

Program Verification

(6EC version only)

Erik Poll

Digital Security

Radboud University Nijmegen



Overview

- Program Verification using Verification Condition Generators
- JML – a formal specification language for Java

Used for the program verification exercise

Program verification

- Formally proving (in the mathematical/logical sense) that a program satisfies some property
 - eg that it does not crash, always terminates, never terminates, meets some functional specification, meets some security requirement, etc
 - *for all possible executions*: ie all possible inputs and all possible scheduling of parallel threads.
- NB in industry, the term **verification** is used for **testing** but testing provides only *weaker* guarantees
 - because testing will only try some executions
 - except in rare case where you can do **exhaustive testing**
- Formal verification provides the **highest level of assurance that code is correct & secure**
 - provided... you can formally verify what it means for the code to be secure

Example software verification for security: Hyper-V

Microsoft Hyper-V Hypervisor

- thin software layer that turns an x64 processor into multiple, isolated virtual x64 processor
- 100 Kloc of C and 5 Kloc of assembly

Verified using VCC tool, that turns code & specifications into verification conditions for theorem prover Z3

Info on VCC

<http://research.microsoft.com/en-us/projects/vcc/>

Video presentation on VCC

<http://channel9.msdn.com/posts/Peli/Michal-Moskal-and-The-Verified-C-Compiler/>

Example: seL4 microkernel

- *microkernel* – OS kernel that is *kept to minimum code size*
 - *in effort to reduce TCB*
- seL4 is 8,700 lines of C code and 600 lines of assembly
- Verified using interactive theorem prover Isabelle/HOL in L4.verified project at NICTA <http://ts.data61.csiro.au/projects/TS/l4.verified>
- Steps in the verification process
 - Developing abstract, executable specification in Haskell
 - Proving that C & machine code implementation behaves identical to (technically – *simulates*) this Haskell prototype
- Proof size 200,000 lines of proof scripts
- Verification effort 11 person-year

Example verification for security: miTLS & HACL*

- **miTLS** is fully verified TLS 1.3 implementation
 - Implementations in
 - functional language **F#**
 - ML-like functional language **F***

<https://mitls.org>

- **HACL*** is a formally verified cryptographic library in **F***
 - can be compiled down to C

<https://github.com/project-everest/hacl-star>

What do we need for program verification?


1. a **formal semantics** of the programming language
2. a **specification language** to express properties
3. a **logic** to reason about programs and specifications
 - aka a program logic
4. a **verification tool** to support all this

These topics are investigation in the field of field of **formal methods**

What to verify? Example

For the program

```
...  
x[4] = false;  
...
```



we might want to verify that **here** **x** is not null and **4** is within the array bounds (and that **x** is a Boolean array)

- Proving **absence of runtime exceptions** (or, in an unsafe language like C, memory-safety, or more generally, the absence of undefined behavior) is a great bottom-line specification to start verification!
- **Typing is a simple form of program verification**, for a limited and relatively weak class of properties, eg “**x is a byte array**”

A type checker can be regarded as an automated program verifier for this class of properties.

How to specify what we want to verify?

A common way to write we want to verify is using **assertions**, ie properties that hold at specific program points

```
...  
// assert x != NULL && x.length > 4;  
x[4] = false;  
...
```

Assertions written as **annotations in code** are also useful for testing, and for generating bug reports.

For methods or procedures, we can give **pre-** and **post-conditions**

How to verify?

Is the assertion below always true?

```
...  
if (x < y) { int z; z = y; y = x; x = z; }  
// assert y <= x
```

How do you verify this?

- You follow all paths in the control flow graph, and check that for each path the property holds using normal logical reasoning

Ways to formalize this reasoning process

- compute **verification conditions** using **weakest precondition** calculation (or strongest postcondition \sim)
- use **symbolic execution** to obtain these verification conditions

Complication 1 : cycles

Is the assertion below always true?

```
...
int i = x-y;
while (i > 0) { y++;
                i--;
            }
// assert y >= x
```

We can't follow all paths through the control flow graph, as the graph contains a cycle.

We need a [loop invariant](#).

Complication 1 : cycles

Is the assertion below always true?

```
...
int i = x-y; // so  $y+i == x$ 
while (i > 0) { y++; // now  $y+i == x+1$ 
                i--; // now  $y+i == x$ 
            }
// now  $i \leq 0$  (because we exited the while loop)
// and  $y+i == x$  (because it is a loop invariant)
// and therefore  $y \geq x$ 
```

Once we realise that $y+i == x$ is a loop invariant, we can split the graph in a finite number of segments, and check that each segment meets the specification

Complication 2 : modularity & the heap

Programming languages offer procedures or methods for modularity. This complicates reasoning.

```
...  
x = 5;  
p();  
// assert x == 5
```

Is the assertion always true?

Complication 2 : modularity & the heap

If **x** is on the **stack**, the assertion is always true

```
proc m() {  
    int x;  
    x = 5;  
    p();  
    // assert x == 5  
}
```

because **x** is out of scope for **p()**

- *assuming that we are in a memory-safe language:
if **p** contains buffer overflows, pointer arithmetic, ...
all bets are off!*

Complication 2 : modularity & the heap

If **x** is on the **heap**, things become tricky

- even in a safe programming language!

In Java, will the assertion below always hold?

```
x = 5;  
o.p();  
// assert x == 5
```

Complication 2 : modularity & the heap

```
class A {  
    static int x = 12; // ie a class field  
  
    public void m() {  
        x = 5;  
        o.p();  
        // assert x == 5  
    }  
    ...  
}
```

Is the assert always true?

No, because `o.p()` might change `A.x`

Complication 2 : modularity & the heap

```
class A {  
    int x = 12;  
  
    public void m() {  
        x = 5;  
        o.p();  
        // assert x == 5  
    }  
  
    ...  
}
```

Is the assert always true?

No, because e.g.

- `o` could be aliased to `this` and `o.p()` could change `x`
- `o` could have a reference to `this` and then change `x` by invoking a method or assignment to `x` if it is not private

Complication 3

```
...  
x = 5;  
// assert x == 5
```

Is the assertion always true?

Complication 3: concurrency & the heap

```
...  
x = 5;  
// assert x == 5
```

Is the assertion always true?

No, not if there is another thread running that may also be accessing **x**

The problem, and possible solutions, are very similar to the problem of modular reasoning about procedures/methods.

Solutions include **separation logic**, **implicit dynamic frames**, or **ownership**

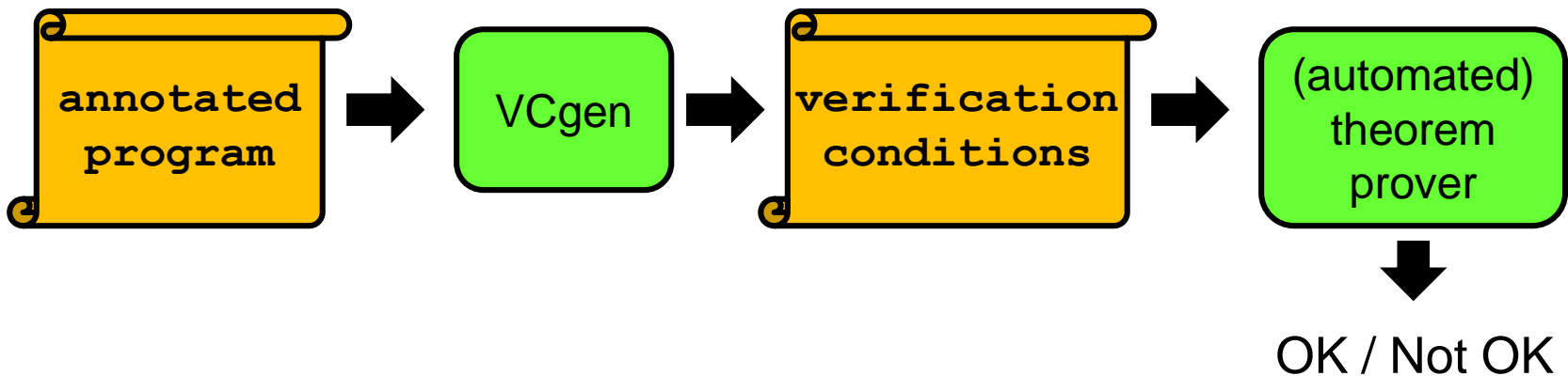
Newer programming languages such as **Rust** might be better suited for reasoning about concurrency

Program Verification
using
Verification Condition Generation

Program Verification using VCGen

One of the standard approaches for program verification:
using [Verification Condition Generator \(VCGen\)](#):

1. Program is [annotated](#) with [properties](#) (the [specification](#))
2. Verification Condition Generator produces a set of [logical properties](#), the so-called [verification conditions](#)
3. If these verification conditions are true, the annotations are correct – ie the program satisfies the specification



Example verification using VCGen

```
//@ requires true;
//@ ensures \result > 5;
public int example(int j)
{
    if (j < 8) {
        int i = 2;
        while (j < 6*i){
            j = j + i;
        }
    }
    return j;
}
```

}

These annotations give a pre- and postcondition that form the specification:

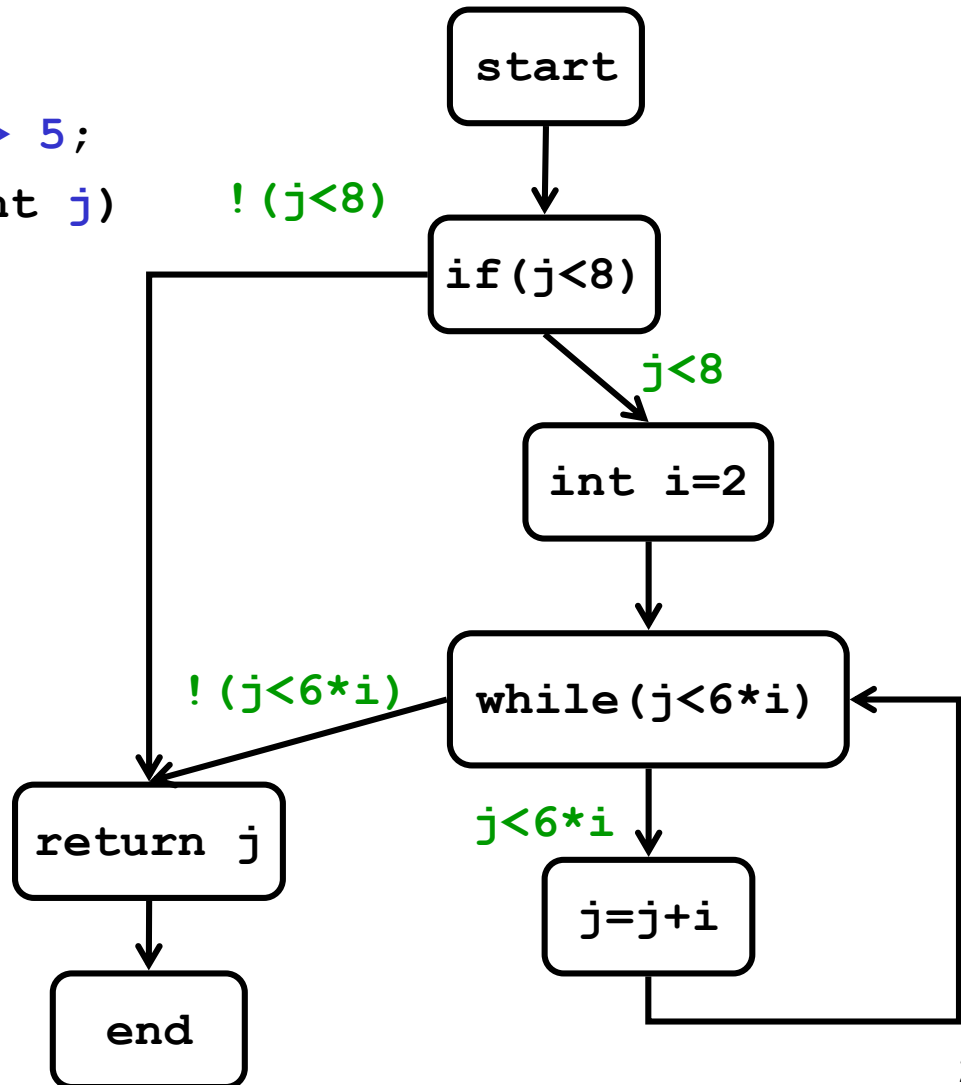
on any input, this method will return a result greater than 5

- is this specification always met?
- how do you know this?
- could an automated tool reproduce your reasoning?

Verification using VCGen

(i) program as graph

```
//@ ensures \result > 5;  
public int example(int j)  
{  
  if (j < 8) {  
    int i = 2;  
    while (j < 6*i) {  
      j = j + i;  
    }  
  }  
  return j;  
}
```

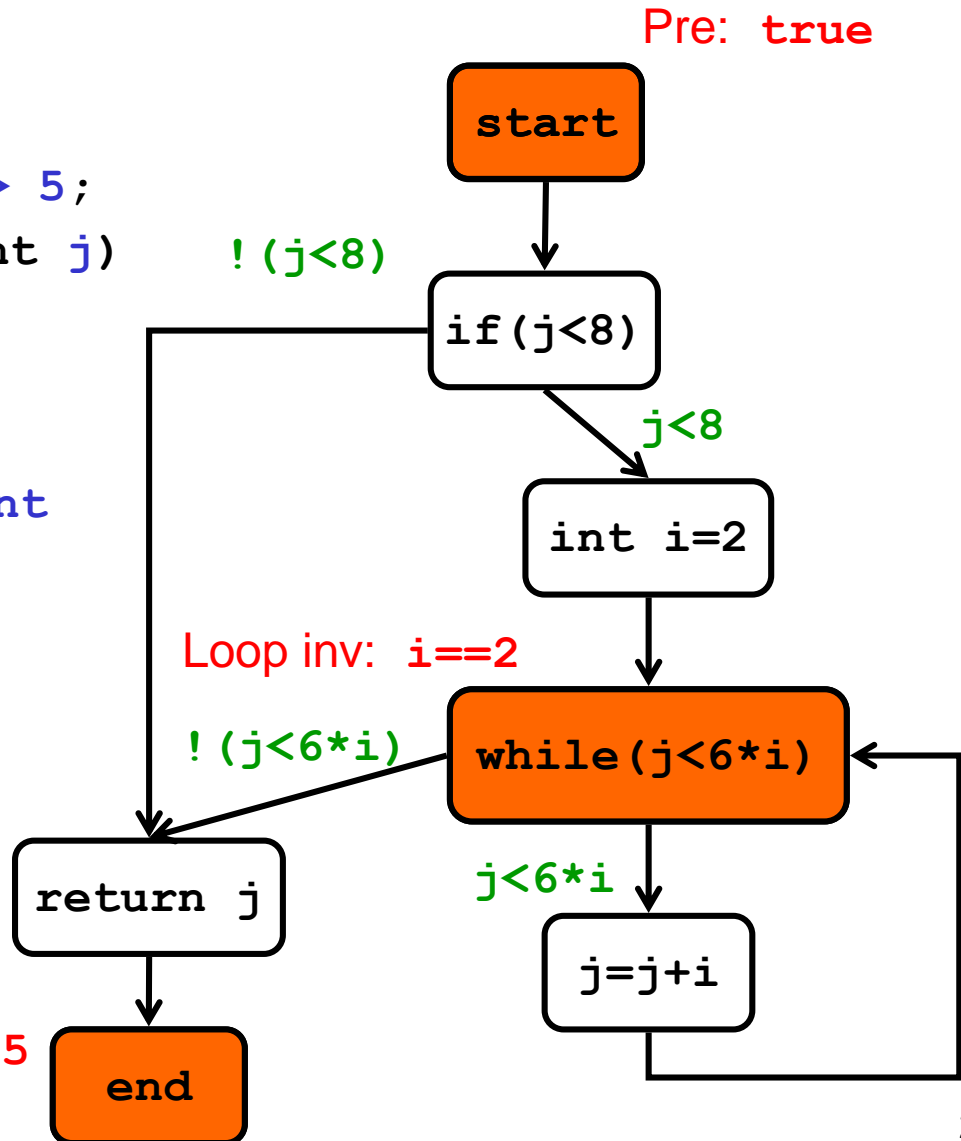


Verification using VCGen

(ii) add assertions

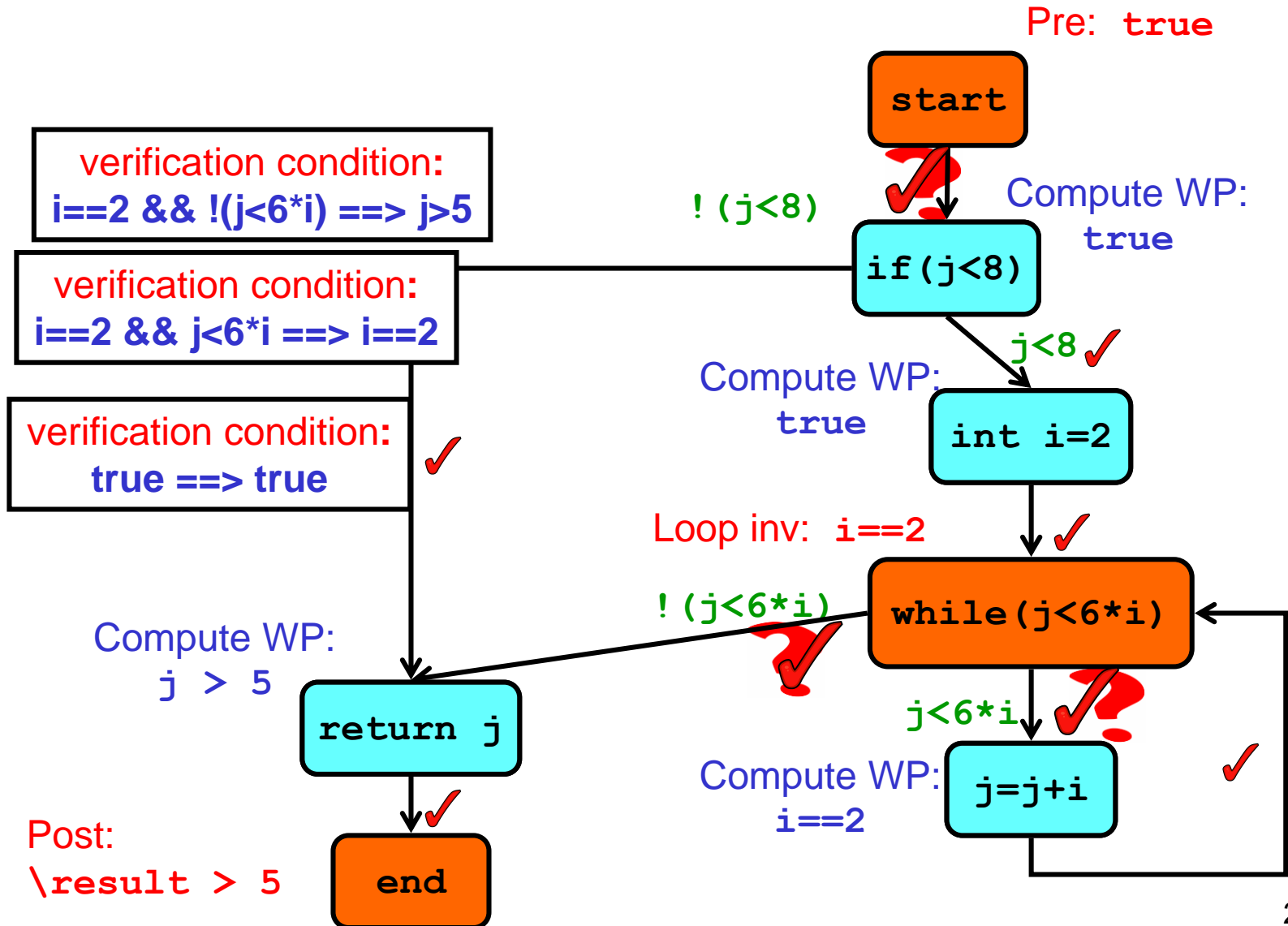
```
//@ ensures \result > 5;  
public int example(int j)  
{  
  if (j < 8) {  
    int i = 2;  
    /*@ loop_invariant  
       i==2;  
    @*/  
    while (j < 6*i) {  
      j = j + i;  
    }  
  }  
  return j;  
}
```

Post: $\text{\result} > 5$



Verification using VCGen

(iii) compute VCs & check



Verification condition generation

Given a postcondition and loop invariants

- compute an assertion P_s for every state s
based on assertions $P_{s'}$ of the states s' reachable from s
 - key idea: P_s is the weakest predicate such that if it holds in state s , and the program goes to state s' then $P_{s'}$ will hold in that state s'
- all that remains to be verified
 - $\text{Pre} \Rightarrow P_0$
the precondition specified in the program implies the assertion computed for the initial state
 - $\text{Loop}_s \Rightarrow P_s$
each loop assertion specified in the program implies the assertion computed for that state

“Opposite” approach: forward instead of backwards

Instead of working backwards from the postcondition of the final state, you can work forward from the precondition in the initial state:

you then compute **strongest postconditions**
instead of **weakest preconditions**

This is very similar to **symbolic execution** of a program.

Tricky issues in program verification

Whatever the approach, the bottlenecks in program verification remain...

1. **pointers / references & the heap**

Reasoning about data on the heap is difficult.

Even in a language with automatic memory management, such as Java or C#, we still have the complication of aliasing

2. **concurrency** aka multi-threading

JML

Formal specification for Java



JML

- Formal specification language for Java
 - Properties can be specified in **Design-By-Contract** style, using **pre/postconditions** and **object invariants**

NB by default, in JML invariants are *object* invariants, not *loop* invariants.

- Various tools to check JML specifications by eg
 - runtime checking
 - program verification

to make JML easy to use

- JML annotations are added as special Java comments, between `/*@ . . @*/` or after `//@`
- JML specs can be in .java files, or in separate .jml files
- Properties specified using Java syntax, extended with some operators
`\old(), \result, \forall, \exists, ==>, ..`
and some keywords
`requires, ensures, invariant,`

Example JML

```
public class ChipKnip{
    private int balance;
    //@ invariant 0 <= balance && balance < 500;

    //@ requires amount >= 0;
    //@ ensures balance <= \old(balance);
    //@ signals (BankException) balance == \old(balance);
    public debit(int amount) {
        if (amount > balance) {
            throw (new BankException("No way"));
        }
        balance = balance - amount;
    }
}
```


JML basics

- preconditions **requires**
- postconditions **ensures**
- exceptional postconditions **signals**
- (object) invariants **invariant**
 - must be **established** by **constructors**
 - must be **preserved** by **methods**
 - ie. assuming invariant holds in pre-state, it must hold in the post-state

Exceptional postconditions: signals

```
//@ requires ...
//@ ensures)...
//@ signals (BankException) balance == \old(balance);
public debit(int amount) throws BankException {
    if (amount > balance) {
        throw (new BankException("No way")); }
    balance = balance - amount;
}
```

But you can ignore this for the practical exercise! There we will always prove that no exceptions can be thrown.

JML convention: a method may only throw exceptions that are explicitly listed in the throws clause. (Java allows implicit Runtime- exceptions, eg Nullpointer- and ArrayIndexOutOfBounds; JML does not!)

non_null

- Lots of invariants and preconditions are about reference not being null, eg

```
int[] a; //@ invariant a != null;
```

- Therefore there is a shorthand

```
/*@ non_null */ int[] a;
```

- But, as [most references are non-null](#), some JML tools adopt this as default, so that only *nullable* fields, arguments and return types need to be annotated, eg

```
/*@ nullable */ int[] b;
```

- We could also use [JSR308 Java tags](#) for this

```
@Nullable int[] b;
```

Defaults specs and joining specs

- Default pre- and postconditions

```
//@ requires true;
```

```
//@ ensures true;
```

can be omitted

- `//@ requires P`

```
//@ requires Q
```

means the same as

```
//@ requires P && Q;
```

but the former may allow tools to give more precise feedback, namely on whether P or Q is not satisfied

What can you do with this?

- Documentation/specification
 - explicitly record detailed design decisions & document assumptions (and hence obligations!)
 - precise, unambiguous documentation
 - parsed & type checked
- Use tools for
 - runtime assertion checking
 - eg when testing code
 - compile time program analysis
 - up to full formal program verification

assert and loop_invariant

Inside method bodies, JML allows

- assertions

```
/*@ assert (\forall int i; 0 <= i && i < a.length;  
           a[i] != null );  
@*/
```

- loop invariants

```
/*@ loop_invariant 0 <= n && n < a.length &  
                  (\forall int i; 0 <= i & i < n;  
                    a[i] != null );  
@*/
```

- Program verification tools, such as ESC/Java2, KeY, Krakatoa, ... can do program verification of JML-annotated Java code

There is a limit to what *fully automated* tools, such as ESC/Java2, can verify

eg. they won't be able to prove Fermat's Last theorem

- So far, only really feasible for small(ish) programs
 - incl. realistic Java Card smart card applications
- In addition to doing the verification, which is a lot of work, a bottleneck is expressing the security property you want to verify

JML for security

JML can be used to specify for instance

1. which – if any - exceptions can be thrown
incorrectly/not handling errors common source of security problems
2. security-critical invariants to be preserved
even when exceptions occur
3. assumptions on input the application relies on
4. any property expressible by security automaton

Simply trying to verify that a program throws no exceptions – or just no Nullpointer-exceptions - will expose many (implicit) invariants and assumptions on input

Related work

- Spec# for C#
by RUSTAN LEINO & CO at Microsoft Research
The back-end of this system also used for Hyper-V verification
- SparkAda for Ada
by Praxis High Integrity System
- ACSL for C
used in the Framac toolset
https://www.youtube.com/watch?v=J_xgbO5-32k

Some industrial usage, also/esp. for *safety*-critical software (notably in avionics) rather than *security*-critical software