instanceof T)} \\ \qquad \texttt{\{try \{e=exc; r\} finally \{s\}\}} \\ \qquad \texttt{else \{s throw exc;\}}\ \ \omega \end{array} \right\rangle \phi, \Delta}{\Gamma \Longrightarrow \texttt{<}\pi\ \texttt{try\{throw exc; q\} catch(T e)\{r\} finally\{s\}}\ \omega\texttt{>}\,\phi, \Delta}$$

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

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

Karlsruhe Institute of Technology

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

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: 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

Part III

Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

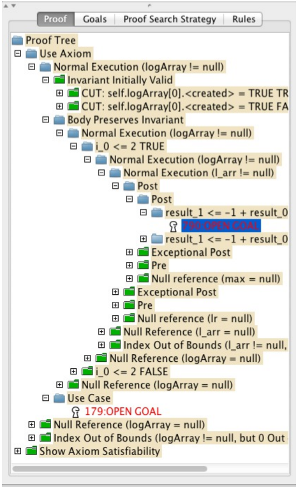
10 A Calculus for 100% JAVA CARD

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



- Proof Tree
  - Use Axiom
    - Normal Execution (logArray != null)
      - Invariant Initially Valid
        - CUT: self.logArray[0].<created> = TRUE TR
        - CUT: self.logArray[0].<created> = TRUE FA
      - Body Preserves Invariant
        - Normal Execution (logArray != null)
          - i_0 <= 2 TRUE
            - Normal Execution (logArray != null)
              - Normal Execution (l_arr != null)
                - Post
                  - Post
                    - result_1 <= -1 + result_0
                      - 790:OPEN GOAL
                    - result_1 <= -1 + result_0
                  - Exceptional Post
                  - Pre
                  - Null reference (max = null)
                - Exceptional Post
                - Pre
                - Null reference (lr = null)
              - Null Reference (l_arr = null)
              - Index Out of Bounds (l_arr != null,
            - Null Reference (logArray = null)
          - i_0 <= 2 FALSE
        - Null Reference (logArray = null)
      - Use Case
        - 179:OPEN GOAL
    - Null Reference (logArray = null)
    - Index Out of Bounds (logArray != null, but 0 Out
  - Show Axiom Satisfiability

Taclet Syntax

$$\text{andLeft}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A \mathbin{\&} B \Longrightarrow \Delta}$$

Taclet

```
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

Taclet Syntax

$$\textsf{andLeft}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A \mathbin{\&} B \Longrightarrow \Delta}$$

Taclet

```
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

Taclet Syntax

$$\text{andLeft}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A \mathbin{\&} B \Longrightarrow \Delta}$$

Taclet

```
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

Taclet Syntax

$$\text{andLeft}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A \mathbin{\&} B \Longrightarrow \Delta}$$

Taclet

```
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

Taclet Syntax

$$\text{andLeft}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A \mathbin{\&} B \Longrightarrow \Delta}$$

Taclet

```
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

Taclet Syntax

Some rules are only sound in a certain context

$$\text{modusPonens}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A, A \texttt{->} B \Longrightarrow \Delta}$$

Taclet

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

Taclet Syntax

Some rules are only sound in a certain context

$$\text{modusPonens}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A, A \mathrel{-\!\!>} B \Longrightarrow \Delta}$$

Taclet

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

Taclet Syntax

Some rules are only sound in a certain context

$$\text{modusPonens}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A, A \mathbin{\texttt{->}} B \Longrightarrow \Delta}$$

Taclet

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

Taclet Syntax

Proof Splitting: andRight

$$\frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma \Longrightarrow B, \Delta}{\Gamma \Longrightarrow A \mathbin{\&} B, \Delta}$$

```
andRight {
  \find ( ==> A & B )
  \replacewith (==> A );
  \replacewith (==> B )
};
```

Variable Conditions: allRight

$$\frac{\Gamma \Longrightarrow \{x/c\}\Phi, \Delta}{\Gamma \Longrightarrow \forall T\ x; \Phi, \Delta}, \text{c new}$$

```
allRight {
  \find ( ==> \forall x;phi )
  \varcond(\new(c,\dependingOn(phi)))
  \replacewith ( ==> {\subst x;c}phi )
};
```

Taclet Syntax

Proof Splitting: andRight

$$\frac{\Gamma \Longrightarrow A, \Delta \quad \Gamma \Longrightarrow B, \Delta}{\Gamma \Longrightarrow A \mathbin{\&} B, \Delta}$$

```
andRight {
   \find ( ==> A & B )
   \replacewith (==> A );
   \replacewith (==> B )
};
```

Variable Conditions: allRight

$$\frac{\Gamma \Longrightarrow \{x/c\}\Phi, \Delta}{\Gamma \Longrightarrow \forall T\ x; \Phi, \Delta}, \text{c new}$$

```
allRight {
   \find ( ==> \forall x;phi )
   \varcond(\new(c,\dependingOn(phi)))
   \replacewith ( ==> {\subst x;c}phi )
};
```

Taclets for Program Transformations

$$\frac{\Gamma \Longrightarrow \left\langle \begin{array}{l} \pi\ \texttt{if (exc == null) \{} \\ \quad \texttt{try\{ throw new NPE(); catch(T e) \{r\};} \\ \texttt{\} else if (exc instanceof T) \{e=exc; r\}} \\ \texttt{else throw exc;}\ \omega \end{array} \right\rangle \phi}{\Gamma \Longrightarrow \texttt{<}\pi\ \texttt{try\{throw exc; q\} catch(T e)\{r\};}\ \omega\texttt{>} \phi}$$

```
\find ( <.. try { throw #se; #slist }
            catch ( #t #v0 ) { #slist1 } ...> post )
\replacewith (
  <.. if (#se == null) {
        try { throw new NullPointerException(); }
        catch (#t #v0) { #slist1 }
      } else if (#se instanceof #t) {
        #t #v0 = (#t) #se;
        #slist1
      } else throw #se; ...> 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
- partion 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

$$\Leftrightarrow$$

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
- partion 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

$$\Leftrightarrow$$

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
- partion 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

$$\Leftrightarrow$$

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
- partion 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 $\Leftrightarrow$

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
- partion 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

$$\Leftrightarrow$$

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
- partion 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

$$\Leftrightarrow$$

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() {                void m_3() {
    l = h;                      if (h>0) {l=1;}
  }                             else {l=2;};
                              }
  void m_2() {
    if (l>0) {h=1;}           void m_4() {
    else {h=2;};                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?

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

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

```
}
```

Examples

Which methods are secure?

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

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

```
}
```

Examples

Which methods are secure?

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

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

}

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 $\Leftrightarrow$

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.

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

- **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

$$\Leftrightarrow$$

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.

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

- **Problem:** not suitable for automatic verification $\leadsto$ 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

$$\Leftrightarrow$$

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.

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

- **Problem:** not suitable for automatic verification ⇝ 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 $\Leftrightarrow$

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.

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

- **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 $\Leftrightarrow$

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.

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

- **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

$$\Leftrightarrow$$

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.

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

- **Problem:** not suitable for automatic verification ⇝ 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 $\Leftrightarrow$

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.

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

- **Problem:** not suitable for automatic verification $\rightsquigarrow$ 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 $\Leftrightarrow$

running two instances of P on the same low values but on arbitrary high values result in low variables which have the same values.

$$\begin{array}{l}\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \; (\\ \quad \{low := in_l \mid\mid high := in_h^1\}[P]out_l^1 = low \\ \quad \wedge \{low := in_l \mid\mid high := in_h^2\}[P]out_l^2 = low \\ \quad \rightarrow out_l^1 = out_l^2 \\ )\end{array}$$

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

$$\Leftrightarrow$$

running two instances of P on the same low values but on arbitrary high values result in low variables which have the same values.

$$
\begin{aligned}
&\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2\ (\\
&\qquad \{low := in_l \mid\mid high := in_h^1\}[P]out_l^1 = low\\
&\quad \wedge \{low := in_l \mid\mid high := in_h^2\}[P]out_l^2 = low\\
&\quad \rightarrow out_l^1 = out_l^2\\
&)
\end{aligned}
$$

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 $\Leftrightarrow$

running two instances of P on the same low values but on arbitrary high values result in low variables which have the same values.

$$
\begin{aligned}
&\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \; (\\
&\qquad \{low := in_l \;||\; high := in_h^1\}[P]out_l^1 = low \\
&\quad \wedge \{low := in_l \;||\; high := in_h^2\}[P]out_l^2 = low \\
&\quad \rightarrow out_l^1 = out_l^2 \\
&)
\end{aligned}
$$

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 $\Leftrightarrow$

running two instances of P on the same low values but on arbitrary high values result in low variables which have the same values.

$$\begin{array}{l} \forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \; (\\ \qquad \{low := in_l \mid\mid high := in_h^1\}[P] out_l^1 = low \\ \quad \wedge \{low := in_l \mid\mid high := in_h^2\}[P] out_l^2 = low \\ \quad \rightarrow out_l^1 = out_l^2 \\ ) \end{array}$$

Program Variables vs. Heaps

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

$$
\begin{aligned}
&\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \, (\\
&\qquad \{low := in_l \mid\mid high := in_h^1\}[P]out_l^1 = low \\
&\qquad \land \{low := in_l \mid\mid high := in_h^2\}[P]out_l^2 = low \\
&\qquad \rightarrow out_l^1 = out_l^2 \\
&)
\end{aligned}
$$

- **Problem:** formulas contain for every program variable a quantifier and an update.
- **Better:** talking about heaps and sets of heap locations ⇝ 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 low and high security locations $high$

$$\Leftrightarrow$$

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.

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

$$\begin{aligned}h^1 \sim_{low} h^2 \leftrightarrow \forall\ Object\ o\ \forall\ Field\ f\ (&\\ &(o, f) \in low \rightarrow\ \{heap := h^1\}o.f = \{heap := h^2\}o.f\ )\end{aligned}$$

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 $high$

$$\Leftrightarrow$$

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.

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

$$h^1 \sim_{low} h^2 \leftrightarrow \forall\ Object\ o\ \forall\ Field\ f\ (\\ (o,f) \in low \rightarrow\ \{heap := h^1\}o.f = \{heap := h^2\}o.f\ )$$

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 $high$

$$\Leftrightarrow$$

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.

$$\begin{aligned} \forall\ Heap\ h^1_{in}, h^2_{in}, h^1_{out}, h^2_{out}\ (& \\ &\{heap := h^1_{in}\}[P]h^1_{out} = heap \\ \wedge\ &\{heap := h^2_{in}\}[P]h^2_{out} = heap \\ \rightarrow\ &(h^1_{in} \sim_{low} h^2_{in} \rightarrow h^1_{out} \sim_{low} h^2_{out})\ ) \end{aligned}$$

$$\begin{aligned} h^1 \sim_{low} h^2 \leftrightarrow \forall\ Object\ o\ \forall\ Field\ f\ (& \\ &(o, f) \in low \rightarrow\ \{heap := h^1\}o.f = \{heap := h^2\}o.f\ ) \end{aligned}$$

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 $high$

$$\Leftrightarrow$$

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.

$$\begin{aligned}\forall\ Heap\ h^1_{in}, h^2_{in}, h^1_{out}, h^2_{out}\ (&\\ &\{heap := h^1_{in}\}[P]h^1_{out} = heap\\ \wedge\ &\{heap := h^2_{in}\}[P]h^2_{out} = heap\\ \rightarrow\ &(h^1_{in} \sim_{low} h^2_{in} \rightarrow h^1_{out} \sim_{low} h^2_{out})\ )\end{aligned}$$

$$\begin{aligned}h^1 \sim_{low} h^2 \leftrightarrow \forall\ Object\ o\ \forall\ Field\ f\ (&\\ &(o, f) \in low \rightarrow\ \{heap := h^1\}o.f = \{heap := h^2\}o.f\ )\end{aligned}$$

Self Composition with Heaps

Karlsruhe Institute of Technology

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 $high$

$$\Leftrightarrow$$

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.

$$\begin{aligned}
&\forall\ Heap\ h^1_{in}, h^2_{in}, h^1_{out}, h^2_{out}\ (\\
&\qquad \{heap := h^1_{in}\}[P]h^1_{out} = heap\\
&\quad \wedge \{heap := h^2_{in}\}[P]h^2_{out} = heap\\
&\quad \rightarrow (h^1_{in} \sim_{low} h^2_{in} \rightarrow h^1_{out} \sim_{low} h^2_{out})\ )
\end{aligned}$$

$$\begin{aligned}
&h^1 \sim_{low} h^2 \leftrightarrow \forall\ Object\ o\ \forall\ Field\ f\ (\\
&\qquad (o, f) \in low \rightarrow\ \{heap := h^1\}o.f = \{heap := h^2\}o.f\ )
\end{aligned}$$

Declassification

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

$$
\begin{aligned}
&\forall\, Heap\ h^1_{in}, h^2_{in}, h^1_{out}, h^2_{out}\ (\\
&\qquad \{heap := h^1_{in}\}[P]h^1_{out} = heap\\
&\quad \wedge \{heap := h^2_{in}\}[P]h^2_{out} = heap\\
&\quad \rightarrow (h^1_{in} \sim_{low} h^2_{in} \wedge \{heap := h^1_{in}\}term = \{heap := h^2_{in}\}term\\
&\qquad\quad \rightarrow h^1_{out} \sim_{low} h^2_{out})\ )
\end{aligned}
$$

Termination-Sensitivity

The formalisation can be made termination sensitive:

$$\begin{aligned}
&\forall\, Heap\ h_{in}^1, h_{in}^2, h_{out}^1, h_{out}^2\ (\\
&\qquad \{heap := h_{in}^1\}[P]h_{out}^1 = heap\\
&\quad\ \wedge \{heap := h_{in}^2\}[P]h_{out}^2 = heap\\
&\quad \rightarrow (h_{in}^1 \sim_{low} h_{in}^2 \rightarrow h_{out}^1 \sim_{low} h_{out}^2)\\
&\quad\ \vee \{heap := h_{in}^1\}[P]false\ \wedge\ \{heap := h_{in}^2\}[P]false\ )
\end{aligned}$$

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

Goal

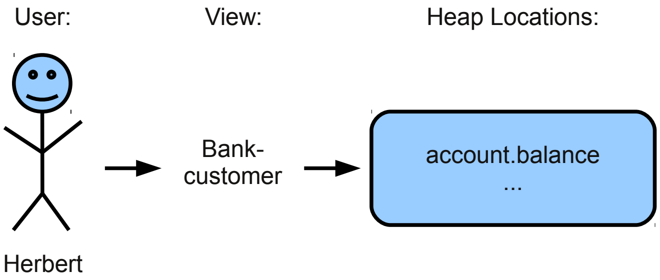
Modular program-level specification of information flow properties for **Java** programs.

- 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:

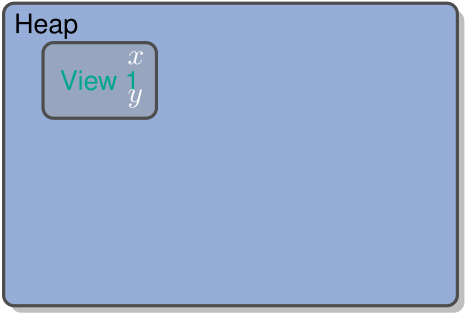
Heap

View 1

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

Implicit Security Policy

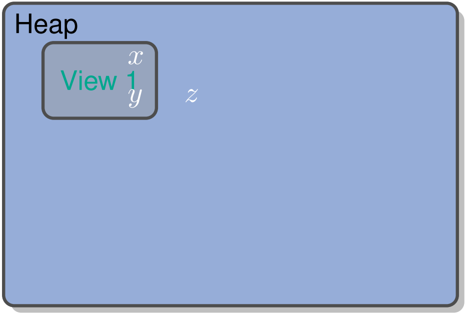
Views define an implicit security policy:



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

Implicit Security Policy

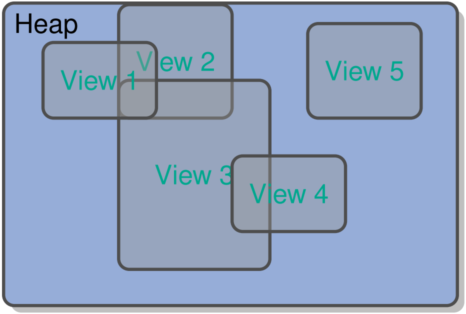
Views define an implicit security policy:



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

Implicit Security Policy

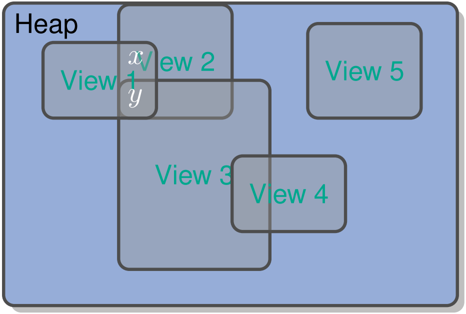
Views define an implicit security policy:



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

Implicit Security Policy

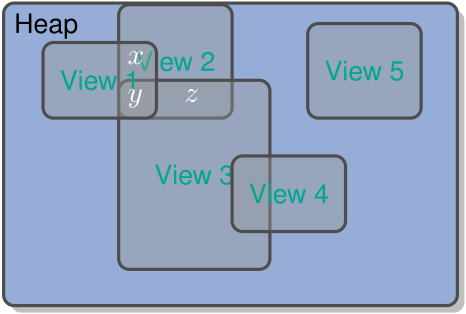
Views define an implicit security policy:



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

Implicit Security Policy

Views define an implicit security policy:

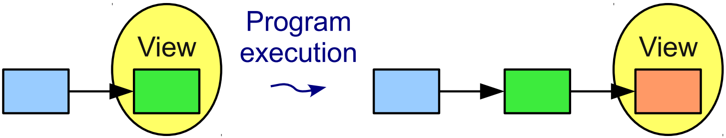


- Information may flow $y \rightarrow x$, but $y \nleftarrow x$.
- Information may flow $y \rightarrow z$, but $y \nleftarrow z$.
- Information may not flow $x \nleftrightarrow z$.

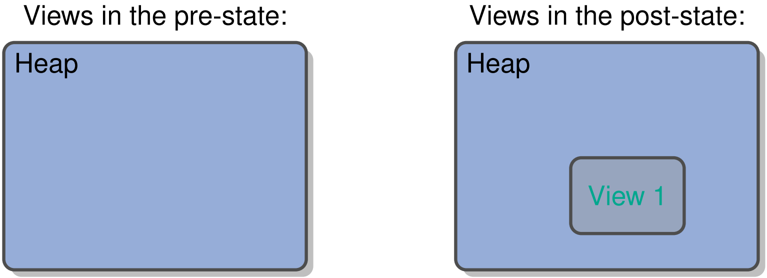
Dynamic Views

Feature:

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

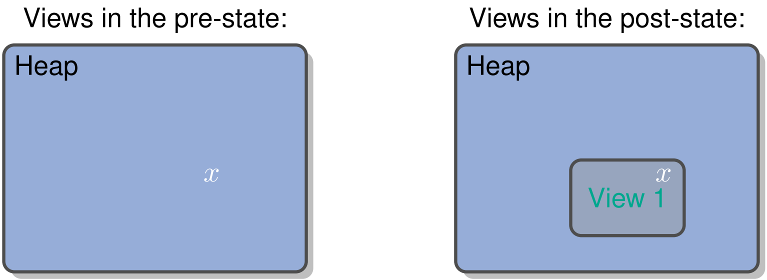


Security Policy for Dynamic Views



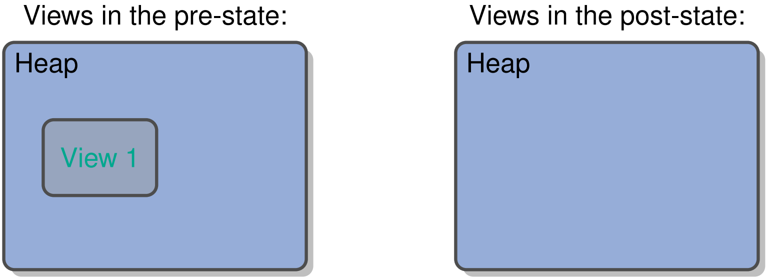
The value of x in the post-state has to be a constant.

Security Policy for Dynamic Views



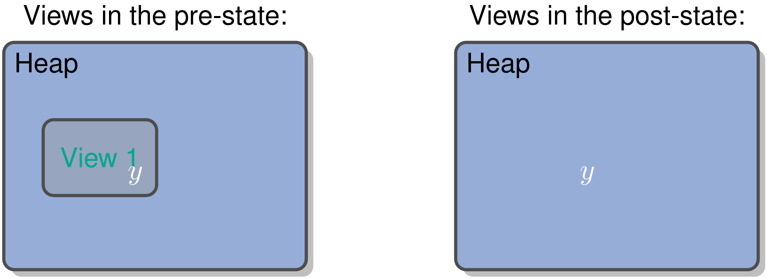
The value of x in the post-state has to be a constant.

Security Policy for Dynamic Views



Nothing has to be checked.

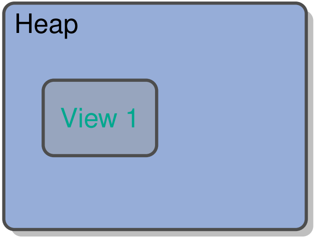
Security Policy for Dynamic Views



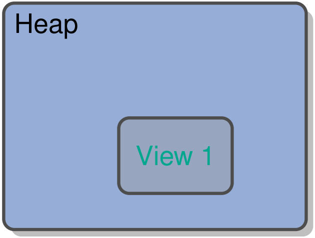
Nothing has to be checked.

Security Policy for Dynamic Views

Views in the pre-state:



Views in the post-state:



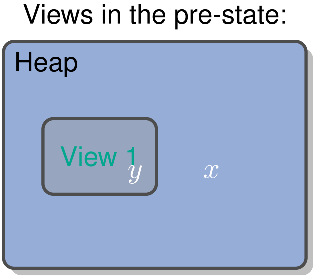
Information may flow

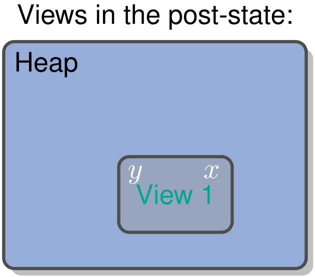
- $y \longrightarrow y$
- $y \longrightarrow x$

Information may *not* flow

- $x \not\longrightarrow y$
- $x \not\longrightarrow x$

Security Policy for Dynamic Views





Information may flow

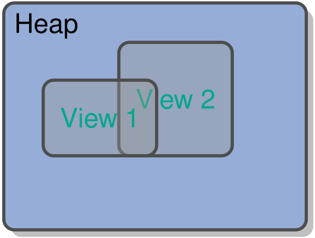
- $y \longrightarrow y$
- $y \longrightarrow x$

Information may *not* flow

- $x \not\rightarrow y$
- $x \not\rightarrow x$

Security Policy for Dynamic Views

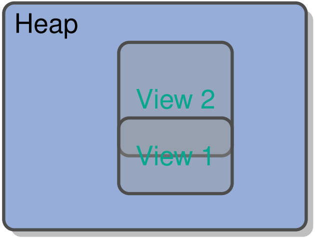
Views in the pre-state:



Information may flow

- $y \longrightarrow y$
- $y \longrightarrow x$
- $z \longrightarrow z$
- $y \longrightarrow z$
- $x \longrightarrow z$

Views in the post-state:

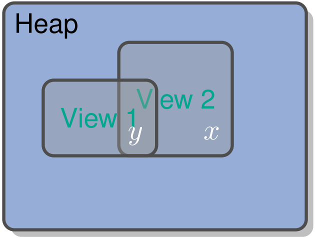


Information may *not* flow

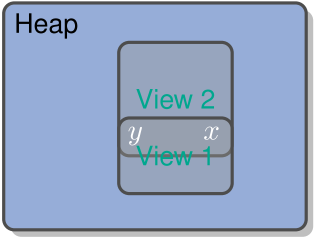
- $x \not\rightarrow y$
- $x \not\rightarrow x$
- $z \not\rightarrow y$
- $z \not\rightarrow x$

Security Policy for Dynamic Views

Views in the pre-state:



Views in the post-state:



Information may flow

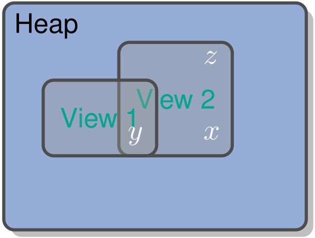
- $y \rightarrow y$
- $y \rightarrow x$
- $z \rightarrow z$
- $y \rightarrow z$
- $x \rightarrow z$

Information may *not* flow

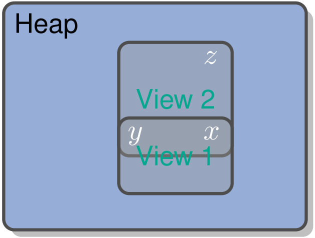
- $x \not\rightarrow y$
- $x \not\rightarrow x$
- $z \not\rightarrow y$
- $z \not\rightarrow x$

Security Policy for Dynamic Views

Views in the pre-state:



Views in the post-state:



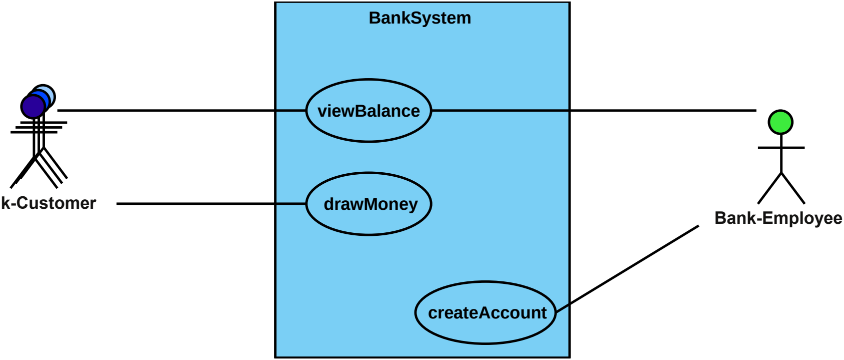
Information may flow

- $y \rightarrow y$
- $y \rightarrow x$
- $z \rightarrow z$
- $y \rightarrow z$
- $x \rightarrow z$

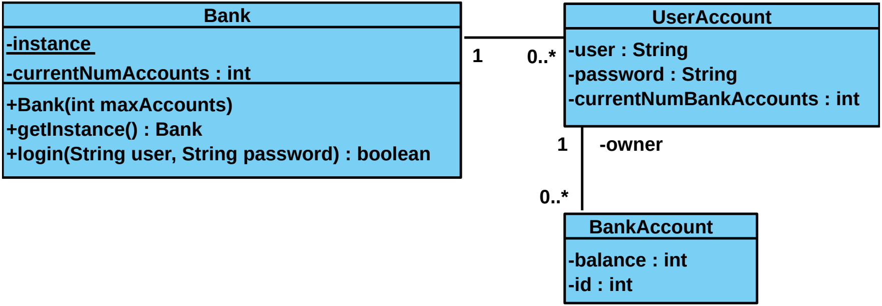
Information may *not* flow

- $x \nrightarrow y$
- $x \nrightarrow x$
- $z \nrightarrow y$
- $z \nrightarrow x$

Example: Banking System – Use-Case Diagram



Example: Banking System – Class Diagram



Bank

<u>-instance</u>

-currentNumAccounts : int

+Bank(int maxAccounts)

+getInstance() : Bank

+login(String user, String password) : boolean

UserAccount

-user : String

-password : String

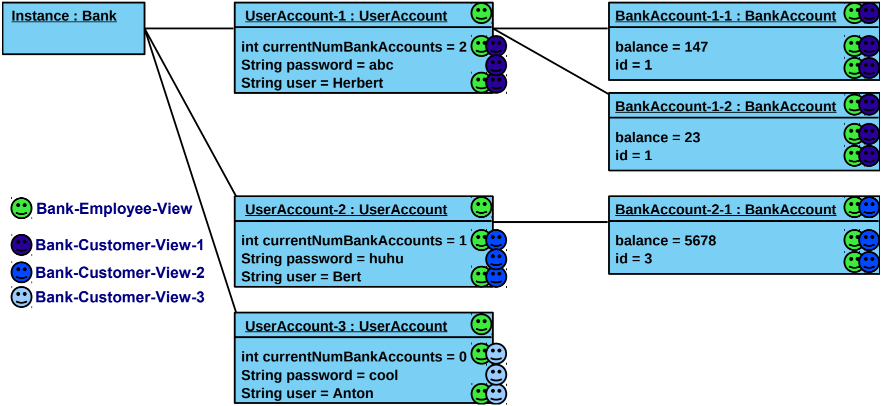
-currentNumBankAccounts : int

BankAccount

-balance : int

-id : int

Example: Banking System – Object Diagram



Instance : Bank

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

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

UserAccount-3 : UserAccount
int currentNumBankAccounts = 0
String password = cool
String user = Anton

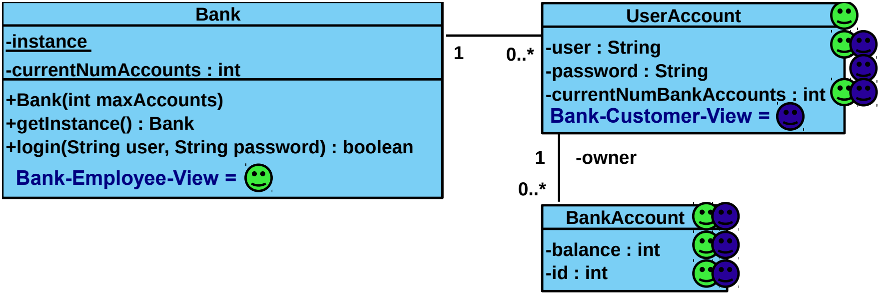
BankAccount-1-1 : BankAccount
balance = 147
id = 1

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

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

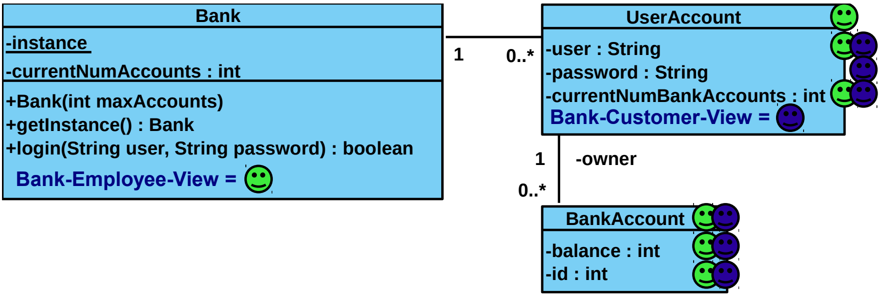
☺ Bank-Employee-View
☺ Bank-Customer-View-1
☺ Bank-Customer-View-2
☺ Bank-Customer-View-3

Example: Banking System – Views



$$\begin{aligned}\text{Bank-Employee-View} = {} & userAccounts[*] \\ & \cup\, userAccounts[*].user \\ & \cup\, userAccounts[*].currentNumBankAccounts \\ & \cup\, userAccounts[*].bankAccounts[*] \\ & \cup\, userAccounts[*].bankAccounts[*].balance \\ & \cup\, userAccounts[*].bankAccounts[*].id\end{aligned}$$

Example: Banking System – Views



Bank

-instance

-currentNumAccounts : int

+Bank(int maxAccounts)

+getInstance() : Bank

+login(String user, String password) : boolean

Bank-Employee-View = ☺

UserAccount ☺

-user : String ☺☻

-password : String ☻

-currentNumBankAccounts : int ☺☻

Bank-Customer-View = ☻

BankAccount ☺☻

-balance : int ☺☻

-id : int ☺☻

$$\begin{aligned}\text{Bank-Customer-View} = {} & user \\ & \cup\ password \\ & \cup\ currentNumAccounts \\ & \cup\ bankAccounts[*] \\ & \cup\ bankAccounts[*].balance \\ & \cup\ bankAccounts[*].id\end{aligned}$$

Non-Interference Specifications in JML

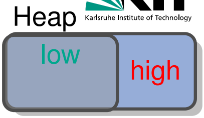
Specification as method contracts:

- 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)*.

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

- Specification of intentional information leakage *(declassify-clause)*.

Non-Interference Specifications in JML



$\text{low} \longrightarrow \text{high}$

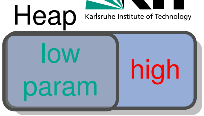
$\text{low} \not\leftarrow \text{high}$

```
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

```
public int low;
private int high;

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



low $\longrightarrow$ high

low $\not\longleftarrow$ high

param $\longleftrightarrow$ low

param $\longrightarrow$ high

param $\not\longleftarrow$ high

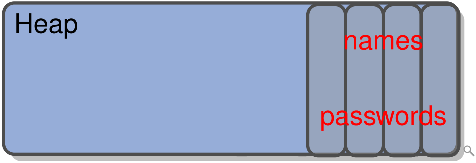
- **Views in JML:** expressions of type `\locset`.
- **Views can be named:** definition of model fields.
- Approach complies to the principle of information hiding!

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;
  }
}
```

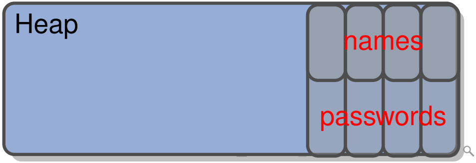
Example: Password Checker

```
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;
  }
}
```



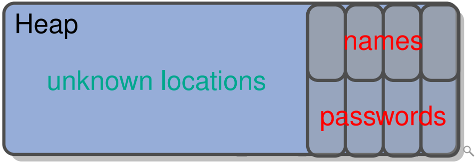
Example: Password Checker

```
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

```
/*@ respects
  @   {names[0]},
  @   {names[0], passwords[0]},
  @   {names[1]},
  @   {names[1], passwords[1]},
  @   ...;
  @*/
public boolean check(int user, int password) { ...
```

JML Specification Example

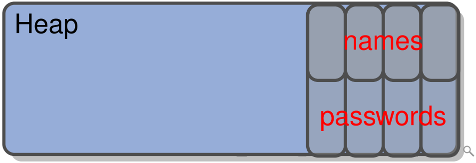
```
/*@ model int userIndex;
  @ represents userIndex \such_that
  @       0 <= userIndex
  @    && userIndex < names.length;
  @*/

/*@ respects
  @   {names[userIndex]},
  @   {names[userIndex], passwords[userIndex]};
  @*/
public boolean check(int user, int password) { ...
```

JML Specification Example

```
/*@ 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) { ...
```

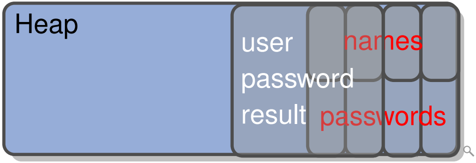
Example: Password Checker

```
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;
  }
}
```



Heap

user

names

password

result

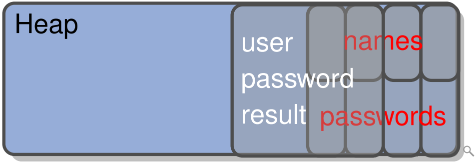
passwords

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;
  }
}
```



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)

JML Specification Example

```
/*@ 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[*]}
  @                 \to anyUser;
  @*/
public boolean check(int user, int password) { ...
```

Notes

- 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

Part V

Wrap Up

14 Further Usage of Verification Technology

15 Directions of Current Research in KeY

16 Different Approaches

Further Usage of Verification Technology

- 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

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

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

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

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

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

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

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 *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

THE END

(for now)