Java Applet Correctness Kit: a tutorial

L. Burdy

INRIA Sophia Antipolis
Lilian.Burdy@sophia.inria.fr

1 Introduction

This document constitutes a tutorial to the JACK tool. JACK is an eclipse plugin allowing to ensure the correctness of Java application annotated with JML. The version of the tools referenced in this document are :

- Jack v1.8.0
- eclipse 3.1

2 Installation

You need to have eclipse installed and to have access to a Jack update site, then you install the plugins using the classical eclipse framework.Independtly you need to install the provers you want to use. The different available provers are:

- B: you can choose to install the Atelier B or the B4Free toolkit.
- Coq: you need to have coq installed and also an interactive proof interface: coqIde, pcog, proofgeneral.
- PVS: you need to have PVS installed and also emacs or xemacs
- Simplify: you need to install the simplify prover by getting the executable on the ESC-Java web site.

3 First step with JACK

3.1 Running Jack

Create a Java project, import the jackTutorial.zip archive file into the project. You will obtain a project with a package jack.tutorial containing different classes.

You can now run Jack on the Java files: select Intro.java and launch Jack:

- using the “c” button or

– in the contextual menu associated to the file, select Jack>Verify Source.

After few seconds, the proof obligations for the file Intro.java are generated: you can now view the lemmas by:

– using the "e" button or
– in the contextual menu associated to the file, select Jack>Edit.

This will open the Jack perspective and edit the proof obligations into the different views. The presentation of this perpespective is detailled latter.

3.2 Preferences

Before continuing using Jack, lets look at its configuration parameters. Configuration parameters exist at two levels: global preferences and project properties

Open the menu, open Window>Preferences>Jack>compiler

Here you can set different preferences:

– subdirectory for Jack files (default value is JPOs): Jack generates different files containing the generated proof obligations and their translation into the different provers input languages. Normally, you do not have to change this value.
– generating obvious lemmas (default value is false): Jack contains a minimal prover that allows to eliminate really obvious lemmas at generation time. You may change this value if you want to see all the proof obligation generated by the tool but on some "big" file you may emcountered some memory overflow.
– generating well definedness lemmas (default value is false): This optin has not been really tested for the moment, so you may not change the value.
– generate ... output file (default values are true): you can choose to generate or not a translation of the lemmas into the different provers plugin language that are installed. If you do not project to use one of the prover, you nay set its preference to false, this will increase the proof obligation generation time.

Open the menu, open Window>Preferences>Jack>lemma viewer

Here you can set different preferences:

– the font of the source viewer in the Jack perspective

– showing ... view (default is true): the lemmas are displayed in different views associated to each prover plugin. You may choose to display or not this view.
– JML colors on the source viewer of the Jack perspective

Open the menu, open Window>Preferences>Jack>provers

Here you can set different preferences:

– the number of parallel proof task (default value is 3): you can launch in parallel many automatic proof task, Jack will run them sequentially limitating the number of parallel tasks: you may change the value depending on your computer power.
– use ... obvious prover (default is false): some prover plugin implement an obvious prover (forthe moment only simplify do it). This prover is called after the lemma generation and before the translation into provers language. It is recommended to use an obvious prover since it will reduce the size of the generated files and the time to create them. Nevertheless, on "big" file, the time for proving is not transparent to user.

Preferences can also be set for the different prover plugins (see this section)

3.3 ProjectProperties

Other configuration parameters are associated to the project.

In the package explorer, select your project and in the contextual menu: Properties>Jack.

Here you can

– subdirectory for Jack files (default value is the global preference value)
– set a JML path (default value is the Java class path): Jack needs to load and link the different classes used by the project. If you do not need to use annotated API, you can select to use the default class path but if you have to use a particular API, you need to specify them in the JML path.
– generate an serialized image (default value is false): if you do not have to modify an annotated API you use in youtr project, it is really usefull to collect all the stable classes, to load and link them and to store them in a serialized image: this will increase drastically the proof obligation generation time.

- Jml clauses default values : you can set the default value for JML clauses (those values will be used for all methods that are not annotated):
  - requires : true or false
  - ensures : true or false
  - modifies/assignable : \everything or \nothing
  - exsures/signals : all exceptions, all runtime exceptions, some runtine exceptions, ...

Select \nothing for the modifies clause. Usually it is simpler to set by default that a method do modify nothing and then to add the modifies clause in your current application considering that all the APIs do not modify your variables.

3.4 Obvious provers

You can now begin to validate your application.

select Intro.java and compile it with the different obvious provers options

You will obtain:

- 185 lemmas when obvious lemmas are generated and simplify is not used as obvious prover
- 21 lemmas when obvious lemmas are not generated and simplify is not used as obvious prover
- 10 lemmas when simplify is used as obvious prover

3.5 Lemma viewers

Edit now Intro.java and lets look at the Jack perspective: this perspective contains 3 views:

- the case explorer view: this view displays a tree that allows to navigate into the lemmas generated for the file. Lemmas are displayed in a tree containing as node :
  - a root node for each class of the file
  - for each class, a node for these methods and constructors and a node for its static initialization
  - for each methods, ... a node by control flow path (called case)
  - for each case, e leef by goal.

The tree can be filtered to show only some lemmas : proved/unproved, poor proved classes or methods, lemmas proving some particuliar aspect : invariant, ensures clause, ...
The cases can be displayed with a flat or a hierarchical layout.
Selecting a case or a goal, highlight its corresponding piece of source in the source viewer.

- the source view: this view displays the edited file. When a lemma is selected, coloration are added to the source in manner to help the developper to understand whhat is to be proven:
  - green code corresponds to code that is in the current control flow
  - green annotation corresponds to annotation to prove
  - blue code corresponds to code that is in the current control flow, moreover some tooltip text is associated to it
  - red code corresponds to point where an exception has been considered to be launched.
- the lemma view displays the lemma in different languages: JPOL and all the other prover plugins language. It contains two parts: hypothesis and goal. Hypothesis are prefexed with a letter depending on its origin:
  - I : invariant
  - L: local hypothesis of the method
  - R: requires clause of the method
  - A: assertions
  - Li : loop invariant
  - E: ensures clause of a called method
  - X: exsures clause of a called method

In the lemma viewer, select runJack(), it has one case with one goal: select the goal: its a goal due to a wrong modifies clause, correct the method specification and re-run the tool.

The exercice consists now in correcting the annotations in order to obtain a file with only one unproved proof obligations due to the fact that one need to annotate the Java API if one want to prove it.

4 Annotation generation

5 Proving with interactive provers

6 Checking security property

7 WP at bytecode level