Run-time Assertion Checker for JML using 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 translator would be developed which translates Java classes together with their JML annotations into logical constructs.

1 Introduction

JML(for Java Modeling Language) is a specification language tailored to Java, primarily developed at Iowa State University. It allows assertions to be included in Java code, specifying for instance pre- and postconditions and invariants in the style of Eiffel and the design –by-contract apprach.

In this project we present J2PL, a framework for enhancing interoperability between Java and Prolog based on SwiProlog. J2PL would support smooth language-interoperability by first introducing an API for modeling first-order logic terms, promoting expressiveness and safety. On top of it, an annotation layer is then introduced that extends Java with the ability of implementing parts of the application code using Prolog.

2 Problem Statement

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.

Accordingly, J2PL trsanslates source-level code from Java to Prolog, which essentially is in a logical theory format that can serve as input for theorem provers, which can then be used to prove properties of the Java program, thus achieving a high level of reliability for this program. Another advantage of using formal specification language is that it becomes possible to provide tool support. Current work in this direction for JML focuses on the generation of run-time checks on preconditions for testing.

3 Objectives

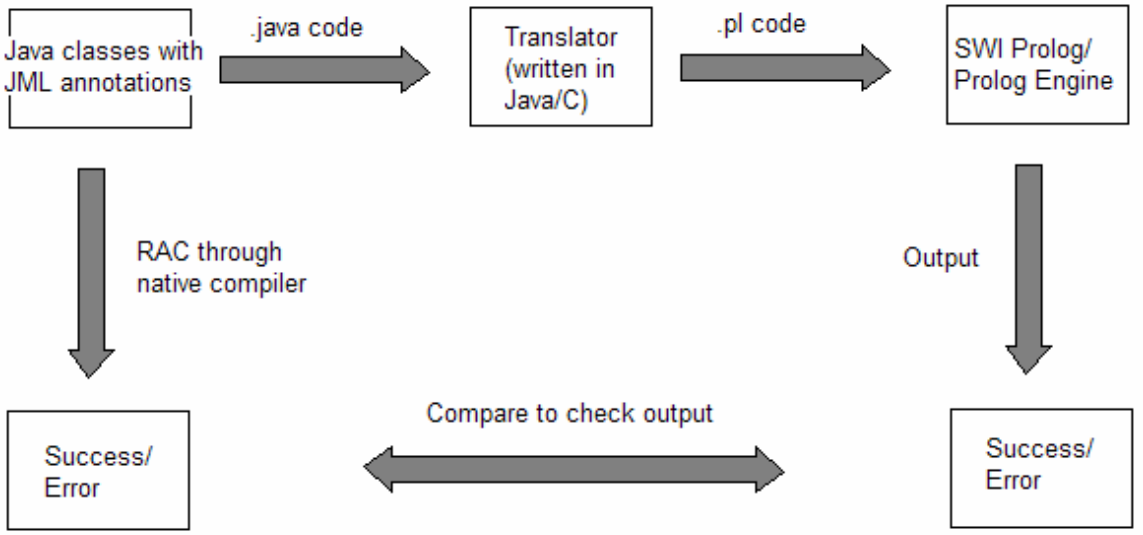
In this project, a formal semantics would be developed for essentially a part of the sequential Java. These formal semantics would present a logic for reasoning about(sequential) Java programs.

To simulate the actual behaviour of the formal semantics(described above), a translator is to be built, the J2PL tool, which translates a Java program into prolog code. The J2PL tool would be able to support only JML, so that it can verify JML-annotated Java source code.

4 Method

The semantical and logical approach to Java within the J2PL project is bottom-up: it starts from an(automatic) translation of Java programs ino what is ultimately a series of logical statements tailored to be send as input to Prolog Inference Engine. From this step onwards, several steps would be taken up the abstraction ladder.

1. At first, the results to be proved (about the Java program under consideration) would be formulated in the logical formula. Only relatively small programs can be handled like this, despite the usefulness of automatic rewriting.
2. In a further abstraction step, the results to be proved is to be checked using SwiProlog for which the translation from JML to Prolog is to be automated. A hand-coded translator(written either in Java/ C) is to be built.
3. In a final step – the output of the SwiProlog is to be compared with the result of RAC, so that the translation and formalisation is indeed correct and feasible.



5 Resources

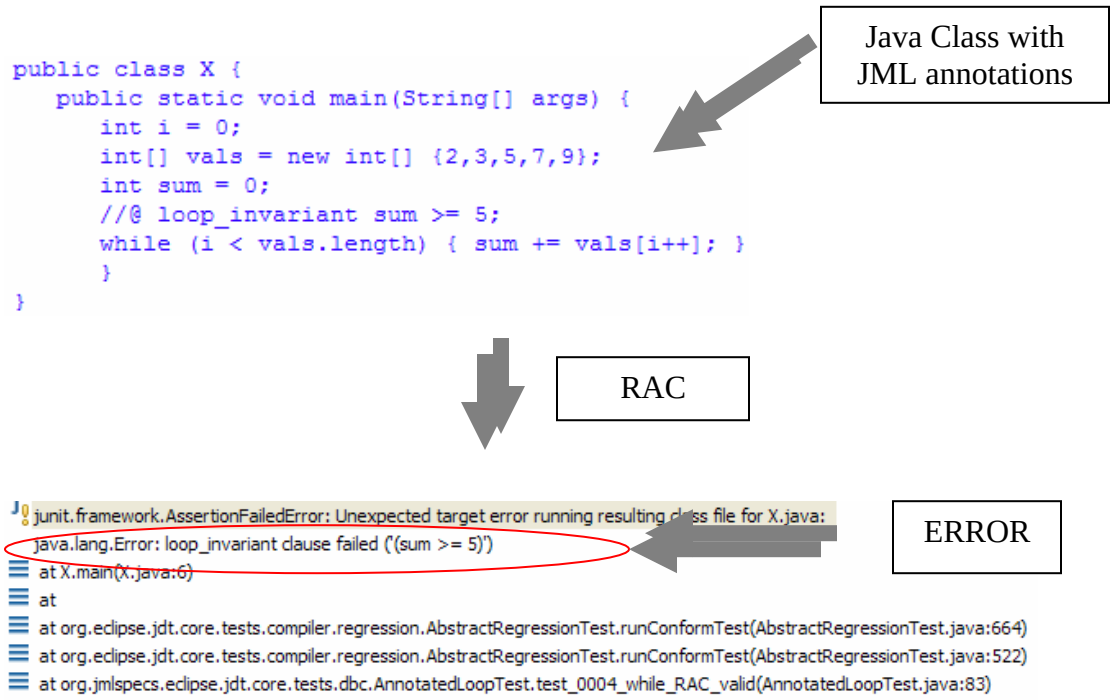
In order to accompish this project, sufficient amount of knowledge in formal notations, logic and compiler/ translator design is required. The tools which we are going to use for the development of our application are:

1. C/Java – to design the translator.
2. Eclipse – an open-source Java IDE where we are going to test run the java classes with jml annotations.
3. SwiProlog – the test bed where we are going to check the translated code works in accordance to our design specifications and our expected result.

6 Early Results

This section shows that such an application is feasible.

Here we hava a java class X. This class has some JML annotations. It is fed into a native compiler (java) and through Run-time assertion checking it produces an output.



We see that the RAC outputs an error showing that the invariant sum>=0 is not satisfied in the while loop.

The same java code is then translated(now using hand, later an automation of translated code would be generated) into prolog code and is then fed into SwiProlog.

```
main :-
        check(0,[2,3,5,7,9],0,5).
check(Sum,List,I,Length):-
        ( Sum >= 5 -> body(Sum,List,I,Length)
         ; write('error')
        ).
check(Sum,[],I,Length):-
        write('success').
body(Sum,[H|T],I,Length):-
        check(Sum+H,T,I+1,Length).
```

Translated Prolog code from java code

SWI Prolog

```
Warning: (c:/documents and settings/subhadeep/desktop/rac.pl:7):
        Singleton variables: [Sum, I, Length]
% c:/Documents and Settings/Subhadeep/Desktop/rac.pl compiled 0.00 sec, 1,168 bytes
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.45)
Copyright (c) 1990-2007 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- main.
error
More? ;

No
 2 ?-
```

ERROR

Hence we see that the system so designed adheres to our expected result.