Improving OpenJML by adding Java Verification of Quantifier Comprehensions

Group 22, Sponsored by Dr. Gary Leavens

Colin Herzberg Robert Derienzo Robinson Vasquez
Sachin Shah

Spring 2022

Introduction

Coding is *easy*, but coding correctly is *difficult*

Introduction

Coding is *easy*, but coding correctly is *difficult*

Tests mean success!

Introduction

Coding is *easy*, but coding correctly is *difficult*

Tests mean success!

- Unit
- Component
- Functional
- Integration
- API
- UI
- Performance
- Security

Problem with the Testing Paradigm

1. Operational overhead of creating different types

Problem with the Testing Paradigm

1. Operational overhead of creating different types
2. Impossible to prove “bug-free” code

Problem with the Testing Paradigm

1. Operational overhead of creating different types
2. Impossible to prove “bug-free” code
3. No relation between tests and code specifications

Problem with the Testing Paradigm

1. Operational overhead of creating different types
2. Impossible to prove “bug-free” code
3. No relation between tests and code specifications

There should be a single and complete mechanism to verify program correctness

OpenJML Solution

The **Java Modeling Language** was designed to specify the behavior of a Java program

OpenJML uses this language to verify program correctness by matching the user intent with the written code

- Reduces development time
- Guarantees program correctness
- Reduces ambiguity of method purpose

OpenJML Solution

- **Runtime Assertion Checker**: creates pre/post condition functions to check function validity
  - Does not provide pre-production confidence
  - Lower barrier to entry
- **Extended Static Checker**: similar to type-checking, statically verifies program correctness
  - Can be slow for complex programs
  - Stronger confidence in program correctness

OpenJML Example

```
public class Max {

  //@ ensures \result >= i && \result >= j && \result >= k;
  //@ ensures \result == i || \result == j || \result == k;
  public static int max(int i, int j, int k) {
    int t = i > j ? i : j;
    return t > k ? t : k;
  }
}
```

```
Note: Summary:
    Valid:          2
    Invalid:        0
    Infeasible:     0
    Timeout:        0
    Error:          0
    Skipped:        0
  TOTAL METHODS: 2
  Classes:         1 proved of 1
  Model Classes: 0
  Model methods: 0 proved of 0
  DURATION:          2.9 secs
```

Limitations with OpenJML

```
//@ ensures (\forall int i; 0 <= i < array.length;
    array[i] <= \result);
//@ ensures (\exists int i; 0 <= i < array.length;
    array[i] == \result);
public int getMax(int[] array) {
    int bestMax = array[0];
    for (int i = 1; i < array.length; i++)
        if (bestMax < array[i])
            bestMax = array[i];
    return bestMax;
}
```

Limitations with OpenJML

```
//@ ensures (\max int i; 0 <= i < array.length; array[i])
    == \result;
public int getMax(int[] array) {
    int bestMax = array[0];
    for (int i = 1; i < array.length; i++)
        if (bestMax < array[i])
            bestMax = array[i];
    return bestMax;
}
```

Limitations with OpenJML

```
public int sum(int[] array) {
    int total = 0;
    for (int i = 0; i < array.length; i++)
        total += array[i];
    return total;
}
```

Why Use Quantifiers?

- Why do programming languages have built in functions?
- Implementing quantifiers provides many of same benefits using Math.max() in Java.
  - Reduces amount of code
  - Provides a consistent and correct method of completing the action
  - Allows the user to write more complex code without the need to fully understand the workings of the quantifier

Increasing Adoption of OpenJML

By making OpenJML easier to use, we can increase adoption of program verification

- Verifying simple statements must be equally simple and intuitive

System Diagram

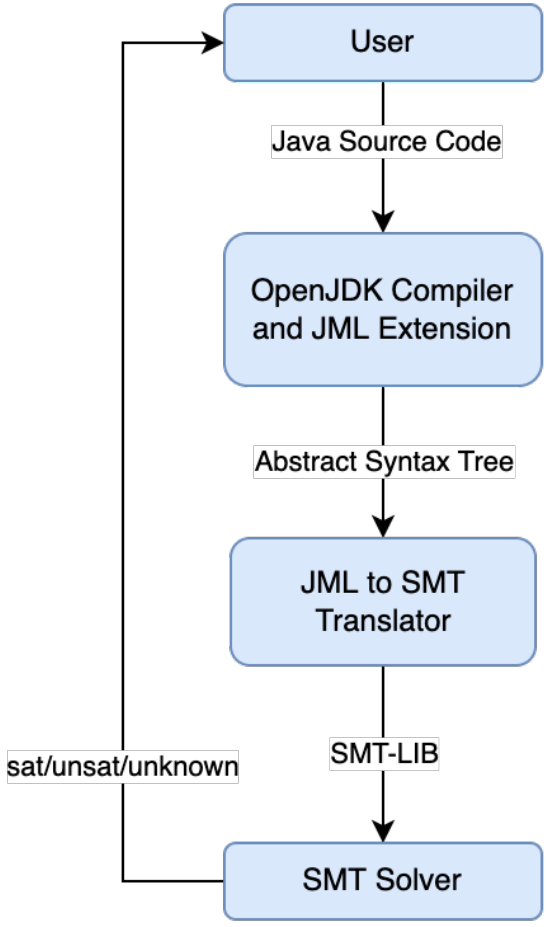

Figure: System Diagram of OpenJML and Quantifier Comprehension

Requirements

Three generalized quantifiers that need implementation:

- Sum
- Number of
- Product

Each quantifier needs:

- Java tests that verify and do not verify
- Hand translate Java tests to SMT-LIB format
- Create a general algorithm to translate JML to SMT-LIB
- Patch OpenJML with this algorithm

OpenJML Quantifier Syntax

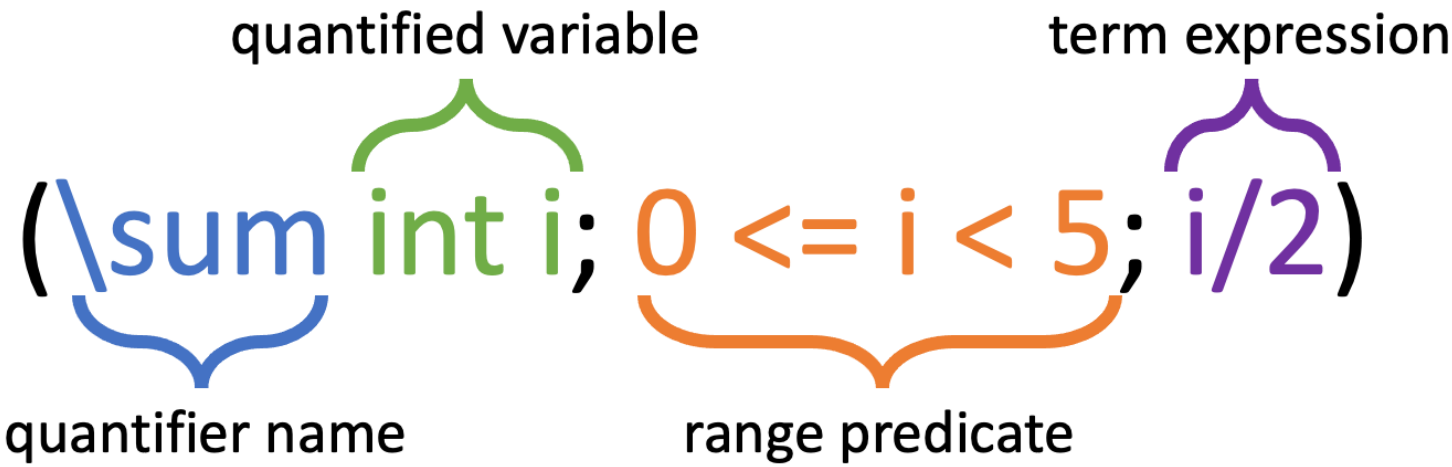

Figure: JML Quantifiers have four components

Examples

Quantifier comprehension can be used for a variety of problems that are encountered daily by programmers

- Priority queue
- Q-learning
- Cost of a path in a graph
- Integral approximations
- Geometric mean

Test case

```
//@ requires 0 < a.length < 10 && a != null;
//@ ensures (\sum int j; 0 <= j && j < a.length; (a[j] -
    mean)*(a[j] - mean)) == \result;
public int SSD(int[] a, int mean) {
    int result = 0;

    //@ loop_invariant 0 <= i <= a.length;
    //@ loop_invariant (\sum int j; 0 <= j && j < i; (a[j] -
        mean)*(a[j] - mean)) == result;
    for(int i = 0; i < a.length; i++) {
        result = result + (a[i] - mean)*(a[i] - mean);
    }
    return result;
}
```

Static Checking: Generalized Quantifiers

The primary obstacle is proving statements about unknown data

Our solution uses basic mathematical properties to prove statements that contain them:

Empty ranges:

$$\forall lo, hi \bullet hi \leq lo \Rightarrow \text{sum}\{\text{int k in (lo : hi); a[k]}\} = 0$$

Induction:

$$\forall lo, hi \bullet lo \leq hi \Rightarrow$$
$$\text{sum}\{\text{int k in (lo : hi+1); a[k]}\} =$$
$$\text{sum}\{\text{int k in (lo : hi); a[k]}\} + a[hi]$$

Static Checking: Axioms

- Unit
- Induction below
- Induction above
- Split range
- Same terms

Static Checking: Axioms

- Unit
- Induction below
- Induction above
- Split range
- Same terms

The implementation of these axioms in SMT involve matching triggers to guide solving
These come at the cost of causing infinite matching loops, so a bonus "connection" axiom is created

Budget

- Free and open source software under GPLv2
- Hosting code on Github

Project Timeline

	Task	Status
Dec. 11th	OpenJML and SMT-LIB research	Complete
Jan. 8th	Two Java test cases per quantifier	Complete
Jan. 15th	All Java test cases finished	Complete
Jan. 31st	Initial SMT translations	Complete
Feb. 14th	All SMT translations finished	In Progress
Feb. 23th	Translation algorithm review	Not Started
Mar. 1st	Integrate into OpenJML	Not Started
Mar. 8th	Review and Test Pipeline	Not Started
Mar. 25th	Submit Pull Request	Not Started

Distribution of work

- Fall Semester
  - Colin: Researched run-time assertion checking. Created tests for Sum and Product
  - Robert: Researched SMT background and project limitations. Created tests for Number Of
  - Robinson: Researched first order logic. Created tests for Min
  - Sachin: Researched solvers and static checking. Created tests for Max
- Spring Semester
  - Working as a group to create translation logic by...
  - Creating SMT-Lib tests individually
  - Using tests to create algorithms as a team

Project Status

- Developed 4 test that verify and 4 that do not verify for each quantifier

- Started writing axioms in python Z3 wrapper
- Can prove summation and product over fixed functional array is equal to their expected value
- ...