Automated Program Verification

Winter 2011

Guaranteeing Program Correctness

- Programs should behave how we want them to
  - Example: not crashing with an unexpected exception
- To guarantee this:
  1. **Specify** what a program's behavior should be
  2. **Check / enforce** that a program satisfies the specification

Method Specifications

- **Preconditions**: must be true when the method is called
- **Postconditions**: must be true when the method exits if the preconditions were met
  - Return value
  - Exceptions that are raised and under what conditions
  - Side-effects

REMEMBER: What does it mean for a method to have stronger preconditions than another method? Stronger postconditions?

Representation Invariants

- Must be true at the end of a constructor
- Must be true before and after every **public** method
- In CSE331, you check these at *runtime* with a `checkRep()` method

Banking Example

```
public class BankingExample{
   //RI: balance != null
   //    0 <= balance <= MAX_BALANCE
   private Integer balance;

   //@effects this.balance = 0
   public BankingExample { balance = 0; }

   //@requires amount != null
   //@requires 0 < amount && amount + balance < MAX_BALANCE
   //@ensures  new this.balance = old this.balance + amount
   public void credit(Integer amount) { balance += amount; }

}
```

Has Specs: ☺

Specs True: ???

Banking Example: Runtime Assertions

```
public class BankingExample{
   //RI: balance != null
   //    0 <= balance <= MAX_BALANCE
   private Integer balance;

   //@effects this.balance = 0
   public BankingExample { balance = 0; }

   //@requires amount != null
   //@requires 0 < amount && amount + balance < MAX_BALANCE
   //@ensures   new this.balance = old this.balance + amount
   public void credit(Integer amount) {
        checkRep(); balance += amount; checkRep();
   }

   private void checkRep(){
        assert(balance != null);
        assert(0 <= balance && balance <= MAX_BALANCE);
   }
}
```

Run-time checks that the program satisfies the specification

Banking Example: Pluggable Type Checking

```
public class BankingExample{
   //RI: balance != null
   //    0 <= balance <= MAX_BALANCE
   private /*@NonNull*/ Integer balance;

   //@effects this.balance = 0
   public BankingExample { balance = 0; }

   //@requires amount != null
   //@requires 0 < amount && amount + balance < MAX_BALANCE
   //@ensures  new this.balance = old this.balance + amount
   public void credit(/*@NonNull*/ Integer amount) { . . . }

   private void checkRep(){
        ~~assert(balance != null);~~
        assert(0 <= balance && balance <= MAX_BALANCE);
   }
}
```

Unnecessary! The type checker enforces this for us!

Banking Example: Formal Proof

```
public class BankingExample{
   //RI: balance != null
   //    0 <= balance <= MAX_BALANCE
   private Integer balance;

   //@effects this.balance = 0
   public BankingExample { balance = 0; }

   //@requires amount != null
   //@requires 0 < amount && amount + balance < MAX_BALANCE
   //@ensures  new this.balance = old this.balance + amount
   public void credit(Integer amount) { balance += amount; }

}
```

Manually find weakest preconditions, inductive properties, and loop invariants (as in PS5)

Specification Approach Comparison

Method	Checked at compile-time	Automatically checked	Documentation consistency	Express all properties
Assertions	☹	☺	☹	☺
Pluggable Type Checking	☺	☺	😐	☹
Formal Proofs	☺	☹	☺	☺

Automated formal proofs	☺	☺	☺	☺

Expressing Rich Specifications

- Need to express conditions such as
  - `this.balance = old this.balance + amount`
  - `returns x if x >= 0 and -x otherwise`
  - `all elements of the array are less than 5`

  in a way that a computer can understand and (hopefully) check automatically
- Our expression language needs support for:
  - logic (e.g., if / else, quantification)
  - programming concepts (return values, side-effects)

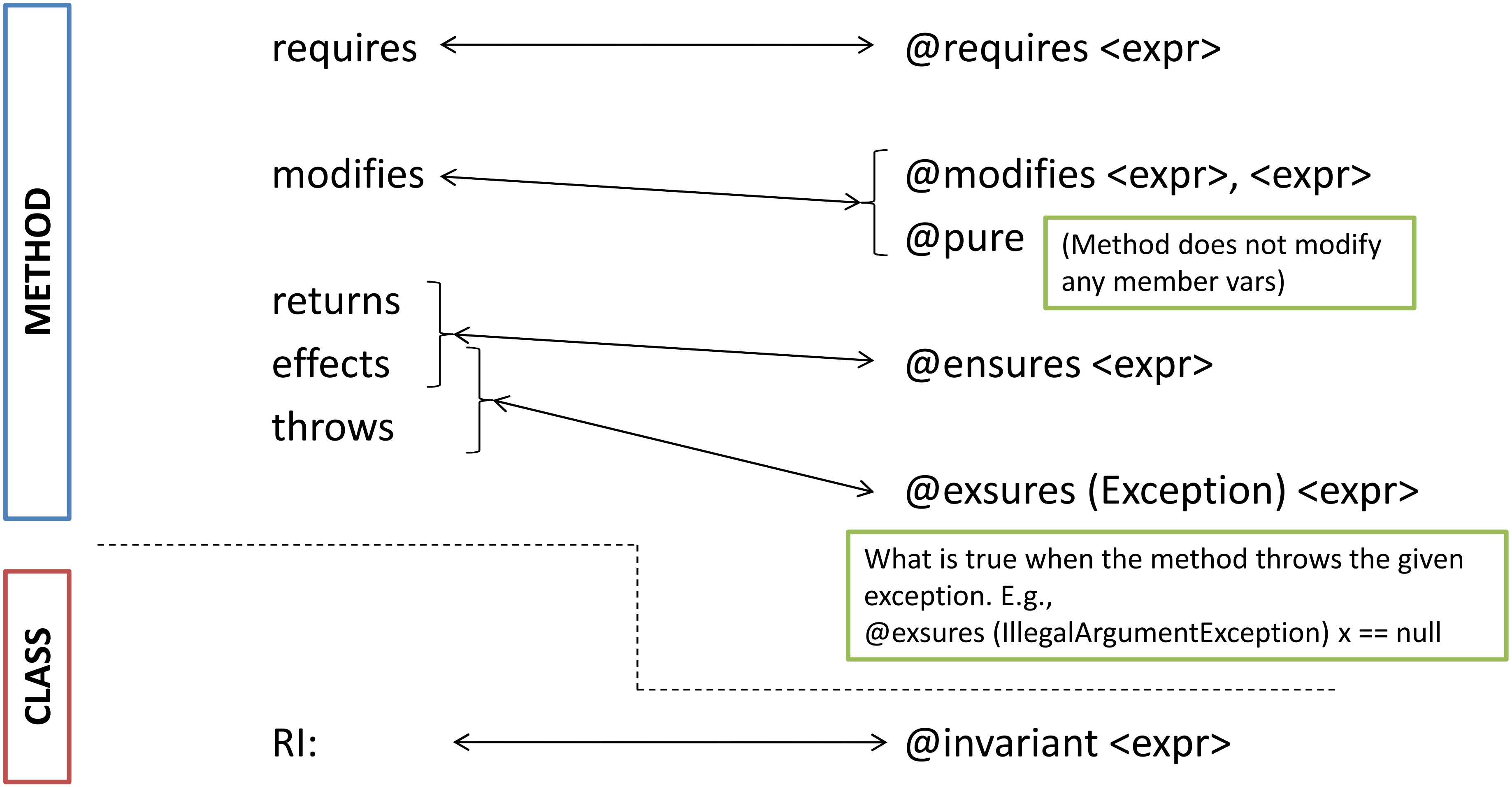
Java Modeling Language (JML)

- Formal language for writing specifications
- Advantages / disadvantages of using a formal language instead of natural language:
  - Precision
  - Expressiveness
- Write in program comments; numerous tools can use the specification to:
  - Generate documentation
  - Automatically generate unit tests
  - Check that the code meets the specification
- Website: http://www.eecs.ucf.edu/~leavens/JML/

CSE331 vs. JML Specifications

CSE331 Specification

JML Specification



METHOD

requires

@requires <expr>

modifies

@modifies <expr>, <expr>

@pure

(Method does not modify any member vars)

returns

effects

@ensures <expr>

throws

@exsures (Exception) <expr>

What is true when the method throws the given exception. E.g.,
@exsures (IllegalArgumentException) x == null

CLASS

RI:

@invariant <expr>

JML Expressions

Expression	Meaning
a ==> b	a implies b
a <==> b	a is true if, and only if, b is true; same as a == b
\result	the return value of the method
\old(<expr>)	Refers to the value of <expr> at the entry of the method
`\forall <decl>; <expr>`	Universal quantification
a && b	Just like in Java
a || b	Just like in Java
!a	Just like in Java

Banking Example in JML

States that variable can be used in public specifications, even though it is private

```
public class BankingExample{
   /*@spec_public */ private Integer balance;
   //@invariant balance != null
   //@invariant 0 <= balance && balance <= MAX_BALANCE

   //@ensures this.balance = 0
   public BankingExample { balance = 0; }

   //@requires amount != null
   //@requires 0 < amount && amount + balance < MAX_BALANCE
   //@modifies balance
   //@ensures  this.balance = \old(this.balance) + amount
   public void credit(Integer amount) {…}
}
```

\result example

```
boolean foo(int x, int y){
   if (x < y){
        return true;
   }else{
        return false;
   }
}
```

They’re all correct!

Which post-condition is correct?

```
//@ensures (x < y) ==> (\result == true)
//@ensures (x >= y) ==> (\result == false)
```

```
//@ensures (x < y) <==> (\result == true)
```

```
//@ensures (x < y) <==> \result
```

```
//@ensures (x < y) == \result
```

Universal Quantification

- Used to express that a fact holds over a range of values:

```
\forall <decl>; <expr>
```

- Example:

```
 \forall int i;
   (0 <= i && i < arr.length) ==> arr[i] < 5
```

Use ==> (implication) to guard against non-sense values

- Implication truth-table:

a ==> b

	b = true	b = false
a = true	**TRUE**	FALSE
a = false	**TRUE**	**TRUE**

Extended Static Checking

- ESC/Java2 takes a program description in JML and a Java program and determines:
  - If the program meets the specification
  - If the program might throw an unexpected exception (e.g., ArrayIndexException)
- You don't have to write any proofs ☺
- Like pluggable type-checkers, some perfectly good programs won't pass (false alarms)

ESC/Java Demo

VeriWeb: A Better (?) Interface to ESC/Java2

- Runs in web browser: no setup required for users
- Drag and drop interface for writing pre- and post- conditions
- You work on a method at a time; representation invariants are determined implicitly

VeriWeb Demo

Conclusion

- JML is a language for writing Java program specifications
- ESC/Java2 verifies JML specifications
- VeriWeb is a web interface to ESC/Java2
- Other tools can use JML specs to:
  - Generate documentation
  - Generate tests
  - Statically check whether or not the program meets the specification