Automatic Verification & Testing

Programming with Contracts

Juan Pablo Galeotti, Alessandra Gorla
Saarland University, Germany
Programming with Contracts

Contract

A (formal) agreement between

Method M (callee)  Callers of M

Rights  Responsabilities  Rights  Responsabilities
Example

Contract

Compute square root of a real number

Method (callee):
get-square-root

Caller
get-fibonacci-number

Expects non negative numbers
Return the square root
Invoke method with non negative numbers
Obtain the square root
**Contract**: Agreement between parts
- In this case: method and user method
- The method pre & postconditions defines an agreement between caller and callee.
  - The client (caller) must **ensure the precondition** and **assume the postcondition**
  - The method (callee) may **assume the precondition**, but it must **ensure the postcondition**

<table>
<thead>
<tr>
<th></th>
<th>Responsibilities</th>
<th>Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller</td>
<td>Pre_Callee</td>
<td>Post_Callee</td>
</tr>
<tr>
<td>Callee</td>
<td>Post_Callee</td>
<td>Pre_Callee</td>
</tr>
</tbody>
</table>
Specifying contracts in components

- A component implements some entity or important element for our solution
  - Component = set of classes
  - Class = set of methods

- Contracts
  - At method level: Requires, Ensures
  - At class level: object/class invariants
  - At component level: ownerships + invariantes among different objects
Dataflow analysis and typestate checking are very effective for dealing with “weak” specifications
- Very simple correction properties
- Null pointers, zero division, API usage, etc.

They are inadequate for dealing with complex/complete specifications (“strong” specifications)
- This functions computes an invoice for a customer
- The candidate declared as winner is the one who has more votes

Can we write several weak specifications to express a strong specification?
The Verifying Compiler

- The Verifying Compiler
  - automatically checks that a program conforms to its specification
  - The correction can be specified using types, assertions or any other redundant annotation to the program

- First Proposal: 1969
Soundness:
- If the formula is true, then the program satisfies the specification
The Verifying Compiler

- Programming Language
- Specification Language
- Logical representation of the program
- Automatic Decision Procedure

JAVA

Program

Specification

JML

ESC/Java2

Translator

Logical Formula

Verifier

Weakest Precondition (Dijkstra)

SMT-Solver (Simplify)

Automatic Theorem Prover

Valid

Invalid
Desired properties of a Verifying Compiler

- **Soundness**: If the verifier reports no failure, then the program does not fail.

- **Completeness**: If the verifier reports a failure, then the program fails.

- **Termination**: Given any program $P$, the verifier finishes the analysis of $P$ (even with an unknown result).
The Specification Language

JAVA

JML

Program

Specification

Translator

Logical Formula

Automatic Theorem Prover

Verifier

Valid

Invalid
JML: Java Modeling Language

- Formal specification language for Java
- **Objective:** Design a specification language easy to use for most of the programmers
- **Origin:** runtime assertion checking
- **Inspiration:** Eiffel language (Design by Contract™)
- For C#: Spec#, CodeContracts™
JML Annotations

- Within comments in the Java code using \/*@...@*/, or after //@ ....

- Boolean Java expressions extended with some new operators
  - (\old, \forall, \result, \sum...)

- Several kinds of annotations
  - Modifiers: pure, non_null, nullable...
  - Method level: requires, ensures, signals,
  - Class level: invariant
JML: pre-, post-conditions (1)

- \texttt{old}(...) returns the value of the expression before the execution of the method
- \texttt{result} refers to the return value of the method

```java
/*@
@ requires amount>=0;
@ ensures balance == \texttt{old}(balance)-amount;
@ ensures \texttt{result} == balance
@*/
public int debit(int amount) {...}
```
JML: pre-, post-conditions (2)

- JML specifications can be as weak (or strong) as we want them to be.
- This specification is stronger or weaker then the previous one?

```java
/*@ requires amount>=0;
@ ensures \result == balance
@*/
public int debit(int amount) {...}
```
If the program signals an exception of type `BankException`, then the predicate holds
JML: exceptions handling

- All exceptions are allowed by default (ensures only applies to normal termination).
- To change this:
  - Forbid all exceptions

/*@ normal_behaviour
  @ requires ...
  @ ensures ...
  @*/
JML: exceptions handling

- All exceptions are allowed by default (ensures only applies to normal termination).
- To change this:
  - Forbid all exceptions
  - Forbid one exception type E

//@ signals (E) false;
All exceptions are allowed by default (ensures only applies to normal termination).

To change this:

- Forbid all exceptions
- Forbid one exception type E
- Allow only some exceptions types E₁,…,Eₙ

//@ signals_only E₁,…,Eₙ;
This means: if an exception \( e \) of type \( \text{Ex} \) is thrown, then \( P(e) \) holds

Can we say: if this precondition holds, then the exception \( \text{Ex} \) is thrown?
```java
/*@ normal_behavior
  @ requires amount<=this.balance;
  @ also
  @ exceptional_behavior
  @ requires amount>this.balance;
  @ signals (BankException e) e.getReason().equals(…);
  @*/
public int debit(int amount) throws BankException {...}
```

- `normal_behavior` implicitly includes a clause:
  - `signals (Exception ex) false;`
An assertion specifies a property that holds at a given program point

```c
if (i<=0 || j<0) {
    ...
} else if (j<5) {
    //@ assert i>0 && 0<j && j<5;
    ...
}
```
JML assertions have more expressive power since we can include JML operators

```java
for (n=0; n<a.length; n++) {
    if (a[n]==null) break;
    //@ assert (\forall int i; 0<=i && i<n; a[i]!=null);
    ...
}
```
JML: assume

- Like JML assertions, but we restrict ourselves to traces where the condition is true

```java
//@ assume b!=null && b.length>0;
b[0]=2;
```

- Useful during development
  - We can document assumptions
  - They “help” the automatic theorem prover
JML: frame conditions

- A frame conditions constraints the side-effects of a given method

```java
/*@ requires amount>0;
    @ assignable this.balance
    @ ensures this.balance==\old(this.balance) – amount;
    @*/
public int debit(int amount) {...}
```

- Can we constraint side-effects by adding postconditions?

- By default: ///*@ modifies \everything
A method with no side-effects is called “pure”.

```
@pure
public int getBalance() {...}
```

The pure annotation is equivalent to

```
@assignable nothing;
```

Only pure methods can be used in specifications

```
@requires this.getBalance()>0;
```
Problem: dealing with assignable annotations can be VERY ANNOYING.

```java
public class Timer{
    int time_hrs, time_mins, time_secs;
    int alarm_hrs, alarm_mins, alarm_secs;

    //@ assignable time_hrs, time_mins, time_secs;
    public void tick() {...}

    //@ assignable alarm_hrs, alarm_mins, alarm_secs;
    public void setAlarm(int hrs, int mins, int secs){...}
}
DataGroups: allow us to specify a recurrent subset of assignable locations

```java
public class Timer{
    JMLDataGroup time, alarm;
    int time_hrs, time_mins, time_secs; //@ in time;
    int alarm_hrs, alarm_mins, alarm_secs; //@ in alarm;

    //@ assignable time;
    public void tick() {...

    //@ assignable alarm;
    public void setAlarm(int hrs, int mins, int secs){...}
```
Class invariants are properties that must be preserved by all methods.

```
public class Wallet {
    public static final short MAX_BAL = 1000;
    private short balance;
    /*@ invariant 0<=this.balance &&
            @         this.balance<=MAX_BAL;
            @*/
```

- They are implicitly included in all methods
- They must be preserved even in case of abnormal termination
public class ArrayOps {

private /*@ spec_public @*/ Object[] a;

//@ public invariant 0 < a.length;
//@ requires 0 < arr.length;
//@ ensures this.a == arr;
//@

public void init(Object[] arr) {
    this.a = arr;
}
}
This modifier allows us to specify if a given field, argument or variable can be null.

By default: all fields (!), arguments, return types and quantified variables (∀forall, ∃exists) have an implicit `non_null` modifier.

The opposite annotation of `non_null` is `nullable`.

Example:

- `/*@ nullable @*/ Integer i;`
- `/*@ non_null @*/ Object o;`
Loop invariant

- A predicate describing how the program state changed by executing the loop.

- Part of the reasoning we do (our subconscious?) while writing a loop
  - Formal description:
    - What we assume before loop execution
    - How the program state evolved at the end of the iteration

- For verification purposes they are fundamental (unless we have a loop free program)
int sumX (int x) {
    int s = 0, i = 0;
    while (i < x) {
        // state s1
        i = i + 1;
        s = s + i;
        // state s2
    }
    return s;
}

//@ requires x >= 0;
//@ ensures \result == \sum(i: int; 0<=i<=x; i)

int sumX (x: Int) {
    int s = 0, i = 0;
    while (i < x) {
        // state s1
        i = i + 1;
        s = s + i;
        // state s2
    }
    return s;
}

//@ requires x >= 0;
//@ ensures \result == \sum(i: int; 0<=i<=x; i)

int sumX (int x) {
    int s = 0, i = 0;
    while (i < x) {
        // state s1
        i = i + 1;
        s = s + i;
        // state s2
    }
    return s;
}

//@ requires x >= 0;
//@ ensures \result == \sum(i: int; 0<=i<=x; i)

int sumX (int x) {
    int s = 0, i = 0;
    while (i < x) {
        // state s1
        i = i + 1;
        s = s + i;
        // state s2
    }
    return s;
}

//@ requires x >= 0;
//@ ensures \result == \sum(i: int; 0<=i<=x; i)

int sumX (int x) {
    int s = 0, i = 0;
    while (i < x) {
        // state s1
        i = i + 1;
        s = s + i;
        // state s2
    }
    return s;
}

//@ requires x >= 0;
//@ ensures \result == \sum(i: int; 0<=i<=x; i)
JML: loop annotations

- JML allows us to annotate a loop invariant and a variant function:
  
  ```
  //@ loop_invariant product==m*i && i>=0 && i<=n && n>0;
  //@ decreases n-i;
  while (i < n) {...}
  ```

- `loop_invariant`: the loop invariant
- `decreases`: variant function

- Why do we need a variant function for?
A *ghost* field is a regular field, except for the fact that we can only refer to it from the specification

- Example: `//@ ghost` Object F;

JML provides the special statements *set* for updating the value of a *ghost* field

- Instead of assigning a new value, the update is captured by using a *condition*.
- Example: `//@ set` F==null;
JML: ghost fields

class Animal {
    //@ ghost Zoo owner;
    ....
}

Class Zoo {
    void add(Animal a) {
        ...
        //@ set a.owner==this;
    }

    //@ requires a.owner==this;
    void feed(Animal a) {...}
}
Reachability in JML: {\texttt{\textbackslash reach}(...)}

- Returns the set of “reachable” objects

- \texttt{\textbackslash reach} captures the reflexive-transitive closure of a binary relation
  - \( R^* = \emptyset \cup R \cup (R;R) \cup (R;R;R) \cup (R;R;R;R) \cup \ldots \)

- The expression value is a JMLObjectSet (empty if only null is reachable)
Some `\(\text{reach}(\ldots)\)` flavours

- `\text{reach}(\text{this.f})`
  - All objects that are reachable by using any field in the reachable objects starting from this.f

- `\text{reach}(\text{this.f, T, f2})`
  - All objects that are reachable starting from this.f BUT
    - Only traversing field f2
    - Objects of type T
class List { Node header; }
class Node { Node next; Object data; }

- Write an invariant for class List such that all reachable nodes form an **acyclic list**.

```*/
@ invariant (\forall Node n;
  \(\text{reach}(\text{this.header}, \text{Node}, \text{next})).\text{has}(n)\);
@  !(\text{reach}(n.\text{next}, \text{Node}, \text{next})).\text{has}(n)\);
@*/
```
The invariant in detail

/*@
@ invariant (\forall Node n;
  \@ \(\text{\textbf{reach}}(\text{this.header, Node, next}).\text{has}(n) ;
  \@ !\(\text{\textbf{reach}}(n.\text{next, Node, next}).\text{has}(n) ) ;
@*/
What is wrong in this class?

```java
public class Counter {
    private /*@ spec_public @*/ int val;
    //@ modifies val;
    //@ ensures val == \old(val + y.val);
    //@ ensures y.val == \old(y.val);
    public void addInto(Counter y) {
        val += y.val;
    }
}
```
Example

- What is wrong with this class?

```java
public class Counter {
    private /*@ spec_public @*/ int val;
    //@ modifies val;
    //@ requires y!=this;
    //@ ensures val == \old(val + y.val);
    //@ ensures y.val == \old(y.val);
    public void addInto(Counter y){
        val += y.val;
    }
}
```