Adding native specifications to JML

Julien Charles
INRIA Sophia-Antipolis, France

Tuesday, july the 4th 2006

Introduction

- Contex : verify programs written in Java annotated with JML with Jack
- Static verification of programs :
  $\longrightarrow$ done in an interactive way with Coq.
- Motivation : some properties where not easy to express on the JML side.
- We needed some kind of uninterpreted functions symbols to express them

General Plan

1. Basics
2. Native Methods
3. Native Types

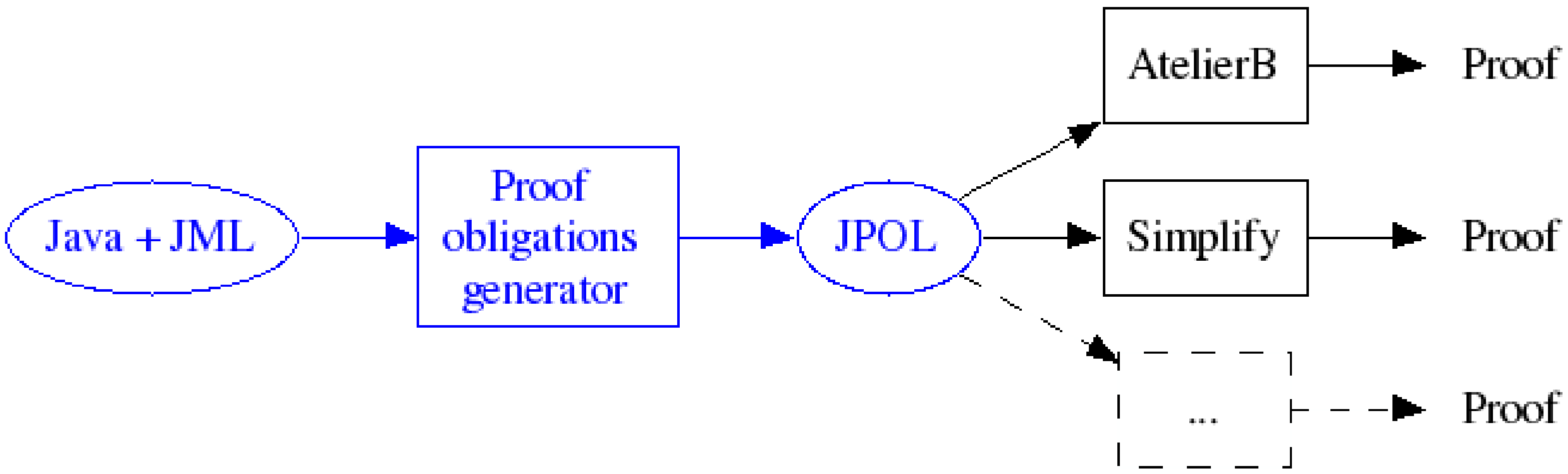
Basics

1. JML, Jack
2. Coq
3. Purity

JML and Jack

The Java Modelling Language (JML) is used to annotate the Java programs we want to verify with Jack.
Jack :

- An Eclipse plugin
- Uses a weakest precondition calculus to generate proof obligations
- The proof obligations are decomposed w.r.t. the different execution cases



Jack with Coq

Coq is based on the calculus of inductive constructions.
You can express : types, axioms, functions variables, definitions, lemmas...

It can easily express the logics for Jack.

The files generated by Jack in Coq can be separated into 3 categories :

- The prelude containing the logic used :
  - A static part (jack_arith.v jack_references.v jack_tactics.v)
  - A dynamic part generated for a specified class (myClass_classes.v myClass_subtypes.v myClass.v)
- The proof obligation (1 file)
- A file containing some custom tactics written by the user

To edit the files we use the ProverEditor plugin, an integrated editor for Coq files in Eclipse.

Purity in JML

Pure methods are methods you can use in specifications in JML.

- No 'visible' side effect
- A constructor can be pure

The main problem for proofs readability : keeping track of the pure method name

Purity in Jack

- In Jack constructors are not pure
- it is translated into a couple definition / hypothesis

```
Definition myfun_1 : A1Type → A2Type → ResType → Prop :=
fun (arg1 arg2 Res) => Context1 → Res = true. (* ResType = bool *)

Definition myfun_2 : A1Type → A2Type → ResType → Prop :=
fun (arg1 arg2 Res) => Context2 → Res = false.

...
(* Used within a hypothesis : *)
...  → myfun_2 this l_var1 Res_23 → Res_23 = true.
```

Native methods

1. Macros
2. Predicates
3. Native methods

Specification macros, predicates in JML

- Pure methods can be used to do some specification macros :

```
/*@ requires true;
  @ ensures \result == ((tab != null) && (i >= 0) && (i < tab.length));
  @*/
public/*@ pure @*/ boolean withinBounds(Object[] tab, int i) {
      return (tab != null) && (i >= 0) && (i < tab.length);
}
```

- Go further in this direction :

  directly defining the pure method in Coq

Native methods

How the predicates are defined for Jack :

- Within JML :

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

- In the file user_extensions.v :

  Definition withinBounds :

  Reference $\rightarrow$ (Reference $\rightarrow$ t_int $\rightarrow$ Reference) $\rightarrow$ t_int $\rightarrow$ bool :=

  fun tab intelements value =>

  and (tab != null) (and (0 <= value) (value < (arraylength tab))).

Static libraries

These functions can define some prover specific libraries :

- They permit to have clearer annotations withinBounds(tab, i) instead of (i $>=$ 0) && (i $<$ tab.length)
- We can prove properties on them at the Coq level

These are a way to explicitly declare the type of uninterpreted function symbols :
$\longrightarrow$ alike ESC/Java dttfsa construct and Why's Parameter construct, less 'tool specific'
... but it's not enough

Using native types

1. Definition
2. Method calls
3. Application

Native types

- To express more complex properties, we'd like special datatypes
  → Have sets easily in Coq, not in JML
- A new construct to directly use a Coq defined datatype :
  //@ public native class ObjectSets
  In the file user_extensions.v :
  Definition ObjectSet : set Reference.
- The types are not some standard Java/JML class types :
  - it does not inherit from the class object
  - no constructors, no casts
  - it is not an instance, etc...
  - more of a functional type : modifiers create new objects and are 'static'

Method calls

```
/*@ public native class MyNativeType {
  @      public native static boolean staticMethod1() ;
  @      public native static boolean staticMethod2() ;
  @      public native boolean nativeMethod() ;
  @ }
  @*/
```

Alike for Java defined classes :

- Static method calls :

  ```
  //@ assert MyNativeType.staticMethod1() !=
  MyNativeType.staticMethod2() ;
  ```

- Instance method calls :

  ```
  //@ ghost MyNativeType myNativeObject ;
  ...
  //@ assert myNativeObject.nativeMethod() ;
  ```

Application : A set library

We can now define a set library binded on some Coq types, to use in annotations :

```
/*@ public native class ObjectSet {
  @     public native static ObjectSet create();
  @     public native static ObjectSet add(ObjectSet os, Object o);
  @     public native boolean member(Object o);
  @     public static native ObjectSet toSet(Object [] tab);
  @ }
  @*/
```

```
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
```

Using native libraries

- Must prove properties over the library to make it usable
  $\longrightarrow$ Usually longer, but the proofs are reusable
  (We have done it for a quick sort proof using a Coq list library)
- It can be used with ghost variables, and also with model variables.
  $\longrightarrow$ We can refine some Coq datatypes to some Java libraries.
  (Work in progress : proof of the correctness of a bytecode verifier written in Java, against a semantic defined in Coq)

Conclusion

- We have added the native keyword for types and methods in Jack
- We can now use some Coq functions and datatypes in Jack's JML annotations, as well as any prover Jack generates proof obligations to.
- We have a way to define simply JML's model libraries
  $\longrightarrow$ It enables to do some refinement from prover datatypes to Java datatypes
- The next steps :
  - Adapt proofs using the native construct
  - Soundness of the method