Playing with Coq in Jack

Julien Charles

Tuesday, May the 30th 2006

Introduction

- Goal : verify Java components annotated with JML
- Static verification of these annotated components :
  - often done with some automatic provers (...Simplify)
  - but there is a need for interactive
- I have worked on the inclusion of the Coq interactive proof assistant to solve the proof obligations of the tools : Jack

General Plan

1. Basic principles
2. The Coq plugin
3. ProverEditor

1. Basic principles

1. Some basics
2. Jack
3. Interacting with the user
4. Interacting with the prover

Some basics

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

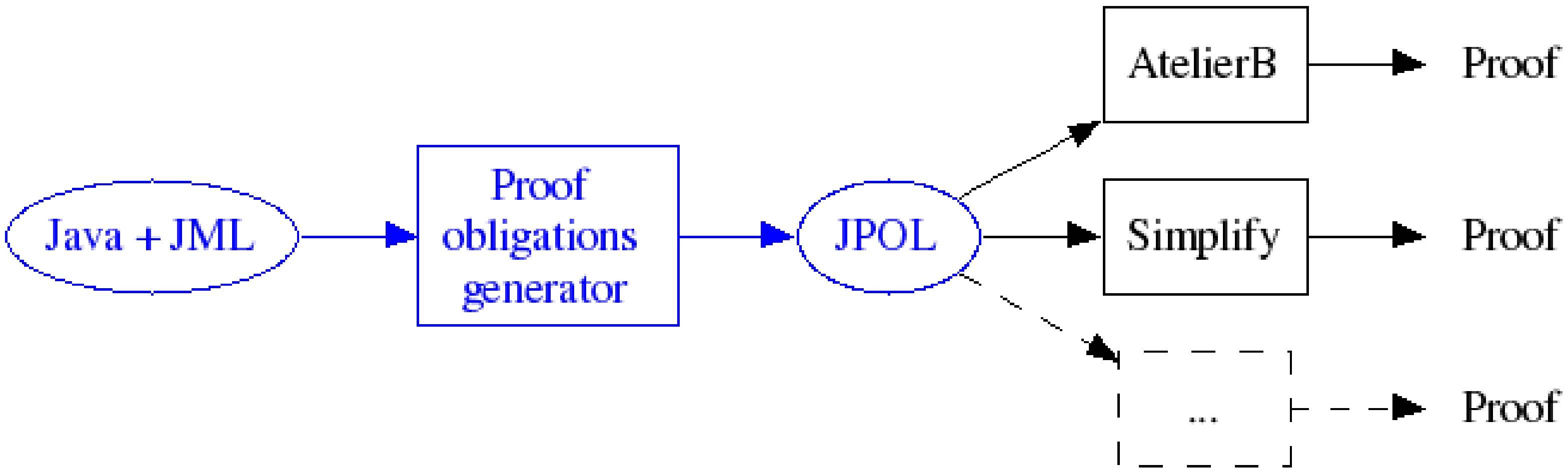
Jack and ESC/Java use a weakest precondition calculus to generate the proof obligations.

In JML you can express loop invariants, assertions method's pre and post conditions.

JML has a really large syntax so every tool (except maybe JMLrac) implements only a portion of its syntax.

Jack

- At first single prover (B)
- Multi-prover interface (Simplify, Coq, PVS)
- Fully integrated in Eclipse as a plugin, and use a plugin architecture to include new provers



Some features

Jack supports most JML functionnalities although the parser is old.
It adds 2 construct :

- `loop_modify` keyword
- `native` keyword for JML annotations

Jack's IDE is also used by Mariela Pavlova's weakest precondition calculus over bytecode

Interacting with the user

Jack is interactive prover friendly :

- Case Explorer
- Ability to prove only one proof obligation
- Can try to prove several goals automatically

POs are generated for each execution path.

Interacting with the prover

Jack interacts with provers through some prover plugins.

Jack has only one way to interact with provers :
$\longrightarrow$ It passes a list of proof obligations (PO) to the prover.
If there is only one PO is most likely done from an interactive mode.
$\longrightarrow$ the context from which a prover is called depends on the informations given by the prover plugin.

3. The plugin for Coq

1. Presentation
2. Memory model
   (a) Types
   (b) Memory model
3. Proofs
   (a) Lemmas
   (b) Human interface
   (c) Tactics

Coq

Coq is based on the calculus of inductive constructions.

You can express : types, axioms, functions variables, definitions, lemmas...

Lemmas have to be proved.

To prove a lemma you have to build a proof term out of the types, axioms, variables...

To build this proof term you use special commands called tactics within a proof script.

Custom tactics can be created composing more primitive ones.

It can easily express the logics for Jack.

General architecture

3 parts are generated :

- The prelude containing the logic used
  - Several files, with some generated dynamically (for the properties on class fields, and the subtyping relation)
- The proof obligation (1 file)
- A file containing some custom tactics

Types

Types for variable :

- Variable with Java primitive types (int, boolean...) have their types mapped to corresponding Coq types (Z, bool...)
- Variable can be also of type REFERENCES.

A variable of type REFERENCES subtypes a Java Type :

Inductive Types : Set :=

| class : Classes $\rightarrow$ Types

| array : Types $\rightarrow$ Z $\rightarrow$ Types.

Subtyping (1)

- In the dynamic prelude :

```
Definition subtypes (t1 : Types) (t2 : Types) : Prop :=
    match t2 with
    | (class c_int) =>
            ...
    | (class c_MemoryState) =>
            match t1 with
            |(class c_MemoryStateJavaLight) => True
            | (class c_MemoryState) => True
            | _ => False
            end
            ...
    | _ => t1 = t2
    end.
```

Subtyping (2)

We need an axiom of transitivity :
$\longrightarrow$ It could be proved for the Jack version, but the proof takes lots of time to compute.

```
Axiom subtypes_trans :
forall a b , subtypes b a  →
             forall c, subtypes c b  → subtypes c a.
```

Memory model

In Jack fields modifications/access and array modifications are done with overriding constructs :

- overridingCoupleRef
- overridingCoupleZ
- overridingArray

Definition overridingCouple f obj inst res :=

```
        if (= obj inst) then
              res
        else
              f (obj).
```

The variable is replaced by the overriding.

Array access is done with the relation :

Variable myTypeelement : REFERENCES $\rightarrow$ Z $\rightarrow$ myType.

Expressions types

In JML forall and exist expressions are of type boolean

$\longrightarrow$ not the case at all in Coq

Coq use Proposition (type Prop).

- Jack does the differenciation in the proof tree

  $\longrightarrow$ No problems

POs generated by Jack

It gives each hypothesis with some hints from their origin :

- bad for the performances of first order provers
- useful informations for the proof script (makes the lemma nearly human readable)

Making more readable the POs generated :

- once translated it is passed to Coq to pretty print
- separation between the different kind of hypothesis

$\longrightarrow$ Cleaning with tactics

Solving proofs

To prove lemmas in Coq we have to use tactics
$\longrightarrow$ LTac

- The model used is not really oriented for first order provers
- The tactics are necessary especially when POs are big.

In fact, having lightweight and powerful tactics is crucial.

Tactics

- Lightweight and general :
  - elimAnd, elimOr
  - tryApply
  - solveInc
  - cleanUp
- Tool specific :
  - solveOver, heap management axioms
  - arrtac, array heap management axioms
  - absurd
- The starting tactic
- Custom `auto` with hints database

Automatic POs resolution

Automation in Coq can be done several ways :

- Light way
- Tough way (nearly unusable !)
- Custom way

It is really worst than with Simplify, but on the samples Simplify was not able to solve all the POs whereas they can be solved interactively with Coq.

Some stats (from a bcv sample)

Classes :	VerificationException	State	Instruction	Verifier
Nbr of code lines :	13	14	47	66
Annotations lines :	5	20	54	81
POs (Jack) :	60	26	129	627
Simplify (Jack) :	60	26	53	5
POs (ESC/Java) :	3	6	14	3
Simplify - warnings :	6	2	9	5

3. ProverEditor

1. Existing editors
2. Motivations
3. Description
4. Some screenshots

Existing editors

- Standard tools :
  - Proof General (emacs)
  - CoqIde (ocaml)
- More related works :
  - PCoq (Java)
  - The new proof general (integrated in eclipse, but only for Isabelle)

Motivations

At first :

- Editor not included in Jack's Coq plugin
- The use of CoqIde or ProofGeneral to edit Jack's Coq files

What for ?

- Separating the library controlling Coq from Jack
- Jack is already in Eclipse, why not edit Coq files in it too ?
- Proof General is getting in Eclipse but for now not it looks 'dead'
- need a light-weight multi-prover interface in Eclipse

ProverEditor

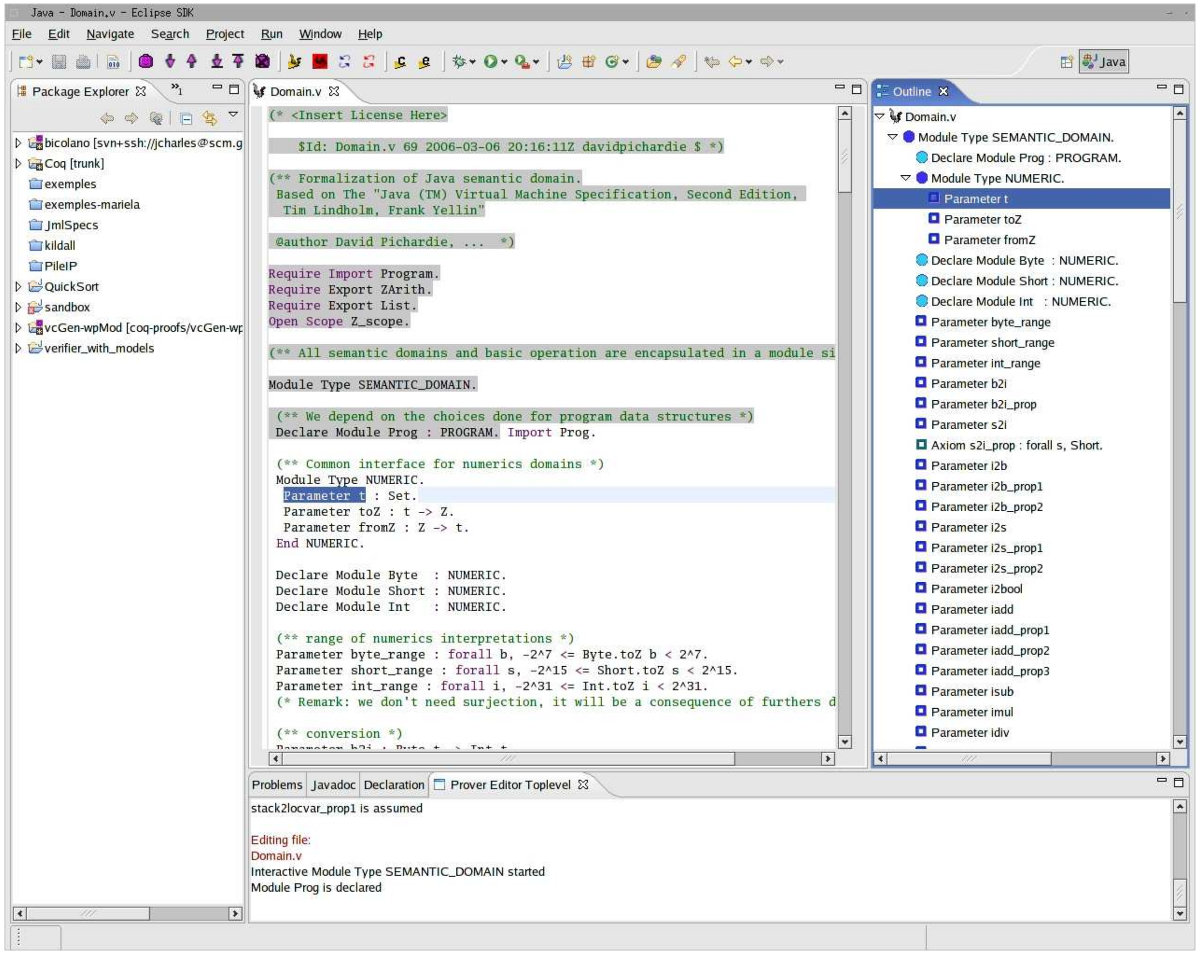
It has :

- syntax highlighting for both the Coq file and the proof view window
- the same keyboard shortcuts as CoqIde

Gain :

- Full integration within eclipse :
  - no weird pop-up except if the user wants to use another editor
  - the managing of the proof files made a bit easier
- As a separate developpement : better handling of Coq than what was done within Jack, and also usable with ESC/Java
- Handling of large files ($>$ 1 Mo)

Demo...



Java - Domain.v - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Package Explorer
bicolano [svn+ssh://jcharles@scm.g
Coq [trunk]
exemples
exemples-mariela
JmlSpecs
kildall
PileIP
QuickSort
sandbox
vcGen-wpMod [coq-proofs/vcGen-wp
verifier_with_models
Domain.v
(* <Insert License Here>
$Id: Domain.v 69 2006-03-06 20:16:11Z davidpichardie $ *)
(** Formalization of Java semantic domain.
Based on The "Java (TM) Virtual Machine Specification, Second Edition,
Tim Lindholm, Frank Yellin"
@author David Pichardie, ... *)
Require Import Program.
Require Export ZArith.
Require Export List.
Open Scope Z_scope.
(** All semantic domains and basic operation are encapsulated in a module si
Module Type SEMANTIC_DOMAIN.
(** We depend on the choices done for program data structures *)
Declare Module Prog : PROGRAM. Import Prog.
(** Common interface for numerics domains *)
Module Type NUMERIC.
Parameter t : Set.
Parameter toZ : t -> Z.
Parameter fromZ : Z -> t.
End NUMERIC.
Declare Module Byte : NUMERIC.
Declare Module Short : NUMERIC.
Declare Module Int : NUMERIC.
(** range of numerics interpretations *)
Parameter byte_range : forall b, -2^7 <= Byte.toZ b < 2^7.
Parameter short_range : forall s, -2^15 <= Short.toZ s < 2^15.
Parameter int_range : forall i, -2^31 <= Int.toZ i < 2^31.
(* Remark: we don't need surjection, it will be a consequence of furthers d
(** conversion *)
Outline
Domain.v
Module Type SEMANTIC_DOMAIN.
Declare Module Prog : PROGRAM.
Module Type NUMERIC.
Parameter t
Parameter toZ
Parameter fromZ
Declare Module Byte : NUMERIC.
Declare Module Short : NUMERIC.
Declare Module Int : NUMERIC.
Parameter byte_range
Parameter short_range
Parameter int_range
Parameter b2i
Parameter b2i_prop
Parameter s2i
Axiom s2i_prop : forall s, Short.
Parameter i2b
Parameter i2b_prop1
Parameter i2b_prop2
Parameter i2s
Parameter i2s_prop1
Parameter i2s_prop2
Parameter i2bool
Parameter iadd
Parameter iadd_prop1
Parameter iadd_prop2
Parameter iadd_prop3
Parameter isub
Parameter imul
Parameter idiv
Problems Javadoc Declaration Prover Editor Toplevel
stack2locvar_prop1 is assumed
Editing file:
Domain.v
Interactive Module Type SEMANTIC_DOMAIN started
Module Prog is declared

Conclusion

- Jack has a working plugin for Coq, with an editor inside of Eclipse
- Still some works on :
  - The JML parser (it has to be changed !)
  - The tactics
  - Better integration to Eclipse / ProverEditor for PVS

Download links

Jack :

`http://www-sop.inria.fr/everest/soft/Jack/`

ProverEditor :

`http://gforge.inria.fr/projects/provereditor/`