JAG: JML Annotation Generation for Verifying Temporal Properties[*]

Alain Giorgetti and Julien Groslambert

Université of Franche-Comté - LIFC,
16 route de Gray - 25030 Besancon cedex France
{giorgetti, groslambert}@lifc.univ-fcomte.fr

Abstract. We present a tool for verifying temporal properties on Java/ JML classes by generating automatically JML annotations that ensure the verification of the temporal properties.

1 Introduction

We present JAG, a JML (Java Modeling Language) annotation generator for verifying temporal properties. JAG consists of a translator that transforms formulae expressed in a temporal language dedicated to Java - first introduced in [7] - into JML annotations that ensure the satisfaction of the temporal formulae.

JML (Java Modeling Language) [5] is a specification language for Java developped (by G.T. Leavens) at IOWA State University. JML annotations are introduced as Java comments using the key character '`@`'. The main annotations are `invariant`, `constraint`, `requires` and `ensures`. An `invariant` clause defines a property that must be satisfied in all visible states of the class, i.e., states before the invocation or after the termination of a method. An history `constraint` relates the value of the current state and the one of the pre-state denoted with the key word `\old`. Methods are described with preconditions (`requires`) and-postconditions (`ensures`). JML allows to declare specification variables (`ghost`) which can be assigned using a `set` clause.

The JML temporal logic extension [7] is inspired by Dwyers *Specification Pattern* [4]. It can deal with exceptional termination of methods and can express both *safety* and *liveness properties.* The semantics of the temporal formulae and the translation rules are given in details in [7] for *safety properties* and in [1] for *liveness properties.*

Take the example of a buffered transaction system (Fig. 1) encoded in Java, with a method `beginTransaction()`, which starts a new transaction, two methods `commitTransaction()` and `abortTransaction()` to respectively validate and abort (rollback) the current transaction and a `modify()` method which writes the modification in a buffer. We would like to verify on this Java class the following security properties describing the behavior of the class: (i) the buffer must be empty before beginning a new transaction and (ii) each started transaction must terminate.

[*] Research partially funded by the french ACI *GECCOO.*

```
package example.transacSystem;
public class TransactionSystem {

//@ ghost boolean trDepth = false;
//@ ghost int bufferFree;
//@ ghost int max = 100;
//@ invariant bufferFree <= max;
//@ constraint max == \old(max);
...
```

```
  /*@ private  normal_behavior
    @ requires trDepth == false;
    @ ensures trDepth == true;
    @*/
  public void beginTransaction() {
    //@ set trDepth = true;
    ...
  };
}
```

Fig. 1. A part of a JML specification: a Buffered Transaction System

The first property is a *safety* property ("something wrong must not happen"). The second one is a *liveness* property ("under certain conditions, something good must inevitably happen").

These properties can be encoded as restrictions on infinite Java execution sequences. However, it is not easy to translate them directly to JML annotations. Therefore, we propose to designers a compact temporal logic language to express such properties, and an automatic translation into standard JML annotations that are directly inserted into the Java code under verification.

The properties (i) and (ii) can be easily expressed in the temporal logic language of [7] by the following (`bufferFree` is the variable counting the free space of the buffer):

```
(i)  after commitTransaction() normal, abortTransaction() normal
     \always {bufferFree == max}
     \unless beginTransaction() called;
(ii) after beginTransaction() called \always true \until
     abortTransaction() called, commitTransaction() called
     under_invariant {true} variant {bufferFree};
```

The first formula means that `after` a transaction is finished - when `commitTransaction()` or `abortTransaction()` terminates `normally` – and unless (`\unless`) a new transaction starts (`beginTransaction() called`), the buffer must always (`\always`) be empty. The second formula means that `after` the start of a transaction, the transaction must inevitably be (`until`) finished by a commit or a rollback. The second formula is completed with a `variant` clause which is a Java expression returning a natural number. This variant must decrease each time a method is called until the method `commitTransaction()` or `abortTransaction()` is invoked. Notice also that there is a `under_invariant` keyword, here set to `true`, that allows to define a local invariant, that permits to express an extra hypothesis for the liveness proof.

2 Description of the Tool

The JAG tool parses a Java file - possibly already JML annotated - with the Iowa State University JML tools parser and takes as other input a file containing temporal formulae (Fig. 2).

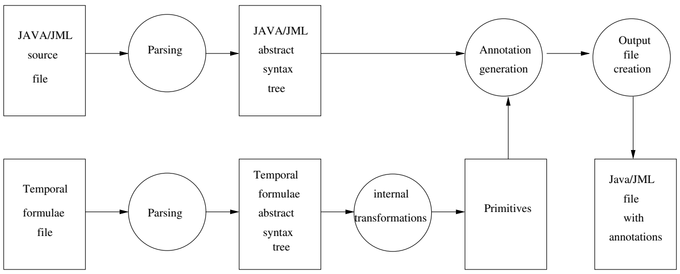

Fig. 2. Internal Structure of the Tool

Translating Temporal Formulae into Intermediate Primitives. The tool reduces each temporal property into one or more intermediate primitives that are semantically equivalent [7, 1]. The Inv primitive represents the safety part of a property. The Loop primitive represents the liveness part of a property. The Witness primitive represents special past marker on the class (for example to know if a method has already been called during the execution).

Translating Intermediate Primitives into Standard JML Annotations. Each Inv primitive is translated into a JML `invariant`. Each Loop is translated into a set of `invariant`s and history `constraint`s that imply the decrease of the variant and the deadlockfreeness of the system. Each Witness is translated as a JML `ghost` variable.

The tool generates an output file including the original file enriched with the generated JML annotations. This file can be used with other JML tools [2] to validate or prove the temporal formulae.

Trace Preservation. The tool is able to keep the trace of the generated annotations, *i.e.* it is possible, given a generated annotation, to find the original intermediate primitive and the original temporal property.

3 Experimental Results

The tool has been used on several examples. Table 1 summarizes the results obtained with the JACK [3] tool as back-end theorem prover.

Table 1. Results

Example Name	Number of temporal properties to verify	Number of line annotation generated	Number of PO (automatically proved)
TransactionSystem	2	18	92 (91)
AtmTransaction	2	21	171 (171)

4 Conclusion

The JAG tool generates JML annotations that imply the satisfaction of temporal properties (both liveness and safety) of the language defined in [7]. The particularity of this work is that the annotations are standard and can be used with all the tools taking JML files as an input. We first plan a better integration of our tool into some back-end tools and second to extend our work to other specification input, like PLTL formulae. JAG can be downloaded from the following page: `http://lifc.univ-fcomte.fr/~groslambert/JAG`.

References

1. F. Bellegarde, J. Groslambert, M. Huisman, J. Julliand, and O. Kouchnarenko. Verification of liveness properties with JML. Technical Report RR-5331, INRIA, 2004.
2. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In Th. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.
3. L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In *Formal Methods (FME'03)*, number 2805 in LNCS, pages 422–439. Springer, 2003.
4. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *International Conf. on Software Engineering*, pages 411–420. IEEE Computer Society Press/ACM Press, 1999.
5. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. Technical report, Iowa State University, Dept. of Computer Science, 1998.
6. G. Nelson. *Techniques for Program Verification.* PhD thesis, Stanford University, 1980.
7. K. Trentelman and M. Huisman. Extending JML Specifications with Temporal Logic. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology (AMAST'02)*, number 2422 in LNCS, pages 334–348. Springer, 2002.