Verification, Testing, and Debugging with the KeY System

Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel,
Vladimir Klebanov, Peter H. Schmitt

ITP
Nijmegen
23 August 2011

Part I

The KeY System – an Overview

Overview

annotated JAVA program

↓

KeY verification browser

↓

KeY problem file

↓

first-order proof obligations

↓

successful or failed proof attempt

Overview

annotated JAVA program

fully automatic translation

KeY verification browser

KeY problem file

first-order proof obligations

successful or failed proof attempt

Overview

annotated JAVA program

↓ fully automatic translation

KeY verification browser

↓ automatic translation tiggered by user

KeY problem file

↓

first-order proof obligations

↓

successful or failed proof attempt

Overview

annotated JAVA program

fully automatic translation

KeY verification browser

automatic translation tiggered by user

KeY problem file

rule-based symbolic execution using KeY

first-order proof obligations

successful or failed proof attempt

Overview

annotated Java program

↓ fully automatic translation

KeY verification browser

↓ automatic translation tiggered by user

KeY problem file

↓ rule-based symbolic execution using KeY

first-order proof obligations

↓ KeY + SMT solvers

successful or failed proof attempt

Overview

annotated JAVA program

↓ fully automatic translation

KeY verification browser

↓ automatic translation tiggered 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

Overview

JML*

annotated JAVA program

↓ fully automatic translation

KeY verification browser

↓ automatic translation tiggered 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

Overview

JML*

annotated JAVA program

fully automatic translation

KeY verification browser

automatic translation tiggered by user

Dynamic Logic

KeY problem file

rule-based symbolic execution using KeY

first-order proof obligations

KeY + SMT solvers

successful or failed proof attempt

KeY rule base

Overview

JML*

annotated JAVA program

fully automatic translation

KeY verification browser

automatic translation tiggered by user

Dynamic Logic

KeY problem file

rule-based symbolic execution using KeY

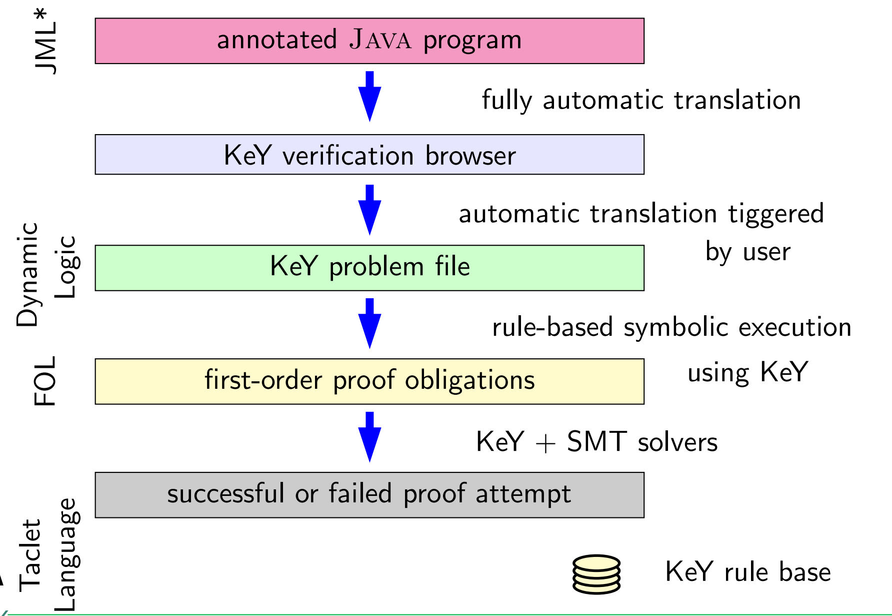
FOL

first-order proof obligations

KeY + SMT solvers

successful or failed proof attempt

Overview



Specific Features of the KeY Approach

Specific Features of the KeY Approach

- Integration of verification into the software development process.

Specific Features of the KeY Approach

- Integration of verification into the software development process.

Specific Features of the KeY Approach

- Integration of verification into the software development process.
- Program logic, explicit JAVA in the logic, not translated away

Specific Features of the KeY Approach

- Integration of verification into the software development process.

- Program logic, explicit JAVA in the logic, not translated away
- Forward symbolic execution instead of backwards wp generation.

Specific Features of the KeY Approach

- Integration of verification into the software development process.

- Program logic, explicit JAVA in the logic, not translated away
- Forward symbolic execution instead of backwards wp generation.

Specific Features of the KeY Approach

- Integration of verification into the software development process.
- Program logic, explicit JAVA in the logic, not translated away
- Forward symbolic execution instead of backwards wp generation.
- Interleaving of vcg and interactive proving.

Specific Features of the KeY Approach

- Integration of verification into the software development process.

- Program logic, explicit JAVA in the logic, not translated away
- Forward symbolic execution instead of backwards wp generation.

- Interleaving of vcg and interactive proving.

Specific Features of the KeY Approach

- Integration of verification into the software development process.
- Program logic, explicit JAVA in the logic, not translated away
- Forward symbolic execution instead of backwards wp generation.
- Interleaving of vcg and interactive proving.
- Additional benefits: test case generation, symbolic debugging.

Specific Features of the KeY Approach

- Part II (P.H.Schmitt)
  - Integration of verification into the software development process.
- Part III (V.Klebanov)
  - Program logic, explicit JAVA in the logic, not translated away
  - Forward symbolic execution instead of backwards wp generation.
- Part IV (R.Bubel)
  - Interleaving of vcg and interactive proving.
- Part V (W.Ahrendt)
  - Additional benefits: test case generation, symbolic debugging.

Part II

The Java Modeling Language

Specification of `commonEntry`

```
class SITA3{  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) { ... }
}
```

Specification of `commonEntry`

```
class SITA3{  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 annotation occur as special comments in source programm

Specification of `commonEntry`

```
class SITA3{  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) { ... }
}
```

precondition

Specification of commonEntry

```
class SITA3{  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) { ... }
}
```

postcondition

Specification of `commonEntry`

```
class SITA3{  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) { ... }
}
```

Specification of the set of locations that may at most be changed.
Here: `commonEntry` is a pure method

Specification of `commonEntry`

```
class SITA3{  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) { ... }
}
```

Specification of `commonEntry`

```
class SITA3{  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.

Specification of `commonEntry`

```
class SITA3{  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 of the parameters

Specification of `commonEntry`

```
class SITA3{  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 of the parameters such that the two arrays have the same entry.

Specification of `commonEntry`

```
class SITA3{  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.

Specification of `commonEntry`

```
class SITA3{  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.

Loop Invariant for commonEntry

```
class SITA3{  public int[] a1,a2;   ...
  public int  commonEntry(int l, int r){ int k = l;
/*@ loop_invariant   l <= k && k <= r &&
  @ (\forall int i; l <= i && i < k; a1[i] != a2[i] );
  @  assignable \nothing;
  @  decreases a1.length - k;
  @*/
   while(k < r){ if(a1[k] == a2[k]){break;}   k++;}
   return k;}
}
```

Loop Invariant for `commonEntry`

```
class SITA3{  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 Invariant for commonEntry

```
class SITA3{  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 Invariant for `commonEntry`

```
class SITA3{  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 <= l && l <= r` follows from the preconditions and

Loop Invariant for `commonEntry`

```
class SITA3{  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 <= l && l <= r` follows from the preconditions and quantification is empty

Loop Invariant for `commonEntry`

```
class SITA3{  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 executed in a state satisfying the invariant it will terminate in a state satisfying the invariant.

Loop Invariant for `commonEntry`

```
class SITA3{  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 executed in a state satisfying the invariant it will terminate in a state satisfying the invariant.
Distinguish `break` and non-`break` case

Loop Invariant for `commonEntry`

On termination of the loop

Loop Invariant for `commonEntry`

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

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

imply the postcondition

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

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

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

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

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.

Assignable Clause

```
/*@ 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 Clause

```
/*@ 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;}
```

This is an example with a non-trivial assignable clause.

Assignable Clause

```
/*@ 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 `a[pos1], a[pos2]` may be changed by method `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;}}}}
```

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 the contracts of these methods not their code.

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

DEMO

Model Fields

```
class SITA4{ ....
 /*@ 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

```
class SITA4{ ....
 /*@ 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(){....}
```

JML provides model fields, that do not occur in the code, but are solely used for specification.

Model Fields

```
class SITA4{ ....
 /*@ 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

```
class SITA4{ ....
 /*@ 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(){....}
```

JML `represents` clauses fix the semantics of model fields.

Model Fields

```
class SITA4{ ....
 /*@ 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 function `array2seq(a)` yields the abstract sequence associated with array *a*.

Model Fields

```
class SITA4{ ....
 /*@ 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

```
class SITA4{ ....
 /*@ 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(){....}
```

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

Model Fields

```
class SITA4{ ....
 /*@ 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.

Part III

Dynamic Logic

Part III

Dynamic Logic

Java Card DL

Sequent Calculus

Rules for Programs: Symbolic Execution

A Calculus for 100% Java Card

Part III

Dynamic Logic

Java Card DL

Sequent Calculus

Rules for Programs: Symbolic Execution

A Calculus for 100% Java Card

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)

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:

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)

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

- Programs are "first-class citizens"
- Real Java syntax

Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic

Hoare triple $\{\psi\}\ \alpha\ \{\phi\}$ equiv. to DL formula ψ -> $[\alpha]\ \phi$

Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic

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

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

Dynamic Logic Example Formulas

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

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

Dynamic Logic Example Formulas

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

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

- Program formulas can appear nested

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

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

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

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

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

Dynamic Logic Example Formulas

```
  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

Validity

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

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

Dynamic Logic

Java Card DL

Sequent Calculus

Rules for Programs: Symbolic Execution

A Calculus for 100% Java Card

Part III

Dynamic Logic

Java Card DL

Sequent Calculus

Rules for Programs: Symbolic Execution

A Calculus for 100% Java Card

Sequents and their Semantics

Syntax

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

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

Sequents and their Semantics

Syntax

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

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

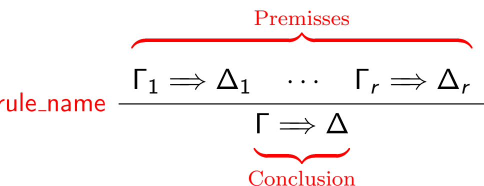
Semantics

Same as the formula

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

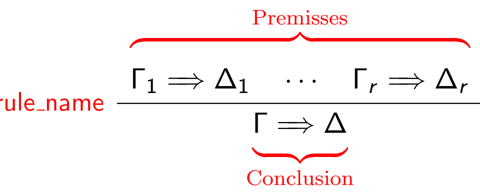
Sequent Rules

General form



Sequent Rules

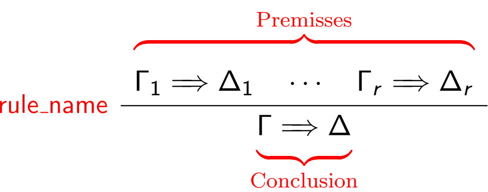
General form



($r = 0$ possible: closing rules)

Sequent Rules

General form



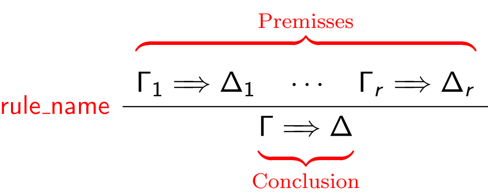
($r = 0$ possible: closing rules)

Soundness

If all premisses are valid, then the conclusion is valid

Sequent Rules

General form



($r = 0$ 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

$$\text{not_left} \quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

Some Simple Sequent Rules

$$\text{not_left} \quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

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

Some Simple Sequent Rules

$$\text{not_left} \quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

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

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

Some Simple Sequent Rules

$$\text{not_left} \quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

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

$$\text{close_goal} \quad \frac{}{\Gamma, A \Longrightarrow A, \Delta} \qquad \text{close_by_true} \quad \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

Some Simple Sequent Rules

$$\text{not_left} \quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

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

$$\text{close_goal} \quad \frac{}{\Gamma, A \Longrightarrow A, \Delta} \qquad\qquad \text{close_by_true} \quad \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

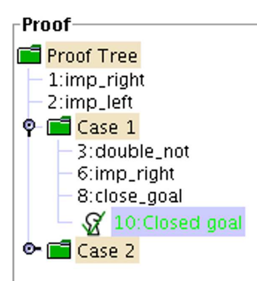
$$\text{all_left} \quad \frac{\Gamma, \backslash\texttt{forall}\ t\ x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\texttt{forall}\ t\ x; \phi \Longrightarrow \Delta}$$

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

Sequent Calculus Proofs

Proof tree

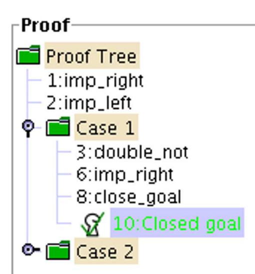
- Proof is tree structure with goal sequent as root



Sequent Calculus Proofs

Proof tree

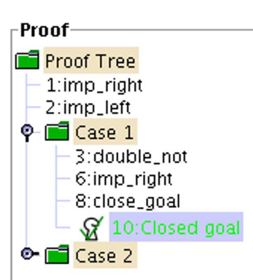
- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)



Sequent Calculus Proofs

Proof tree

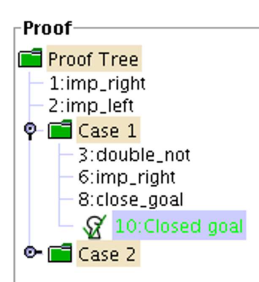
- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch



Sequent Calculus Proofs

Proof tree

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



Part III

Dynamic Logic

Java Card DL

Sequent Calculus

Rules for Programs: Symbolic Execution

A Calculus for 100% Java Card

Part III

Dynamic Logic

Java Card DL

Sequent Calculus

Rules for Programs: Symbolic Execution

A Calculus for 100% Java Card

Proof by Symbolic Program Execution

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

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; }}
```

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; }}
```

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

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

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

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.

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

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

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

Extending DL by Explicit State Updates

Updates

explicit syntactic elements in the logic

Extending DL by Explicit State Updates

Updates

explicit syntactic elements in the logic

Elementary Updates

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

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

Extending DL by Explicit State Updates

Updates

explicit syntactic elements in the logic

Elementary Updates

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

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

$$\{\mathit{loc}_1 := t_1 \mid\mid \cdots \mid\mid \mathit{loc}_n := t_n\}\,\phi$$

no dependency between the n components (but 'right wins' semantics)

Why Updates?

Updates are:

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

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)

$\Longrightarrow$ $x < y$ -> `<int t=x; x=y; y=t;>` $y < x$

Symbolic Execution with Updates (by Example)

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

$$\vdots$$

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

Symbolic Execution with Updates (by Example)

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

$$\vdots$$

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

$$\vdots$$

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

Symbolic Execution with Updates (by Example)

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

$$\vdots$$

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

$$\vdots$$

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

$$\vdots$$

$$\implies \mathtt{x} < \mathtt{y} \mathbin{\texttt{->}} \mathtt{<int\ t=x;\ x=y;\ y=t;>}\ \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates (by Example)

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

$$\vdots$$

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

$$\vdots$$

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

$$\vdots$$

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

$$\vdots$$

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

Symbolic Execution with Updates (by Example)

$$\mathtt{x} < \mathtt{y} \;\Longrightarrow\; \{\mathtt{x:=y} \,\|\, \mathtt{y:=x}\}\texttt{<>}\; \mathtt{y} < \mathtt{x}$$

$$\vdots$$

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

$$\vdots$$

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

$$\vdots$$

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

$$\vdots$$

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

$$\vdots$$

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

Symbolic Execution with Updates (by Example)

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

$$\vdots$$

$$\texttt{x} < \texttt{y} \implies \{\texttt{x:=y} \,||\, \texttt{y:=x}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \implies \{\texttt{t:=x} \,||\, \texttt{x:=y} \,||\, \texttt{y:=x}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \implies \{\texttt{t:=x} \,||\, \texttt{x:=y}\}\{\texttt{y:=t}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

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

$$\vdots$$

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

$$\vdots$$

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

Program State Representation

Local program variables

Modeled as non-rigid constants

Program State Representation

Local program variables

Modeled as non-rigid constants

Heap

Modeled with theory of arrays:

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

Program State Representation

Local program variables

Modeled as non-rigid constants

Heap

Modeled with theory of arrays:

$heap: \quad \rightarrow Heap$ (the heap in the current state)
$select: \quad Heap \times Object \times Field \rightarrow Any$
$store: \quad Heap \times Object \times Field \times Any \rightarrow Heap$

Heap axioms (excerpt)

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

Handling Abrupt Termination

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

Example

TRY-THROW

$$\Gamma \Longrightarrow \langle \texttt{try\{throw exc; q\} catch(T e)\{r\} finally\{s\}}\ \omega \rangle \phi, \Delta$$

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

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

Part III

Dynamic Logic

Java Card DL

Sequent Calculus

Rules for Programs: Symbolic Execution

A Calculus for 100% Java Card

Part III

Dynamic Logic

Java Card DL

Sequent Calculus

Rules for Programs: Symbolic Execution

A Calculus for 100% Java Card

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

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

Pro: Feature needs not be handled in calculus
Contra: Modified source code
Example in KeY: Very rare: treating inner classes

Java—A Language of Many Features

Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly

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

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

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

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

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

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

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`

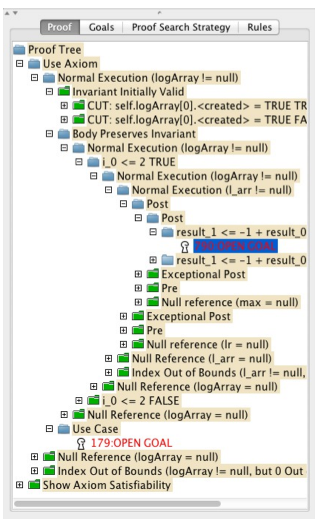
Part IV

Proof Construction

Taclets – KeY's Rule Description Language

Taclets ...

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



Taclet Syntax—Basics

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

Taclet Syntax—Basics

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

Taclet

```
andLeft {

};
```

- Unique name

Taclet Syntax—Basics

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

Taclet

```
andLeft {
    \find ( A & B ==> )


};
```

- Unique name
- Find expression:
  - Formula (Term) to be modified

Taclet Syntax—Basics

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

Taclet

```
andLeft  {
    \find  ( 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.

Taclet Syntax—Basics

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

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—Basics (II)

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 {

};

Taclet Syntax—Basics (II)

Some rules are only sound in a certain context

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

Taclet

```
modusPonens {


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

Taclet Syntax—Basics (II)

Some rules are only sound in a certain context

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

Taclet

```
modusPonens {
  \assumes (A ==>)

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

Taclet Syntax—Basics (II)

Some rules are only sound in a certain context

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

Taclet

```
modusPonens {
  \assumes (A ==>)

  \find  ( A -> B   ==>)
  \add              (B   ==>)
};
```

Taclet Syntax—Basics (III)

Proof Splitting: andRight

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

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

Taclet Syntax—Basics (III)

Proof Splitting: andRight

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

```
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 (==> ∀x; φ)
  \varcond(\new(c,\dependingOn(Φ)))
  \replacewith (==> {\subst x; c }Φ)
};
```

Well-Formed Taclets

Syntactic checks avoid common user errors

- prevent introduction of free variables

```
\find (\forall v;(A & B) ==>)

\replacewith (A & \forall v;B ==>)
```

Well-Formed Taclets

Syntactic checks avoid common user errors

- prevent introduction of free variables

```
\find (\forall v;(A & B) ==>)
\varcond(\notFreeIn(v, A))
\replacewith (A & \forall v;B ==>)
```

Well-Formed Taclets

Syntactic checks avoid common user errors

- prevent introduction of free variables

```
\find (\forall v;(A & B) ==>)
\varcond(\notFreeIn(v, A))
\replacewith (A & \forall v;B ==>)
```

- accidental capturing of variables

Well-Formed Taclets

Syntactic checks avoid common user errors

- prevent introduction of free variables

```
\find (\forall v;(A & B) ==>)
\varcond(\notFreeIn(v, A))
\replacewith (A & \forall v;B ==>)
```

- accidental capturing of variables
- prevent ambiguous variable binding

Well-Formed Taclets

Syntactic checks avoid common user errors

- prevent introduction of free variables

```
\find (\forall v;(A & B) ==>)
\varcond(\notFreeIn(v, A))
\replacewith (A & \forall v;B ==>)
```

- accidental capturing of variables
- prevent ambiguous variable binding

Limited reflection support for non-program taclets

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 )
```

Interactive Proof Construction: Demo

Taclets can be compiled into GUI

- Highlighted term matched against all find expressions
- Successful match returns (partial) instantiation of schema variables
- Possibly missing instantiations
  - user provided (dialog box)
  - drag and drop
- User can select from matching rules which to apply
- Application generates new sequents (goals) and appends them on the proof tree (*explicit proof object*)

Automated Proof Construction

- Automated proof search strategies implemented in Java
- Automated built-in prover uses same datastructure as interactive prover
  - Produces complete proof trace browsable by user
  - User can take over at any time
- Highlights (built-in prover):
  - Instantiation-based quantifier instantiation (triggers adopted from Simplify)
  - Linear and non-linear arithmetic reasoning (e.g., Groebner bases computation,Fourier-Motzkin, omega test etc.)
- Can export sub problems to SMT provers

Part V

Software Engineering Applications

Further Usage of Verification Technology

- Verification performs deep *Program Analysis*
- Information in (partial) proofs usable for other purposes

Further Usage of Verification Technology

- Verification performs deep *Program Analysis*
- Information in (partial) proofs usable for other purposes
- we use (partial) proofs for

Further Usage of Verification Technology

- Verification performs deep *Program Analysis*
- Information in (partial) proofs usable for other purposes
- we use (partial) proofs for
  1. Test Case Generation

Further Usage of Verification Technology

- Verification performs deep *Program Analysis*
- Information in (partial) proofs usable for other purposes
- we use (partial) proofs for
  1. Test Case Generation
  2. Symbolic Debugging

Verification Driven Test Generation

- Specification- and code-based approach

Verification Driven Test Generation

- Specification- and code-based approach
- Achieve strong hybrid coverage criteria

Verification Driven Test Generation

- Specification- and code-based approach
- Achieve strong hybrid coverage criteria
- Exploit strong correspondence:
  proof branches $\leftrightarrow$ program execution paths

Verification Driven Test Generation

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

Using Symbolic Execution and Updates

$\mathit{PRE} \implies$ `<x=x+y; y=x-y; x=x-y; if(x>2*y)` α `else` β`>` POST

Using Symbolic Execution and Updates

$$\mathit{PRE} \implies \{\texttt{x:=y} \parallel \texttt{y:=x}\}\texttt{<if(x>2*y)}\ \alpha\ \texttt{else}\ \beta\texttt{>}\ \mathit{POST}$$

⋮

$$\mathit{PRE} \implies \texttt{<x=x+y; y=x-y; x=x-y; if(x>2*y)}\ \alpha\ \texttt{else}\ \beta\texttt{>}\ \mathit{POST}$$

Using Symbolic Execution and Updates

$$PRE, \texttt{y}>\texttt{2*x} \implies \{\texttt{x:=y} \parallel \texttt{y:=x}\}\texttt{<}\alpha\texttt{>}\, POST$$

$$PRE \implies \{\texttt{x:=y} \parallel \texttt{y:=x}\}\texttt{<if(x}>\texttt{2*y)}\ \alpha\ \texttt{else}\ \beta\texttt{>}\, POST$$

$$PRE \implies \texttt{<x=x+y; y=x-y; x=x-y; if(x}>\texttt{2*y)}\ \alpha\ \texttt{else}\ \beta\texttt{>}\, POST$$

Using Symbolic Execution and Updates

$$PRE, \mathtt{y}\leq\mathtt{2*x} \implies \{\mathtt{x:=y} \mid\mid \mathtt{y:=x}\}\mathtt{<}\beta\mathtt{>}\ POST$$

$$PRE, \mathtt{y}>\mathtt{2*x} \implies \{\mathtt{x:=y} \mid\mid \mathtt{y:=x}\}\mathtt{<}\alpha\mathtt{>}\ POST$$

$$PRE \implies \{\mathtt{x:=y} \mid\mid \mathtt{y:=x}\}\mathtt{<if(x}>\mathtt{2*y)}\ \alpha\ \mathtt{else}\ \beta\mathtt{>}\ POST$$

$$PRE \implies \mathtt{<x=x+y;\ y=x-y;\ x=x-y;\ if(x}>\mathtt{2*y)}\ \alpha\ \mathtt{else}\ \beta\mathtt{>}\ POST$$

From (partial) Proofs to Test Cases

1. Extract 'Execution Tree'
   (stripped down proof tree)

From (partial) Proofs to Test Cases

1. Extract 'Execution Tree'
   (stripped down proof tree)

2. For each leaf, generate at least one test
   - Isolate input constraint

From (partial) Proofs to Test Cases

1. Extract 'Execution Tree'
   (stripped down proof tree)

2. For each leaf, generate at least one test
   - Isolate input constraint
   - Generate data satisfying the constraint
     (Model Generation, SMT solving)

From (partial) Proofs to Test Cases

1. Extract 'Execution Tree'
   (stripped down proof tree)

2. For each leaf, generate at least one test
   - Isolate input constraint
   - Generate data satisfying the constraint
     (Model Generation, SMT solving)
   - Construct test input(s) in JUnit format

Postcondition not used for Generating Test Inputs

$PRE, \mathtt{y} \leq \mathtt{2*x} \implies \{\mathtt{x:=y} \mid\mid \mathtt{y:=x}\}\mathtt{<}\beta\mathtt{>}\ POST$

$PRE, \mathtt{y} > \mathtt{2*x} \implies \{\mathtt{x:=y} \mid\mid \mathtt{y:=x}\}\mathtt{<}\alpha\mathtt{>}\ POST$

$PRE \implies \{\mathtt{x:=y} \mid\mid \mathtt{y:=x}\}\mathtt{<if(x>2*y)}\ \alpha\ \mathtt{else}\ \beta\mathtt{>}\ POST$

$PRE \implies \mathtt{<x=x+y;\ y=x-y;\ x=x-y;\ if(x>2*y)}\ \alpha\ \mathtt{else}\ \beta\mathtt{>}\ POST$

Postcondition not used for Generating Test Inputs

$PRE, \texttt{y} \leq \texttt{2*x} \implies \{\texttt{x:=y} \,||\, \texttt{y:=x}\}\texttt{<}\beta\texttt{>}\ \textit{true}$

$PRE, \texttt{y} > \texttt{2*x} \implies \{\texttt{x:=y} \,||\, \texttt{y:=x}\}\texttt{<}\alpha\texttt{>}\ \textit{true}$

$PRE \implies \{\texttt{x:=y} \,||\, \texttt{y:=x}\}\texttt{<if(x>2*y)}\ \alpha\ \texttt{else}\ \beta\texttt{>}\ \textit{true}$

$PRE \implies \texttt{<x=x+y; y=x-y; x=x-y; if(x>2*y)}\ \alpha\ \texttt{else}\ \beta\texttt{>}\ \textit{true}$

Test Oracle Generation

Test Oracle program routine to check test success or failure

Test Oracle Generation

Test Oracle program routine to check test success or failure

Traditionally specific for each test case

Test Oracle Generation

Test Oracle program routine to check test success or failure

Traditionally specific for each test case

Desired one generic oracle per method

Test Oracle Generation

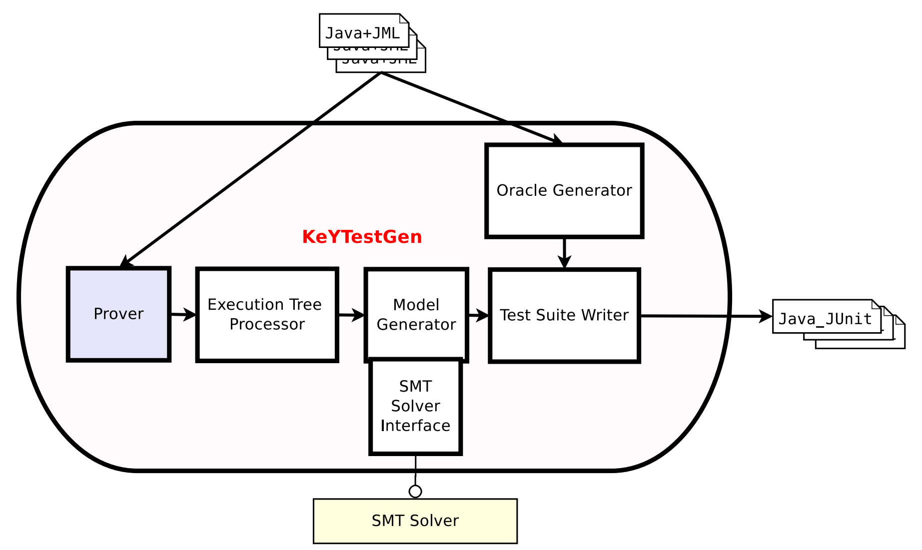
Test Oracle program routine to check test success or failure

Traditionally specific for each test case

Desired one generic oracle per method

KeYTestGen constructs generic oracle from post-condition of tested code

Architecture of KeYTestGen



Eclipse integration

- Eclipse plugin for KeYTestGen (we)

Eclipse integration

- Eclipse plugin for KeYTestGen (we)
- Eclipse plugin for JUnit (standard)

Achieved Coverage Criteria

- Structural Coverage Criteria

Achieved Coverage Criteria

- Structural Coverage Criteria
  - Statement Coverage

Achieved Coverage Criteria

- Structural Coverage Criteria
  - Statement Coverage
  - Branch Coverage

Achieved Coverage Criteria

- Structural Coverage Criteria
  - Statement Coverage
  - Branch Coverage
- Logical Coverage Criteria

Achieved Coverage Criteria

- Structural Coverage Criteria
  - Statement Coverage
  - Branch Coverage
- Logical Coverage Criteria
  - Top level Boolean Decision Coverage (DC)

Achieved Coverage Criteria

- Structural Coverage Criteria
  - Statement Coverage
  - Branch Coverage
- Logical Coverage Criteria
  - Top level Boolean Decision Coverage (DC)
  - Atomic Boolean Condition Coverage (CC)

Achieved Coverage Criteria

- Structural Coverage Criteria
  - Statement Coverage
  - Branch Coverage
- Logical Coverage Criteria
  - Top level Boolean Decision Coverage (DC)
  - Atomic Boolean Condition Coverage (CC)
  - Multiple Boolean Condition Coverage (MCC)
    (all possible combinations)

Achieved Coverage Criteria

- Structural Coverage Criteria
  - Statement Coverage
  - Branch Coverage
- Logical Coverage Criteria
  - Top level Boolean Decision Coverage (DC)
  - Atomic Boolean Condition Coverage (CC)
  - Multiple Boolean Condition Coverage (MCC)
    (all possible combinations)
  - Modified Condition/Decision Coverage (MCDC)
    (required in Avionics certification standards)

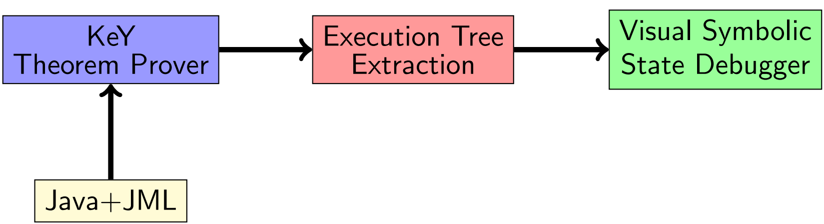
Test Generation Case Study

- applied to implementations of 40 methods in Real-Time Java API

Test Generation Case Study

- applied to implementations of 40 methods in Real-Time Java API
- KeYTestGen produced a failing test case,
  i.e., detected a bug in Real-Time Java library code

Visual Symbolic State Debugger



- Symbolic execution tree extraction (as for KeYTestGen)
- Enhanced omniscient debugging
  - enhanced: *all* possible execution paths
  - classic debugger navigation
  - variable inspection: symbolic variable values
  - semantic watchpoints
- Heap visualization

Part VI

Wrap Up

Further Directions of Current Research in KeY

Extending the scope of verification

- Concurrency and distribution

Further Directions of Current Research in KeY

Extending the scope of verification

- Concurrency and distribution
- Information-flow properties

Further Directions of Current Research in KeY

Extending the scope of verification

- Concurrency and distribution
- Information-flow properties
- Floating-point arithmetic

Further 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)

Further 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)

Further 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

Further 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

Further 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

Further 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

Modelling and specification

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

Further 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

Modelling and specification

- Modular specification of heap structures
  dynamic frames, abstract data types
- Compositional models of concurrency and distribution

Further 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

Modelling and specification

- Modular specification of heap structures
  dynamic frames, abstract data types
- Compositional models of concurrency and distribution
- Refinement

Further 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

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

Different Approaches Software Verification

General Purpose Systems

Different Approaches Software Verification

General Purpose Systems

- General purpose

Different Approaches Software Verification

General Purpose Systems

- General purpose
- Elaborate support for theories, abstract data types

Different Approaches Software Verification

General Purpose Systems

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

Different Approaches 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

Different Approaches 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

Different Approaches 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

Different Approaches 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

Different Approaches 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

Security Case Studies: Java Card Software

Safety/security properties specified in dynamic logic

- 'Only certain exceptions can be thrown'
- Transactions are properly used
  (do not commit or abort a transaction that was never started,
  all started Transactions are also closed)
- Data consistency
  (also if a smartcard is "ripped out" during operation)
- Absence of overflows for integer operations

Two studies in this area (for which some critical parts were verified)

- Demoney (about 3000 lines):
  Electronic purse application provided by Trusted Logic S.A.
- SafeApplet (about 600 lines): RSA based authentication applet

Safety Case Study

Computation of Railway Speed Restrictions

- Software by DBSystems for computing schedules for train drivers: Speed restrictions, required break powers
- Software formally specified using UML/OCL (based on existing informal specification)
- Program translated from Smalltalk to JAVA

Avionics Software

- JAVA implementation of a Flight Manager module at Thales Avionics
- Comprehensive specification using JML, emphasis on class invariants
- Verification of some nested method calls using contracts

Virtual Machine for Real Time Secure Java

- Verification of some library functions of the Jamaica VM from Aicas