Playing with Coq in Jack and ESC/Java2

Julien Charles
Benjamin Grégoire

Monday, december the 12th 2005

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 : ESC/Java2 and Jack

General Plan

1. Basic principles
2. Interfacing
3. Modelling in Coq
4. CoqEditor

1. Basic principles

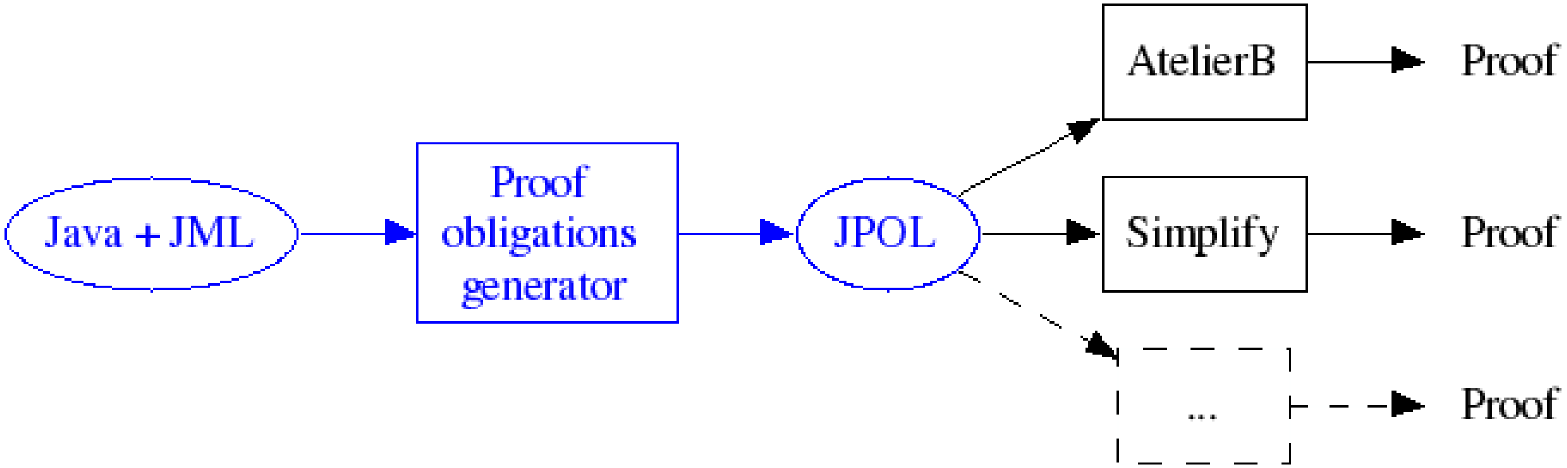
1. Some basics
2. Jack
3. ESC/Java

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.

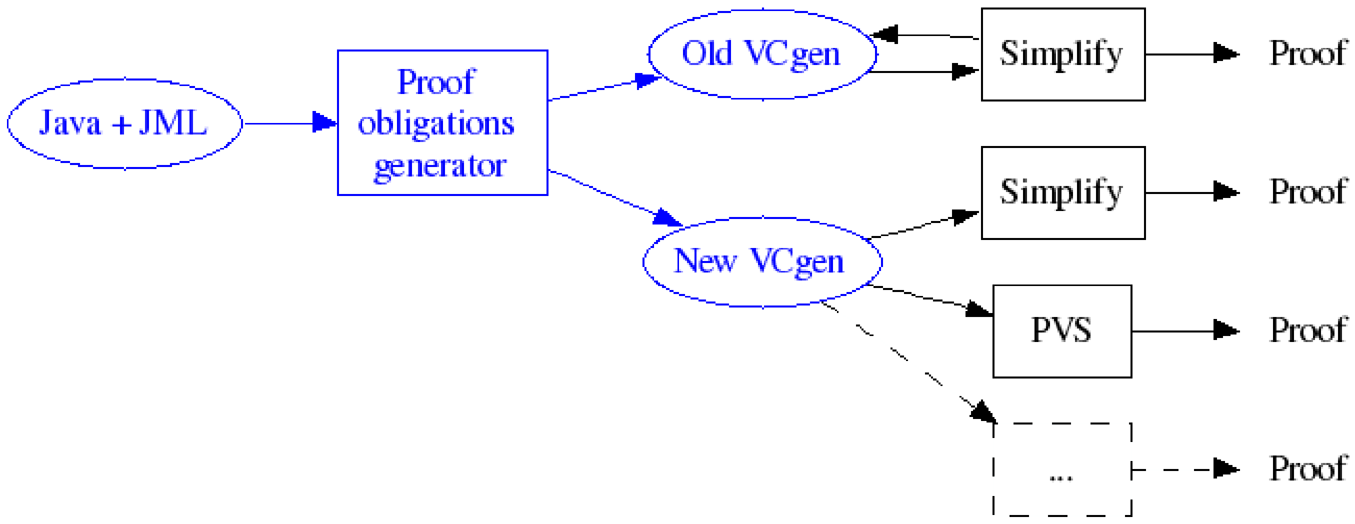
Jack

- Programmed mainly by Lilian Burdy
- 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



ESC/Java

- Joe Kiniry et al.
- At first single prover (Simplify)
- Now beginning to be multi-prover (PVS, Coq...)
  $\longrightarrow$ the work of Clément Hurlin and Carl Pulley
  and wishing to be fully integrated in eclipse !



2. Interface

1. Interacting with the user
2. Interacting with the prover

Interacting with the user (1)

ESC/Java has 2 interfaces :

- A command line interface with many switches
- An eclipse interface which basically add some gui over the command line version, and permits an easier way to see the warnings of ESC/Java

POs are generated for each method.

Interacting with the user (2)

Jack is much more 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 (1)

ESC/Java interacts with provers through some prover extensions.

ESC/Java has two kind of interaction with provers :

- Simplify-like :
  each goal is tried with the prover and ESC/Java awaits a respond to continue verifying the program.
- Interactive, using the new vcGen, and recently developped interfaces $\longrightarrow$ no interaction with the prover at all, it only generates some files containing the lemmas to prove

Interacting with the prover (2)

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. Modelling in Coq

1. Presentation
2. Memory model
   (a) Types
   (b) Memory model
   (c) Pure methods
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 ESC/Java and Jack.

General architecture

3 parts are generated :

- The prelude containing the logic used
  - ESC/Java : 1 file 'static'
  - Jack : 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 for Jack and ESC/Java

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 :

- In Jack :

  Inductive Types : Set :=

  | class : Classes → Types

  | array : Types → Z → Types.

- In ESC/Java there is a separation between types for class and for arrays

  → There are some special axioms for the types of arrays

Subtyping (1)

- Jack : 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)

- ESC/Java : in the proof obligation

For Jack and for ESC/Java 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 $\rightarrow$ forall c, subtypes c b $\rightarrow$ subtypes c a.

Memory model (1)

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.

Memory model (2)

In ESC/Java fields access/modifications and array access are done through a sort of heap (more uniform than Jack) :

Variable select : S → S → A.
Variable store : S → S → A → S.

Axiom select_store1 :
    forall (var obj :S)(val :A), (select (store obj var val) var) = val.

Variable arrselect : S → Z → A.
Variable arrstore : S → Z → A → S.

Problems :

- the array select/store should be different (not the case on the proof tree)
- select/store and their axioms have to be parametized for each variable type

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

- ESC/Java does tries to do differenciation (recent work...)

  $\longrightarrow$ Not finished yet, but when the new vcGen is finished it should be corrected

  $\longrightarrow$ the Coq translator must deduce from the context if an 'and' should be boolean or between two Prop

The pure methods

Pure methods are some methods you can use in your specification in JML.

- No 'visible' side effect
- A constructor can be pure
- We'd like them to be deterministic

The main problem : keeping track of the pure method name

Jack approach

- At first pure methods were directly unfolded within the proof obligations
- Now there is a couple definition / hypothesis

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

Definition myfun_2 : A1Type → A2Type → ResType → Prop :=
fun (h arg1 arg2 Res) => Context2 → Res = false.
...
(* Used within an hypothesis : *)
...  → myfun_2 this l_var1 Res_23 → Res_23 = true.
```

ESC/Java approach

- False heap with a variable / hypothesis

Variable myfun : Heap → A1Type → A2Type → ResType.

...

Variable state_1 : Heap.

Variable state_2 : Heap.

...

(* Used within an hypothesis : *)

... → myfun state_1 this l_var1 = true.

(* Used within another hypothesis : *)

... → myfun state_2 this l_var1 = false.

What we would like to have

- The same kind as the new one in Jack but using a heap.
  $\longrightarrow$ The first step in order to have within JML functions that can be directly defined in the theorem prover language in order to gain expressiveness

Definition myfun : Heap $\rightarrow$ A1Type $\rightarrow$ A2Type $\rightarrow$ ResType $\rightarrow$ Prop :=
fun (h arg1 arg2 Res) => (Context1 h $\rightarrow$ Res = true) $\wedge$
(Context2 h $\rightarrow$ Res = false).

- Problem : it would totally change the memory model in Jack and partly change it in ESC/Java.

POs generated by Jack and ESC/Java

- ESC/Java, refutation based :

  pre $\longrightarrow$ (not (npost))

  where npost is the weakest precondition of the negation of the postcondition. False has to be proved :

  - good for first order provers (Simplify)
  - not so for interactive proof assistants (our case)

- 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)

POs presentation

Making more readable the POs generated :

- ESC/Java :
  - before the translation to Coq the tree is cleaned from useless informations (buggy)
  - problem : the form of the lemma
- Jack :
  - once translated it is passed to Coq to pretty print
  - separation between the different kind of hypothesis
- Cleaning with tactics

Solving proofs

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

- Most of ESC/Java POs can be solved using the `firstorder` tactic due to its Simplify-Theorem-Prover history.
- For Jack : 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

- ESC/Java :

  The starting tactic + `intuition` or `firstorder` solve nearly all the one Simplify manages to solve, but testing on bigger examples is needed

  If we turn off proof simplification we get easily POs of more than 1Mo of text (in Coq)

- Jack :

  Automation in Coq 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

4. CoqEditor

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 for Coq

CoqEditor

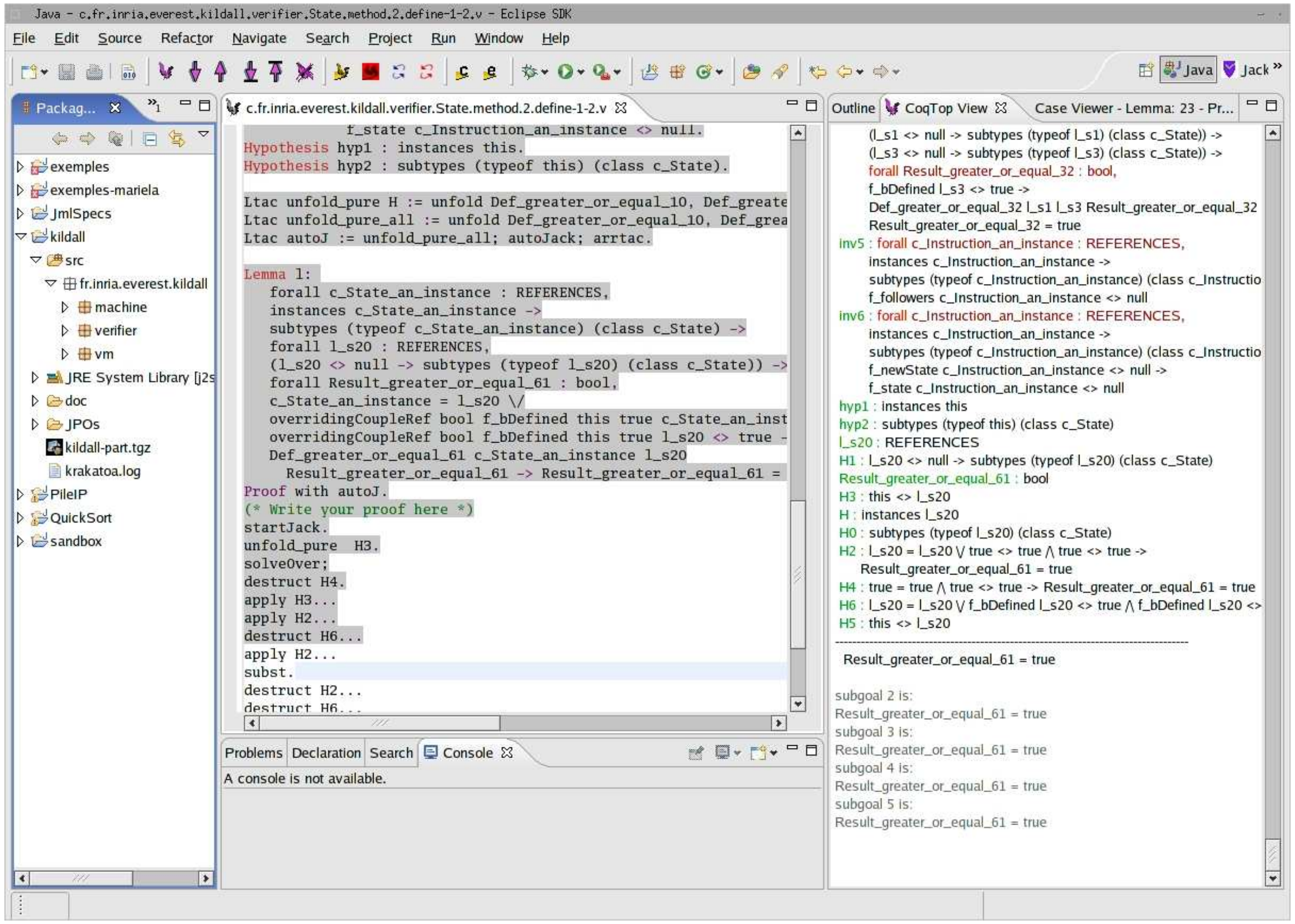
It has :

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

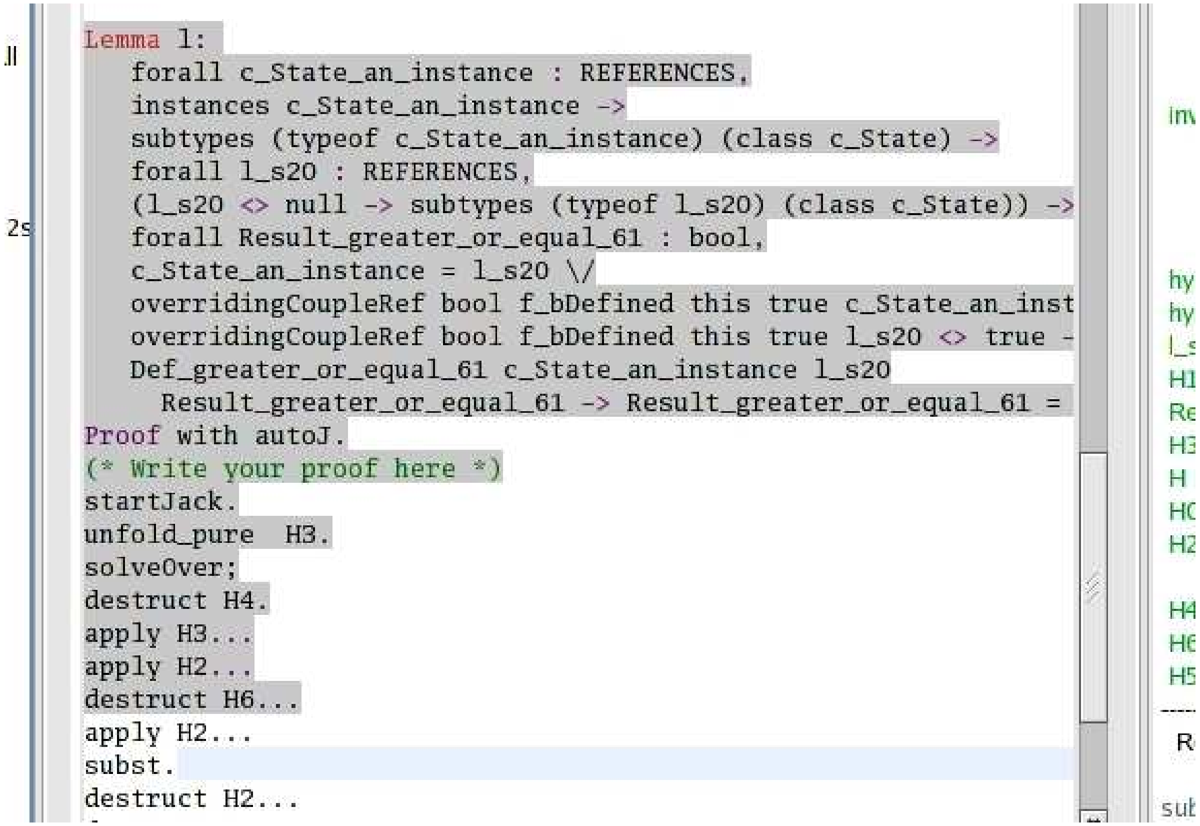
Gain :

- 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)

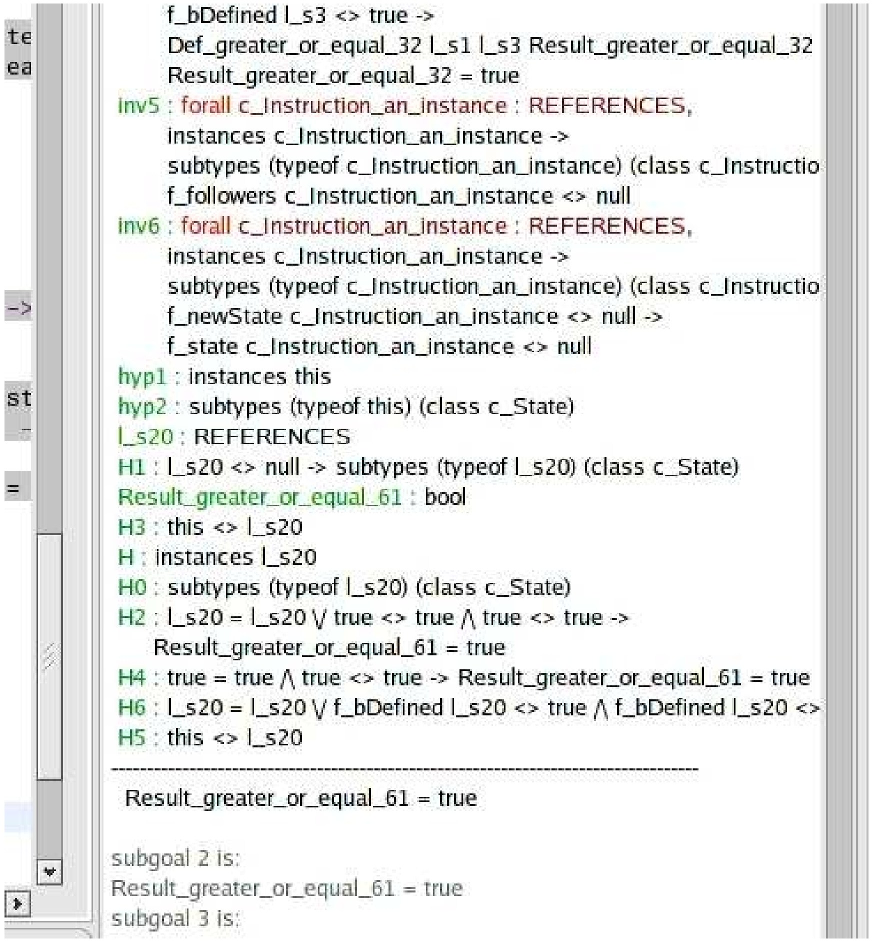
Some Screenshots (1)



Some Screenshots (2)



Some Screenshots (3)



Conclusion

- Jack has a working plugin for Coq, with an editor inside of Eclipse
- ESC/Java2 has now an output to Coq but it needs further testing
- Still some works on :
  - The question on the pure methods (and the memory model)
  - The tactics
  - ESC/Java in Eclipse, with a working and typed vcGen

$\longrightarrow$ Integrate everything in ESC/Java

Download links

Jack :

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

ESC/Java2 :

`http://secure.ucd.ie/products/opensource/ESCJava2/`

CoqEditor (only works with the CVS version of Coq !) :

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