Formal Specification And Verification

Winter 2010/2011
Prof. P. H. Schmitt
Formal Semantics for JML
Goals: What, How, Why?

- Provide a precise mathematical semantics of JML independent of any specific logic formalism
- Independent of a formalization of Java semantics
- As a basis for a common understanding

Quote from JML Reference Manual p.1

We include both informal (intentions) and where possible [we will eventually include] formal semantics (describing when an implementation satisfies a specification).

Challenge

Bridge the Gap between Logical and Program Semantics

Formal Specification And Verification
Goals: What, How, Why?

- Provide a precise mathematical semantics of JML
Goals: What, How, Why?

- Provide a precise mathematical semantics of JML
- independent of any specific logic formalism

Quote from JML Reference Manual p.1

We include both informal (intentions) and where possible [[we will eventually include]] formal semantics (describing when an implementation satisfies a specification)
Goals: What, How, Why?

- Provide a precise mathematical semantics of JML
- independent of any specific logic formalism
- independent of a formalization of Java semantics
Goals: What, How, Why?

- Provide a precise mathematical semantics of JML
- independent of any specific logic formalism
- independent of a formalization of Java semantics
- as a basis for a common understanding
Goals: What, How, Why?

▶ Provide a precise mathematical semantics of JML
▶ independent of any specific logic formalism
▶ independent of a formalization of Java semantics
▶ as a basis for a common understanding
▶ Quote from JML Reference Manual p.1
Goals: What, How, Why?

▶ Provide a precise mathematical semantics of JML
▶ independent of any specific logic formalism
▶ independent of a formalization of Java semantics
▶ as a basis for a common understanding
▶ Quote from JML Reference Manual p.1

*We include both informal (intentions) and where possible [[we will eventually include]] formal semantics (describing when an implementation satisfies a specification)*
Goals: What, How, Why?

- Provide a precise mathematical semantics of JML
- independent of any specific logic formalism
- independent of a formalization of Java semantics
- as a basis for a common understanding
- Quote from JML Reference Manual p.1

We include both informal (intentions) and where possible [[we will eventually include]] formal semantics (describing when an implementation satisfies a specification)

Challenge

Bridge the Gap between Logical and Program Semantics
A First Method Specification Example
Meaning Of A Contract

A contract

/*@ public normal_behavior
  @ requires pre;
  @ ensures post;
  @*/
mod void meth(p1,..,pk) {...}

is satisfied
A contract

```c
/*@ public normal_behavior
   @ requires pre;
   @ ensures post;
   @*/
mod void meth(p1,..,pk) {...}
```

is satisfied if for any state $s_1$
A contract

```java
/*@ public normal_behavior
    @ requires pre;
    @ ensures post;
    @*/
mod void meth(p1,..,pk) {...}
```

is satisfied if for any state $s_1$

- if the preconditions $pre$ are satisfied in $s_1$
Meaning Of A Contract

A contract

```java
/*@ public normal_behavior
  @ requires pre;
  @ ensures post;
  @*/
mod void meth(p1, .., pk) {...}
```

is satisfied if for any state $s_1$

- if the preconditions $\text{pre}$ are satisfied in $s_1$
- then the method $\text{meth}$ terminates, say in state $s_2$, and $s_2$ satisfies the postconditions $\text{post}$ and no exception is thrown.
Meaning Of A Contract

A contract

```java
/*@ public normal_behavior
 @ requires pre;
 @ ensures post;
 @*/
mod void meth(p1,..,pk) {...}
```

is satisfied if for any state $s_1$

if the preconditions $pre$ are satisfied in $s_1$

then the method $meth$ terminates, say in state $s_2$,

and $s_2$ satisfies the postconditions $post$

and no exception is thrown.

$\Rightarrow s_1$ is the state after passing of the parameters $p1, \ldots, pk$. 
Meaning Of A Contract

A contract

```java
/*@ public normal_behavior
  @ requires pre;
  @ ensures post;
  @*/
mod void meth(p1,..,pk) {...}
```

is satisfied if for any state $s_1$

- if the preconditions $pre$ are satisfied in $s_1$
- then the method $meth$ terminates, say in state $s_2$,
  and $s_2$ satisfies the postconditions $post$
  and no exception is thrown.

$\Rightarrow$ $s_1$ is the state after passing of the parameters $p_1, \ldots, p_k$.
$\Rightarrow$ Exceptions may be thrown as long as they are caught
Meaning Of A Contract

A contract

```java
/*@ public normal_behavior
    @ requires pre;
    @ ensures post;
    @*/

mod void meth(p1,..,pk) {...}
```

is satisfied if for any state \( s_1 \)

if the preconditions \( pre \) are satisfied in \( s_1 \)
then the method \( meth \) terminates, say in state \( s_2 \),
and \( s_2 \) satisfies the postconditions \( post \)
and no exception is thrown.

What happens if the precondition is not satisfied?
public class P3{
    public int p1, p2, p3;
    /*@ public normal_behavior
       @ requires p1!=p2 && p1!=p3 && p2!=p3;
       @ ensures result==p1 || result==p2 || result==p3;
       @ ensures result>p1 || result>p2 || result>p3;
       @ ensures result<p1 || result<p2 || result<p3;
       @ assignable nothing;
       @*/
    public int middle(){ }
}
public class P3{
  public int p1, p2, p3;
  /*@ public normal_behavior @ . . . . . */
  @* /
  public int middle() {
    int m = 0;
    boolean c12, c23, c13;
    c12 = p1 < p2; c13 = p1 < p3; c23 = p2 < p3;
    if (((!c12 || !c13) && (c12 || c13)) {m=p1;})
    if (((!c23 || c12) && (c23 || !c12)) {m=p2;})
    if (((c23 || c13) && (!c13 || !c23)) {m=p3;})
    return m;
  }
}
A Call Site of the `middle()` Method

```java
/*@ public normal_behavior
@ ensures \result \geq \text{p1} || \result \geq \text{p2} || \result \geq \text{p3};
@ ensures \result \leq \text{p1} || \result \leq \text{p2} || \result \leq \text{p3};
@ assignable \nothing ;
@*/

public int mid () {
    int m=0;
    if (p1 == p2 || p1 == p3) {m=p1;}
    else {if (p2 == p3) {m=p2;}
      else {m = middle();}}}
    return m;
}
```
A Call Site of the `middle()` Method

```java
/*@ public normal_behavior
 @ ensures \result\geq p1 || \result\geq p2 || \result\geq p3;
 @ ensures \result\leq p1 || \result\leq p2 || \result\leq p3;
 @ assignable \nothing;
 @*/

public int mid() {
    int m=0;
    if (p1 == p2 || p1 == p3) {m=p1;}
    else { if (p2 == p3) {m=p2;}
    else { m = middle();} }
    return m;
}
```

Contract for method `mid()` is satisfied, since when `middle()` is called its precondition is satisfied.
The AddPos Class

```java
public class AddPos{
    public int a, b, end;

    /** @ public normal_behavior
        @ requires a >= 0 && b >= 0;
        @ ensures \result >= 0;
        @ assignable \nothing;
        @*/
    public int add(int a, int b){
        return a + b;
    }
}
```
public class AddPos{
    public int a, b, end;
    public int add(int a, int b){ ... }
    /*@ public normal_behavior
    @ requires 0 <= d && d <= 2 && a>0;
    @ ensures end >= 0;
    @ assignable end;
    @*/
    public void discount(int d){
        if (d==0){b = 0;}
        if (d==1){b = -(a/3);}
        if (d==2){b = -(a/2);}
        end = add(a, b);
    }
}
public class AddPos{
    public int a,b,end;
    public int add(int a,int b){ ... }
    /**@ public normal_behavior
       @ requires 0 <= d && d <= 2 && a>0;
       @ ensures end >= 0;
       @ assignable end;
       @*/
    public void discount(int d){ if (d==0){b = 0;}
                                if (d==1){b = -(a/3);}
                                if (d==2){b = -(a/2);}
                                end = add(a,b);}
}

Contract of discount is not satisfied, since when add is called its precondition is not satisfied.
The AddPos Class (Continued)

```java
public class AddPos{
    public int a, b, end;
    public int add(int a, int b){  
        . . .
    }  
    /*@ public normal_behavior
       @ requires 0 <= d && d <= 2 && a > 0;
       @ ensures end >= 0;
       @ assignable end;
       @*/
    public void discount(int d){  
        if (d==0){b = 0;}
        if (d==1){b = -(a/3);}
        if (d==2){b = -(a/2);}
        end = add(a, b);
    }
    }
}
```

Contract of `discount` is not satisfied, since when `add` is called its precondition is not satisfied. **Blocking contract semantics**
The AddPos Class (Continued)

```java
public class AddPos{
    public int a, b, end;
    public int add(int a, int b){ ... }
    /*@ public normal_behavior
    @ requires 0 <= d && d <= 2 && a>0;
    @ ensures end >= 0;
    @ assignable end;
    @*/
    public void discount(int d){
        if (d==0){b = 0;}
        if (d==1){b = -(a/3);}
        if (d==2){b = -(a/2);}
        end = add(a, b);
    }
}
```

If we replace the call to `add` by its code the contract of `discount` is satisfied.
The AddPos Class (Continued)

```java
public class AddPos {
    public int a, b, end;
    public int add(int a, int b) {
        ... }
    } /*@ public normal_behavior
        @ requires 0 <= d && d <= 2 && a > 0;
        @ ensures end >= 0;
        @ assignable end;
        @*/
    public void discount(int d) {
        if (d == 0) { b = 0; }
        if (d == 1) { b = -(a / 3); }
        if (d == 2) { b = -(a / 2); }
        end = add(a, b); }
}
```

If we replace the call to `add` by its code the contract of `discount` is satisfied. **Action:** Improve contract for `add`
States
General Definition

A state $s$ is a function that

1. maps any type symbol $T$ into a set of values $T^s$, 

2. for any field $f$ of $T_1$ with type $T_2$ and any element $o \in T^s_1$ maps the pair $(o, f)$ into $(o, f) \in T^s_2$, 

3. maps local variables and method parameters $v$ of declared type $T$ to $v \in T^s$, 

4. provides a value callstack $s$ that is a finite sequence of pairs $(\mu, r)$ with $\mu$ a method instantiation and $r$ an object. The entry $\mu$ consists of a method name, its signature (parameter types and return type), the class $T$ that contains the code to be executed (thus dynamic dispatch is already taken care of). If $\mu$ is a non-static method then $r \in T^s$ is the calling object, in case $\mu$ is a constructor $r$ is the generated object. If $\mu$ is a static method then $r$ is missing.
A state \( s \) is a function that

1. maps any type symbol \( T \) into a set of values \( T^s \),

2. for any field \( f \) of \( T_1 \) with type \( T_2 \) and any element \( o \in T_1^s \) maps the pair \( (o,f) \) into \( (o.f)^s \in T_2^s \).
General Definition

A state $s$ is a function that

1. maps any type symbol $T$ into a set of values $T^s$,

2. for any field $f$ of $T_1$ with type $T_2$ and any element $o \in T^s_1$ maps the pair $(o,f)$ into $(o.f)^s \in T^s_2$,

3. maps local variables and method parameters $v$ of declared type $T$ to $v^s \in T^s$. 

Formal Specification And Verification
General Definition

A state \( s \) is a function that

1. maps any type symbol \( T \) into a set of values \( T^s \),

2. for any field \( f \) of \( T_1 \) with type \( T_2 \) and any element \( o \in T_1^s \) maps the pair \((o, f)\) into \((o.f)^s \in T_2^s\),

3. maps local variables and method parameters \( v \) of declared type \( T \) to \( v^s \in T^s \).

4. provides a value callstack\(^s\) that is a finite sequence of pairs \((\mu, r)\) with \( \mu \) a method instantiation and \( r \) an object. The entry \( \mu \) consists of a method name, its signature (parameter types and return type), the class \( T \) that contains the code to be executed (thus dynamic dispatch is already taken care of). If \( \mu \) is a non-static method then \( r \in T^s \) is the calling object, in case \( \mu \) is a constructor \( r \) is the generated object. If \( \mu \) is a static method then \( r \) is missing.
A state $s$ is a function that

1. maps any type symbol $T$ into a set of values $T^s$,

2. for any field $f$ of $T_1$ with type $T_2$ and any element $o \in T_1^s$ maps the pair $(o, f)$ into $(o.f)^s \in T_2^s$, heap

3. maps local variables and method parameters $v$ of declared type $T$ to $v^s \in T^s$.

4. provides a value callstack$^s$ that is a finite sequence of pairs $(\mu, r)$ with $\mu$ a method instantiation and $r$ an object. The entry $\mu$ consists of a method name, its signature (parameter types and return type), the class $T$ that contains the code to be executed (thus dynamic dispatch is already taken care of). If $\mu$ is a non-static method then $r \in T^s$ is the calling object, in case $\mu$ is a constructor $r$ is the generated object. If $\mu$ is a static method then $r$ is missing.
A state $s$ is a function that

1. maps any type symbol $T$ into a set of values $T^s$,

2. for any field $f$ of $T_1$ with type $T_2$ and any element $o \in T_1^s$ maps the pair $(o,f)$ into $(o.f)^s \in T_2^s$, heap

3. maps local variables and method parameters $v$ of declared type $T$ to $v^s \in T^s$. stack

4. provides a value callstack$^s$ that is a finite sequence of pairs $(\mu, r)$ with $\mu$ a method instantiation and $r$ an object. The entry $\mu$ consists of a method name, its signature (parameter types and return type), the class $T$ that contains the code to be executed (thus dynamic dispatch is already taken care of). If $\mu$ is a non-static method then $r \in T^s$ is the calling object, in case $\mu$ is a constructor $r$ is the generated object. If $\mu$ is a static method then $r$ is missing.
Type Restrictions

1. For a primitive type $T$ the value set $T^s$ is as described in the Java language specification. E.g.; $long^s = \{-2^{63}, \ldots, +2^{63} - 1\}$. Thus $T^s$ is independent of $s$ and we write $V_T$. 

Formal Specification And Verification
Type Restrictions

1. For a primitive type $T$ the value set $T^s$ is as described in the Java language specification. E.g.; $long^s = \{-2^{63}, \ldots, +2^{63} - 1\}$. Thus $T^s$ is independent of $s$ and we write $V_T$.

2. For the JML model class $\text{\texttt{bigint}}$ we have $T^s_{\text{\texttt{bigint}}} = \mathbb{Z}$. 

Formal Specification And Verification
Type Restrictions

1. For a primitive type $T$ the value set $T^s$ is as described in the Java language specification. E.g.; $long^s = \{-2^{63}, \ldots, +2^{63} - 1\}$. Thus $T^s$ is independent of $s$ and we write $V_T$.

2. For the JML model class \texttt{bigint} we have $T_{\texttt{bigint}}^s = \mathbb{Z}$.

3. If $T_1$ is a subtype of $T_2$ ($T_1 \subseteq T_2$) then $T_1^s \subseteq T_2^s$. 
Type Restrictions

1. For a primitive type $T$ the value set $T^s$ is as described in the Java language specification. E.g.; $long^s = \{-2^{63}, \ldots, +2^{63} - 1\}$. Thus $T^s$ is independent of $s$ and we write $V_T$.

2. For the JML model class $\texttt{bigint}$ we have $T_{\texttt{bigint}}^s = \mathbb{Z}$.

3. If $T_1$ is a subtype of $T_2$ ($T_1 \subseteq T_2$) then $T_1^s \subseteq T_2^s$.

4. For every reference type $T$ the values set $T^s$ is infinite. It is a reservoir of all objects that could be generated. $\texttt{created}$ is an implicit field in type $\texttt{Object}$ with the intention that
$$\{o \in T^s \mid (o.\texttt{created})^s = \text{true}\}$$

is the set of objects that exist in state $s$. 
Type Restrictions

1. For a primitive type $T$ the value set $T^s$ is as described in the Java language specification. E.g.; $long^s = \{-2^{63}, \ldots, +2^{63} - 1\}$. Thus $T^s$ is independent of $s$ and we write $V_T$.

2. For the JML model class $\texttt{bigint}$ we have $T^s_{\texttt{bigint}} = \mathbb{Z}$.

3. If $T_1$ is a subtype of $T_2$ ($T_1 \subseteq T_2$) then $T_1^s \subseteq T_2^s$.

4. For every reference type $T$ the values set $T^s$ is infinite. It is a reservoir of all objects that could be generated. $\texttt{created}$ is an implicit field in type $\texttt{Object}$ with the intention that $
\{ o \in T^s \mid (o.\texttt{created})^s = true \}$ is the set of objects that exist in state $s$.

This approach is called the constant domain assumption. As a consequence the set of possible values of a reference type $T$ does not depend on the state $s$ and we may denote it again by $V_T$. 
5. For every reference type $T$ there is a null object $o_{null} \in T^s$ and $null^s = o_{null}$.
Type Restrictions (Continued)

5. For every reference type $T$ there is a null object $o_{\text{null}} \in T^s$ and $\text{null}^s = o_{\text{null}}$.

6. If $T$ is an interface then $o \in T^s$ only if there is a subtype $T_0 \subseteq T$ with $o \in T_0^s$. 

The JML manual (Subsect. 7.1.2.2) introduces a type \textsc{TYPE} whose interpretation is declared to be \texttt{java.lang.Class}. We will not cover \textsc{TYPE} here.
5. For every reference type $T$ there is a null object $o_{null} \in T^s$ and $null^s = o_{null}$.

6. If $T$ is an interface then $o \in T^s$ only if there is a subtype $T_0 \sqsubseteq T$ with $o \in T_0^s$.

7. The JML manual (Subsect. 7.1.2.2) introduces a type $\backslash TYPE$ whose interpretation is declared to be `java.lang.Class`. We will not cover $\backslash TYPE$ here.
Field Restrictions

1. For any array type $T[]$ there are implicit fields `length` and $[n]$ for $n \in \mathbb{N}$.
   For $o \in (T[])^s$, as expected, $o.length^s$ is the length of array $o$ while $o[n]^s$ is the $n$-th entry in $o$. 
Field Restrictions

1. For any array type $T[]$ there are implicit fields $\text{length}$ and $[n]$ for $n \in \mathbb{N}$.
   For $o \in (T[])^s$, as expected, $o.\text{length}^s$ is the length of array $o$ while $o[n]^s$ is the $n$-th entry in $o$.

2. For static fields $f$ in a class $C$ we require $(o_1.f)^s = (o_2.f)^s$ for all $o_1, o_2 \in C^s$. 
Field Restrictions

1. For any array type $T[]$ there are implicit fields $\text{length}$ and $[n]$ for $n \in \mathbb{N}$.
   For $o \in (T[])^s$, as expected, $o\. \text{length}^s$ is the length of array $o$ while $o[n]^s$ is the $n$-th entry in $o$.

2. For static fields $f$ in a class $C$ we require $(o_1.f)^s = (o_2.f)^s$ for all $o_1, o_2 \in C^s$.

3. For objects $o_1, o_2$ if $o_1.f\. \text{created} = true$ then $o_1\. \text{created} = true$. 
Further Restrictions

1. Stack restrictions

1.1 We assume that there is a mechanism that disambiguates names of local variables and method parameters.

2. Call stack restrictions

2.1 For all objects $o$ occurring as receiver object in an entry in the callstack $s$ we have $o$ created.
Further Restrictions

1. Stack restrictions
   1.1 We assume that there is a mechanism that disambiguates names of local variables and methods parameters.
Further Restrictions

1. Stack restrictions
   1.1 We assume that there is a mechanism that disambiguates names of local variables and methods parameters.

2. Call stack restrictions
Further Restrictions

1. Stack restrictions
   1.1 We assume that there is a mechanism that disambiguates names of local variables and methods parameters.

2. Call stack restrictions
   2.1 For all objects $o$ occurring as receiver object in an entry in the sequence $\text{callstack}^s$ we have $o.\backslash created = true$. 
The Java Black Box
Definition

The Java Black Box takes as input a system state \( s \) and piece of JML annotated code \( \pi^A \) assuming that \( \pi^A \) is started in \( s \) and returns
Definition

The Java Black Box takes as input a system state $s$ and piece of JML annotated code $\pi^A$ assuming that $\pi^A$ is started in $s$ and returns

1. An object $\Omega$ indicating

   - abrupt termination, in this case $\Omega$ is the thrown exception object
   - normal termination, in this case $\Omega = \text{null}$

2. The information whether $\pi^A$ terminates when started in $s$ and in the positive case the resulting final state $s_2$

3. Sets $L_{AC}$, $L_{AS}$ of locations that have been accessed, respectively assigned during execution of $\pi^A$

4. A return value $\rho$ in case $\pi^A$ is the implementation of a method with non-void return type and normal termination. If $\pi^A$ is a constructor $\rho$ will hold the newly generated object.

5. A run $R$ of the program $\pi^A$ starting in $s$, i.e., an infinite or finite sequence of states. (needed only in few cases)
Definition

The Java Black Box takes as input a system state $s$ and piece of JML annotated code $\pi^A$ assuming that $\pi^A$ is started in $s$ and returns

1. An object $\Omega$ indicating
   1.1 abrupt termination, in this case $\Omega$ is the thrown exception object
Definition

The Java Black Box takes as input a system state \( s \) and piece of JML annotated code \( \pi^A \) assuming that \( \pi^A \) is started in \( s \) and returns

1. An object \( \Omega \) indicating
   1.1 abrupt termination, in this case \( \Omega \) is the thrown exception object
   1.2 normal termination, in this case \( \Omega = null \)
Definition

The Java Black Box takes as input a system state \( s \) and piece of JML annotated code \( \pi^A \) assuming that \( \pi^A \) is started in \( s \) and returns

1. An object \( \Omega \) indicating
   1.1 abrupt termination, in this case \( \Omega \) is the thrown exception object
   1.2 normal termination, in this case \( \Omega = \text{null} \)

2. The information whether \( \pi^A \) terminates when started in \( s \) and in the positive case the resulting final state \( s_2 \),
Definition

The Java Black Box takes as input a system state $s$ and piece of JML annotated code $\pi^A$ assuming that $\pi^A$ is started in $s$ and returns

1. An object $\Omega$ indicating
   1.1 abrupt termination, in this case $\Omega$ is the thrown exception object
   1.2 normal termination, in this case $\Omega = \text{null}$

2. The information whether $\pi^A$ terminates when started in $s$ and in the positive case the resulting final state $s_2$,

3. Sets $L_{AC}, L_{AS}$ of locations that have been accessed, respectively assigned during execution of $\pi^A$, 
The Java Black Box takes as input a system state \( s \) and piece of JML annotated code \( \pi^A \) assuming that \( \pi^A \) is started in \( s \) and returns

1. An object \( \Omega \) indicating
   1.1 abrupt termination, in this case \( \Omega \) is the thrown exception object
   1.2 normal termination, in this case \( \Omega = null \)

2. The information whether \( \pi^A \) terminates when started in \( s \) and in the positive case the resulting final state \( s_2 \),

3. Sets \( L_{AC}, L_{AS} \) of locations that have been accessed, respectively assigned during execution of \( \pi^A \),

4. A return value \( \rho \) in case \( \pi^A \) is the implementation of a method with non-void return type and normal termination. If \( \pi^A \) is a constructor \( \rho \) will hold the newly generated object.
Definition

The Java Black Box takes as input a system state \( s \) and piece of JML annotated code \( \pi^A \) assuming that \( \pi^A \) is started in \( s \) and returns

1. An object \( \Omega \) indicating
   1.1 abrupt termination, in this case \( \Omega \) is the thrown exception object
   1.2 normal termination, in this case \( \Omega = \text{null} \)

2. The information wether \( \pi^A \) terminates when started in \( s \) and in the positive case the resulting final state \( s_2 \),

3. Sets \( L_{AC}, L_{AS} \) of locations that have been accessed, respectively assigned during execution of \( \pi^A \),

4. A return value \( \rho \) in case \( \pi^A \) is the implementation of a method with non-void return type and normal termination. If \( \pi^A \) is a constructor \( \rho \) will hold the newly generated object.

5. a run \( R \) of the program \( \pi^A \) starting in \( s \), i.e., an infinite or finite sequence of states. (needed only in few cases)
The Java Black Box

Black Box (JVM)
The Java Black Box

Black Box (JVM)

Start State

Code Fragment
The Java Black Box

Start State

Code Fragment

Black Box (JVM)

Stop State

read fields

written fields

result value

Exception

Run
The Java Black Box

Start State

Code Fragment

Black Box (JVM)

ghost fields

Stop State
read fields
written fields
result value
Exception
Run
The Java Black Box

Start State

Code Fragment

model fields

ghost fields

Black Box (JVM)

Stop State
read fields
written fields
result value
Exception
Run

Formal Specification And Verification

19/42
Undefinedness
public class MyUtil {
    /*
     * @ ensures \result == 
     *   @ (a1.length <= a2.length?a1.length:a2.length);
     * @ assignable \nothing;
     */
    public static int minLen(int[] a1, int[] a2);

    /*
     * @ requires n <= a.length;
     * @ ensures \result == (\bsum int i; 0 ; n ; a[i]);
     * @ assignable \nothing;
     */
    public static int sumUpTo(int[] a, int n);
}

public class PairSum {
    /**
     * public normal_behavior
     * @ ensures true;
     */
    public static int pairSum(int[] a, int[] b) {
        int n = MyUtil.minLen(a, b);
        return MyUtil.sumUpTo(a, n) + MyUtil.sumUpTo(b, n);
    }
}
The Class PairSum

```java
public class PairSum {
  /**
   * @public normal_behavior
   *   @ ensures true;
   * @*/
  public static int pairSum(int[] a, int[] b) {
    int n = MyUtil.minLen(a, b);
    return MyUtil.sumUpTo(a, n) + MyUtil.sumUpTo(b, n);
  }
}
```

Using the contract for `minLen` and `sumUpTo` this contract can be proved correct. (ESCJava2, KeY).
public class PairSum {
    /*@ public normal_behavior @ ensures true; @*/
    public static int pairSum(int[] a, int[] b) {
        int n = MyUtil.minLen(a, b);
        return MyUtil.sumUpTo(a, n) + MyUtil.sumUpTo(b, n);
    }
}

Using the contract for minLen and sumUpTo this contract can be proved correct. (ESCJava2, KeY).
But, at runtime it will throw an exception.
The Problem

public class MyUtil {
    /*@ ensures \result ==
        @  (a1.length <= a2.length?a1.length:a2.length);
        @ assignable \nothing;
        @*/
    public static int minLen(int[] a1, int[] a2);
}

Postcondition is undefined if a1 == null or a2 == null.
The Problem

```java
public class MyUtil {
    /*@ ensures \result ==
        @    (a1.length <= a2.length?a1.length:a2.length);
        @ assignable \nothing;
        @*/
    public static int minLen(int[] a1, int[] a2);
}
```

Postcondition is undefined if `a1==null` or `a2==null`. 
Within JML annotations, Java expressions have the values that are defined in the Java Language Specification . . . (This) . . . means that exceptions may arise during evaluation of subexpressions within assertions. These exceptions should be avoided by the specifier and tools are encouraged to warn users when they detect that an exception may arise during assertion evaluation.
Welldefinedness Predicate

\[ \text{wd}_s(a \mid\mid b) \iff a^s = \text{true} \text{ and } \text{wd}_s(a) \text{ or } \text{wd}_s(a) \text{ and } \text{wd}_{s_1}(b) \]

\[ \text{wd}_s(a \&\& b) \iff a^s = \text{false} \text{ and } \text{wd}_s(a) \text{ or } \text{wd}_s(a) \text{ and } \text{wd}_{s_1}(b) \]

\[ \text{wd}_s(x/y) \iff \text{wd}_s(x) \text{ and } \text{wd}_{s_2}(y) \text{ and } y^{s_2} \neq 0 \]
Welldefinedness Predicate

\[
wd_s(a || b) \iff a^s = \text{true} \text{ and } wd_s(a) \text{ or } wd_s(a) \text{ and } wd_{s1}(b)
\]

\[
wd_s(a \&\& b) \iff a^s = \text{false} \text{ and } wd_s(a) \text{ or } wd_s(a) \text{ and } wd_{s1}(b)
\]

\[
wds(x/y) \iff wd_s(x) \text{ and } wds_2(y) \text{ and } y^{s2} \neq 0
\]

Here \( s_1 \) is the state reached after evaluating \( a \) and \( s_2 \) the state after evaluating \( x \).
public class MyUtil2 {
   /*@ public normal_behavior
      @ requires a1!=null && a2!=null;
      @ ensures \result ==
      @ (a1.length <= a2.length?a1.length:a2.length);
      @ assignable \nothing;
      @ also
      @ public exceptional_behavior
      @ requires a1==null || a2==null;
      @ signals_only NullPointerException;
      @*/
   public static int minLen(int[] a1, int[] a2);
}
public class MyUtil2 {
...
/*@ public normal_behavior
  @ requires a!=null && n <= a.length;
  @ ensures \result == (\bsum int i; 0 ; n ; a[i]);
  @ assignable \nothing;
  @ also
  @ public exceptional_behavior
  @ requires a==null || n > a.length;
  @ signals_only NullPointerException ,
  @ ArrayIndexOutOfBoundsException;
  @ signals (NullPointerExc) a == null;
  @ signals (ArrayIndexOutOfBoundsExc) n>a.length;
/*@*/
public static int sumUpTo(int[] a, int n);
}
Expressions with Side Effect
A method is called *weakly pure* if it does not change any of the objects existing at the time the method is called.
A method is called *weakly pure* if it does not change any of the objects existing at the time the method is called.

Thus, a weakly pure method may create new objects.
Weakly Pure Methods

Definition

A method is called *weakly pure* if it does not change any of the objects existing at the time the method is called.

Thus, a weakly pure method may create new objects.

A weakly pure method may even return a newly created object.
Why do we want to use weakly pure methods in JML specifications?
Weakly Pure Method Returning An Object

class Date {
}
class Appointment {
    /**
     * get the next day on which a recurrent appointment is scheduled after a
     * given date. May be complex computation e.g. Easter */
    public /*@pure*/ Date getNextDate(Date from); }

class AppointmentChecker {
    public static final Date TODAY;
    /**
     * @ ensures (\\forall Appointment x;
     * @   \creator(x) && x != null;
     * @   result.contains(x) <=>
     * @   x.getNextDate(TODAY).equals(TODAY));
     * @*/
    public List<Appointment> getTodaysAppointments();
}
The Next State Function

Definition of $\omega$

For any state $s$ and any JML expression $t$

$$\omega_s(t)$$

denotes the state reached after evaluating $t$ in $s$. 
Let $b$ be a JML expression of type Boolean and $c, d$ expressions of compatible type and $s$ a state.
Let $b$ be a JML expression of type Boolean and $c, d$ expressions of compatible type and $s$ a state.

$$ (b?c : d)^s = \begin{cases} 
    c^s & \text{if } b^s = \text{true} \\
    d^s & \text{if } b^s = \text{false}
\end{cases} $$
Let $b$ be a JML expression of type Boolean and $c, d$ expressions of compatible type and $s$ a state.

\[ (b?c : d)^s = \begin{cases} 
  c^s & \text{if } b^s = \text{true} \\
  d^s & \text{if } b^s = \text{false}
\end{cases} \]

\[ \omega_s(b?c : d) \iff \]

Formal Specification And Verification
Let $b$ be a JML expression of type Boolean and $c, d$ expressions of compatible type and $s$ a state.

$$(b?c : d)^s = \begin{cases} 
    c^s & \text{if } b^s = \text{true} \\
    d^s & \text{if } b^s = \text{false}
\end{cases}$$

$\text{wd}_s(b?c : d)$ iff

- $b^s = \text{true}$ and $\text{wd}_s(b)$ and $\text{wd}_{\omega_s(b)}(c)$ or
Let \( b \) be a JML expression of type Boolean and \( c, d \) expressions of compatible type and \( s \) a state.

\[
(b?c:d)^s = \begin{cases} 
  c^s & \text{if } b^s = \text{true} \\
  d^s & \text{if } b^s = \text{false}
\end{cases}
\]

\[
w_d(s)(b?c:d) \text{ iff } \quad
\begin{align*}
  b^s &= \text{true} \text{ and } w_d(s)(b) \text{ and } w_{\omega}(s)(c) \text{ or } \\
  b^s &= \text{false} \text{ and } w_d(s)(b) \text{ and } w_{\omega}(s)(d)
\end{align*}
\]
Let $b$ be a JML expression of type Boolean and $c, d$ expressions of compatible type and $s$ a state.

\[
(b?c : d)^s = \begin{cases} 
    c^s & \text{if } b^s = \text{true} \\
    d^s & \text{if } b^s = \text{false}
\end{cases}
\]

\[
\omega_s(b?c : d) = \begin{cases} 
    \omega_s(b)(c) & \text{if } b^s = \text{true} \\
    \omega_s(b)(d) & \text{if } b^s = \text{false}
\end{cases}
\]

\[
wds(b?c : d) \text{ iff } \begin{align*}
    b^s &= \text{true} \text{ and } wds(b) \land wds(b)(c) \\
    b^s &= \text{false} \text{ and } wds(b) \land wds(b)(d)
\end{align*}
\]
Consider a weakly pure, non-static method \textit{meth} without parameters declared in type \( T \), an expression \( a \) of type \( T \) and a state \( s \).
Semantics of Weakly Pure Methods in JML Specs

Consider a weakly pure, non-static method \( \text{meth} \) without parameters declared in type \( T \), an expression \( a \) of type \( T \) and a state \( s \).

\[
\begin{align*}
\Downarrow & \quad (a.meth)^s = \rho \\
& (\rho \text{ is the return value of } a.meth \text{ according to the black box})
\end{align*}
\]
Semantics of Weakly Pure Methods in JML Specs

Consider a weakly pure, non-static method \( \text{meth} \) without parameters declared in type \( T \), an expression \( a \) of type \( T \) and a state \( s \).

- \( (a.\text{meth})^s = \rho \)
  \( (\rho \) is the return value of \( a.\text{meth} \) according to the black box)\)
- \( \text{wd}_s(a.\text{meth}) \iff \text{wd}_s(a) \) and \( a^s \neq \text{null} \) and \( (a.\text{\textbackslash created})^s = \text{true} \) and \( \Omega = \text{null} \)
  \( (\Omega = \text{null} \) signals that method call \( a.\text{meth} \) terminates normally.
  First item in black box definition)
Consider a weakly pure, non-static method \( \text{meth} \) without parameters declared in type \( T \), an expression \( a \) of type \( T \) and a state \( s \).

1. \( (a.\text{meth})^s = \rho \)
   (\( \rho \) is the return value of \( a.\text{meth} \) according to the black box)

2. \( wd_s(a.\text{meth}) \iff \) \( wd_s(a) \) and \( a^s \neq \text{null} \) and \( (a.\text{\textbackslash created})^s = \text{true} \) and \( \Omega = \text{null} \)
   (\( \Omega = \text{null} \) signals that method call \( a.\text{meth} \) terminates normally. First item in black box definition)

3. \( \omega_s(a.\text{meth}) = s_2 \)
   (here \( s_2 \) is the terminating state of the method call \( a.\text{meth} \). Item 2 in definition of black box)
Weakly Pure Method Violating An Invariant

```java
class Unsound {
   int f;
   /*@ invariant f != 0; */

   /*@ private normal_behavior */
   @ assignable f;
   @ ensures this.f == 0; /*
   /*@ pure helper */ private Unsound() { f = 0; }
   */

   /*@ private normal_behavior */
   @ requires v == (new Unsound()).f;
   @ assignable \nothing; /*
   int divide(int v) { return 5 / v; }
   int showIt() { return divide(0); }
}
```
Invariants
When Should Invariants Be Valid?

class Inv{
    int a, b, c;
    // @ invariant c >= 0; @*/

    // @ private normal_behavior
    // @ requires b >= a;
    // @ assignable c;
    // @*/
    void balance(int a, int b) {
        c = c-a;
        c = c+b;
    }
}
When Should Invariants Be Valid?

class Inv{
    int a, b, c;
    /*@ invariant c >= 0; @*/

    /*@ private normal_behavior
        @ requires b >= a;
        @ assignable c;
        @*/

    void balance(int a, int b) {
        c = c-a;
        c = c+b;
    }
}

Invariant is violated after command c = c-a;
When Should Invariants Be Valid?

class Inv{
  int a, b, c;
  //!< invariant c >= 0; @*/

  //!< private normal_behavior
  @ requires b >= a;
  @ assignable c;
  @*/

  void balance(int a, int b) {
    c = c-a;
    c = c+b;
  }
}

Invariant is violated after command c = c-a;
but satisfied again after termination of the balance method.
Visible States

A state $s$ is called *visible* for an object $o$ if one of the following is true:

1. $s$ is the poststate of a non-helper constructor that initialises $o$,
2. $s$ is the prestate or poststate of a non-static non-helper method call with receiver object $o$,
3. $s$ is the prestate or poststate of a static non-helper method call in class $T$ where $T$ is a superclass of the class of $o$,
4. in $s$ no constructor or non-static method invocation as described in item 2, or static method invocation for a method as described in item 3 is in progress.
Visible States

A state $s$ is called *visible* for an object $o$ if one of the following is true:

1. $s$ is the poststate of a non-helper constructor that initialises $o$,
2. $s$ is the prestate or poststate of a non-static non-helper method call with receiver object $o$,
3. $s$ is the prestate or poststate of a static non-helper method call in class $T$ where $T$ is a superclass of the class of $o$,
4. in $s$ no constructor or non-static method invocation as described in item 2, or static method invocation for a method as described in item 3 is in progress.

A state $s$ is called *visible* for class $T$ if one of the following is true:

1. $s$ is the poststate after static initialisation of $T$ is complete
2. there is an object $o$ of type $T$ such that $s$ is visible for $o$. 
Visible States Semantics

Proof Obligation

In any state $s$ visible for object $ob$ all invariants of $ob$ are true.
Visible States Semantics

Proof Obligation

In any state $s$ visible for object $ob$ all invariants of $ob$ are true.

This assumes an appropriate notion of invariant for one object. Invariants referring to more than one element or using quantifiers are problematic.
Difference between visible and observable states

```java
public class Invariants{
    private int z = 1;
    /*@ private invariant z > 0; @*/

    public void a() {
        z += 0;
    }

    public void b() {
        z = 0;
        a();
    }
}
```
class List { private int[] a; int n;
/*@ public invariant a != null
   @ && 0<=n && n<=a.length &&
   @ (\forall int i; 0<=i && i<n; a[i]>=0) &&
   @ (\forall int i; n<=i && i<a.length; a[i]==0);
   @ requires ... @*/
public void add(int k){if(n==a.length){resize();}
    a[n] = k; n++;}

class BagWithMax {
private List Li; int max;
/*@ public invariant Li != null &&
   @ (\forall int i;0<= i && i<Li.n; Li.a[i] <= max);
   @ requires k>=0; ... @*/
public void insert(int k) { Li.add(k);
    if (k>max) {max = k;} }
// other methods and constructors omitted }
Problems With Visible State Semantics

class List { private int[] a; int n;
/*@ public invariant a != null
   @ && 0<=n && n<=a.length &&
   @ (\forall int i; 0<=i && i<n;a[i]>=0) &&
   @ (\forall int i; n<=i && i<a.length; a[i]==0);
   @ requires ... @*/
public void add(int k){if(n==a.length){resize();}
a[n] = k; n++;}
class BagWithMax {
private List Li; int max;
/*@ public invariant Li != null &&
   @ (\forall int i;0<= i && i<li.n;Li.a[i] <= max);
   @ requires k>=0; ... @*/
public void insert(int k) { Li.add(k);
   if (k>max) {max = k;}}
// other methods and constructors omitted }

This line is visible for object Li. Thus all its invariants are true.
Problems With Visible State Semantics

class List { private int[] a; int n;
   /*@ public invariant a != null 
     @ && 0<=n && n<=a.length && 
     @ (\forall int i; 0<=i && i<n; a[i]>=0) && 
     @ (\forall int i; n<=i && i<a.length; a[i]==0)); 
     @ requires ... @*/
public void add(int k){if(n==a.length){resize();}
   a[n] = k; n++;}

class BagWithMax {
private List Li; int max;
   /*@ public invariant Li != null && 
     @ (\forall int i;0<= i && i<Li.n;Li.a[i] <= max); 
     @ requires k>=0; ... @*/
public void insert(int k) { Li.add(k);
   if (k>max) {max = k;}}
   // other methods and constructors omitted }

This is not a visible state for object this since a method with this as caller is still active.
class List { private int[] a; int n;
/*@ public invariant a != null
 @ && 0<=n && n<=a.length &&
 @ (\forall int i; 0<=i && i<n; a[i]>=0) &&
 @ (\forall int i; n<=i && i<a.length; a[i]==0);
 @ requires ... @*/
public void add(int k){if(n==a.lenght){resize();}
a[n] = k; n++;}
}
class BagWithMax {
private List Li; int max;
/*@ public invariant Li != null &&
 @ (\forall int i;0<= i && i<Li.n;Li.a[i] <= max);
 @ requires k>=0; ... @*/
public void insert(int k) { Li.add(k);
  if (k>max) {max = k;}}
// other methods and constructors omitted }

Problem with visible state semantics: An analysis tool has to keep track if methods still execute on object.
THE END