Run-time Assertion Checker for JML in Prolog

Amritam Sarcar
Dept. Computer Science, University of Texas, El Paso
500 West University Avenue
El Paso, TX - 79968
amritamsarcar@yahoo.co.in

Abstract. In this paper, we describe a specialised logic for proving specifications in the Java Modeling Language (JML). JML is a behavorial interface specification language for Java. It allows assertions like invariants, constraints, pre- and post-conditions amongst other things, in a design-by-contract style. A special compiler-cum-translator is built which translates Java classes together with their JML annotations into logical constructs. Although object-oriented languages are nowadays the mainstream of application development, several research contexts suggest that a multi-paradigm approach is worth pursuing. In particular, a declarative, logic-based paradigm could fruitfully add functionalities related to automatic reasoning, adaptivity, and conciseness in expressing algorithms. In this paper we present J2PL, a framework for enhancing interoperability between Java and Prolog based on the SWIProlog open-source Prolog engine for Java. J2PL supports smooth language-interoperability by first introducing an API for modeling first-order logic terms based on JML semantics, promoting expressiveness and safety.

1 Introduction

The growing complexity in modern software systems comes in two main ways: interaction and intelligence. First, systems are typically thought of as sub-systems that interact so as to achieve a global system goal: object-oriented languages typically provide valuable tools to this end, flexibly allowing decomposing a system in terms of interacting sub-systems like objects, processes, components, agents. Second, systems must generally embed automatic and complex reasoning, adaptiveness, and reconfiguration capabilities in order to achieve their task in open and unpredictable environments: handling these aspects through mainstream object-oriented languages (such as Java) are however often difficult. Instead, declarative logic-based programming provides valuable constructs for building software components based on logic theories, where the concepts of reasoning, inference, and declarative specification of algorithms and behaviour are more naturally understood and implemented. Accordingly, integrating object-oriented and logic-based programming has been the subject of several researches and corresponding technologies. Such proposals either attempt at joining the two paradigms [9, 16], or simply provide an interface library for accessing a Prolog engine from a mainstream object-oriented language such as Java [12, 8, 14]. Both solutions have however drawbacks: in the first case, the conceptual integrity of the programming model is sacrificed since the new language is typically more complex, thus making application development an harder task; in the second case, there is no true language integration, and some "boilerplate code" has to be implemented each time to fix the paradigm mismatch. What would be needed, hence, is an extension that better trades off language integration and programming integrity. We aim at achieving this result in Java starting from an existing Prolog engine, and developing on top of it a framework based on Java generics and annotations which promotes seamless exploitation of Prolog programming in Java. As existing Prolog engine, we consider the SWIProlog open-source project. Developed on top of SWIProlog, we introduce J2PL, a framework for extending Java programming with Prolog-based abilities. In particular, J2PL is structured in two stacked layers, each one further reducing the semantic gap between Java and Prolog:

- Generic API layer: a new hierarchy of semantics is introduced that models Prolog terms, so that user code written with Java annotation can be deterministically translated from object-oriented to logic-oriented representation of data [13].
- Annotation layer: a true Prolog-based extension of Java programming is added, providing custom Java annotations to be used for embedding into Prolog theories, so as to specify Prolog code as a possible implementation of given Java methods.

Other than Java-Prolog integration, we believe this paper provides general hints on how annotations, along with the expressiveness of the JML semantic system, can successfully turn into smooth language extensions, making Prolog a flexible platform for customising the programming model according to the application needs.

2 SWIProlog Background

SWI-Prolog offers a comprehensive Free Software Prolog environment, licensed under the Lesser GNU Public License. Together with its graphics toolkit XPCE, its development started in 1987 and has been driven by the needs for real-world applications. These days SWI-Prolog is widely used in research and education as well as for commercial applications. As an example, assuming that we want to exploit the following Prolog theory for generating all permutations of a list:

```
any([X|Xs],X,Xs).
any([X|Xs],E,[X|Ys]):-any(Xs,E,Ys).
permutation([],[]).
permutation(Xs,[X|Ys]):-any(Xs,X,Zs),
                        permutation(Zs, Ys).
```

Predicate any/3 takes a list, any element of it, and the list after removing the element, while predicate permutation/2 takes a list and a permuted version of it—syntax [X|Xs] stands for a list with head X and tail Xs as usual. Though this is just an explanatory example, a Java programmer might enjoy how the permutation algorithm is easily resolved in Prolog, and accordingly be willing to use it in a Java application to compute permutations of Java collections. The following sections would help us to understand the actual translation.

3 Illustrating JML with an Example

JML is used to specify the behaviour of Java modules. It is presented as annotations embedded within the Java code, starting with a comment-like syntax so that they do not interfere with usual Java tools, but specialized JML tools may take care of them.

The example in figure 1 presents a simplified electronic purse specification that illustrates the possibilities of JML. This class contains a field named balance which represents the amount of money stored in the purse, and a static field named max which designates the maximal amount that the purse may contain.

This specification illustrates the main clauses of JML, such as the class invariant (invariant), specifying that the balance should always be greater or equal to zero, or history constraints (constraint) specifying that the maximal balance, max, should never be modified. Notice the presence of the \old(*x*) operator in the before-after predicates, which expresses that the expression *x* has to be considered at its before value.

Each method specification clause is described by a keyword indicating its kind (e.g. requires for preconditions, ensures for normal postcondition, signals for exceptional postcondition, etc.), followed by a first-order logic predicate or an explicit keyword (e.g. \nothing, \not_specified, etc.). The assignable clause in the method specifications is used to list the fields which may be modified by the execution of the method. The signals clause is used to describe the postcondition the method establishes when the considered method throws an exception of the given type. In our example, the exception NoCreditException is raised when the amount to withdraw is greater than the value of the balance.

```
class Purse {
//@ invariant balance >= 0;
short balance;
//@ constraint max == \old(max);
static short max = 32767;

/*@ behavior
  @ requires a > 0;
  @ assignable balance;
  @ ensures balance == \old(balance) - a;
  @ signals (NoCreditException e)
  @ balance == \old(balance) &&
  @ a > \old(balance);
  @*/
public void withdraw(short a)
throws NoCreditException {...}

/*@ normal_behavior
  @ requires b > 0 && b <= max;
  @ assignable balance;
  @ ensures getBalance() == b;
  @*/
public Purse(short b) {...}

/*@ normal_behavior
  @ assignable \nothing;
  @ ensures \result == balance;
  @*/
public /*@ pure @*/ short getBalance() {...}
}
```

Fig.1. The JML specification of the Purse example

3.1 Problem

The use of formal models makes it possible to check the coherence of the specification (*verification*) and also to check the conformance of the specification with the initial requirements (*validation*). Good tool support for these verification and validation processes is always appreciated by users of the modeling language. A key technique for validation is to use model checkers or Program verifiers for the model. This is a semi-automated process, which simulates the execution of the specification, allowing the author to check that his specification has the desired behaviour.

However, the existing implementation cannot be proved using either a Theorem Prover or a Model Checker. However, in the current approach Java classes (annotated with JML specifications) can be used to verify and validate the client code in runtime. This approach has its own short-comings. Figure 2 illustrates how Java+JML fails to statically prove verify or validate or both.

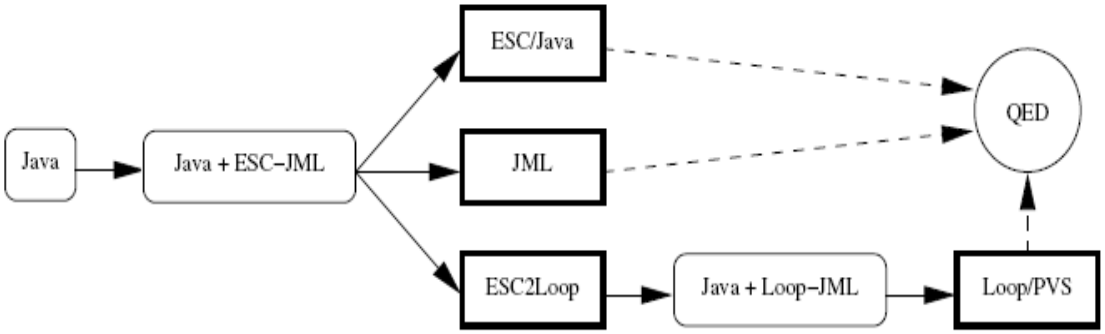

Fig.2. The current problem in runtime assertion checking approach.

4 J2PL Generic API Layer

In this section we describe the set of formal semantics introduced to fill the gap between object- and logic-based representation of data. Every data value in Prolog is a term—and even clauses and goals can be represented as such—with syntax:

t ::= a|p|V |f(t1, . . . , tn)|[t1, . . . , tn]

A term t is either an atom a (an unstructured literal), a primitive value p (an integer, a boolean, and so on), a logic variable V (a variable that can be bound to a value during computation, expressed as a literal starting with a capital letter), or a compound term where f is the functor name and each t_i is a term (n > 0). Terms can also be lists of the kind [t_1, . . . , t_n] (or [t_h|t_t] where t_h is the head and t_t is the tail), which Prolog implementations handle as special cases of compound terms.

The first problem J2PL has to face is to map these concepts in a very precise way into an object-oriented structure. However before we focus ourselves more into the translation scheme, we require to formalize JML semantics into Logical form. The general structure of every JML construct can be thought to be a sequence of predicates, which require to be fulfilled before we can process further.

4.1 Formalization of JML/Java statements

We take help of 4 helper predicates which would help us to solve the current problem. Since Java is an object-oriented program, it is enclosed within classes. We denote every class in Java to specification predicate with spec-name mapping to the class name. The declaration predicate is 4 tuple predicate where each parameter is responsible

to correctly identify the different methods within a Java program. Every statement in Java is deterministically transformed into Prolog terms with syntax:

specification(*spec-name).*
declaration(*spec-name, data-kind, data-name, data-type).*
operation(*spec-name, operation-name).*
predicate(*spec-name, pred-kind, pred-id, predicate)*

data-kind: static | variable | input(*op-name)* | *output(op-name)* | *local(op-name)*
data-type: atom | int| set(data-type) | pair(data- type, data-type)
pred-kind: static | invariant | initialization | pre(*op-name)* | *post(op-name)*

This formalization technique can be used in translating Java to logical format. Let us illustrate with an example. The following snippet of code in figure 3 describes a Java class Date with a simple getDay method. It shows how each construct JML/Java statements are translated to Prolog predicates as discussed above.

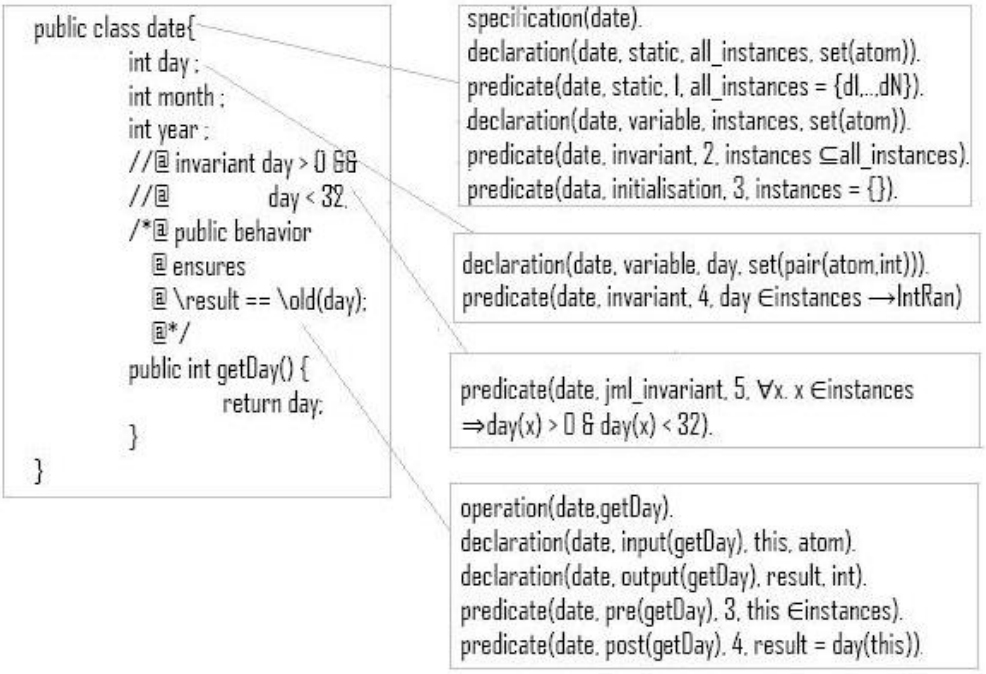

Fig.3. Translation of Java to Logical Format

4.2 Framework of Prolog Code

This section very briefly describes about how each prolog code look like. As stated earlier, one of our implementation goals is to check that every prolog code conforms to a particular format. Hence every such code is designed as per the below mentioned framework.

class_<class name> :-	Field Declaration,
	Call Main.
Constructor declaration: -	<<body of the constructor>>
Method_Name:-	Pre_Spec_<method Name>(),
	<<body of the method (contains sequence of statements)>>,
	Post_Spec_<method Name> ().
Pre_Spec_<method Name>:-	jml Clause.
Post_Spec_<method Name> :-	jml Clause.

As we see that the first predicate that is called from the console is class_<class name> . It first declares any field types and then calls main. Every method has a suffix _<method name> . To differentiate between Pre_spec of many methods, each of them has a suffix ie. _<method name>.

5 The architecture of the J2PL tool

As shown in figure 4, the J2PL tool accepts 2 languages with object oriented features namely, Java, and JML. It serves as a front end for a theorem prover which in this figure is PVS. The theorem prover is used to actually prove properties about the classes in the input languages, on the basis of logical theories generated by the tool itself.

5.1 Input Languages

The first input language is Java-one of the most popular object oriented programming languages. Our semantics for sequential Java ie java without threads, closely follows the java language specification(JLS).

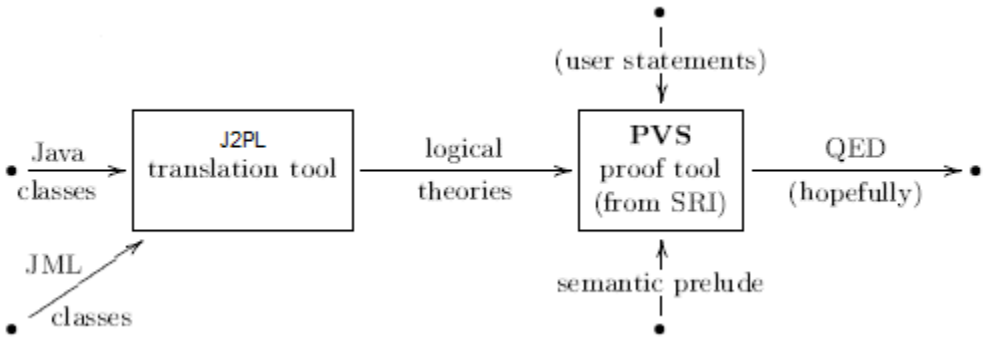

Fig4. Overview of the Tool

The second input language in JML short for Java Modeling Language. JML is a behavioral interface specification language ,tailored to java and primarily developed at Iowa State University. It is designed to be easy to use for programmers with limited experience in logic. Therefore it extends java such that a user can write (class)invariants and pre and post conditions for methods and constructors within the source code making use of java expressions (extended with various logical operators) to formulate the desired properties. All extensions of JML are enclosed between Java's comments, markers, and will therefore not influence the program's behavior. A typical specification for a method m looks as follows:

```
/*@ behaviour
   @ requires : <precondition>
   @  modifiable : <fields>
   @       ensures :  <postcondition>          // when terminating normally
   @       signals : (E) <postcondition>       //when terminating abruptly
   @                                           //because of exception E
   @*/
void m() {….}
```

5.2 J2PL Internals

In figure 5 the view on the Loop tool is enlarged. Here a view is considered where the tool accepts Java classes (and interfaces). The first three passes can be viewed as the first of a standard Java compiler. Standard techniques are used to build a lexer and parser, following the definition of the Java syntax in the JLS. During parsing, unknown types-class and interface types-are not resolved. These types are stored (tagged)strings in the abstract syntax tree and resolved in a later pass.

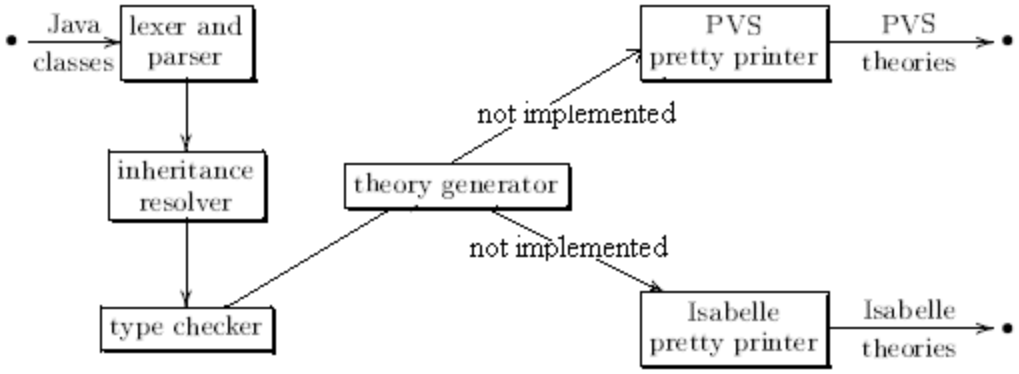

Fig5. The exploded view on the J2L tool

6 Integration with IDE

Applied formal methods has turned a corner over the past few years. Various groups in the semantics, specification, and verification communities now have sufficiently developed mathematical and tool infrastructures that automatic directly realize this theoretical framework in a modern software development environment (IDE) as an Open Source initiative. Specifically, our current work is focused on the integration of the verification technologies behind Eclipse, an open-source IDE initially developed by IBM. The key idea is to build a single environment whereby as much verification as possible happens automatically, and thus use of interactive verification only happens when necessary. (In those situations where developers wish to delay completion of the interactive proofs, it will be possible to insert run-time assertion checking code to perform compensatory verification during code execution.)

6.1 Eclipse IDE

Eclipse is a plug-in based application platform. An Eclipse application consists of the Eclipse plug-in loader (Platform Runtime component), certain common plug-ins (such as those in the Eclipse Platform package) along with application specific plug-ins. Well known bundles of Eclipse plug-ins include the Eclipse Software Development Kit (SDK) and the Eclipse Rich Client Platform (RCP). While Eclipse is written in Java, it does not have built-in support for Java. Like all other Eclipse features, Java support is provided by a collection of plug-ins—called the Eclipse Java Development Tooling (JDT)—offering, among other things, a standard Java compiler and debugger.

6.1 Architectural Overview

The main packages of interest in the JDT are the ui, core, and debug. As can be gathered from the names, the core (non-UI) compiler functionality is defined in the core package; UI elements and debugger infrastructure are provided by the components in the ui and debug packages, respectively. One of the rules of Eclipse development is that public APIs must be maintained *forever*. This API stability helps avoid breaking client code. The following convention was established by Eclipse developers: only classes or interfaces that are *not* in a package named internal can be considered part of the *public API*. Hence, for example, the classes for the JDT's internal AST are found in the org.eclipse.jdt.internal.compiler.ast package, where as the public version of the AST is (partly) reproduced under org.eclipse.jdt.core.dom. For JML4 we have generally made changes to internal components (to insert hooks) and then moved most of the JML specific code to org.jmlspecs.eclipse.jdt. At the top-most level, JML4 consists of:

• a customized version of the org.eclipse.jdt.core package (details will be given below) that is used as a drop-in replacement for the official Eclipse JDT core.

• JML specific classes contained in org.jmlspecs.eclipse.jdt including core classes (most of which are subclasses of the JDT Abstract Syntax Tree (AST) node hierarchy) and ui classes (e.g. for JML related preferences).
These packages are shown in bold in Figure 6.

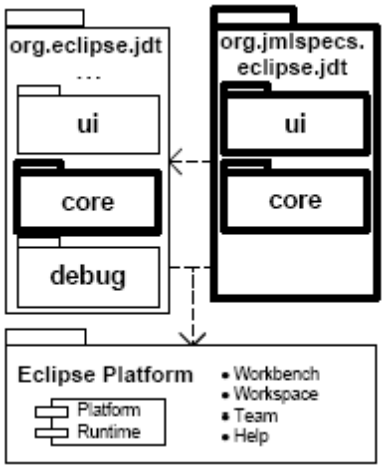

Fig6. High-level package view

The main steps of the compilation process performed by JML4 are illustrated in Figure 7. In the Eclipse JDT (and JML4), there are two types of parsing: in addition to a standard full parse, there is also a diet parse, which only gathers signature information and ignores method bodies. When a set of JML annotated Java files is to be compiled, all are diet parsed to create (diet) ASTs containing initial type information, and the resulting type bindings are stored in the lookup environment (not shown). Then each compilation unit (CU) is fully parsed. During the processing of each CU, types that are referenced but not yet in the lookup environment must have type bindings created for them. This is done by first searching for a binary (*.class) file or, if not found, a source (*.java) file. Bindings are created directly from a binary file, but a source file must be diet parsed and added to the list to be processed. In both cases the bindings are added to the lookup environment. If JML specifications for any CU or referenced type are contained in a separate external file (e.g. a *.jml file), then these specification files are diet parsed and the resulting information merged with the CU AST (or associated with the code generation are performed. Extended static checking is treated as a distinct phase between flow analysis and code generation.

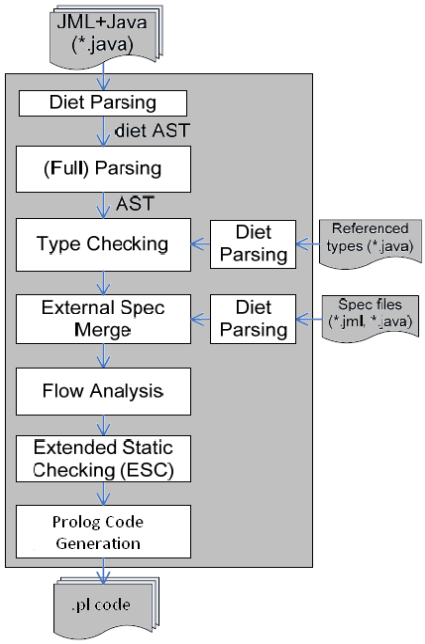

Fig7. The compilation phases of J2PL (integrated into Eclipse JDT)

7 Evaluation

The following snippet of code is used as a test suite against validating whether the framework outlined in this paper supports the feasibility about translating Java to Prolog code.

```
class TestI{
        // field declaration
        // author:amritam
        int amritam;
        int sarcar = 98;
        int fieldx;
        int fieldy;
        TestI(){}
        TestI(int p,int q){
        int Z;
        Z = p + q;
        }
        public static void main(String args[]){
                TestI p = new TestI();
                //p.functionI(5);
                p.functionI(2);
                pI.function2(I);
                TestI pI = new TestI(5,6);
        }
        public void function2(int X){
                int Y = X;
        }

        //@ requires Y>3;
        //@ assignable \nothing;
        //@ ensures Y>3;
        public int functionI(int Y){
                int X = Y + I;
                Y = Y + X * 4 + 3 * 3 + 4 / 3 - 2;
        return Y;
        }
}
```

This code after undergoing changes through the translation process from Java to prolog gives us the following output (prolog code) TestI.pl

```
class_TestI:- AMRITAM is 0 , SARCAR is 98 , FIELDX is 0 , FIELDY is 0 , main( AMRITAM , SARCAR , FIELDX , FIELDY ).
testI.
testI( P , Q , AMRITAM , SARCAR , FIELDX , FIELDY ):- Z is (P + Q) .
main( AMRITAM , SARCAR , FIELDX , FIELDY ):- prespec_main , testI , functionI( 2 , AMRITAM , SARCAR , FIELDX , FIELDY ) , testI( 5 , 6 , AMRITAM ,
SARCAR , FIELDX , FIELDY ) , function2( I , AMRITAM , SARCAR , FIELDX , FIELDY ) ,  postspec_main .
function2( X , AMRITAM , SARCAR , FIELDX , FIELDY ):- prespec_function2  ,  postspec_function2 .
functionI( Y , AMRITAM , SARCAR , FIELDX , FIELDY ):- prespec_functionI( Y , AMRITAM , SARCAR , FIELDX , FIELDY  ) , X is (Y + I) , YI is ((((Y + (X * 4)) +
(3 * 3)) + (4 / 3)) - 2) , postspec_functionI( YI , AMRITAM , SARCAR , FIELDX , FIELDY  ) .
prespec_main.
postspec_main.
prespec_function2.
postspec_function2.
prespec_functionI( Y , AMRITAM , SARCAR , FIELDX , FIELDY  ) :- ( (Y > 3) -> ! ; write_ln('PreCondition Failure ((Y > 3)')) .
postspec_functionI( Y , AMRITAM , SARCAR , FIELDX , FIELDY  ) :- ( (Y > 3) -> ! ; write_ln('PostCondition Failure ((Y > 3)')) .
```

When this prolog code is run on SWI Prolog, it gives the following output.

```
I ?- class_TesI.
PreCondition Failure ((Y > 3)
true.
```

The java code, on compiling using the native JML compiler, and then running it we get the following error:

Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLInternalPreconditionError: by method TestI.functionI regarding specifications at
File "TestI.java", line25, character 24 when
"Y" is 2
at TestI.main(TestI.java:396).

Hence we can conclude that the translated code is in conformance to the actual JML class.

8 Conclusion

In this paper we showed J2PL, a framework to promote seamless integration of Java into Prolog applications, exploiting SWIProlog technology [19]. We believe that the J2PL framework will encourage those programmers who are familiar with Java mainstream programming to easily incorporate declarative features into their programs. J2PL is structured in a compositional way, since Prolog integration is achieved through two layers; moreover, the high degree of customisation provided by the J2PL annotation layer makes it possible for the user to choose among many different interoperability paradigms.

9 Future Works

Some of the immediate future work where we can focus our attention are enlisted below.

- Judiciously extend the subset of RAC Implementation.
  - Include Level 0 and Level 1 JML annotations.
  - Include floating points.
  - Include concepts of Object Orientation.
    - Eg : Objects, polymorphism, inheritance.
- Execute generated RAC code from within Java.
- The .pl code obtained should be verified using theorem prover.

10 Related Works

There are several non-Java Prolog engines providing a Java interface such as SICStus Prolog [4], SWI-Prolog [5] and k-Prolog [2]; moreover, several Prolog engines have been written entirely in Java such as JLog [12] and Minerva [3]. Japlo [9] is an hybrid Java and Prolog language that has been built on top of the JLog engine, whose aim is to bring declarative features into the Java programming language; this is done at the language level: Japlo is an extension of the Java programming language providing some declarative features such as strongly-typed Prolog list declarations, theory definition, and so on. A significant difference between J2PL and Japlo concerns compatibility and deployment requirements: Japlo comes with its own compiler which translates a Japlo program into plain Java bytecode. The idea of using annotations for extending the Java language derived mostly from AspectJ /AspectWerkz [18], which are very popular aspect-oriented extensions of the Java programming language. AspectJ uses Java annotations for declaring aspects, pointcuts, and advices. Similarly, in [6] a framework is described that supports pluggable type systems in the Java programming language. This framework heavily relies upon Java annotations in order to define custom constraints on Java types. Other remarkable applications of Java annotations include simplifying code of J2EE applications, associating semantics to Java interfaces [15], building frameworks for detecting anomalies (e.g. deadlocks) in concurrent Java programs [10]. Moreover, it would be interesting to model the state of objects as a logic theory, and adding those inference-like abilities that could bridge the gap between object-oriented and agent-oriented programming.

Acknowledgements

This proposal is based upon the course work that I took under Dr. Ceberio. Many of the work outline in this paper has been done as a Research Assistant under Dr.Cheon.

References

[1] C.-B. Breunesse. On JML: Topics in Tool-assisted Verification of Java Programs. PhD thesis, Radboud University Nijmegen, 2005. in preparation.
[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens,K. M. Leino, and E. Poll. An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer, Feb. 2005.
[3] Z. Chen. Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison-Wesley Publishing Company, 2000.
[4] M. Clavel, F. Dur´an, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In Proceedings of the World Congress on Formal Methods in the Development of Computing Systems, 1999.
[5] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In Proceedings, CASSIS 2004, Marseille, France, 2004. Elsevier Science, Inc. In press.
[6] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003- 148, HP Labs, July 2003.
[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002), pages 234– 245, 2002.
[8] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts), volume 3114 of Lecture Notes in Computer Science, pages 175–188. Springer, 2004.
[9] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. Addison-Wesley Publishing Company, first edition, Aug. 1996.
[10] B. Jacobs. JavaCard program verification. In R. Boulton and P. Jackson, editors, Theorem Proving in Higher Order Logics TPHOL 2001, volume 2151 of Lecture Notes in Computer Science, pages 1–3. Springer–Verlag, 2001.
[11] B. Jacobs. Counting votes with formal methods. In C.Rattray, S. Majaraj, and C. Shankland, editors, Algebraic Methodology and Software Technology, volume 3116 of Lecture Notes in Computer Science, pages 241–257. Springer–Verlag, 2004.
[12] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML Reference Manual. Department of Computer Science, Iowa State University, 226 Atanasoff Hall, draft revision 1.94 edition, 2004.
[13] S. Ranise and D. Deharbe. Light-weight theorem proving for debugging and verifying units of code. In International Conference on Software Engineering and Formal Methods SEFM 2003, Canberra, Australia, Sept. 2003. IEEE Computer Society.
[15] RTI: Health, Social, and Economics Research, Research Triangle Park, NC. The economic impacts of inadequate infrastructure for software testing. Technical Report Planning Report 02-3, NIST, May 2002.
[16] T. Wahls, G.T. Leavens, and A.L. Baker. Executing Formal Specifications with Concurrent Constraint Programming. *Automated Software Engineering*, 7(4):315 – 343, December 2000.
[17]. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.