Adding native specifications to JML

Julien Charles

INRIA Sophia-Antipolis, France
Julien.Charles@sophia.inria.fr

Abstract. In the specification language JML we can see pure methods as a way to express user-defined predicates that will simplify the annotations. We take this idea a step further in allowing to only declare these predicates in JML without giving an explicit definition. The explicit definition is done directly in the language to which the Java program and the specifications are translated. To this end we introduce a new keyword to JML, the keyword `native`. To facilitate these definitions we have enabled the user to define also `native` types in the same way. In this paper we will describe these new constructs as well as their implementation in Jack, and their application to JML's libraries and model fields.

1 Introduction

The Java Modeling Language (JML) is a widely used specification language used to annotate Java programs for both runtime verification and static verification. It has a vast syntax defined in its manual and most of the tools only implement a part of its syntax. The part of JML we will be interested to in this paper is one that is common in most of the tools using JML [5]: pure methods, ghost variables and model fields.

When doing program verification with JML, one of the difficulties that appears is to handle the connection between the specifications and the interpreted program specifications.We have decided to add the keyword `native` to specify some specification-only methods and types that would be directly defined to the environment to which specifications are interpreted. This keyword can be used in dynamic verification of programs, but it is in a static verification context, that it can be the most helpful. It can be used to define predicates, as well as give a real definition to which lemmas can be proven upon.

In order to test our new construct and our modifications of the JML language we have used a tool that does static program verification, Jack (the Java Applet Correctness Kit)[15, 4]. Jack takes as an entry Java programs annotated with JML. The proof obligations are generated to an intermediate language called JPOL (the Java Proof Obligation Language) and afterward are translated into proof languages, like Coq, Simplify, AtelierB and PVS. Jack handles most of JML constructs, and is integrated as a plugin within Eclipse [8].

The latest developments in Jack concern its Coq output. The interest of this output is that it permits to prove interactively some proof obligations that would not be provable automatically in a first order prover like Simplify [1]. Together with the Simplify output this is the output which is used the most.

Coq is a proof assistant based on the calculus of inductive construction [2]. It can express higher-order logic which first-order automatic provers cannot. In Jack's Coq output, the logic of Java is expressed through Coq as axioms, definitions, inductive definitions as well as recursive functions (in more sparse cases).

In section 2, we will discuss of the pure methods and their interpretation in Jack. We will then introduce the `native` keyword and its use to define predicates. In section 3, we will define the `native` types and we will see the application of these constructs as a way to implement JML's model classes. Finally, in section 4, we will show applications of the native libraries with ghost variables and model fields.

2 Pure methods

JML's pure methods are methods that can be used in specifications. They cannot mutate already existing objects but they can allocate new objects. Nonetheless, the pure keyword is not an alias for JML `modifies \nothing`, it implies also that the methods terminates, giving a result or throwing an exception. For instance if a constructor only modifies the object that is being created and terminates properly it can be considered as pure. A mean to verify whether a method is pure or not according to JML can be found in [16].

In dynamic program verification pure methods are usually built from their source code. The method is first thought on the Java level, without side effects, and afterward the user writes its specifications on the JML level; in order to be able to use it in JML annotations.

In static program verification, pure methods can be built directly from their specifications, since most of the tools replace pure method's calls by the instantiation of the pure method's specifications.

2.1 Jack's implementation

In Jack the notion of purity used is as in JML a kind of observational purity [14], but a constructor that modifies the fields of the newly created objects is not considered as pure. In Jack's weakest precondition calculus the specifications are considered as lightweight: the method calls are replaced by their specifications inside the calculus. The replacement with its specifications is done with the normal specifications in case the method terminated normally or with the exceptional specification if the pure method terminated on an exception.

For a method defined in a Java file:

```
/*@ requires tab != null;
  @ modifies \nothing;
  @ ensures \result == (0 ≤ i && i ≤ tab.length);
  @ exsures false;
```

```
  @*/
public static /*@ pure @*/ withinBounds(int[] tab, int i) {
     return 0 ≤ i && i ≤ tab.length;
}
```

with this method call within the annotations:

```
withinBounds(tab,i)
```

the method call will be directly replaced in the weakest precondition calculus by:

```
(tab != null) → (0 ≤ i) ∧ (i ≤ tab.length)
```

Since the specification of the method has an `exsures false` clause, there is no exceptional case. In a way, this method will be replaced by its specifications like for a macro.

2.2 Specification macros

When specifying a program with JML one of the main problems is the growth of the size of the annotations. The way static verification tools usually define the handling of pure methods, we can use them to do some specification macros. The method calls will be replaced by their specifications when the annotations will be interpreted. It is useful to avoid the growth.

If we have for instance a property to tell an array is sorted we would prefer read the annotation:

```
is_sorted(tab)
```

instead of:

```
(\forall int i; 0 ≤ i && i < tab.length;
         (\forall int j; 0 ≤ j && j < tab.length;
                        (i < j) ==> tab[i] ≤ tab[j]));
```

This method makes the annotations clearer, but as annotations grows big, proof obligations grows big too. In order to ease the readability of the proof obligations, we would like to keep track of the pure method name that was used as the macro in order to see what part of the specification we are proving. That's why we changed the pure method's substitution in Jack.

In Jack, we decided to have a couple Definition / hypothesis. Now a functional definition is generated of the form:

```
mypurefun_norm Args Result := (requires Args) → (ensures Args Result)
mypurefun_exc Args Result := (requires Args) → (exsures Args Result)
```

where `requires` is a predicate that is on the arguments of the pure function and which correspond to JML `requires` clause the same for `ensures` and `exsures` which are predicates that correspond to JML's `ensures` and `exsures` clause respectively. These functions are then called within the hypothesis at the places where they were used in the code.

It is nearly what is done in Krakatoa[13], as Krakatoa use a functional definition of the pure method if it can generate it but otherwise use an axiomatisation of it like in ESC/Java[6]. The axiomatisation is done in 3 parts: the pure method is first declared as a variable, there is some hypothesis using it and giving it its properties (which correspond to its specifications), and then the variable is used within the lemma which has to be proved (for more detailed comparison between the different technique see: [7, 9]).

This way of defining Definition/Hypothesis doesn't change anything for automatic proof of the proof obligations with prover like Simplify. With Coq, it facilitate the readability of the proof obligation for the user which is a critical point, notably when doing an interactive proof.

2.3 Pure as Predicates

Some of the properties we have to express are not so easy to deal with on the JML level. We want to be able to prove lemmas concerning pure methods, and also have relations over variables without specifying any property on the relation.

So we decided to be able to define pure methods directly within the language in which the proof obligations are generated or the JML annotations are interpreted. We added a new keyword to JML in order to allow it: the `native` keyword. If a method is declared within a specification as native, the method will not be defined nor specified in JML at all, it will only be declared. Its specifications will be to the target prover or environment discretion.

Since the native methods are declared within the specifications they must be pure: they must not have any side-effect, they can only create objects, they have to be terminating. But native is more restrictive than pure: a native method must not throw any exception.

For instance we can have the property `withinBounds` declared as native, inside the specification:

```
//@ public native static boolean withinBounds(int [] tab, int i);
```

If interpreted with a dynamic program verification tool, it can be defined with the Java method:

```
public static boolean withinBounds(int [] tab, int i) {
      return (tab != null) && (0 ≤ i) && (i ≤ (tab.length));
}
```

If interpreted with a static program verification tool, it can be defined this way in Coq:

```
Definition withinBounds :=
   fun (tab i) =>
       (and (not (tab = null)) (and (0 ≤ i) (i ≤ (arraylength tab))).
```

In Simplify it will be seen as an uninterpreted function symbol, just a relation on the arguments.

In Jack, the binding from the JML declaration to the `native` language is done automatically. The arguments passed to the method are the same as the one whose the method was declared with except:

– if the method is an instance method, an extra argument `this` is added by Jack at the beginning of the method when it is translated
– if one of the argument is an array, the array dereferencing relation (to do array access) and its length relation are also given

For static program verification, this construct can be really useful, especially if it is used in the pure macro fashion. Even though we lose the ability to express JML's behaviour with these specification macros, we can now easily prove properties over specifications in the target prover language. Once these properties are proved, they can be added as a help to ease the automatic solving to some of the proofs of the proof obligations.

In ESC/Java or Krakatoa it is permitted to define a pure method in specifications only, but its definition/specifications must be written in Java. Krakatoa is indeed a front-end for Java and JML to the Why tool [10]. In this tool there exists a mechanism which allow similar definitions as the native keyword: the `parameter` construct. The main difference with the native keyword is that `parameter` is more tool dependent, it is used implicitly within annotations and it is not used on the JML level.

3 Native types

One of the interest of these native methods would be to use them to define specification-only libraries. But to do so, native methods are not expressive enough: primitive Java types and object types as defined in JML are not really good to handle some Coq native functions and expressions. A good thing would be to be able to use directly some Coq types within our specifications. A solution can be to embed some Coq types within a Java type (like `Reference`). First we need some new axioms to get and set the Coq special values, together with a reduction rule. For instance if we want to manage a list type we would need the relation `setList`:

```
Variable setList: Reference → list → Reference.
```

and the relation `getList`:

```
Variable getList: Reference → list.
```

and the rewriting rule:

```
Axiom getsetList: forall r l, getList (setList r l) = l.
```

This method is bad because we have to add many axioms, and this is not really done in a natural way. To properly do this kind of manipulation we would need some types directly defined in the target environment, some 'native' types.

3.1 Definition

We have extended the native construct to the type definitions, in order to be able to map existing libraries well defined in the target prover language to some specification written in JML. With this construct we can declare a type and some operations on it (with the native methods) that will be entirely defined in the target language.

The syntax is the same as for the 'native' pure methods:

```
/*@ public native class MyNativeType {
  @    public native boolean myNativeMethod();
  @    public native static MyNativeType myStaticMethod1();
  ...
```

The `native` keyword is mandatory for native methods inside the native classes since static verification tools (notably Krakatoa) usually allow defining specification only methods, which are pure methods defined only by their specifications.

These native types are not standard Java/JML classes, they are more akin of a functional type:

- they do not inherit from the Object class, they are outside the Java type hierarchy, and they do not subtype one another;
- they are not instances, in fact they are not even some references but they could be bound to a reference type in the target language;
- they have no default initializer: if a specification variable is declared with this type it must be initialized to a value returned by a method or taken from another variable of the same type;
- it has no constructors: since constructors are used in Java to initialize the object (they return nothing) there is no semantic in initializing the newly created 'object' from a native type

On the other hand they allow method calls à la Java on them. There are two kind of method calls on these types:

- the static call, which is just a normal method call from a method defined inside a specification library.
  For instance we can have this kind of calls:

  ```
  //@ assert MyNativeType.myStaticMethod1() !=
                           MyNativeType.myStaticMethod2();
  ```

  where `myStaticMethod1()` and `myStaticMethod2()` return values of type `MyNativeType`, and are two native static methods.
- the instance call, where the variable on which the native method is called is passed as a parameter. For instance:

  ```
  //@ assert MyNativeType.myStaticMethod1().myNativeMethod();
  ```

  which correspond to a call to `myNativeMethod()`, with only one argument passed to the method: the result of `myStaticMethod1()` call. This is a valid for JML assertion since the method `myNativeMethod()` is pure (native in fact) and returns a boolean.

3.2 Soundness of the method

Defined this way, the native types are sound with respect to the way they are implemented in the target environment.

In Jack, for the Simplify output, the native types and methods are just uninterpreted function symbols. There is no axioms on them, so there is less chances to be able to prove automatically the proof obligations using these types. So the proof obligations are as sound as it would have been without these new constructs. Jack logic for Simplify could be extendable, in order to allow to add axioms on the native methods and types, but it has not been implemented yet.

In Coq the native types, just like the native methods are defined in a library. Jack's Coq plugin was modified in order to call the user defined type everytime a variable tagged as native is translated from Jack to Coq. Letting the user define by himself the type and the properties over them can lead to unsoundness. One way to avoid this is to encourage the user to use only functional definitions, like definitions or fixpoints to define the native types and the operations over them. The aim is to allow to use mostly executable constructs of Coq which will not modify the logic of Jack for Coq, and not add unnecessary axioms.

If we do runtime verification the native types can be mapped to a Java implementation. The only requirement is for every native methods defined to be pure and that it does not throw any exception. The definition will be sound with respect to JML and Java interpretation.

3.3 Native libraries

The native construct in JML enables to easily declare libraries that will be used within the specifications like JML's model classes. For instance, if we want to have a library on sets we could define it this way:

```
/*@ public native class ObjectSet {
  @    public native static ObjectSet create();
  @    public native static ObjectSet add(ObjectSet os, Object o);
  @    public native boolean member(Object o);
  @ ....
  @*/
```

On the Coq level an easy way to bind this to a library, is to map it to the Coq library ListSet:

```
Definition ObjectSet := set Reference.
Definition ObjectSet_create := empty_set.
Definition ObjectSet_add (os: ObjectSet) (o: Reference) := set_add o os.
Definition ObjectSet_member (this: ObjectSet) (o: Reference) := set_mem o this.
....
```

First we define the Coq type on which the `ObjectSet` native type will be bound: it is a set of objects. For Jack, objects are seen as `Reference`, hence the first definition where `ObjectSet` is defined by `set Reference`. Then we define the method `create` by the definition `ObjectSet_create`, which is static and takes no argument. When it is created the set is empty. The `add` method is defined by `ObjectSet_add`; in Coq it is directly bound to the `set_add` method of the library `ListSet` of Coq. It returns a new `ObjectSet` where `o` has been added to the preexisting `ObjectSet os`. The `member` method is an instance method, it tells if the `ObjectSet` on which it is called rightly contain the argument `o` which is an object. It is mapped directly to the `set_mem method` of the `ListSet` library.

In fact when we define a library this way, there are two distinct sort of methods:

- the modifiers which are implemented solely as static methods. Since native types are of a functional nature (all their methods must be without any side-effect), each time we want to modify a data of this type we must create a new object.
- the observers which can be instance methods or static methods.

This way of defining the libraries forces the programmer to make a clear distinction between modifiers and observers.

It gives a way to have a Set library in JML a bit like what is done for `JMLObjectSet` [11], but with differences: modifiers here are all static (which is not the case for `JMLObjectSet`), the `ObjectSet` type is outside the Java class hirarchy since it is a native type, so it does not have to define all the inherited methods from the class Object (namely `equals(Object obj)`, `hashCode()` or `wait()`...) and are not interesting when defining a library to use sets within specifications.

4 Application of the native libraries

4.1 Ghost variables

The direct application of native libraries is their use with ghost variables. In JML ghost variables are specification-only variables. In the Jack implementation as well as for all the other JML tools, these variables are treated as normal variables except they can only be used in specifications and they can only be modified using the JML set construct.

Since they are only defined within the specifications, ghost variables can have a native type. To use our ObjectSet library we could have for instance a variable mySet. It would be declared as followed in a Java program:

```
//@ ghost ObjectSet mySet = ObjectSet.create(); // native types variables
          // must be initialized since they have no default initializer
```

We could add elements to our ObjectSet using JML's set instruction (with the static modifier):

```
//@ set mySet = ObjectSet.add(mySet, new Object());
```

and finally we could then use it within an assertion (with the instance accessor):

```
//@ assert mySet.member(new Object());
```

The only difference in the use of native types for ghost variables instead of Java types is the reduced number of properties and proof obligations we have on them. For instance there's no point for the assertion to generate a proof obligation to verify that the program is not dereferencing a null pointer. We had to initialize the ghost variable with a first value, for which we have *a priori* no hypothesis (except if we know the implementation of the natives, which is prover or target system dependent).

The native types add some expressiveness to JML annotations and are a way to implement easily the JML model classes libraries. Ghost variables are useful when specifying a program, but sometimes we would like to be sure specifcation variables model real program behaviours. That's why we used our native construct with JML's model variables.

4.2 Model fields

Definition Model fields are specification variables defined by a representation function or with a representation relation that maps a program variable to a model variable. Here we will only interest ourselves in defining the model field by a representation function (detailed hints on the implementation of model fields for static verification can be found in [12, 3]). One must use 3 constructs in JML to declare a model field:

- first it must be declared with the model keyword:

  ```
  //@ model MyType myModel
  ```

- then it must be linked to a program variable with an abstraction function:

  ```
  //@ represents myModel ← myFun(progVar);
  ```

- finally we must specify that when the program field is modified the model field is modified too.

  ```
  //@ depends myModel ← progVar;
  ```

The `represents` and `depends` clauses are strong invariants: they must not be broken in any way in any state of the program.

Since one of the aims of model fields is to gain abstraction from the program, we could do this abstraction with a native construct. Program variables can be abstracted to a native type using a native abstraction function that will link the native type with the Java type. The native method would have this signature inside of JML:

```
//@ public native static MyNativeType translate(MyJavaType var);
```

Once defined we can use it smoothly within the `represents` clause.

Modeling an array with sets One of the immediate applications would be to implement such a translation function to model a Java array of Objects with sets, using the sets defined as a native library (as defined in subsection 3.3). To be able to model an array with our native sets, the only thing missing is an abstraction function from array to sets. This function would have this declaration:

```
//@ public static native ObjectSet toSet(Object [] tab);
```

Translated through Jack it would be linked with an `ObjectSet_toSet` definition in Coq:

```
Fixpoint toSet_intern (tab: Reference)
   (refelements: Reference → Z → Reference)
        (len: nat) {struct len} : set Reference :=
   match len with
   | S n => set_add (intelements tab (Z_of_nat n))
                    (toSet_intern tab refelements n)
   | S 0 => empty_set
   end.
Definition ObjectSet_toSet :=
   fun (tab: Reference) (refelements: Reference → Z → Reference)
           (arraylength: Reference → Z) =>
      if (tab = null)
         empty_set
      else
         match (arraylength tab) with
         | Zpos p => toSet_intern tab refelements (nat_of_P p)
         | _ => empty_set
         end.
```

where `refelements` is a dereferencing relation and `arraylength` the relation to get the length of an array. This translation function is built around 2 functions:

- a first one (`ObjectSet_toSet`) determining the value of the result if the `tab` parameter is `null`
- a recursive function (`toSet_intern`) that add to the set each element from the array

The only thing missing from these definition are the lemmas to ease the proofs. Typical examples are:

- ```
  forall tab arraylength, tab <> null →
          (forall i, 0 ≤ i ∧i < (arraylength tab) →
                  (forall refelements, ObjectSet_member
                          (ObjectSet_toSet tab refelements arraylength)
                          (refelements tab i)).
  ```
  each element in the array `tab` is a member of the corresponding set defined by the translation function.
- ```
  forall tab arraylength,
          (forall ref, ObjectSet_member
                  (ObjectSet_toSet tab intelements arraylength) ref →
  ```

```
exists i, (refelements tab i) = ref).
```

for each element of the result set of the translation, there exist an element in the array from which the set was translated.

5 Conclusion

We have added the `native` keyword inside of Jack to help solve the different proofs. It can be used to build native libraries that could replace or at least elegantly complement the model classes, adding more expressiveness to define such types and simplifying their definitions.

Defining abstractions over program variables with model fields and native types enable one to refine Coq data structures to Java programs. The proof obligations using these abstractions are not so easy to prove using Jack. One needs lots of intermediate lemmas to do so. With this methodology, one must first prove some properties on the library, like that the translation function `toSet` has the same number of elements as the originating array, if each element in the array was different otherwise it will have less elements.

We have done the same kind of work over a list library to help implement a QuickSort. The size of the library that only binds the Coq constructs with the JML native methods and types is 251 lines long, the lemmas proved to help do the proof obligations are 569 lines long. With Jack, around 230 proof obligation are generated, half of them are solved automatically, and the proof scripts of each solved interactively range from 2 or 3 lines (most of the proofs) to 20. Here with the model variables about just as much are solved automatically, and afterward the longest proof takes around 10 lines. First we have to prove properties on the library, and after the proof is easier to do. For multiple use, it is really good to have such JML libraries, fully proven, enabling faster proof and program developments.

Jack doesn't check yet if the defined native method or type well-fit with the declared ones. It will be an interesting development to add this verification.

Another work would be to do refinement from the target language not only on the data level but also on the program level. A nice way to handle this would be to use JML's model program construct together with the native methods and types. We could imagine to implement a program with our favorite prover, then bind it to JML's model program construct using the native methods and types, and finally have to prove that the program behave the same as the model program which is in fact the prover-implemented program.

The only problem is that the model program construct is not implemented at all inside the main static program verification tools, neither in ESC/Java2 nor in Krakatoa nor in Jack.

Acknowledgements: *I thank Lilian Burdy for having coined the 'native' keyword and for his valuable help on Jack, I also thank Gilles Barthe, Benjamin Grégoire, Marieke Huisman, Mariela Pavlova, Erik Poll, Sylvain Boulmé, Christine Paulin-Mohring and the anonymous reviewers for their helpful comments.*

References

1. Simplify. Its 'official' website: http://research.compaq.com/SRC/esc/Simplify.html.
2. Bertot and Y. P. Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Series: Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
3. C. Breunesse and E. Poll. Verifying JML specifications with model fields, 2003. In Formal Techniques for Java-like Programs. Proceedings of the ECOOP'2003.
4. L. Burdy et al. The Jack website, 2004-2006. http://www-sop.inria.fr/everest/soft/Jack.
5. L. Burdy, Y. Cheon , D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Rustan, M. Leino et Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 2005.
6. Cok and D. J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.
7. D. Cok. Reasoning with specifications containing method calls and model fields. In *Special Issue: ECOOP 2004 Workshop FTfJP*, volume 4 of *Journal of Object Technology*, pages 77–103, 2005.
8. The Eclipse Consortium. The Eclipse website. http://www.eclipse.org/.
9. Á. Darvas and P. Müller. Reasoning About Method Calls in JML Specifications. In *Formal Techniques for Java-like Programs*, 2005.
10. J.-C. Filliâtre. The Why Tool. http://why.lri.fr/.
11. G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok et J. Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available here: http://www.jmlspecs.org, 2006.
12. K. R. M. Leino and P. Müller. A verification methodology for model fields. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2006.
13. Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.
14. David A. Naumann. Observational purity and encapsulation. volume 3442 of *Lecture Notes in Computer Science*, pages 190–204. Springer-Verlag, 2005.
15. Burdy, A. Requet and L. J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.
16. A. Salcianu and M. Rinard. Purity and side effect analysis for java programs. volume 3385 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag, 2005.