Java Modeling Language
Tools and Applications

Speaker: Ivo Steinmann

Overview

- Introduction to JML / Example
- Presentation of Tools for JML
- Conclusion

Introduction

- JML (Java Modeling Language) is a behavioral interface specification language
  - define the behavior of Java code
  - specify syntactic interface (API)
- Ideas taken from Eiffels Design by Contract
  - preconditions
  - postconditions
  - class and loop invariants
- Goal: Easily understandable by Java programmers

Example - BankAccount interface

```java
public interface BankAccount {

    /**
     * Returns the current balance of the bank account
     *
     * @return balance
     */
    public double getBalance();

    /**
     * Deposit some money to the bank account
     *
     * @param amount of money to deposit. It is expected to be
     * positive and less or equal then 1000000.
     */
    public void deposit(double amount);

    /**
     * Withdraw some money from the bank account
     *
     * @param amount The amount of money to withdraw. Positive value
     * less or equal to 2000.
     * @param pin The pin to access the bank account.
     * @throws AccessException when an invalid pin is used.
     */
    public void withdraw(double amount, byte[] pin) throws AccessException;

}
```

BankAccount - getBalance()

```
public interface BankAccount {

    //@ public model instance double m_balance;
    //@ invariant 0 <= m_balance;

    /**
     * Returns the current balance of the bank account
     *
     * @return balance
     */
    //@ ensures \result == m_balance;
    //@ pure
    public double getBalance();


    ...
}
```

```
JML specifications inserted
after //@ or
between /*@ ... @*/

(Java comment)
```

BankAccount - deposit()

```
public interface BankAccount {
    ...

    /**
     * Deposit some money to the bank account
     *
     * @param amount of money to deposit. It is expected to be
     * positive and less or equal then 1000000.
     */
    //@ requires amount >= 0;
    //@ requires amount <= 1000000;
    //@ ensures m_balance == \old(m_balance) + amount;
    //@ assignable m_balance;
    public void deposit(double amount);

    ...
}
```

BankAccount - withdraw()

```
public interface BankAccount {
    ...

    //@ public model instance byte[] m_pin;
    //@ invariant m_pin != null && m_pin.length >= 4;
    //@ invariant (\forall int i; 0 <= i && i < m_pin.length;
            0 <= m_pin[i] && m_pin[i] <= 9);

    /**
     * Withdraw some money from the bank account
     *
     * @param amount The amount of money to withdraw. Positive value
     * less or equal to 2000.
     * @param pin The pin to access the bank account.
     * @throws AccessException when an invalid pin is used.
     */
    //@ requires amount >= 0;
    //@ requires amount <= 2000;
    //@ requires m_balance >= amount;
    //@ requires pin != null;
    //@ assignable m_balance;
    //@ ensures m_balance == \old(m_balance) – amount;
    //@ ensures java.util.Arrays.equals(m_pin, p);
    //@ signals (AccessException) !java.util.Arrays.equals(m_pin, p);
    public void withdraw(double amount, byte[] p) throws AccessException;

}
```

BankAccount implementation (1)

```
public class BankAccountImpl implements BankAccount {

    //@ private represents m_balance = balance;
    //@ spec_public
    private double balance;

    //@ private represents m_pin = pin;
    //@ spec_public
    private byte[] pin;

    //@ requires b >= 0 && p != null;
    //@ assignable balance, pin;
    //@ ensures balance == b && java.util.Arrays.equals(pin, p);
    public BankAccountImpl(double b, byte[] p) {
        balance = b; pin = (byte[]) p.clone();
    }


    ...
```

BankAccount implementation (2)

```
    ...continued

    public double getBalance() {
        return balance;
    }

    //@ also assignable balance;
    public void deposit(double amount) {
        balance = balance + amount;
    }

    //@ also assignable balance;
    public void withdraw(double amount, byte[] pin) throws AccessException {
        if (!Arrays.equals(this.pin, pin))
            throw new AccessException("invalid pin");
        balance = balance - amount;
    }

}
```

Tools for JML

- Runtime assertion checking and testing
- Static checking and verification
- Generating specifications
- Documentation

Runtime assertion checking (1)

- Run code and report assertion violations
- Translates JML assertions into runtime checks
- No side effects: transparancy guaranteed
- JML Compilers
  - jmlc
    - Problems with keeping up with Java features
  - jml4c
    - Based on Eclipse JDT
    - Java 5 features
    - Up to 3 times faster than jmlc

Runtime assertion checking (2)

- Possibility to autocreate JUnit testcases
  - Automates unit testing
- Side effects: Detects also bugs in JML assertions
- Generators
  - jmlunit
  - jmlunitng
    - Java 5 features

- But: Quality of created testcases depends on quality of JML specifications

Static checking and verification

- Verify the code specifications statically
  - Not decidable in general!
- Tools
  - ESC/Java(2)
    - Tries to prove correctness at compile time
    - Not sound: may miss errors that can occur
    - Not complete: may warn of errors that can not occur
  - JACK
    - Weakest precondition calculus
    - Interface to automatic theorem prover B
      - But hides the complications of the theorem prover
    - Eclipse Plugin

Generating specifications

- So far: Tools expected the existence of JML specifications
- Writing these maybe very time-consuming
- Daikon
  - Detect invariants
  - Accuracy depends on quality and completness of testcases
  - Actual behavior ↔ intended behavior?

Documentation

- Create documentation out of JML specifications
- Tool: jmldoc
  - Collect JML annotations accross overriden methods
  - Translate JML annotations to Javadoc
  - Combine with existing Javadoc Text
  - Create browsable HTML pages

Conclusion

- JML is easy to learn
  - No need to learn another language
- Source code *is* the formal model
- Can be used in existing code and API's
  - Possible to introduce JML *gradually*
- There are many tools supporting JML
  - But: Keep up with new Java versions is a challenge
- Still many open research issues

Questions?

Static checking and verification (Ext)

- TACO: Translation of Annotated Code
- Bounded verification technique
  - Examine execution up to a user-provided heap bound
  - ... and loop unrollings
- Java 1.6
- Translates to JDynAlloy

JML2 Eclipse Plug-In

- ETH Chair of Programming Methodology
  - This Eclipse plug-in provides a basic integration of the Common JML2 tools into the Eclipse IDE.
  - Originally, this plug-in started as a small part of the Universe type system inference tools, but was spun off later. See the research page for an overview of the Universe type system and the tools page for information about the tools we provide.

- See: http://www.pm.inf.ethz.ch/research/universes/tools/eclipse/

Links

• JML Specs	www.jmlspecs.org
• Jml4c	www.cs.utep.edu/cheon/download/jml4c
• Jmlunitng	http://formalmethods.insttech.washington.edu/software/jmlunitng/
• JACK	http://www-sop.inria.fr/everest/soft/Jack/jack.html
• TACO	http://www.dc.uba.ar/inv/grupos/rfm_folder/TACO
• Daikon	http://groups.csail.mit.edu/pag/daikon/dist/doc/daikon.html
• JML2 Eclipse Plugin	http://www.pm.inf.ethz.ch/research/universes/tools/eclipse