Improving OpenJML by adding Java Verification of Quantifier Comprehensions

Group 22, Sponsored by Dr. Gary Leavens

Colin Herzberg Robert Derienzo Robinson Vasquez
Sachin Shah

UCF - 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 <u>single</u> and <u>complete</u> 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** (RAC): creates pre/post condition functions to check function validity
  - Provides little pre-production confidence
  - Lower barrier to entry than Extended Static Checker
- **Extended Static Checker** (ESC): similar to type-checking, statically verifies program correctness
  - Can be slow for complex programs
  - Stronger confidence in program correctness than RAC

OpenJML Example

```
public class Example {
    //@ ensures \result == x + y;
    public static int add(int x, int y) {
        return x + y;
    }
}
```

OpenJML Example

```
public class Example {
    //@ ensures \result == x + y;
    public static int add(int x, int y) {
        return x + y;
    }
}
```

Overflow!

Limitations with OpenJML

```
//@ ensures \result == array[0] + array[1] + array[2] ...
public int sum(int[] array) {
    int total = 0;
    for (int i = 0; i < array.length; i++)
        total += array[i];
    return total;
}
```

OpenJML Quantifier Syntax

$$(\underbrace{\texttt{\textbackslash sum}}_{\text{quantifier name}}\ \overbrace{\texttt{int i}}^{\text{quantified variable}};\ \underbrace{\texttt{0 <= i < 5}}_{\text{range predicate}};\ \overbrace{\texttt{i+2}}^{\text{term expression}})$$

Figure: JML Quantifiers have four components

How Do Quantifiers Solve The Problem

- **Provability** - allows previously un-provable methods
- **Consistency** - eliminates the need for complex hacks and is provably correct
- **Completeness** - JML Specification

Example Use Cases

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

Increasing Completeness of OpenJML

More useful toolkit for proving functions

- Broadens applications of OpenJML
- Provides intuitive mechanisms to prove simple programs

System Diagram

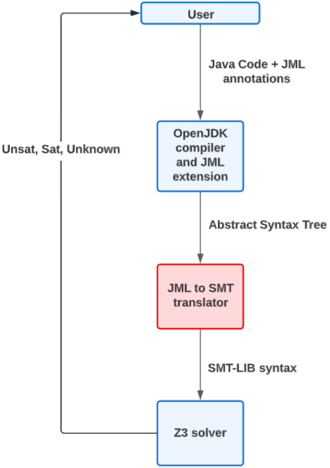


Figure: OpenJML ESC

SMT and Solvers

- SMT-LIB is a common language used by many SMT solvers
- OpenJML is released with both Z3 and CVC4
- OpenJML generates SMT-LIB to preserve solver compatibility

SMT-LIB Example

```
; Variable declarations
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun c () Int)

; Constraints
(assert (> a 0))
(assert (> b 0))
(assert (> c 0))
(assert (= (+ (* a a) (* b b)) (* c c)))

; Solve
(check-sat)
(get-model)
```

Requirements

Three generalized quantifiers 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

Budget

- Budget \$0.00, Spent \$26.32
- Free and open source software under GPLv2
- Code hosting on Github
- Communication with Zoom and Discord

Distribution of Work

- Fall Semester
  - Conducted research and created basic test cases
  - Collaborated on discussions and writing
- Spring Semester
  - Worked as a group to develop translation logic
  - Created SMT-LIB tests individually
  - Used tests to create algorithms as a team

Lacking Initial Understanding

- Referred to *Reasoning about Comprehensions with First-Order SMT Solvers* by Leino & Monahan (2009)
- Too focused on trying to implement every axiom presented without understanding them fully
- Could not produce sat or unsat using all of the axioms presented in the paper

Testing With z3py

- Started testing with z3py
- Could write simple proofs, but could not use quantifiers
- Found the pythonic abstractions lead to confusion

Simplifying the Approach

- Proved simple static sums with basic axioms from Leino & Monahan
- "What even is it that we are trying to do?"
- Wrote a proof for sum and equated it to $\frac{n(n-1)}{2}$
- Simple Sum and Product examples worked, but lacked vision of how this ties into OpenJML

Simplifying the Approach

- Learned about if-then-else statements in SMT-LIB from Dr. Cok and Dr. Leavens
- No longer needed to deal with implications, and could create a one line function definition for quantifiers
- Found that `define-fun-rec` allows for inline declaration and definition

Leveraging New Tools

- Created Simple SMT-LIB cases that represented how an OpenJML SMT-LIB file proves
- Developed a simple python script that translated from JML to SMT for very simple programs
- Began OpenJML implementation

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:

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

Static Checking: Recursion

- **Base case**: when the current index is less than the lowest value in the range, return either 0 for sums and counting or 1 for products
- **Recursive step**: accumulate the result of recursion on *low* and $high - 1$ with the value at *high* if the filter at *high* is true otherwise, the base case value
- **Usage**: Extract the *low* and *high* bounds from the range extremes and call the previously defined function

Static Checking: SMT Sketch

Quantifier Name

Quantified Variables

Base Case

Recursive Call

Body

```
(define-fun-rec sum
  ((lo Int)(i Int)) Int
    (ite (< i lo)
      0
        (+
          (sum lo (- i 1))
          (ite (and (< 0 i) (<= i 3)) i 0)
        )
      )
  )
```

Filter/Range Expression

Body for False Filter

Figure: SMT-LIB Translation for a Simple Sum

Static Checking: Filters

```
(\sum int i; 0 < i <= 3; i);
```

To evaluate elements within a filter use 'If-Then-Else' (`ite`) logic

```
(ite (and (< 0 i) (<= i 3)) i 0)
```

Static Checking: Bounds Extracting

“Split ranges” include more than one interval.

```
(\sum int i; 0 <= i < 3 && 5 <= i < 7; i);
```

This might translate to:

```
(sum 0 7)
```

Static Checking: Bounds Extracting

What happens, when the bounds are variables ?

```
(\sum int i; low <= i < mid1 && mid2 <= i < high; i);
```

```
(sum
    (ite (< low mid2) low mid2)
    (ite (> mid1 high) mid1 high)
)
```

Static Checking: \sum

```
(\sum int i; 0 < i && i <= 3; i);
```

This might translate to:

```
(define-fun-rec sum
  ((lo Int) (i Int)) Int
  (ite (< i lo)
    0
    (+
      (sum lo (- i 1))
      (ite (and (< 0 i) (<= i 3))
        i 0
      )
)))
```

Static Checking: \product

```
( \product int i; 0 < i && i <= 3; i);
```

Only change the function name, and operator and base cases.

```
(define-fun-rec product
  ((lo Int) (i Int)) Int
  (ite (< i lo)
    1
    ( *
      ( product lo (- i 1))
      (ite (and (< 0 i) (<= i 3))
        i 1
      )
)))
```

Static Checking: \num_of

```
(\num_of int i; 0 < i && i <= 3; true);
```

\num_of is equivalent to a summation

```
(\sum int i; 0 < i && i <= 3 && true ; 1 );
```

```
(define-fun-rec sum
  ((lo Int) (i Int)) Int
  (ite (< i lo)
    0
    (+
      (sum lo (- i 1))
      (ite (and (and (< 0 i) (<= i 3)) true )
        1 0
      )
)))
```

Demo

Project Timeline

	Task	Status
Dec. 11th	OpenJML and SMT-LIB research	Complete
Jan. 8th	Two Java test cases per quantifier	Complete
Jan. 31st	Initial SMT translations	Complete
Feb. 14th	All SMT translations finished	Complete
Feb. 23th	Translation algorithm review	Complete
Mar. 1st	Sum Integration into OpenJML	Complete
Mar. 8th	Implement Bounds Extractor	Complete
Mar. 15th	Product Integration into OpenJML	Complete
Mar. 21st	NumOf Integration into OpenJML	Complete
Apr. 1st	Review and Test Pipeline	Complete
Apr. 8th	Document successes and failures	Complete
Apr. 19th	Submit Pull Requests	Complete

Accomplishments

- Algorithm for translating JML quantifiers to SMT
- Proves quantifier expressions
- Handles complex ranges using the bounds extractor
- Allows quantifier expression nesting

Limitations

1. Large ranges are slow
   - SMT Solvers unravel and try to solve the proofs manually
     - This could be due to the filter function
   - No support for infinite ranges from floating point quantified variables
   - Limited warnings around infinite ranges, leading to possible long solve times
2. No support for the specified syntax for multiple quantified variables
   - Alternative is nesting

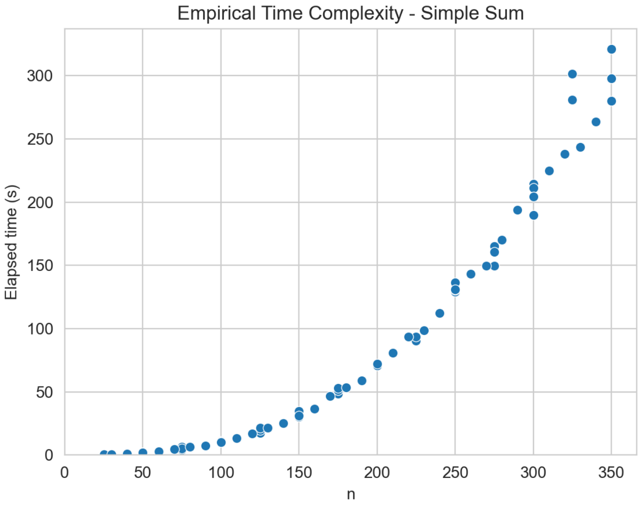
Nesting Quantifiers

```
(\sum int i, j; 0 <= i < j; i+j)
```

translates to

```
(\sum int i; true; (\sum int j; 0 <= i < j; i+j))
```

Empirical Time Complexity



Empirical Time Complexity

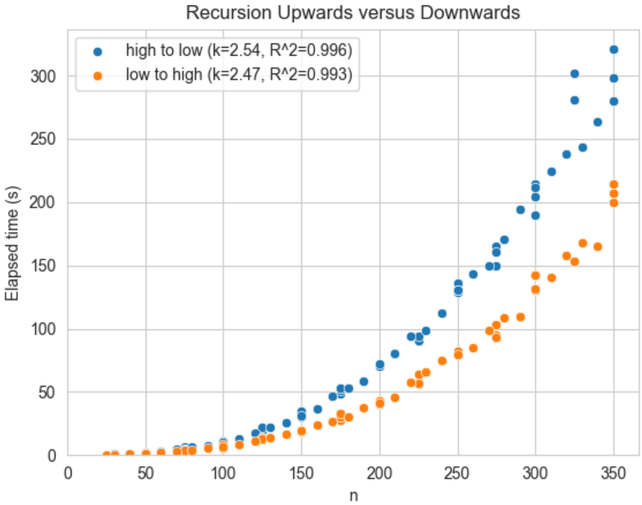
$$y = ax^k$$
$$\log y = k \log x + \log a$$

If $Y = \log y$ and $X = \log x$, a linear regression can be fit

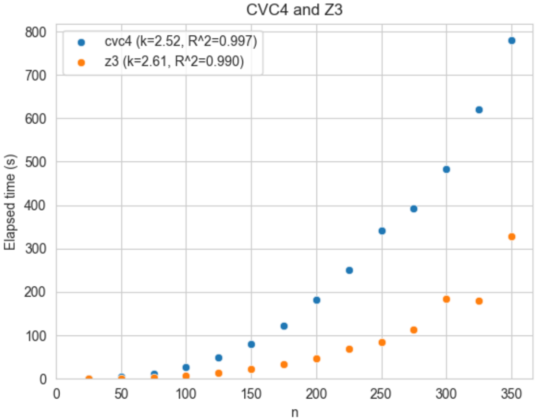
$$Y = 2.5394X - 3.9984 \text{ with } R^2 = 0.9964$$

$$O\left(N^{2.54}\right)$$

Better Exponent



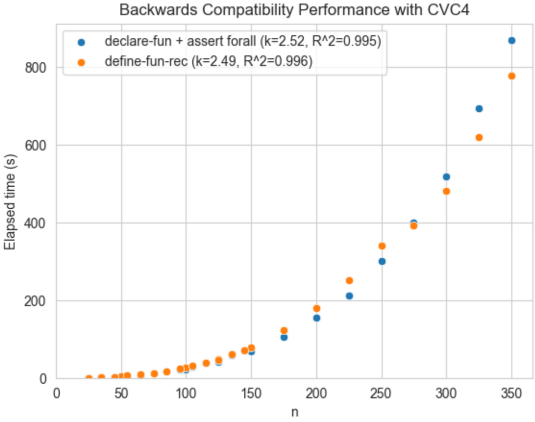
Solver Differences



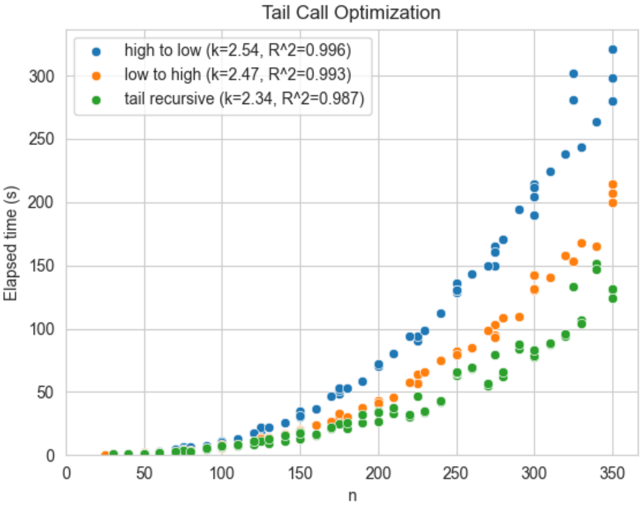
Although CVC4 has a better exponent, the constant $10^{-8.2}$ is an order of magnitude worse than z3's constant of $10^{-9.9}$

Backwards Compatibility

`define-fun-rec` is new syntactic sugar for `declare-fun` and `(assert (forall ...))`. z3 does some additional work under the hood, but cvc4 operates the same.



Even Better Exponent



Future Work

1. Understand why the unraveling happens and how we can prevent it
2. Solve this issue to allow verification with large ranges
3. Remove warnings around large and infinite ranges
4. Enable syntactic sugar for nesting
5. Add support for floating point quantifier variables