JACK: Java Applet Correctness Kit

Lilian Burdy   Antoine Requet

Gemplus
Software Research Labs
Phones +33 (0)4 42 36 45 33, +33 (0)4 42 36 61 66
Fax + 33 (0)4 42 36 55 55
`{lilian.burdy,antoine.requet}@gemplus.com`

Abstract. The paper presents a solution to improve the applet quality by allowing proof on Java Card^{TM} annotated applets. It presents the chosen annotation language: JML that allows to formally specify each method and to give properties on fields. The innovative part of the paper is the presentation of the tools developed in the Gemplus Software Research Labs which allows proving the annotation by translating them in a formal language. To reduce the difficulty of using formal techniques, the tools aim to provide a user-friendly interface which hides to developers most of the formal features and provides him a "Java view" of proofs.

Introduction

As the card is becoming a software platform closer to classical OS, applets also become software closer to classical Java programs with problems inherited from, mainly validation cost and documentation.

Moreover, the card remains a specific domain with costly post-issuance corrections due to the deployment process and the mass product. Currently, works on correctness validation focus on validating platforms or OS components. However, as the card becomes a software platform, being able to ensure correctness of applications is becoming more and more important. This emphasizes the need for adapted tools and methods that would allow increasing the assurance in the developed software.

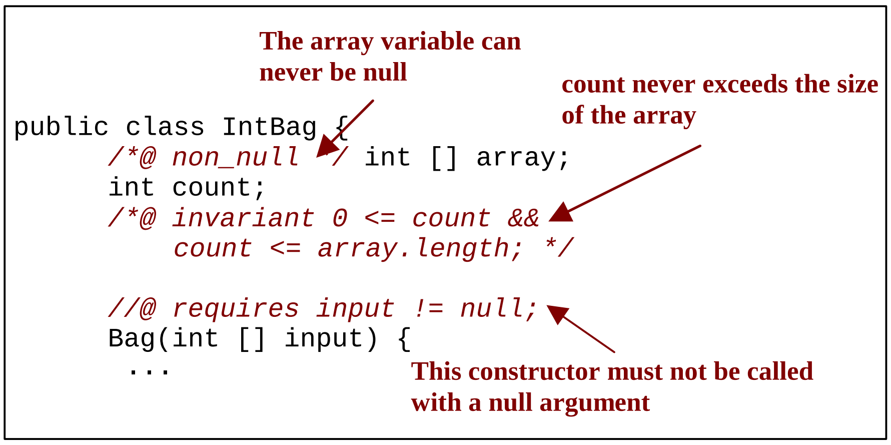
In order to address those issues, it is proposed to use the JML (**J**ava **M**odeling **L**anguage) [1] notation for applet development. This use should be made possible at different levels for example for documentation, development or correctness proof. This is achieved by the use and the development of tools supporting the JML notation.

The tools considered correspond to:

1. JML: the basic tool supporting the JML language. It allows type checking of specification, and generation of JavaDoc like documentation.
2. ESC/Java: Static analysis tool allowing to detect some exceptions (`ArrayOutOfBoundException`, `NullPointerException`, etc...) as well as violations of the JML annotations.
3. JACK (**J**ava **A**pplet **C**orrectness **K**it): Set of tools developed in the Software Research Labs. They allow *proving* properties on Java applets.

The JML language

The JML language is a language that aims to specify Java classes and interfaces. It allows to formally express properties and requirements on those classes.



```
public class IntBag {
      /*@ non_null */ int [] array;
      int count;
      /*@ invariant 0 <= count &&
          count <= array.length; */

      //@ requires input != null;
      Bag(int [] input) {
       ...
```

The array variable can never be null
count never exceeds the size of the array
This constructor must not be called with a null argument

Figure 1 : Example JML annotations

As the JML notation is specially tailored to Java, it can be easily read and written by Java programmers. Figure 1 shows an example of a Java class enriched with JML annotations.

The JML language allows the specification of:

1. *class invariant*: this corresponds to properties on member and class variables that must always hold. The first two JML annotations of Figure 1 correspond to *invariant*.
2. *preconditions*: preconditions are associated to method, and correspond to properties that must hold in order to call the method. The third JML annotation of Figure 1 corresponds to such a precondition expressing that the method must not be called with a null argument.
3. *postconditions*: as preconditions express the properties that must be true when calling a method, postconditions describes the behavior of the method, by expressing the properties ensured by the method. Special postconditions can also be used to describe exceptional behavior, in particular, when the method throws an exception.

JACK

The JACK converter converts a JML file to a B model allowing proving properties on Java program. Developers add the properties in a Java file as special annotations. These annotations are written using the JML syntax. The tool converts the Java JML annotated file in lemmas written in the B language.

The B method [3] is a formal method with different features such as refinement scheme, translation in C code, etc… We only use the B method as a framework that allows to write lemmas in some files and to prove them using a prover. This prover is developed within the Atelier B [2] and has two interfaces: an automatic and an interactive one.

Principles

Our goal is to prove properties on source files written with the Java language. To reach this goal, one has to know how to "translate" a Java source file in a B machine. The two main questions are:

1. How to formalize the object oriented Java features in set theory (the B language is a first order language with set theory) ?
2. How to generate lemmas from Java methods?

Object oriented concept formalization: We bypass this question. The idea is to generate one B machine for one Java file. This B machine contains the context of the current file, that is the class hierarchy that it can access, all the fields and methods that it can use. This information is translated in constants and properties over these constants. They are collected from the current file, the packages and classes imported from the file and from the other classes of the package of the current class.

We do not, for example, formalize inheritance concept; we just associate to each class, a set of seen classes and a subtype relation between classes.

Lemma generation: The tool calculates the lemmas that need to be proven to ensure that a method does not break the invariant and ensures what it claims to ensure. These lemmas are calculated from the source file and are inserted as lemmas associated to each method and constructor. These assertions are calculated by "applying" method body to the formula coming from the JML annotations. This application is based on the weakest precondition calculus, which allows calculating the formula that is sufficient to hold, to ensure that after a statement the requested formula holds. The JML annotations are Java boolean expression without side effects. Though, they are easily translated in a logical formula. The Java statement contains different features like control-flow break. The classical Hoare logic has to be completed to allow the generation of lemmas from methods [4].

Example

This part presents an example of an annotated Java source file with associated lemmas.

```java
/**
 * Returns the minimum value of the three parameters.
 */
//@ ensures
      \result == a || \result == b || \result == c;
//@ ensures
      \result <= a && \result <= b && \result <= c;
int min(int a, int b, int c) {
   return (a = (a <= b ? a : b)) <= c ? a : c;
  }
```

Figure 2: The min method

The source file contains three manners to express that a method returns the minimum value between three parameters:

1. the JavaDoc comment expresses informally what is returned by the method;
2. the JML annotation expresses the property of the value returned by method;
3. the method body contains the code that calculates the returned value.

The example illustrates that the JML annotation can be an intermediate expression between the informal comment and the code. Here, the informal comment is really simple, the code is not really an obvious way to implement such requirement (but it is a small one in term of byte code size) and the JML is the way to be sure that the method is well implemented. In fact, one has only to check, informally, that the property expressed in JML is the expected property, that is, the method returns a value between the three parameters and it is less than each one. The proof ensures formally that the method maintains this property.

As an idea of what is generated by the tool, the figure below shows the lemmas that are generated from the example of figure 2.

```
a <= b && a <= c
==> (a == a || a == b || a == c) && a <= a && a <= b &&
a <= c

a <= b && !a <= c
==> (c == a || c == b || c == c) && c <= a && c <= b &&
c <= c

!a <= b && b <= c
==> (b == a || b == b || b == c) && b <= a && b <= b &&
b <= c

!a <= b && !b <= c
==> (c == a || c == b || c == c) && c <= a && c <= b &&
c <= c
```

Figure 3: Lemmas

There are four lemmas due to the implicit four branches issued from the two conditional operators. The lemmas are JML/Java boolean expression (the conditional operator ==> is part of the JML language).

Development environment integration

JML has the advantage of being a language that can be rapidly and easily learned and used by developers. One can consider that using a prover is not so easy. Nevertheless we believe that formal activities like modeling and proofing are not reserved to expert. To demonstrate this idea, we provide a prover interface understandable to non-expert in formal methods. The JML language is especially well suited for this, since it is close to Java.

```
/**
 * Returns the minimum value of the three parameters.
 */
//@ ensures
       \result == a || \result == b || \result == c;
//@ ensures
       \result <= a && \result <= b && \result <= c;
int min(int a, int b, int c) {
   return (a = (a <= b ? a : b)) <= c ? a : c;
  }
```

Figure 4 : Highlighted min method

The tool provides a user-friendly presentation of the lemmas by highlighting portion of the code that is involved in this lemma. Figure 4 shows this highlighting for the first lemma; it shows that the two tests are passed and that, for each one, a is elected as the minimum of the values.

Conclusion

As Java Card applets become more and more widespread, there are growing requirements for validating them and ensuring their quality. *Proving* that the applet behaves correctly is one means of coping with those new constraints. We show that this approach is feasible by converting JML annotated source code into B lemmas suitable for use within an automated proof tool.
However, due to their mathematical foundations proof tools are usually hard to use by non-experts. In order to reduce this limitation, we provide a simpler interface hiding the mathematical complexity behind a Java view of the lemmas.

Finally, this work will not only allow formal method experts to prove Java applet correctness, but it should also allow non-experts to get in the formal world. This is a necessary point for those validation techniques to be widely used.

References

[1] The JML language home page: http://www.cs.iastate.edu/~leavens/JML.html
[2] The Clearsy home page: http://www.clearsy.com
[3] J.R. Abrial, The B Book, Assigning Programs to Meanings, Cambridge University Press, 1996
[4] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In FASE, volume 1783 of LNCS, pages 284-303. Springer-Verlag, 2000.