JML-Testing-Tools: A Symbolic Animator for JML Specifications Using CLP

Fabrice Bouquet, Frédéric Dadeau, Bruno Legeard, and Mark Utting

Laboratoire d'Informatique (LIFC),
Université de Franche-Comté, CNRS - INRIA,
16, route de Gray - 25030 Besançon cedex, France
{bouquet, dadeau, legeard, utting}@lifc.univ-fcomte.fr

Abstract. This paper describes a tool for symbolically animating JML specifications using Constraint Logic Programming. A customized solver handles constraints that represent the value of instance fields. We have extended a model-based approach to be able to handle object-oriented specifications. Our tool is also able to check properties during the simulation and exhibit counter-examples for false properties. Therefore, it can be used both for semi-automated verification and for validation purposes.

Keywords: Java Modeling Language, model-based, object-oriented, symbolic animation.

1 Motivations

Building formal models of systems is a valuable technique for improving the design of software, and analyzing safety and functionality, particularly when there is good tool support for the formal method. A variety of different modeling languages are used for building the formal models. The Java Modeling Language (JML) [LBR98, LBR99], is an object-oriented modeling language based on Java and designed to be used as well by developers as by modeling engineers.

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 *animation* of 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 behavior.

This paper describes a tool, called JML-Testing-Tools, which is able to symbolically execute a JML specification. It also allows users to specify constrained values as input for the method parameters, which is more general than entering specific values. We use a novel constraint solver to handle the constrained values of the resulting state variables. Moreover, our tool is able to check properties on-the-fly and to display counter examples for properties that fail. Thus, this tool may also be used for verification purposes. This technology has already

N. Halbwachs and L. Zuck (Eds.): TACAS 2005, LNCS 3440, pp. 551–556, 2005.

been applied to the animation of B [Abr96], Z [Spi92] or Statechart [Har87] specifications within the BZ-Testing-Tools environment [ABC+02].

2 Illustrating JML with an Example

JML is used to specify the behavior 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() {...}
}
```

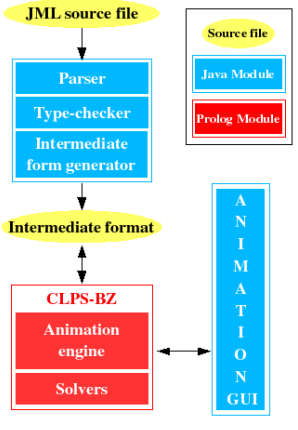
Fig. 1. The JML specification of the Purse example

JML also introduces new kind of method declaration modifiers, including the notion of *purity*, meaning that a method specified as `pure` does not change the value of any field of the considered class. In our example, method `getBalance()` is described to observe the value of the field `balance`. Method specifications may contain method calls, if and only if these methods are described as *pure*, in order to avoid side-effects.

3 Description of JML-Testing-Tools

JML-Testing-Tools – JML-TT – is a recently developed JML specification animator. It relies on a model-based approach, meaning that we only consider the method specifications to simulate the activation of the behaviors of the system, and we do not execute the Java code itself.

This is an extension of the BZ-Testing-Tools technology, a framework for animation and automatic test generation from B, Z or Statechart specifications, extended for handling object concepts. At the present time, only the animation part has been studied and implemented.



JML-TT takes as an input a JML annotated file of a Java class. The tool parses, type-checks and gathers all the referenced and needed classes which are then translated it to an intermediate format file, from which animation is realized. The animation relies on an original constraint solver named CLPS-BZ which handles constraints on the values of state variables, and method input parameters. Indeed, JML-TT makes it possible to constrain the value of an input to execute a method on an instance. Moreover, JML-TT is able to assign to the state variables a value satisfying the constraints store, by valuation of a constrained environment.

4 Animating a JML Specification

JML-Testing-Tools uses the JML annotations describing the specification of the Java module, i.e., class or interface, to symbolically execute it.

The CLPS-BZ animation engine manages an execution environment, which represents the classes, instances, fields and their corresponding values. Animating consists in identifying the predicates representing the different behaviors of a considered method, and interpreting them so that a new execution state is reached. Therefore, the methods are described using before-after predicates whose semantics is close to JML.

During the animation, the user is free to choose which objects he wants to create and which methods he wants to invoke on the created objects. The user

is also asked to input the value of method parameters, which may also be left constrained. This latter creates a constrained variable to represent the value of the parameter, depending on its type and the constraints described in the JML method annotations. New constrained variable may appear to represent instance fields values, if they are related to the constrained parameter. A labeling can be performed at any time to get all the possible values for all the newly introduced constrained variables.

The invocation of a method may create choice-points identifying behaviors in the specification. For example, the following JML-annotated method:

```
/*@ behavior
  @    requires P;
  @    assignable A;
  @    ensures Q;
  @    signals (Exception) S;
  @*/
TypeReturn methodName(TypeParam1 param1, ...) { ... }
```

will induce two behaviors: `P` $\wedge$ `Q` and `P` $\wedge$ `S`, describing the case when the method terminates normally and establishes the normal postcondition `Q`, and the case when the method terminates abnormally by throwing the specified exception and establishes the exceptional postcondition `S`. JML-TT makes it possible to execute each one of them by using a simple backtracking technique.

Each step of the animation is expressed in Java syntax to produce a trace of the symbolic execution performed. This Java instruction sequence may then be exported to a Java test case file and compiled to perform runtime assertion checking as described in [CL02].

Finally, properties can be checked within a specific execution state to ensure the conformance of the dynamic part of the specification –the methods– with the static properties of the system –the invariant and the history constraints. Properties are checked using the principle of refutation, which makes it possible to check either the validity or the satisfiability of the properties and to exhibit a reachable counter-example, when the property is checked to false.

5 Features of JML-Testing-Tools

JML-Testing-Tools has the following features:

- Animation of a JML specification in an environment also displaying all the referenced classes;
- Execution of the methods by activating their behaviors with precondition checking;
- Possibility to leave input values of method parameters unspecified, to create constrained states;
- Valuation of the constrained state to assign all their possible values to the constrained variables, with the possibility to take into account the invariant and/or the history constraints;

- Properties checking (invariant, history constraints) within an execution state, and exhibition of counter-examples for unchecked properties;
- Exportation of the user-defined execution sequence to a Java test case file, that can be checked by a runtime assertion checker;
- Good coverage of JML specifications clauses: class invariant, history constraints, preconditions (requires), postcondition (ensures, signals), delaying (when), divergence (diverges);
- Undo and redo features;
- Possibility to save and open animations.

All these features are realized by the user through a user-friendly Graphical User Interface described hereafter.

6 Description of the GUI

The Graphical User Interface displayed by the tool is presented in figure 2.

The left area (1) displays the state informations: the instances that have been created, the value of their fields, etc. From this area, the user can execute the class methods on the instances, or several specific actions on public fields such as directly assigning a value. The top-right area (2) displays the Java code corresponding to the execution sequence that is being created. The middle-right area (3) recalls information on the selected instance and on the corresponding class. The bottom-right area (4) is used to present the result of properties checking, such as invariant or history constraints. If a property evaluates to false, it is

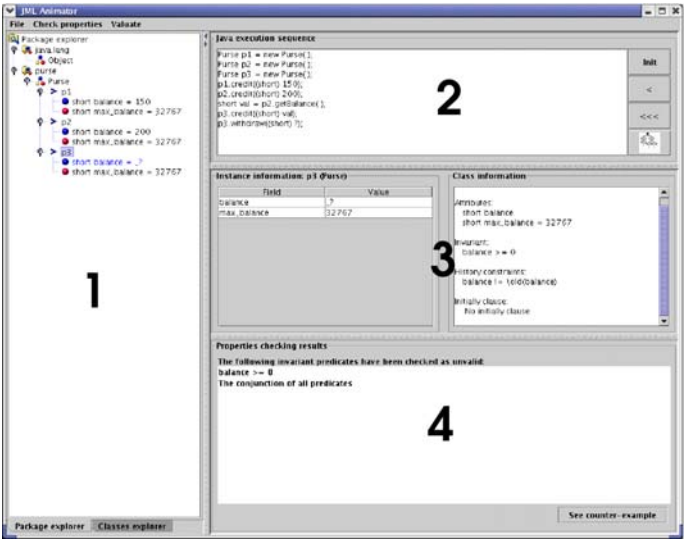


Fig. 2. The JML-Testing-Tools animator main frame

displayed and a counter-example exhibits the state of the system that presents an error. The menu is used to run the verification of properties or the valuation of a constrained environment.

7 General Information

The JML-Testing-Tools has been developed at the Computer Science Laboratory of the University of Franche-Comté CNRS INRIA (France), in the Constraint group led by Professor Bruno Legeard, in partnership with the GECCOO[1] and INRIA CASSIS[2] projects.

JML-Testing-Tools is available for download at the following address:

`http://lifc.univ-fcomte.fr/~jmltt/`

References

[ABC+02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming. In Robert Hierons and Thierry Jerron, editors, *Formal Approaches to Testing of Software, FATES 2002 workshop of CONCUR'02*, pages 105–120. INRIA Report, August 2002.

[Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 1996.

[CL02] Y. Cheon and G.T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[Har87] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Journal of Science of Computer Programming*, 8:231–274, 1987.

[LBR98] G.T. Leavens, A.L. Baker, and C. Ruby. JML: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, October 1998.

[LBR99] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[Spi92] J.M. Spivey. *The Z notation: A Reference Manual.* Prentice-Hall, 2^{nd} edition, 1992. ISBN 0 13 978529 9.

[1] GEneration of Certified Code for Object-Oriented applications
`http://geccoo.lri.fr`

[2] Combining ApproacheS for the Security of Infinite state Systems
`http://www.loria.fr/equipes/cassis/`