SASP Project Report 2012

Implementing Quantified Expressions for OpenJML

A Pattern Matching Approach

Pelle Krøgholt[*] Florian Biermann[†]

May 22, 2012

This project report for the course Advanced Models and Program in spring 2012 documents our efforts to extend the OpenJML framework. Our project focuses on implementing quantified statements into the runtime assertion checker of OpenJML. The report outlines our implementation and emphasizes on the model behind the implementation.

[*]pelle@itu.dk
[†]fbie@itu.dk

Contents

1. Introduction

This project report for the course Advanced Models and Program in spring 2012 documents our efforts to extend the OpenJML[1] framework. Our project focuses on implementing quantified statements into the runtime assertion checker of OpenJML.

1.1. An Overview of OpenJML

OpenJML is a tool to verify the correctness of Java 7 code by specifying the behavior of classes and methods using mathematical models [Cok, 2011a,b]. It is built on top of the OpenJDK[2] compiler and has a Java-like syntax (the JML syntax) to add pre- and post-conditions to source code as well as invariants. These conditions are written by the developer in comments throughout the sources or in a separate file. The tool then comes with three different variants to check the correctness of source code specifications:

- Static analysis
- Extended static analysis (ESC)
- Runtime assertion checker (RAC)

While the static analysis only checks the correctness of the JML statements, the ESC is able to verify the correctness of the program's behavior to a certain extend by implying automated like Yices[3] or interactive provers like Coq[4] [Cok, 2011a, Burdy et al., 2005, Chalin et al., 2006].

The RAC compiles the JML specifications in to the binary code and checks that invariants and pre- and post-conditions hold during executing the program. Because the OpenJDK compiler is part of OpenJML, the AST generated during compile-time can be altered directly so that actual assertions will be executed before and after each call of a method. Using additional tools, it is possible to generate test-suites for the RAC-compiled Java binaries to quickly get huge coverage of unit testing [Cheon and Leavens, 2002, Zimmerman and Nagmoti, 2011].

1.2. Overview of the Report

In this project, we have investigated and implemented a solution for evaluating quantified statements over integers. In the current OpenJML trunk version[5], quantified statements can only be evaluated for one race variable. As the ESC of OpenJML is currently being overhauled entirely, we focused on developing a solution for the OpenJML RAC.

In the following report, we will outline the problem further and give examples of currently not evaluated quantified statements. For brevity, we will focus only on the implementation of the *for all* expression. Next we will describe a number of possible solutions, starting with the most naive approach, and explain our design decisions in the solution. We will then explain our solution in detail, followed by a section to outline future work on our proposed solution

[1] http://jmlspecs.sourceforge.net/
[2] http://www.openjdk.org
[3] http://yices.csl.sri.com/
[4] http://coq.inria.fr/
[5] http://sourceforge.net/apps/trac/jmlspecs/log/OpenJML/trunk/, Revision 2543

2. Problem Outline

In this section we will outline the current state of the implementation of quantified expressions in OpenJML RAC. Further we will elaborate on the problem by giving an example of a naive approach towards solving the evaluation of range expressions and point to more cases where the execution of a quantified expression is difficult.

For brevity and to get a deeper understanding of the underlying implications and problems, we have focused on the *\forall* statement over integers during the project. Since all quantified expressions can be evaluated using the same technique just with a minor alteration, it will be possible to apply our results to general quantifiers in OpenJML.

2.1. Current Implementation of Quantifiers in OpenJML

As of revision [rev] in the OpenJML trunk, the following statement will be compiled into RAC binary code:

```
//@ requires (\forall int i; 0 <= i && i <= 10; p(i));
```

where $p(i)$ is a predicate that should hold for all i. A more general form is:

```
//@ requires (\forall i; R; P);
```

where R is a boolean expression that defines a range and P a boolean expression that defines the predicate which should hold for all i. The notation is similar to the set-builder notation, where the values inside of a set are denoted through a boolean expression:

$$i \in N | 0 \leq i \wedge i \leq 10$$

The OpenJML RAC will compile a check into the method decorated with the above JML statement that runs a *for-loop* for each i for which the given P is asserted. If the check runs over an i for which P does not hold, an Exception is thrown, notifying the user about the violation of the condition.

However, conditions in JML that use the *\forall* statement are likely to become more elaborate. Take the following as an example, where $p(i)$ and $q(j)$ are predicates that must hold for all i and j.

```
//@ requires (\forall int i, j; (100 >= i && i > 0 || i == 200) && (-100 < j
    && 100 > j); p(i) && q(j));
```

Here, we have multiple new issues:

- There is more than one race variable declared in the expression.
- The order of the boolean range description is entirely arbitrary.
- i can not only be inside a single well defined range but additionally become a value outside of $[1, 100]$.

An expression like given in this example will currently (as of trunk revision 2543) not be executed when compiled into RAC in OpenJML. This is mostly due to the declaration of two race variables. However, OpenJML RAC relies on a heuristic to identify the set of values which i and j can take and therefore highly relies on the layout of the expression.

2.2. The Naive Approach

The most naive approach sees the range expression R as a predicate that has to hold for all integers. Code to check for the expression given in Section 2.1 could look as follows.

```
for(int i = Integer.MIN_VALUE; i <= Integer.MAX_VALUE; i++){
  for(int j = Integer.MIN_VALUE; j <= Integer.MAX_VALUE; j++){
    if((100 >= i && i > 0 || i == 200) && (-100 < j && 100 > j)){
      assert p(i) && q(j);
    }
  }
}
```

While this is a valid check, this approach has a runtime of $\mathscr{O}(|Integer|^{Number\,of\,race\,variables})$. This is impractical for conducting actual runtime assertion checks. While it is obvious that running RAC-compiled code is very slow, it should still be runnable within a reasonable time. This illustrates that the naive approach is not a good solution to the question of how to implement quantifiers with multiple race variables.

2.3. Further Difficulties

Of course, quantifiers should not only be applicable to integers as race variables but also for other primitive data types like booleans, chars or even floats and as well for any kind of object in Java, for which a range can be defined. Due to time difficulties we omitted work on these cases and instead tried to build a solid foundation for integer quantification. If our foundational work is properly executed, it may be possible to extend our approaches to include more of the mentioned cases.

However, our approach is mainly targeted towards primitive types as there are other techniques and approaches to assert quantified expressions for arbitrary objects. These aim to manipulate the actual behavior of objects to be able to verify conditions that these must hold without having to generate every possible instance of this type.

3. Implementation

Our implementation has two main features. It relies on a rigorous pattern matcher that tries to eliminate as much heuristics as possible. It is able to procedurally produce code that represents the values defined in the quantified expression by the set-builder notation. This representation in turn relies on our implementation of a class that implements Iterator and therefore can efficiently calculate allowed values on the fly through operations on sets. We will go into further detail in the following subsections.

Our implementation is, however, not yet tightly integrated into the OpenJML infrastructure. Currently it is in an experimental stadium and mostly relies extensively on string comparison. Future work will be to integrate it into the visitor class used in OpenJML that walks the AST generated by the parser and modifies it accordingly to RAC.

3.1. QRange - Implementation of a Pattern Matcher

We implemented a rigorous pattern matcher that analyzes an expression recursive through a unification-like algorithm. The pattern matcher is part of a binary tree that, if asked, will return code that produces another binary tree that represents the actual values. This behavior is implemented in *QRange.java* (see Appendix C). The implementation was inspired by the original QSet class implemented as part of JML2[6] [Cheon and Leavens, 2002].

3.1.1. Rigorous Pattern Matching

JML expressions are, after parsing, represented by a tree structure implemented in *JmlExpression*. The *JmlExpression* processed by the pattern matcher is the expression that defines which values are defined for a given variable name, i.e. the range expression. The idea is to find all sub-expressions that mention a variable name without applying any arithmetic operators to it, therefore actually give a definition for its value range. This assures that only expressions that are intended to define values for a variable name actually define its values. Other sub-expressions are ignored. Further, the boolean expression is broken down until it contains an atomic boolean expression between integers, i.e. an expression of the form $i == j$. To achieve this behavior, we implemented two levels of pattern matcher.

The first level breaks a *JmlBinaryExpr* object down into its subexpressions and stores the representation by matching the operator of this binary expression. After our definition, a higher-level binary expression has either the operator *&&* or ||. The underlying implication here is that this expression is actually building a set. Therefore we translate the operators into their set operation counterparts. *&&* is defined as set intersection, || is defined as a set union. All not matched operators will result in an exception thrown which informs the user that the expression can not be evaluated.

Let *op* be a binary boolean operator and *E1*, *E2* boolean expressions of any form. The pseudo-code is denoted in a functional-style, [] the described function.

```
[E1, op, E2] :=
if var not in E1, E2 then Ignore() else
match op with
| "&&" => Intersection([E1], [E2])
| "||" => Union([E1], [E2])
| ">" | ">=" | "<" | "<=" | "==" | "!=" => Interval(E1, op, E2)
| _ => Exception
```

Note that if the first line's condition is hit, a *QRange* of type *Ignore* is returned. If the expression that is being evaluated is the only given one, our convention for the definition for this variable through Ignore will be *[Integer.MIN_VALUE, Integer.MAX_VALUE]*, as we presume that the user actually wants to check *every* possible integer.

The second level of the pattern matcher performs a unification-like algorithm. First, note that a single atomic boolean expression can only define a single margin value of an interval. Mathematically this means that one of the two boundaries of the interval will be either infinity or negative infinity. Programmatically however, this is not a real problem as the lowest and highest representable integers are discrete values.

[6] http://jmlspecs.cvs.sourceforge.net/viewvc/jmlspecs/JML2/

Because of this fact, the low and high border of an interval are set initially to *Integer.MIN_VALUE*, *Integer.MAX_VALUE* respectively. Then, the pattern matcher determines if the expression defines the lower or the upper border of the interval. The algorithm then unifies the expression so that the variable always takes the left part of the expression. Afterwards it is trivial to infer if the expression defines a high or a low boundary.

Let *op* be a binary boolean operator on integers, *E1*, *E2* either the variable that should be defined or a (reference to an) integer, high and low data fields that together represent an interval. The pseudo-code is again denoted in a functional-style, [[]] the described function.

```
[[E1, op, E2]] :=
if "++" in E1, E2 || "--" in E1, E2 then Exception else
if E2 == var && var not in E1 then [[E2, inverted(op), E1]] else
match op with
| "<=" => high = E2
| ">=" => low = E2
| "<" => [[E1, "<=", E2 - 1]]
| ">" => [[E1, ">=", E2 + 1]]
| "==" => low = E2, high = E2
| "!=" => low = E2 + 1, high = E2 - 1
| _ => Exception
```

Pre- and postfix in- or decrementors are not allowed, as they would alter the value of race variables while defining another one. The rule for the operator *!=* is a way to define an interval where only a single value is not defined. The implementation of *IntervalSet*, which will execute the values computed here, allows this definition and produces valid results, as we will describe in the next subsection. However, there are cases where the pattern-matcher does not apply correctly. We will outline these in the next section.

3.1.2. Code Generation

The outlined algorithm builds a binary tree of objects of type *QRange*, where the operation for union and intersection as well as the actual interval are subtypes of it. Additionally, we implemented an ignore subtype that indicates that this sub-expression should be ignored. After the recursive construction of the binary tree, a method translate can be called on the object that returns a string holding code to create the specified interval using the *IntervalSet* type.

Translate walks the tree recursively. It is implemented on each sub-type differently and therefore returns the type-specific operation. I.e. the union type of *QRange* returns code to build a union of *IntervalSet*, the singleton type returns code to build a singleton of *IntervalSet* and so on. If a ignore type is found, the set operation is omitted and the respective other sub-tree's code is returned exclusively.

3.2. IntervalSet - Binary Tree Representing Set Operations

IntervalSet (see Appendix D) is also implemented as a binary tree. Structure-wise, it is very similar to *QRange*. This makes sense, since *QRange* is a meta-level implementation of *IntervalSet*. During compile time, the structure of a *QRange* instance is projected one-to-one on the structure of an *IntervalSet* instance.

Additionally, *IntervalSet* implements *Iterator* and *Iterable* of *Integer*. This makes it convenient and quick to run over all values in an *IntervalSet* using a shorthand *for-loop*.

```
for(int i: IntervalSet.interval(0, 10)){
  System.out.println(i);
}
```

This will print out all numbers from and including 0 to 10.

IntervalSet has subclasses that represent union, intersection and a singleton, for the reasons outlined. Again, only the singleton type actually holds values that represent interval borders. While in *QRange*, these subclasses mostly only differ in the way they generate source code, they have three methods each that differ in behavior. One simply determines through boolean operations, if a given value is inside the set. The methods *getNextLow* and *getNextHigh* can however calculate a new low or high value given the tree structure is interpreted as set-operations.

The main difference is, that the union subclass can answer two different lows, highs respectively, depending on what the current value is, when iterating over an *IntervalSet*, while intersection will only always answer the higher low or the lower high. If neither high nor low is greater than the current value, the current value itself is returned instead. Following this, the condition to stop iteration is that current equals low and high at the same time.

Due to set-builder notations not necessarily being properly written, it is not given that a balanced tree will result from parsing and evaluating a JML *\forall* statement. Still, walking an unbalanced tree is faster as implementing the naive approach, especially because expressions usually are rather brief and therefore would not result in very deep trees.

There are still certain issues outstanding with this implementation, which we will describe in the next section.

3.3. ForAll - Generating For-Loops for Multiple Race Variables

To generate a quantified expression including multiple race variables, we implemented a class that recurses over each declared variable in the quantified expression (see Appendix B). It looks for a valid definition inside the range expression contained in the quantified expression and generates a nested for-loop for each variable. After the last variable has been processed, an assertion, as it is stated in the quantified expression, is placed in the body of the inner most *for-loop*.

The class returns the generated code on a call to *translate*. In *ForAll*, there are as well still issues that need to be resolved as pointed out in the next section.

3.4. JML Specifications

We provide lightweight JML specifications throughout our implementation in as many places in the code as possible.

3.5. Testing

Additionally to writing specifications, we manually build several test cases to make sure our implementation works correctly. They can be found in Appendix E.

4. Outstanding Issues and Future Work

As hinted before, our solution is not yet complete. There are multiple issues that we were unable to fix given the short period of time during which this project was conducted. The following subsections will go into detail regarding errors in our code.

4.1. Pattern Matching

While the pattern matcher behaves correctly and is error prone than the JML2 pedant for the most expressions, there are cases we do not yet cover. The pattern matcher does not properly take care of relational expressions of the form

```
\forall int i, j; 0 < i && i < 10 && j == i + 1; p(i) && p(j);
```

These are simply disregarded entirely. The pattern matcher will produce arbitrary results as the heuristic we use is not strong enough to determine if an expression mentioning a variable name is defining the variable or defining another one.

Also, recursive definitions are not checked against. This is important to not have the virtual machine crash when crossing recursive variable definitions.

Another important part which the current implementation does not take care of is the use of JML operators such as implications in the range expression. Since these are still extremely useful for writing specifications, it would be vital to support them.

4.2. Interval Representation

The interval representation implemented in *IntervalSet* is complete except for the representation of an interval of the form $[n,n]$. Since in this case

```
curent == low && current == high
```

this interval would fulfill the stop condition on the iterator, hence *hasNext* always returns false. Therefore, no value would ever be produced in a for-loop, even though the method *next* would return the (single only valid) value.

To avoid these issues and to avoid over-complicating the code by introducing several borderline cases, it might be easier to implement the polyhedra analysis algorithm outlined by Charles et al.. Polyhedra analysis seems to provide greater reliability, this technique has a high complexity, the problem is NP-hard [Charles et al., 2009], correctness is of greater importance when doing program analysis.

Still, we will keep working on the problem of interval representation to find a solution without turning to polyhedra analysis.

4.3. Loop Generation

The loop generation in *ForAll* produces nested loops. To do so it relies on the order of the race variable declaration. An expression of the following form would not be accepted and result in an error:

```
\forall int i, j; 0 < j && j < 10 && i == j; p(i) && p(j);
```

To make sure that, regardless of the order of declaration, the loops would be executable, it would be necessary to check for relational definitions and to change the order of loops accordingly.

Additional, it would be easy to extend our approach towards implementing the *\forall* expression by dynamically identifying the type of the quantified expression. Instead of only inserting assertions then, a sum expression could be stated or the assertion could be reversed in its assertion context to provide the functionality for an exists quantifier. We were unable to implement this due to time constraints in this project.

4.4. Validation

Unfortunately, JML2, while still being the most complete tool for checking JML specifications together with ESC/Java2, is not able to perform properly on Java 7. Due to this we ran the current OpenJML extended static checker on it to find the most basic flaws in our design. However, this proved to impossible due to many JML related errors in the OpenJML trunk that have not yet been taken care of. We were unable to check out specifications properly. Hence, this is part of future work on this project.

5. Conclusion

In this report we described our approach towards implementing quantified expressions (more precisely the for all expression) in an efficient way for OpenJML. While the described implementation is still experimental and not using all the internal structures of OpenJML, it should be possible to implement our approach in a more tied-in fashion. This would mean that instead of returning strings containing code, the implementation would modify a given AST before compilation so that the quantified expressions would be executed during RAC, much like it is performed with other checks already.

We showed that it is possible to use pattern matching to efficiently determine interval borders, even though our implementation is still lacking some rigorousness. Relational or recursive definitions are not properly matched yet. However, we believe that it would be possible to implement both similar to how it has been implemented originally in JML2.

Our approach does not try to calculate values initially but represents the constraints as set-operations in a binary tree. Our implementation takes advantage of the *Iterator* and *Iterable* classes, so that a one can simply iterate over the interval representing object. It answers sub-intervals when asked for in combination with the current race value with regard to set operations inferred from the range expression. Therefore we avoid heavy calculations.

We solved the issue of multiple race variables by introducing nested for-loops for quantified expressions with more than one declaration. However, there are still outstanding issues, like relying on declaration order, which can interfere with relational definitions inside the range expression.

Throughout the report we reflected upon these implementations and outstanding issues extensively and pointed out further difficulties that need to be engaged for future work. Our implementation does still produce runnable code for the given test cases, and is, to a major extend, supported by JML specifications.

References

L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005. ISSN 1433-2779. doi: 10.1007/s10009-004-0167-4. URL `http://www.springerlink.com/content/t6qp52glycr4jx68/abstract/`.

P. Chalin, J. Kiniry, G. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In F. de Boer, M. Bonsangue, S. Graf, and W. de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-36749-9. URL `http://www.springerlink.com/content/v5125m8010411665/abstract/`.

P. Charles, J. Howe, and A. King. Integer polyhedra for program analysis. In A. Goldberg and Y. Zhou, editors, *Algorithmic Aspects in Information and Management*, volume 5564 of *Lecture Notes in Computer Science*, pages 85–99. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-02157-2. URL `http://www.springerlink.com/content/a02471013pmj0452/abstract/`.

Y. Cheon and G. T. Leavens. A runtime assertion checker for the java modeling language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas, Nevada, USA*, page 322–328, 2002. URL `http://archives.cs.iastate.edu/documents/disk0/00/00/02/74/00000274-00/jmlrac.pdf`.

D. Cok. OpenJML: JML for java 7 by extending OpenJDK. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer Berlin / Heidelberg, 2011a. ISBN 978-3-642-20397-8. URL `http://www.springerlink.com/content/50p463l035ltv072/abstract/`.

D. R. Cok. The OpenJML user guide DRAFT IN PROGRESS, 2011b. URL `http://jmlspecs.sourceforge.net/OpenJMLUserGuide.pdf`.

D. Zimmerman and R. Nagmoti. JMLUnit: the next generation. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software*, volume 6528 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-18069-9. URL `http://www.springerlink.com/content/9t4r84636062l355/abstract/`.

A. Instructions on How to Execute the Code

Setup OpenJML project Eclipse project Based upon the official OpenJML setup guide[7]. Works for us on OSX 10.6.8 and Ubuntu 12.4.

Prerequisites Java 7

Setup trunk / branches Create a standalone folder/workspace for the code:

```
$ cat open_jml_trunk_sourcecode/svn_update.sh #!/bin/sh
svn co https://jmlspecs.svn.sourceforge.net/svnroot/jmlspecs/JMLAnnotations/
    trunk JMLAnnotations
svn co https://jmlspecs.svn.sourceforge.net/svnroot/jmlspecs/OpenJML/trunk/
    OpenJDK OpenJDK
svn co https://jmlspecs.svn.sourceforge.net/svnroot/jmlspecs/OpenJML/trunk/
    OpenJML OpenJML
svn co https://jmlspecs.svn.sourceforge.net/svnroot/jmlspecs/OpenJML/trunk/
    OpenJML-UpdateSite OpenJML-UpdateSite
svn co https://jmlspecs.svn.sourceforge.net/svnroot/jmlspecs/OpenJML/trunk/
    OpenJMLFeature OpenJMLFeature
svn co https://jmlspecs.svn.sourceforge.net/svnroot/jmlspecs/OpenJML/trunk/
    OpenJMLUI OpenJMLUI
svn co https://jmlspecs.svn.sourceforge.net/svnroot/jmlspecs/Specs/trunk
    Specs
svn co https://jmlspecs.svn.sourceforge.net/svnroot/jmlspecs/OpenJML/vendor
    vendor
```

Create a separate Eclipse workspace which contains no code:

```
$ ls -a open_jml_trunk_workspace
. .. .metadata
```

Get a *fresh* Eclipse SDK (both 32/64 bit 3.7.2 worked for us). Start Eclipse using the *empty* workspace. Then import as existing projects (without copying)

- JMLAnnotations
- OpenJDK
- OpenJML
- OpenJML-UpdateSite
- OpenJMLFeature
- OpenJMLUI
- Specs

Then at least clean projects and build. Turn off auto build since it otherwise will build projects each time saving-actions are done.

[7] http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJmlSetup

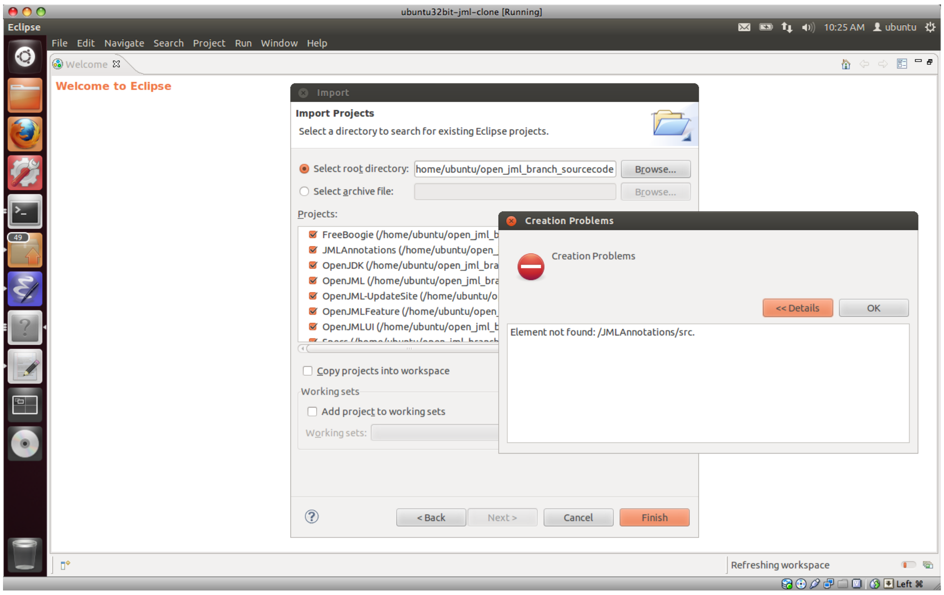

Figure 1: Eclipse Import Error on a Non Existing *src* Folder

Trouble shooting on import Turn on show all *.* resources under the *filters* e.g. add exception for **.svn* which we don't wanna see.

We experienced an import error on a non existing *src* folder (see Figure 1).

Clean and build project again, then it raises the error: "Project 'OpenJML' is missing required Java project: 'Specs"'. Simply import Specs again and clean plus rebuild all projects at the same time again. And now no errors in the log, *only* a bunch of warnings.

Setup OpenJMLExtended Eclipse project Check out the code from svn (public repository):

```
svn co http://sasp-f2012-jml-and-more.googlecode.com/svn/OpenJMLExtended/tags
    /final_handin OpenJMLExtended
```

Then import OpenJMLExtended as Eclipse existing projects (here, copying can safely be done).

Run OpenJMLExtended Eclipse project Open the Eclipse with a workspace installed in the above way. It should look as follow in Figure 2. Run one of the launch configurations for example *TestAllOpenJMLExtended.launch* which executes all the JUnit tests within the OpenJMLExtended Eclipse project.

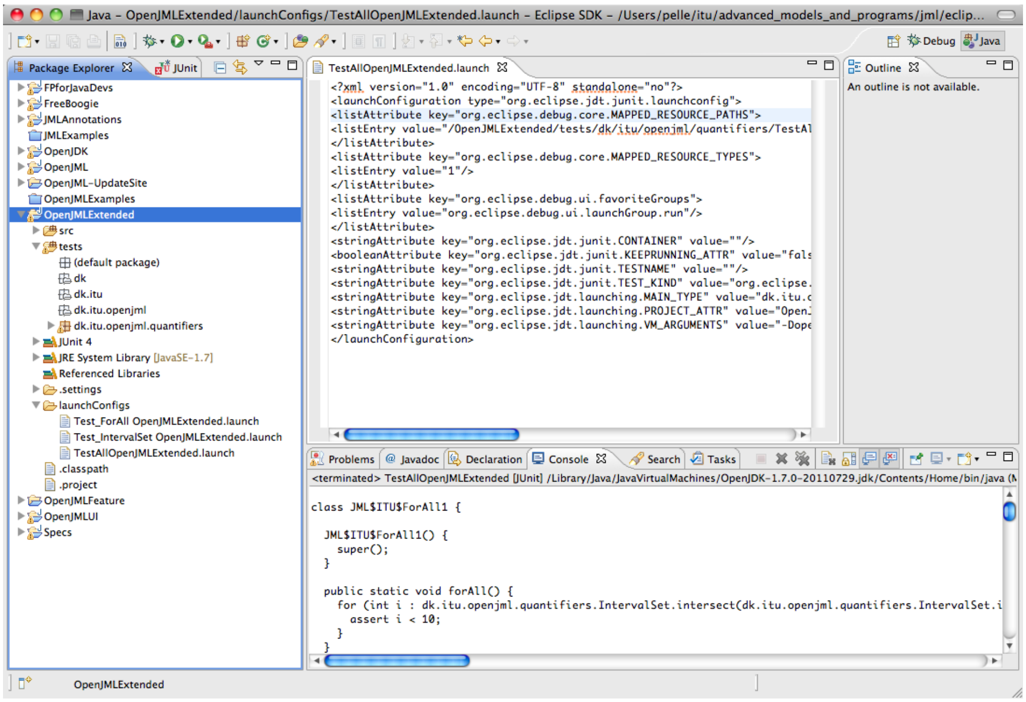

Figure 2: OpenJMLExtendedExlipseProject

B. ForAll.java

```java
package dk.itu.openjml.quantifiers;

import org.jmlspecs.openjml.JmlTree.JmlQuantifiedExpr;

import com.sun.tools.javac.tree.JCTree.JCVariableDecl;
import com.sun.tools.javac.util.ListBuffer;

import dk.itu.openjml.quantifiers.QRange;

/**
 * This is the class that generates code to  evaluate JML \forall
 * expressions using QRange to interpret a range expression and
 * Range and Interval to build a list of numbers that represents
 * all valid values for a variable.
 */
public class ForAll {

  String generated;
  JmlQuantifiedExpr expr;

  private final static String LOOP_START = "for(";
  private final static String LOOP_SEPARATOR = " : ";
  private final static String LOOP_END = ")";
```

```

private final static String BLOCK_START = "{";
private final static String BLOCK_END = "}";

private final static String SEPARATOR = " ";
private final static String STATEMENT_END = ";";

// NOTE: #9
private final static String ASSERT = "assert";

/**
 * Constructs a string that holds code to evaluate the
 * given expression
 * @param e The JmlQuantifiedExpr to generate evaluation code for
 */
//@ assignable generated, expr;
public ForAll(JmlQuantifiedExpr e){
  generated = "";
  expr = e;
  generate();
}

/**
 * Generates the code. If the quantified expression is not executable,
 * the incident gets reported and an empty statement (;) is produced.
 */
protected void generate(){
  try{
    addLoops(getDeclarations());
  } catch (QRange.NotExecutableQuantifiedExpr e){
    // NOTE: #10
    add(STATEMENT_END);
  }
}

/**
 * Adds nested loops to the generated code until no more declarations
 * are left. Then the predicate statement is added to the most inner
 * loop body.
 * @param list
 * @throws QRange.NotExecutableQuantifiedExpr
 */
//@ assignable decls;
protected void addLoops(/*@ non_null @*/ ListBuffer<JCVariableDecl> decls)
    throws QRange.NotExecutableQuantifiedExpr {
  //if(decls != null && !decls.isEmpty()){
  if(!decls.isEmpty()){
    JCVariableDecl d = decls.next(); // same as next / poll
    // Add the loop header
    addLoopHeader(d);
    // Add the next inner loop
    add(BLOCK_START); // {
    addLoops(decls); // <body>
    add(BLOCK_END); // }
```

```
    } else {
      addPredicate();
    }
  }

  /**
   * @return A list containing all JCVariableDecl objects of the expression
       tree
   */
  protected /*@ pure @*/ ListBuffer<JCVariableDecl> getDeclarations(){
    // Note: expr.decls is a ListBuffer which can be turned into a javac List
        w. toList()
    return expr.decls;
  }

  /**
   * Adds a string to the generated code.
   * @param s String that should be added to the code
   */
  //@ requires generated != null;
  //@ assignable s;
  //@ ensures generated.startsWith(\old(generated));
  protected void add(/*@ non_null @*/ String s){
    generated += s;
  }

  /**
   * Adds a for-loop head to the generated code.
   * @param type Type of the variable
   * @param var Variable name
   * @throws QRange.NotExecutableQuantifiedExpr if QRange cannot evaluate a
       proper range for the variable
   */
  //@ requires generated != null;
  //@ assignable decl;
  //@ ensures generated.startsWith(\old(generated));
  protected void addLoopHeader(/*@ non_null @*/ JCVariableDecl decl) throws
      QRange.NotExecutableQuantifiedExpr{

    // Generate code that generates an interval object during runtime
    // - decl.name.toString() should work(TM) though we had odd cases with
        decl.var being *null*
    QRange range = QRange.compute(expr.range, decl.name.toString());

    add(LOOP_START); // for(
    add(decl.toString()); // type var e.g. int i
    add(LOOP_SEPARATOR); // :
    add(range.translate()); // [i_0, ..., i_n]
    add(LOOP_END); // )
  }

  /**
   * Adds a predicate check to the code.
   */
```

```
  //@ requires generated != null;
  //@ ensures generated.startsWith(\old(generated));
  //@ ensures generated.endsWith(STATEMENT_END);
  protected void addPredicate(){
    // NOTE: #9
    add(ASSERT);
    add(SEPARATOR);
    add(expr.value.toString());
    add(STATEMENT_END);
  }

  /**
   * @return Code that evaluates this for all expression during RAC.
   */
  public /*@ pure @*/ String translate(){
    return generated;
  }

  public /*@ pure @*/ String toString(){
    return translate();
  }
}
```

C. QRange.java

```
package dk.itu.openjml.quantifiers;

import com.sun.tools.javac.tree.JCTree.JCBinary;
import com.sun.tools.javac.tree.JCTree.JCExpression;

/**
 * Inspired from the QSet class implemented for JML2
 *
 * This class represents a quantified range over integers.
 * It walks over an expression tree and translate atomic
 * boolean expressions into a set of values. As an atomic
 * boolean expression can only define one margin of a range,
 * the other margin will by default be Integer.MAX_VALUE or
 * respectively Integer.MIN_VALUE. Through operations on
 * sets these "infinite" margins will be reduced to a valid
 * set of ranges which can also have gaps.
 */
public abstract class QRange {

  /**
   * Thrown if a quantified expression can not be evaluated properly so
   * that it can be executed in RAC.
   */
  public static class NotExecutableQuantifiedExpr extends Exception {
    private static final long serialVersionUID = 1L;

    public NotExecutableQuantifiedExpr(String expr){
      super("Cannot evaluate quantified expression [" + expr + "]");
```

```
    }
  }

  // Conjunction and disjunction
  final static String CON = "&&";
  final static String DIS = "||";

  // Implications (NOTE: Currently not supported #12)
  final static String RIMP = "==>";
  final static String LIMP = "<==";
  final static String BIMP = "<==>";

  // Boolean operators on numbers
  final static String GT = ">";
  final static String GEQ = ">=";
  final static String LT = "<";
  final static String LEQ = "<=";
  final static String EQ = "==";
  final static String NEQ = "!=";

  final static String PPLUS = "++";
  final static String MMINUS = "--";

  // Branches
  protected /*@ spec_public @*/ QRange left;
  protected /*@ spec_public @*/ QRange right;

  /**
   * Returns an instance of QRange that represents a (set of) range(s)
   * @param e An expression defining the range
   * @param var A variable for which the range should be computed for
   * @return A new QRange for the given expression and variable
   * @throws NotExecutableQuantifiedExpr Thrown if the expression contains
       logical operations we are not willing or able to process.
   */
  public static QRange compute(JCExpression e, String var) throws
      NotExecutableQuantifiedExpr{

    // Ignore the expression if var is not defined anywhere in this
        expression
    if(!definesVar(e, var)){
      return new IgnoreQRange();
    }

    // We only want to process binary expressions
    if(e instanceof JCBinary){
      if(hasOperator((JCBinary)e, CON)) {
        // Conjunction is an intersection
        return new IntersectionQRange((JCBinary)e, var);

      } else if(hasOperator((JCBinary)e, DIS)) {
        // Disjunction is a union
        return new UnionQRange((JCBinary)e, var);

```

```
      } else if(isAtomic((JCBinary)e)){
        // Leaf node representing actual range definitions
        return new LeafQRange((JCBinary)e, var);
      }
      // NOTE: #13
    }
    throw new NotExecutableQuantifiedExpr(e.toString());
  }

  /**
   * Build a QRange object that holds QRanges through compute
   * @param e A JmlBinay expression which describes a range
   * @param var A variable name for which we want to find the range
   * @throws NotExecutableQuantifiedExpr If e is not an executable statement
   */
  //@ assignable left, right;
  //@ ensures \fresh(left);
  //@ ensures \fresh(right);
  public QRange(JCBinary e, String var) throws NotExecutableQuantifiedExpr{
    left = compute(e.lhs, var);
    right = compute(e.rhs, var);
  }

  /**
   * Empty default constructor
   */
  //@ ensures left == null;
  //@ ensures right == null;
  protected QRange(){
    left = null;
    right = null;
  }

  /**
   * Checks recursively if an expression defines a variable name,
   * i.e. if any subexpression consists of only the variable.
   * @param e The expression tree
   * @param var The variable name
   * @return True if e or any subexpression of e defines var, false otherwise
   */
  private static /*@ pure @*/ boolean definesVar(JCExpression e, String var){
    if(e instanceof JCBinary){
      return definesVar(((JCBinary)e).lhs, var) || definesVar(((JCBinary)e).
          rhs, var);
    }
    // NOTE: #14
    return e.toString().equals(var);
  }

  /**
   * Checks a JmlBinary expression for a given operator
   * @param e A JmlBinary expression
   * @param o Some operator
   * @return True if o is the operator in e
```

```
 */
public static /*@ pure @*/ boolean hasOperator(JCBinary e, String o){
  return getOperator(e).equals(o);
}

/**
 * Returns a string which is the operator of a JCBinary expression
 * @param e A JCBinary expression
 * @return The operator in e
 */
// NOTE: #21
//@ ensures fresh(\result);
public static String getOperator(JCBinary e){
  // NOTE: #15
  String op = e.toString();
  op = op.replace(e.lhs.toString(), "");
  op = op.replace(e.rhs.toString(), "");
  op = op.replace(" ", "");
  return op;
}

/**
 * Checks if a JmlBinary expression is an atomic boolean expression
 * @param e A JmlBinary expression
 * @return True if e is an atomic boolean expression, false otherwise
 */
public static /*@ pure @*/ boolean isAtomic(JCBinary e){
  return hasOperator(e, GT) ||
      hasOperator(e, EQ) ||
      hasOperator(e, NEQ) ||
      hasOperator(e, LT) ||
      hasOperator(e, GEQ) ||
      hasOperator(e, LEQ);
}

/**
 * Generates source code for this range.
 * Returns no operator if one of the child ranges is empty
 * and instead just returns the opposite child's code.
 *
 * @return Source code to build the range defined by this QRange
 */
//@ requires left != null;
//@ requires right != null;
public /*@ pure @*/ String translate(){
  if(left.ignore()){
    return right.translate();
  } else if(right.ignore()){
    return left.translate();
  }
  return getCode();
}

public /*@ pure @*/ String toString(){
```

```java
    return translate();
  }

  /**
   * @return The actual operation for this range to be performed in source
       code
   */
  abstract protected /*@ pure @*/ String getCode();

  /**
   * @return True if the fields left and right are null
   */
  public /*@ pure @*/ boolean isLeaf(){
    return left == null && right == null;
  }

  /**
   * @return True if this is empty, false otherwise
   */
  public /*@ pure @*/ boolean ignore(){
    return this instanceof IgnoreQRange;
  }
}

/**
 * Represents a union of two ranges
 */
class UnionQRange extends QRange {

  public UnionQRange(JCBinary e, String var) throws
      NotExecutableQuantifiedExpr{
    super(e, var);
  }

  /**
   * @returns The code for a union-operation on ranges
   */
  protected /*@ pure @*/ String getCode(){
    return IntervalSet.class.getName() + ".union(" + left.translate() + ", "
        + right.translate() + ")";
  }
}

/**
 * Represents an intersection of two ranges
 */
class IntersectionQRange extends QRange {

  public IntersectionQRange(JCBinary e, String var) throws
      NotExecutableQuantifiedExpr{
    super(e, var);
  }

  /**
```

```
   * Get code
   * - here its a real code call, imagine on the rac.
   * @returns The code for an intersection-operation on ranges
   */
  protected /*@ pure @*/ String getCode(){
    return IntervalSet.class.getName() + ".intersect(" + left.translate() + "
        , " + right.translate() + ")";
  }
}

/**
 * Represents a range defined through a boolean expression
 */
class LeafQRange extends QRange {

  String var;
  String low;
  String high;

  public LeafQRange(){
    low = "Integer.MIN_VALUE";
    high = "Integer.MAX_VALUE";
    left = null;
    right = null;
  }

  public LeafQRange(JCBinary e, String var) throws
      NotExecutableQuantifiedExpr{
    // Store the variable name
    this.var = var;

    low = "Integer.MIN_VALUE";
    high = "Integer.MAX_VALUE";

    String left = e.lhs.toString();
    String op = getOperator(e);
    String right = e.rhs.toString();

    evaluateExpression(left, op, right);
  }

  // #16
  /**
   * Evaluates an expression made from three strings, left,
   * op, right, after these rules:
   *
   * (Note: had to switch rules 1 and 3 and 2 and 4 to guarantee inclusive
       ranges.)
   *
   * e[l o r] =:
   *   if "++", "--" in l, r then Exception else
   *   if r = var && (var not in l) ==> e[l o^(-1) r] else
   *   match o with
   * | "<=" ==> high = r
```

```
 * | ">=" ==> low = r
 * | "<" ==> e[l "<=" (r - 1)]
 * | ">" ==> e[l ">=" (r + 1)]
 * | "!=" ==> low = (r + 1) && high = (r - 1)
 * | "==" ==> low = r && high = r
 * | _ ==> Exception
 *
 * @param left A value or a identifier
 * @param op A operator
 * @param right A value or an identifier
 * @throws NotExecutableQuantifiedExpr if none of the rules apply to this
 *   expression
 */
//@ assignable low, high;
private void evaluateExpression(String left, String op, String right)
    throws NotExecutableQuantifiedExpr{

  if(right.contains(PPLUS) || right.contains(MMINUS) ||
      left.contains(PPLUS) || left.contains(MMINUS)){
    throw new NotExecutableQuantifiedExpr("[" + left + " " + op + " " +
        right + "] contains a pre- or postfix operator.");
  }
  else if(right.equals(var) && !left.contains(var)){
    evaluateExpression(right, changeOrientation(op), left);

  } else if(op.equals(LEQ)){
    high = right;

  } else if(op.equals(GEQ)){
    low = right;

  } else if(op.equals(LT)){
    evaluateExpression(left, LEQ, right + " - 1");

  } else if(op.equals(GT)){
    evaluateExpression(left, GEQ, right + " + 1");

  } else if(op.equals(NEQ)){
    // This is correct for inclusive intervals!
    low = right + " + 1";
    high = right + " - 1";

  } else if(op.equals(EQ)){
    // right is the only defined number
    low = right;
    high = right;

  } else {
    throw new NotExecutableQuantifiedExpr(left + " " + op + " " + right);
  }
}

/**
 * Swaps orientation of a logical inequality operator
```

```
   * @param op The operator to switch
   * @return The switched operator
   */
  private /*@ pure @*/ String changeOrientation(String op){
    // NOTE: #24
    switch(op){
    case GEQ: return LEQ;
    case LEQ: return GEQ;
    case GT:  return LT;
    case LT:  return GT;
    }
    return op;
  }

  /**
   * @returns The code for a range expression limited by two variables
   */
  public /*@ pure @*/  String translate(){
    return IntervalSet.class.getName() + ".interval(" + low + ", " + high + "
        )";
  }

  /**
   * @returns Empty string
   */
  protected /*@ pure @*/ String getCode(){
    return "";
  }
}

class IgnoreQRange extends LeafQRange {

  /**
   * Creates a leaf that has no meaning for an expression
   */
  public IgnoreQRange(){
    super();
    low = "Integer.MIN_VALUE";
    high = "Integer.MAX_VALUE";
  }

  public /*@ pure @*/ String translate(){
    return IntervalSet.class.getName() + ".interval(" + low + ", " + high + "
        )";
  }
}
```

D. IntervalSet.java

```
package dk.itu.openjml.quantifiers;

import java.util.Iterator;

```

```java
/**
 * A set of intervals over integers. The Interval is a
 * subtype of IntervalSet and can be regarded as a singleton
 * of IntervalSet.
 */
public abstract class IntervalSet implements Iterator<Integer>, Iterable<Integer>{

  protected /*@ nullable @*/ IntervalSet left;
  protected /*@ nullable @*/ IntervalSet right;
  private boolean initialized;
  protected int low;
  protected int high;
  protected int current;

  /**
   * Cheap trick! But it works(TM) #18
   *
   * Normally we would return new SomeIterator<Integer>().
   *
   * @returns This, as it is also implements Iterator and for that
   * reason full fills the interface.
   */
  public Iterator<Integer> iterator() {
    return this;
  }

  /**
   * Performs union on two IntervalSets
   * @param l Left IntervalSet
   * @param r Right IntervalSet
   * @return An IntervalSet of type UnionIntervalSet
   */
  //@ ensures \fresh(\result);
  public static IntervalSet union(IntervalSet l, IntervalSet r){
    return new UnionIntervalSet(l, r);
  }

  /**
   * Performs intersection on two IntervalSets
   * @param l Left IntervalSet
   * @param r Right IntervalSet
   * @return An IntervalSet of type IntersectionIntervalSet
   */
  //@ ensures \fresh(\result);
  public static IntervalSet intersect(IntervalSet l, IntervalSet r){
    return new IntersectionIntervalSet(l, r);
  }

  /**
   * Only factory that can produce an IntervalSet without two IntervalSets
   * - both values are inclusive in the interval.
   *
   * @param low The lower boundary
```

```
   * @param high The upper boundary
   * @return An IntervalSet of type Interval describing an interval over integers
   */
  //@ ensures \fresh(\result);
  public static IntervalSet interval(int low, int high){
    return new Interval(low, high);
  }

  /**
   * Internal constructor
   * @param l Left side of the set
   * @param r Right side of the set
   */
  protected IntervalSet(/*@ nullable @*/ IntervalSet l, /*@ nullable @*/ IntervalSet r){
    left = l;
    right = r;
    current = Integer.MIN_VALUE;
    initialized = false;
  }

  //@ requires !initialized || current == high;
  //@ assignable low, high, current;
  //@ also
  //@ requires low <= current;
  //@ requires current < high;
  //@ ensures \result == true;
  //@ also
  //@ requires low > current || current >= high;
  //@ ensures \result == false;
  public boolean hasNext() {
    // Get the next valid range
    if(!initialized || current < high){
      initialized = true;
      getNextRange();
    }

    // This assures that false is returned if we reached
    // the end of the interval and the interval is right inclusive!
    if(low == current && high == current){
      return false;
    }

    return low <= current && current <= high;
  }

  public Integer next(){
    // Check sets up all the ranges, just in case
    if(hasNext()){
      return current++;
    }

    return current;
```

```
    }

    /**
     * This is implemented because Iterator requires it.
     */
    public void remove(){
          throw new UnsupportedOperationException();
    }

    /**
     * Find the next valid range for this IntervalSet
     */
    //@ ensures current == low;
    //@ assignable low, high, current;
    //@ ensures \old(current) <= current;
    protected void getNextRange(){
      low = getNextLow(current);
      high = getNextHigh(current);

      // Set current to the new low
      current = low;
    }

    /**
     * Checks if a given number is inside this IntervalSet (inclusive)
     * @param current The number to check against
     * @return True if current is inside, false otherwise
     */
    abstract protected boolean isInside(int current);

    //@ invariant initialized ==> isInside(current);

    /**
     * Returns the next valid low after current
     * @param current The current number active.
     * @return The new low. If there is no valid new low, return current again.
     */
    abstract protected /*@ pure @*/ int getNextLow(int current);

    /**
     * Returns the next valid high for current
     * @param current
     * @return
     */
    abstract protected /*@ pure @*/ int getNextHigh(int current);
  }

  /**
   * Represents a union of two IntervalSets
   */
  class UnionIntervalSet extends IntervalSet {

    protected UnionIntervalSet(IntervalSet l, IntervalSet r){
      super(l, r);
```

```
  }

  //@ ensures \result == left.isInside(current) || right.isInside(current);
  protected /*@ pure @*/ boolean isInside(int current){
    return left.isInside(current) || right.isInside(current);
  }

  /**
   * Union can answer two different lows!
   */
  protected /*@ pure @*/ int getNextLow(int current){
    int l = left.getNextLow(current);
    int r = right.getNextLow(current);

    // If both are higher than current
    if(current < l && current < r){
      // Return the smaller low
      return l < r ? l : r;

    // Else return the one that is higher than current
    } else if(current < l){
      return l;
    } else if(current < r){
      return r;
    }

    // If none of this holds, return current again
    return current;
  }

  /**
   * Union can answer two different highs!
   */
  protected /*@ pure @*/ int getNextHigh(int current) {
    int l = left.getNextHigh(current);
    int r = right.getNextHigh(current);

    // If both are higher than current
    if(current < l && current < r){
      // Return the smaller high
      return l < r ? l : r;

    // Else return the one that is higher than current
    } else if(current < l){
      return l;
    } else if(current < r){
      return r;
    }

    // If none of this holds, return current again
    return current;
  }
}
```

```java
/**
 * Represents an intersection of two IntervalSets
 */
class IntersectionIntervalSet extends IntervalSet {

  protected IntersectionIntervalSet(IntervalSet l, IntervalSet r){
    super(l, r);
  }

  //@ ensures \result == left.isInside(current) && right.isInside(current);
  public /*@ pure @*/ boolean isInside(int current){
    return left.isInside(current) && right.isInside(current);
  }

  /**
   * Intersection can only answer one low.
   */
  protected /*@ pure @*/ int getNextLow(int current) {
    int l = left.getNextLow(current);
    int r = right.getNextLow(current);

    // When current is lower than any of these, return
    // the higher low.
    if(current < l || current < r){
      return l < r ? r : l;
    }

    // Else just return current
    return current;
  }

  /**
   * Intersection can only answer one high.
   */
  protected /*@ pure @*/ int getNextHigh(int current) {
    int l = left.getNextHigh(current);
    int r = right.getNextHigh(current);

    // When current is lower than any of these, return
    // the lower high.
    if(current < l || current < r){
      return l < r ? l : r;
    }

    // Else just return current
    return current;
  }
}

/**
 * Represents a singleton of IntervalSet
 * It is left-inclusive and right-exclusive!
 */
class Interval extends IntervalSet {
```

```

  /**
   * Creates an actual left-inclusive right-exclusive interval
   * @param low Lower boundary
   * @param high Upper boundary
   */
  //@ assignable left, right, low, high, current;
  //@ ensures left == null && right == null;
  protected Interval(int low, int high){
    super(null, null);
    this.low = low;
    this.high = high;
    this.current = this.low;
  }

  //@ ensures low <= current && current <= high;
  protected /*@ pure @*/ boolean isInside(int current){
    return low <= current && current <= high;
  }

  //@ ensures \result == isInside(current);
  public /*@ pure @*/ boolean hasNext(){
    return isInside(current);
  }

  //@ ensures \result == this.low;
  protected int getNextLow(int current) {
    return low;
  }

  //@ ensures \result == this.high;
  protected int getNextHigh(int current) {
    return high;
  }
}
```

E. JUnit Tests

E.1. Test_ForAll.java

```
package dk.itu.openjml.quantifiers;

import java.io.File;
import java.util.ArrayList;
import java.util.List;

import org.jmlspecs.openjml.API;
import org.jmlspecs.openjml.JmlTree.*;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import com.sun.tools.javac.tree.JCTree.JCParens;
```

```

/**
 * This test class requires to be run with the launch configuration:
 * - "Test_ForAll OpenJMLExtended.launch"
 */
public class Test_ForAll {

  List<String> qExprsJml;
  List<JmlQuantifiedExpr> qExprsAst;
  API openJmlApi;

  final static String FORALL_CLASS_HEAD = "class JML$ITU$ForAll";
  final static String FORALL_CLASS_TOP = "{ public static void forAll() {";
  final static String FORALL_CLASS_BOTTOM = "}}";

  // Do not remove escape sequences!
  final static String TEST_CLASS_HEAD = "class Test";
  final static String TEST_CLASS_TOP = "{\n";
  final static String TEST_CLASS_BOTTOM = "\npublic static void test() {}}";

  public void addExpressions(List<String> s) {
    s.add("//@ requires (\\forall int i; 0 <= i && i < 10; i < 10);");
    s.add("//@ requires (\\forall int i; i >= 5 || i < 10; i < 10);");
    s.add("//@ requires (\\forall int i; i >= 5 || i < 10 && i < 300; i > 0);
        ");
    s.add("//@ requires (\\forall int i; i >= 5 || i < 10 && i < 300 && i !=
        500; i > 10 );");
    // # 28
    //s.add("//@ requires (\\forall int i, j; 0 <= i && i < 10 && j == i + 1;
         i == (j - 1));");
    s.add("//@ requires (\\forall int i, j, h; 0 <= i && i < 10 && 50 < j &&
        j <= 100; i == (j - 1));");
    s.add("//@ requires (\\forall int i; -100 < i && i < 0 || 0 < i && i <
        100; i != 0);");
    // #27
  }


  /**
   * @param t AST containing JML-annotated Java sources
   * @return Only the JmlQuantifiedExpr subtree
   */
  public static JmlQuantifiedExpr pullOutQuantifier(JmlCompilationUnit t){
    if(t.defs.head instanceof JmlClassDecl){
      JmlClassDecl a = (JmlClassDecl)t.defs.head;
      if(a.defs.head instanceof JmlMethodDecl){
        JmlMethodDecl b = (JmlMethodDecl)a.defs.head;
        if(b.cases.cases.head.clauses.head instanceof JmlMethodClauseExpr){
          JmlMethodClauseExpr c = (JmlMethodClauseExpr)b.cases.cases.head.
              clauses.head;
          if(c.expression instanceof JCParens){
            JCParens d = (JCParens)c.expression;
            if(d.expr instanceof JmlQuantifiedExpr){
              return (JmlQuantifiedExpr)d.expr;
```

```
            }
          }
        }
      }
    }
    return null;
  }


  @Before
  public void setUp() throws Exception {

    qExprsJml = new ArrayList<String>();
    qExprsAst = new ArrayList<JmlQuantifiedExpr>();
    addExpressions(qExprsJml);

    openJmlApi = new API();

    // Add all expressions to AST list
    int count = 1;
    for(String s: qExprsJml) {
      JmlCompilationUnit t = openJmlApi.parseString("test$" + count,
          TEST_CLASS_HEAD + count + TEST_CLASS_TOP + s + TEST_CLASS_BOTTOM);
      qExprsAst.add(pullOutQuantifier(t));
      count++;
    }

    openJmlApi.setOption("-noPurityCheck");
    openJmlApi.parseAndCheck(new File("src/dk/itu/openjml/quantifiers/
        IntervalSet.java"));
  }

  /**
   * Runs the ForAll class on a JmlQuantifiedExpr and typechecks the result.
   */
  @Test
  public void testForAll() {
    int count = 1;
    for(JmlQuantifiedExpr t: qExprsAst) {
      ForAll f = new ForAll(t);
      try{
        JmlCompilationUnit cForAll = openJmlApi.parseString("forAll$" + count
            , FORALL_CLASS_HEAD + count + FORALL_CLASS_TOP + f.translate() +
            FORALL_CLASS_BOTTOM);
        Assert.assertEquals(f.toString(), 0, openJmlApi.enterAndCheck(cForAll
            ));
        System.out.println(openJmlApi.prettyPrint(cForAll, false));
      } catch (Exception e){
        Assert.fail(t.toString() + ", " + f.toString() + ", " + e.toString())
            ;
      } finally {
        count++;
      }
    }
```

```
  }

}
```

E.2. Test_QRange.java

```
package dk.itu.openjml.quantifiers;

import org.jmlspecs.openjml.JmlTree.JmlQuantifiedExpr;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import dk.itu.openjml.quantifiers.QRange;

public class Test_QRange extends Test_ForAll {

  @Before
  public void setUp() throws Exception {
    super.setUp();
  }

  @Test
  public void testCompute() {
    for(JmlQuantifiedExpr t: qExprsAst) {
      System.out.println(t);
      String p = "";
      try{
        p = QRange.compute(t.range, "i").translate();
      } catch (Exception e){
        Assert.fail(e.toString());
      }
      System.out.println(p);
    }
  }

}
```

E.3. Test_IntervalSet.java

```
package dk.itu.openjml.quantifiers;

import static org.junit.Assert.*;

import java.util.Iterator;

import junit.framework.Assert;

import org.junit.Before;
import org.junit.Test;

/**
 *
 * Test class for the IntervalSet
```

```
 *
 * This test class requires to be run with the launch configuration:
 *  - "Test_IntervalSet OpenJMLExtended.launch"
 *
 * Keep the specific min /max values in <b>fresh </b> mind:
 * Min value: -2147483648
 * Max value: 2147483647
 *
 */
public class Test_IntervalSet {

  @Before
  public void setUp() throws Exception {
  }


  @Test
  public void testIntervalSetBasic() {
    IntervalSet i = IntervalSet.interval(0, 10);
    assertNotNull(i);
  }

  @Test
  public void testIntervalSetBasicForEach() {
    IntervalSet i = IntervalSet.interval(0, 10);
    int count = 0;
    for(int n: i){
      assertNotNull(n);
      count++;
    }
    assertEquals(11, count);
  }

  @Test
  public void testIntervalSetBasicIterator() {
    IntervalSet i = IntervalSet.interval(0, 10);
    int count = 0;
    Iterator<Integer> ite = i.iterator();
    while(ite.hasNext()){
      assertNotNull(ite.next());
      count++;
    }
    assertEquals(11, count);
  }


  @Test
  public void testUnionGap() {
    IntervalSet u = IntervalSet.union(IntervalSet.interval(11, 20),
        IntervalSet.interval(0, 9));
    try{
      for(int n: u){
        assertTrue("Failed with " + n, 0 <= n && n <= 9 || 11 <= n && n <=
            20);
```

```
      }
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testIntersection() {
    IntervalSet i = IntervalSet.intersect(IntervalSet.interval(0, 11),
        IntervalSet.interval(5, 15));
    try{
      for(int n: i){
        assertTrue("Failed with " + n, 0 <= n && n <= 11 && 5 <= n && n <=
            15);
      }
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testIntersectedUnion() {
    IntervalSet u = IntervalSet.union(IntervalSet.interval(20, 30),
        IntervalSet.interval(0, 10));
    IntervalSet iu = IntervalSet.intersect(u, IntervalSet.interval(5, 25));
    try{
      for(int n: iu){
        assertTrue("Failed with " + n, 0 <= n && n <= 10 || 20 <= n && n <=
            30 && 5 <= n && n <= 25);
      }
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testUnitedIntersection() {
    IntervalSet i = IntervalSet.intersect(IntervalSet.interval(0, 100),
        IntervalSet.interval(50, 150));
    IntervalSet ui = IntervalSet.union(i, IntervalSet.interval(40, 60));
    try{
      for(int n: ui){
        assertTrue("Failed with " + n, 0 <= n && n <= 100 && 50 <= n && n <=
            150 || 40 <= n && n <= 60);
      }
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testNotInside() {
    IntervalSet ni = IntervalSet.union(IntervalSet.interval(0, 100),
        IntervalSet.interval(51, 49));
```

```
    try{
      for(int n: ni){
        assertTrue("Failed with " + n, 0 <= n && n <= 100 && n != 50);
      }
    } catch (Exception e){
      Assert.fail();
    }
  }

  /**
   * (2147483645, 2147483647] =>
   * (2147483645, 2147483646)
   */
  @Test
  public void testUnionMaxValue() {
    IntervalSet u = IntervalSet.interval(Integer.MAX_VALUE-2, Integer.
        MAX_VALUE);
    try{
      int count = 0;
      for(int n: u){
        assertTrue("Failed with " + n, Integer.MAX_VALUE-2 <= n && n <=
            Integer.MAX_VALUE);
        count++;
      }
      assertEquals(3, count);
    } catch (Exception e){
      Assert.fail();
    }
  }

  /**
   * (-2147483648, -2147483646] =>
   * (-2147483648, -2147483647)
   */
  @Test
  public void testUnionMinValue() {
    IntervalSet u = IntervalSet.interval(Integer.MIN_VALUE, Integer.MIN_VALUE
        +2);
    try{
      int count = 0;
      for(int n: u){
        assertTrue("Failed with " + n, Integer.MIN_VALUE <= n && n <= Integer
            .MIN_VALUE+2);
        count++;
      }
      assertEquals(3, count);
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testSingleValue() {
    IntervalSet i = IntervalSet.interval(0, 0);
```

```
    try{
      int count = 0;
      for(int n: i){
        assertTrue(n == 0);
        count++;
      }
      assertEquals(1, count);
    } catch (Exception e){
      Assert.fail();
    }
  }


  // NOTE: #26
  @Test
  public void testUnionSingleton() {
    IntervalSet u = IntervalSet.union(IntervalSet.interval(0, 0), IntervalSet
        .interval(10, 10));
    try{
      int count = 0;
      for(int n: u){
        assertTrue(n == 0 || n == 10);
        count++;
      }
      assertEquals(2, count);
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testOverlappingIntersection() {
    IntervalSet i = IntervalSet.intersect(IntervalSet.interval(0, 20),
        IntervalSet.interval(0, 10));
    try{
      for(int n: i){
        assertTrue("Failed with " + n, 0 <= n && n <= 20 && n <= 10);
      }
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testOverlappingUnion() {
    IntervalSet u = IntervalSet.union(IntervalSet.interval(0, 20),
        IntervalSet.interval(0, 10));
    try{
      for(int n: u){
        assertTrue("Failed with " + n, 0 <= n && (n <= 20 || n <= 10));
      }
    } catch (Exception e){
      Assert.fail();
    }
```

```
  }
}
```

E.4. TestAll.java

```
package dk.itu.openjml.quantifiers;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses( {
  Test_ForAll.class,
  Test_IntervalSet.class,
  Test_QRange.class,
  Test_ForAllCompiledForRAC.class
} )

/**
 * This test case requires to be run with the launch configuration:
 * - TestAllOpenJMLExtended.launch
 */
public class TestAll {

}
```

E.5. Test_ForAllCompiledForRAC.java

```
package dk.itu.openjml.quantifiers;

import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertEquals;
import junit.framework.Assert;

import org.junit.Test;

/**
 * This test class *runs* the code compiled with the ForAll.java class.
 * - the code is slightly modified because of JUnit requires use of its
 * own asserts not the build in Java <b>assert something</b>.
 */
public class Test_ForAllCompiledForRAC {

  @Test
  public void testJML$ITU$ForAll1() {
    try{
      JML$ITU$ForAll1.forAll();
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testJML$ITU$ForAll2() {
```

```
    try{
      JML$ITU$ForAll2.forAll();
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testJML$ITU$ForAll3() {
    try{
      JML$ITU$ForAll3.forAll();
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testJML$ITU$ForAll4() {
    try{
      JML$ITU$ForAll4.forAll();
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testJML$ITU$ForAll5() {
    try{
      JML$ITU$ForAll5.forAll();
    } catch (AssertionError a){
      assertEquals("java.lang.AssertionError: ", a.toString());
    } catch (Exception e){
      Assert.fail();
    }
  }

  @Test
  public void testJML$ITU$ForAll6() {
    try{
      JML$ITU$ForAll6.forAll();
    } catch (Exception e){
      Assert.fail();
    }
  }

}


/**
 * s.add("//@ requires (\\forall int i; 0 <= i && i < 10; i < 10);");
 * P, predicate holds always true for this one.
 */
class JML$ITU$ForAll1 {

```

```
  JML$ITU$ForAll1() {
    super();
  }

  public static void forAll() {
    for (int i : dk.itu.openjml.quantifiers.IntervalSet.intersect(dk.itu.
        openjml.quantifiers.IntervalSet.interval(0, Integer.MAX_VALUE), dk.
        itu.openjml.quantifiers.IntervalSet.interval(Integer.MIN_VALUE, 10 -
        1))) {
      assert i < 10;
      // Repeat for JUNIT:
      assertTrue(i < 10);
    }
  }
}

/**
 * s.add("//@ requires (\\forall int i; i >= 5 || i < 10; i < 10);");
 * P, predicate holds always true for this one.
 */
class JML$ITU$ForAll2 {

  JML$ITU$ForAll2() {
    super();
  }

  public static void forAll() {
    for (int i : dk.itu.openjml.quantifiers.IntervalSet.union(dk.itu.openjml.
        quantifiers.IntervalSet.interval(5, Integer.MAX_VALUE), dk.itu.
        openjml.quantifiers.IntervalSet.interval(Integer.MIN_VALUE, 10 - 1)))
         {
      assert i < 10;
      // Repeat for JUNIT:
      assertTrue(i < 10);
    }
  }
}

/**
 * s.add("//@ requires (\\forall int i; i >= 5 || i < 10 && i < 300; i > 0)
     ;");
 * P, predicate holds always true for this one.
 */
class JML$ITU$ForAll3 {

  JML$ITU$ForAll3() {
    super();
  }

  public static void forAll() {
    for (int i : dk.itu.openjml.quantifiers.IntervalSet.union(dk.itu.openjml.
        quantifiers.IntervalSet.interval(5, Integer.MAX_VALUE), dk.itu.
        openjml.quantifiers.IntervalSet.intersect(dk.itu.openjml.quantifiers.
        IntervalSet.interval(Integer.MIN_VALUE, 10 - 1), dk.itu.openjml.
```

```
            quantifiers.IntervalSet.interval(Integer.MIN_VALUE, 300 - 1)))) {
        assert i > 0;
        // Repeat for JUNIT:
        assertTrue(i > 0);
      }
    }
}

/*
 * //@ requires (\\forall int i; i >= 5 || i < 10 && i < 300 && i != 500; i >
       10 )
 */
class JML$ITU$ForAll4 {

  JML$ITU$ForAll4() {
    super();
  }

  public static void forAll() {
    for (int i : dk.itu.openjml.quantifiers.IntervalSet.union(dk.itu.openjml.
        quantifiers.IntervalSet.interval(5, Integer.MAX_VALUE), dk.itu.
        openjml.quantifiers.IntervalSet.intersect(dk.itu.openjml.quantifiers.
        IntervalSet.intersect(dk.itu.openjml.quantifiers.IntervalSet.interval
        (Integer.MIN_VALUE, 10 - 1), dk.itu.openjml.quantifiers.IntervalSet.
        interval(Integer.MIN_VALUE, 300 - 1)), dk.itu.openjml.quantifiers.
        IntervalSet.interval(500 + 1, 500 - 1)))) {
      assert i > 10;
      // Repeat for JUNIT:
      assertTrue(i > 10);
    }
  }
}

/*
 * //@ requires (\\forall int i, j, h; 0 <= i && i < 10 && 50 < j && j <=
      100; i == (j - 1))
 * P, predicate does NOT hold always true for this one.
 */
class JML$ITU$ForAll5 {

    JML$ITU$ForAll5() {
      super();
    }

    public static void forAll() {
      for (int i : dk.itu.openjml.quantifiers.IntervalSet.intersect(dk.itu.
          openjml.quantifiers.IntervalSet.interval(0, Integer.MAX_VALUE), dk.
          itu.openjml.quantifiers.IntervalSet.interval(Integer.MIN_VALUE, 10
          - 1))) {
        for (int j : dk.itu.openjml.quantifiers.IntervalSet.intersect(dk.itu.
            openjml.quantifiers.IntervalSet.interval(50 + 1, Integer.
            MAX_VALUE), dk.itu.openjml.quantifiers.IntervalSet.interval(
            Integer.MIN_VALUE, 100))) {
```

```
            for (int h : dk.itu.openjml.quantifiers.IntervalSet.interval(
                Integer.MIN_VALUE, Integer.MAX_VALUE)) {
              assert i == (j - 1);
              // Repeat for JUNIT:
              assertTrue(i == (j - 1));
            }
          }
        }
      }
    }


/*
 * //@ requires (\\forall int i; -100 < i && i < 0 || 0 < i && i < 100; i !=
      0)
 * P, predicate holds always true for this one.
 */
class JML$ITU$ForAll6 {

    JML$ITU$ForAll6() {
      super();
    }

    public static void forAll() {
      for (int i : dk.itu.openjml.quantifiers.IntervalSet.union(dk.itu.openjml.
          quantifiers.IntervalSet.intersect(dk.itu.openjml.quantifiers.
          IntervalSet.interval(-100 + 1, Integer.MAX_VALUE), dk.itu.openjml.
          quantifiers.IntervalSet.interval(Integer.MIN_VALUE, 0 - 1)), dk.itu.
          openjml.quantifiers.IntervalSet.intersect(dk.itu.openjml.quantifiers.
          IntervalSet.interval(0 + 1, Integer.MAX_VALUE), dk.itu.openjml.
          quantifiers.IntervalSet.interval(Integer.MIN_VALUE, 100 - 1)))) {
        assert i != 0;
        // Repeat for JUNIT:
        assertTrue(i != 0);
      }
    }
}
```

F. Weblinks

- Google Code SVN
  - https://sasp-f2012-jml-and-more.googlecode.com/svn/OpenJMLExtended/tags/final_handin