

jmle: A Tool for Executing JML Specifications via Constraint Programming

Ben Krause and Tim Wahls

Department of Mathematics and Computer Science
Dickinson College
P.O. Box 1773
Carlisle, PA 17013, USA
{krauseb, wahlst}@dickinson.edu

Abstract. Formal specifications are more useful and easier to develop if they are executable. In this work, we describe a system for executing specifications written in the Java Modeling Language (JML) by translating them to constraint programs, which are then executed via the Java Constraint Kit (JCK). Our system can execute specifications written at a high level of abstraction, and the generated constraint programs are Java implementations of the translated specifications. Hence, they can be called directly from ordinary Java code.

1 Introduction

The ability to execute a formal specification while it is being developed greatly eases the development process. Running a specification allows the developer to validate that the meaning of the specification is what was intended, providing strong intuition that the specification is correct and complete. As formal specifications are intrinsically abstract, this ability to get “hands on” experience with a specification makes developing and understanding the specification much easier. Additionally, executable formal specifications can serve as prototypes of the final implementation, and test oracles for back-to-back testing of that implementation. These benefits are accessible to nontechnical users, as they do not require reading mathematical notation, understanding proofs of system properties, and so on. Given these benefits, it is not surprising that a large number of techniques for implementing specifications have been developed [1–5].

In this work, we describe the `jmle` tool, which is used to execute specifications written in the Java Modeling Language (JML) [6]. JML is a behavioral interface specification language for Java - class definitions and method prototypes are written using Java syntax, and a JML class specification can only be correctly implemented by a Java class definition. JML is also a model-based specification language - the language provides a rich set of mathematical types (sets, sequences, bags, functions, relations, . . .), which are implemented as Java classes. Each method is specified with first order pre- and postconditions written over these types, as well as the built-in types of Java. An example of a JML specification is presented in Section 3.

`jmle` executes JML specifications by translating them to constraint programs. It is important that executability does not compromise the abstraction and freedom from implementation bias that makes JML specifications useful, and using constraint programming techniques allows specifications written at a relatively high level of abstraction to be executed. The resulting constraint programs are executed using the Java Constraint Kit (JCK) [7], a system for creating Java implementations of constraint solvers. All parts of our system are implemented in Java, so users do not need to install any additional programming environments, and our system is completely compatible with existing tools for JML (as described in [6]). The programs that we generate are Java implementations of the corresponding specifications, and so can be called directly from Java code.

2 Implementation of `jmle`

We have adapted `jmle` (the JML tool that generates runtime assertion checking code) [8] to automatically compile JML specifications to JCK programs as follows:

- a JML class specification is compiled to a Java class
- the model (specification only) fields become actual fields of this class
- each JML method specification is compiled to a method implementation.

The body of each translated method creates goal constraints corresponding to the original JML specification. When the method is called, these constraints become the initial constraint store, which is then simplified using special-purpose JCK solvers. The solvers can execute a large subset of JML, including `\old` expressions (which allow pre-state values to be used in postconditions), universally and existentially quantified assertions where the domain of the quantified variable can be determined and is finite, and nondeterministic specifications (which are executed via backtracking). The system will report an error at compile time for specifications that use features outside of this subset. Additionally, `jmle` will fail to execute specifications that use only these features if they do not provide sufficient information to allow the system to construct post-state values, or to simplify all of the constraints in the store. In these situations, `jmle` will throw an exception to indicate that it could not execute the specification.

We considered each of the Java primitive types and the JML classes implementing the `JMLCollection` interface (that is, the classes representing mathematical sets, sequences, bags, functions and relations) as constraint domains, and defined constraints corresponding to each operation on each of these types. We then implemented solvers for each of these domains using JCK rewriting rules. Targeting our solvers specifically for Java and JML types allowed us to compensate somewhat for the performance disadvantages of a Java implementation (as compared to the solvers found in full constraint programming languages).

3 Example

The following JML specification of class `IntList` demonstrates the kind of implicit and abstract specifications that `jmle` can execute. Instances of class `IntList` contain a list of integers, modeled as a `JMLObjectSequence` holding `java.lang.Integer` objects. `JMLObjectSequence` is a `JMLCollection` class that implements sequences in which elements are compared using `==` (rather than the `equals` method). The constructor and other natural methods of the class have been omitted in the interest of space.

```
//@ model import org.jmlspecs.models.JMLObjectSequence;

public class IntList {
    //@ public model JMLObjectSequence theList;

    /*@ assignable theList;
       ensures theList.int_size() == \old(theList.int_size()) &&
              (\forall Integer i; \old(theList.has(i));
               theList.count(i) == \old(theList.count(i))) &&
              (\forall int j; 0 <= j && j < \old(theList.int_size()) - 1;
               ((Integer) theList.itemAt(j)).intValue() <=
                ((Integer) theList.itemAt(j + 1)).intValue()); */
    public void sort();
}
```

The postcondition for the `sort` method asserts that the pre- and post-state values of the model field `theList` are of the same size, and then uses a universal quantifier to state that the post-state value of `theList` contains the same number of occurrences of each element as occurs in the pre-state value. Together, these assertions ensure that the post-state value is a permutation of the pre-state value. The final universally quantified assertion forces the post-state value to be sorted. When this specification is executed, the `count` constraints on the post-state value are used to search partial permutations of the pre-state value (backtracking as soon as an unsorted prefix is discovered) until a sorted permutation is found. As the running time is exponential in the length of the sequence, only small inputs (up to about 5 elements) can be used for validating this specification.

`jmle` first compiles this specification to a JCK program, and then to ordinary Java bytecode that uses JCK library code. Hence, the compiled specification appears to client code exactly as any Java implementation of the JML specification would (except that it is much larger and slower than a hand-coded implementation). The specification can then be executed using ordinary Java “driver” code that creates an instance of the class and calls its methods, by writing JUnit tests for the class specification, or in any other manner that a hand-coded implementation could be used. This client code can then be re-used without modification when testing other implementations of the specification.

4 Conclusion

jmle is related to several other systems that translate specifications to constraint programs (such as [1, 5]), and particularly the jml-tt tool [3], which animates JML specifications. One practical difference is that these other systems do not translate specifications to implementations that can be called directly from client code. Rather, specifications are animated using an external interface. Additionally, jml-tt does not provide constraint support for executing specifications that use the `JMLCollection` classes.

Although jmle can execute a large subset of JML, much remains to be done. Java and JML constructs that are currently not supported include exceptional behavior specifications, signals clauses, history constraints, inheritance of specifications from interfaces, and the features added in Java 1.5. Perhaps the most critical area for future work is using jmle to execute specifications that are being developed in industrial applications, in order to investigate the usefulness of the system in practice. As such, we encourage anyone who is interested in obtaining and evaluating jmle to contact the second author via email.

References

1. Grieskamp, W.: A computation model for Z based on concurrent constraint resolution. In Bowen, J.P., Dunne, S., Galloway, A., King, S., eds.: ZB 2000: Formal Specification and Development in Z and B, First International Conference of Z and B Users. Volume 1878 of Lecture Notes in Computer Science., York, UK, Springer-Verlag (2000) 414 – 432
2. Wahls, T., Leavens, G.T., Baker, A.L.: Executing formal specifications with concurrent constraint programming. *Automated Software Engineering* **7**(4) (2000) 315 – 343
3. Bouquet, F., Dadeau, F., Legeard, B., Utting, M.: Symbolic animation of JML specifications. In: Proceedings of the International Conference on Formal Methods 2005 (FM'05). Volume 3582 of Lecture Notes in Computer Science., Springer-Verlag (2005) 75 – 90
4. Wahls, T.: Compiling formal specifications to Oz programs. In van Roy, P., ed.: MOZ 2004, The Second International Mozart/Oz Conference. Volume 3389 of Lecture Notes in Computer Science., Springer-Verlag (2005) 66 – 77
5. Leuschel, M., Butler, M.: ProB: A model checker for B. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME 2003: Formal Methods. LNCS 2805, Springer-Verlag (2003) 855–874
6. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)* **7**(3) (2005) 212–232
7. Abdennadher, S., Krämer, E., Saft, M., Schmauss, M.: JACK: A Java constraint kit. In Hanus, M., ed.: *Electronic Notes in Theoretical Computer Science*. Volume 64., Elsevier (2002)
8. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java Modeling Language (JML). In Arabnia, H.R., Mun, Y., eds.: Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24–27, 2002, CSREA Press (2002) 322–328