Final Year Project Report

A PVS Eclipse Plug-in

Robert Finlay

A thesis submitted in part fulfilment of the degree of

BA/BSc (hons) in Computer Science

Supervisor: Dr. Joe Kiniry

Moderator: Dr. Damian Dalton

UCD School of Computer Science and Informatics
College of Engineering Mathematical and Physical Sciences
University College Dublin

March 23, 2006

Table of Contents

Abstract

PVS is a formal specification language and mathematical theorem prover for higher-order logics. PVS is one of the three most popular theorem provers in the world. The aim of this project is to develop a PVS plug-in for the Eclipse environment.

Acknowledgments

Acknowledgement is given to Mr. Julian Charles at INRIA, Paris, France for providing the Coq Plug-in. This Plug-in formed the basis for the project and proved to be a valuable tool.

Dr.Joe Kiniry of the School of Computer Science and Informatics at UCD for giving guidance and feedback for the duration of the project.

Chapter 1: **Introduction**

PVS is a prototype environment for developing rigorous mathematical formalisations[6]. PVS is one of the three most popular theorem provers in the world. PVS is the product of over 20 years of research and development. The aim of this project is to integrate a PVS plug-in into the Eclipse environment.

Eclipse is an Integrated Development Environment for developing software applications. This report outlines the various technologies researched for the duration of the project such as PVS, Eclipse, Coq, Isabelle and Proof General. The project goals are laid out with detail given on how these aims were designed and implemented.

1.1 Project Specification

Project 61

A PVS Eclipse Plug-in

Supervisor Dr Joseph Kiniry Subject Area Software Engineering
Pre-requisite Strong knowledge of Java and basic software engineering principles
Co-requisite Eclipse use and plug-in development
Subject Coverage Integrated Development Environments, Higher-order Theorem Provers
Project Type Design & Implementation
Software Java, Common Lisp, PVS
Hardware Any machine running x86/Linux or Mac OS X

Description
PVS is a higher-order theorem prover that uses Emacs as a front-end, much like nearly all other higher-order theorem provers.

Many other provers like Isabelle, HOL, and Coq used the same Emacs-based front-end, called Proof General. Proof General is somewhat inappropriate for PVS, as the former's interactive style is semi-incompatible with the latter.

PVS's front-end, on the other hand, is highly tuned for the kinds of specifications and proofs accomplished in PVS. Additionally, that interface has been significantly enriched by other projects mentioned on this web site.

A new Eclipse-based Proof General is now in development. The purpose of this project is to evaluate this new interface and the general possibility of using an Eclipse plug-in framework for the development of a new PVS front-end.

Mandatory

1. Show expertise with the Eclipse Platform, particularly the plug-ins architecture. 2. Specify a JML-annotated Java interface for the I/O interface and proof interface of PVS. 3. Design, implement, and test a PVS plug-in for the Eclipse Platform supporting PVS specification authoring and proof interaction. 4. Release the plug-in to the world under an appropriate FLOSS license.

Discretionary

1. Support plug-in indefinitely and help kick start a community focused on the evolution and maintenance of the plug-in.

Chapter 2: **Background Reading**

2.1 Introduction

For this project, extensive background reading was necessary to understand the project objectives. This chapter describes the research done on the topics involved in this projects. The topics covered are Eclipse[1], Plug-ins[15], PVS[2], Coq[10], Isabelle[12], Proof General[13]. Eclipse, Plug-ins and PVS represent mandatory inclusions in the project. The latter products represent close links to the above mentioned. Research into all these software products was necessary to postulate possible methods to implement the specifications for the project.

2.2 Eclipse

Eclipse is an Integrated Development Environment(IDE)[1]. Eclipse is a platform independent software framework available under the open source license. Eclipse runs on many major platforms including Windows, Mac OS X and Linux. Eclipse is very extensible and supports a wide range of plug-ins including the popular Java development tools. Eclipse was begun by IBM in the early 90's. Eclipse is now managed by the Eclipse Foundation, an independent non-profit organisation of software industry vendors. Many notable software tool vendors have embraced Eclipse as a future framework for their IDEs[14].

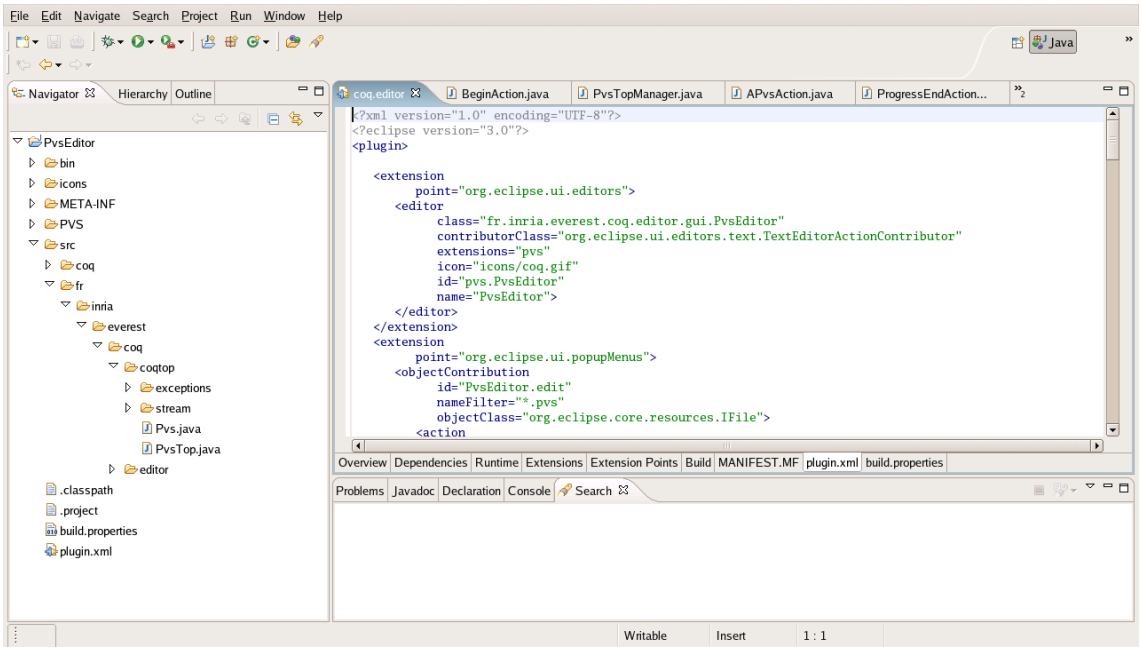

Figure 2.1: The Eclipse IDE

2.2.1 Eclipse Architecture

Eclipse has an extensible design that supports many programming languages and plug-ins. Eclipse was not created with any particular language in mind although Java has been the main focus since its inception. Java was the language of choice for many of the thousands of developers responsible for Eclipse. The development of Eclipse has been collaborative, open and transparent. This development process has allowed Eclipse to be an environment of creativity, originality, innovation and freedom[1]. The Eclipse IDE encourages users to develop projects of varying sizes and complexities in a variety of languages.

Eclipse supports the integration of plug-ins. Eclipse employs plug-ins in order to provide all of its functionality on top of its platform. Plug-ins permit the IDE to interact with programmes outside of Eclipse. For example, the CVS version control plug-in that allows a developer to use CVS from within the IDE. Plug-in integration was a principle reason for choosing Eclipse as the environment in which to develop the project.

2.2.2 What is a Plug-in

In order to develop a plug-in in the Eclipse IDE, it was first necessary to understand what is meant by a plug-in. It was also needed to understand why the creation of such a plug-in will be helpful. A plug-in is a single independent program which must interact with other programmes linked to it in order to achieve a specific functionality. Plug-ins have a well defined boundary of possible sets of actions. Interaction is usually relied upon through their interface. Plug-ins enhance the way users view and interact with programs. This is also applicable in terms of the project aims. It would therefore be of great benefit for developers to have a mathematical theorem prover integrated into Eclipse.

Such a plug-in has the ability to provide added functionality to an existing theorem prover and present it in an appealing manner. For example, a plug-in includes graphical buttons on a menu bar that when pressed perform the commands most commonly used in the theorem prover. One such theorem proving system is PVS.

2.3 PVS

PVS is an abbreviation for Prototype Verification System. It is a theorem prover which also comprises of a specification language and support tools such as utilities and documentation. PVS was developed by SRI International[3]. The current version is PVS 3.1. The first version was released to users in 1993 and the most recent version was released in 2003. PVS is free to download under the open source license from the PVS website[2]. PVS is constantly evolving with newer versions being released all the time. This integrated theorem prover provides a mechanised method of aiding in the development, analysis and verification of interactive proofs. It has had various applications since its creation.

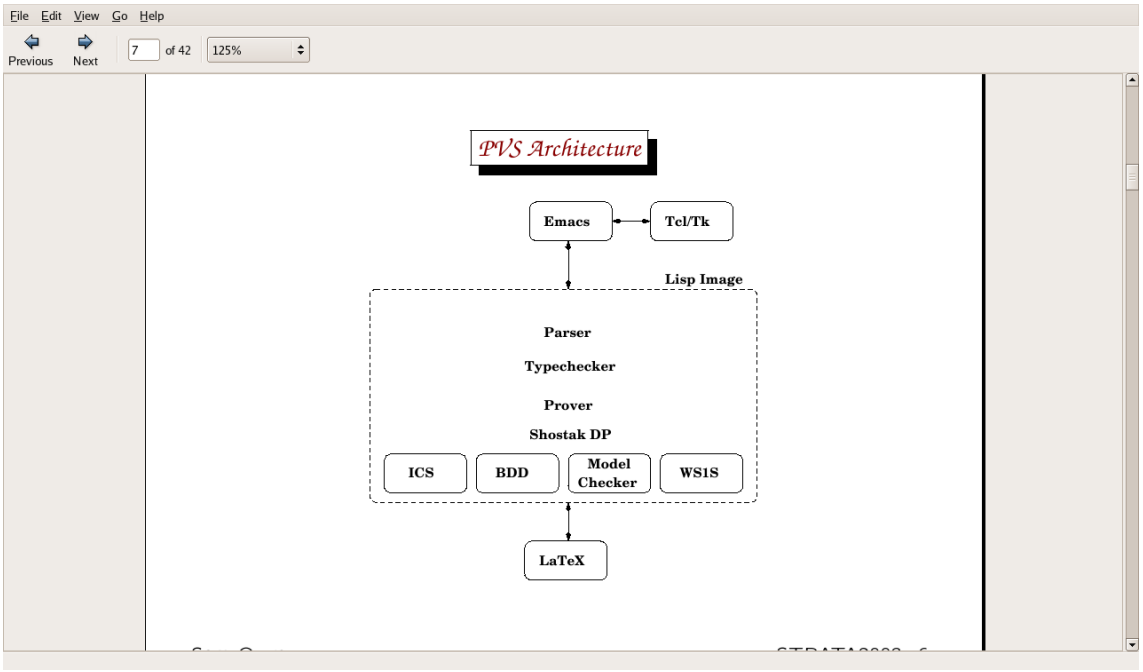

Figure 2.2: PVS Architecture

2.3.1 Applications of PVS

PVS is predominantly used for analysing and producing solutions for difficult and detailed problems. Such problems that PVS has been applied to include the axiomatization of a Unity-like system for reasoning about real-time computations and specification of the IEEE-854 Floating Point standard (by Paul Miner)[16]. Additionally, portions of algorithms used in a space shuttle flight control system have been specified via a number of collaborations with N.A.S.A. and other aerospace companies. Microprocessors have also been designed for aircraft flight control systems using PVS.

2.3.2 PVS Specification Language

The specification language of PVS is based on a simply typed higher-order logic[4]. A higher-order logic is suitable compared to other logics such as first order logics. The is because higher-order logics are beneficial for expressing the language whilst also enforcing strict typing. A specification language is a formal language. A specification language differs from most programming languages because it specifies program design requirements.

Each specification is a set of parameterised theories. Each theory consists of types and formulae. There are simple base types such as booleans, integers, etc. and constructor types such as sets, tuples, records and functions. Constructor types support the definition of types within theories through the implementation of constructor types on base types. Constrained types may be imposed by dependant types on other types.

For example, a simple theory TH1 is defined below[5]:

```
TH1 : THEORY
  BEGIN
   n: VAR nat
   m = n*(n + 2)/4
  END TH1
```

This example shows the definition of a theory involving a basic arithmetic operation.

The main part of a PVS specification is a declaration. Declarations are used to introduce variables and identifiers. A declaration usually consists of an identifier, any optional bindings and a body[4]. An identifier names a language entity such as a variable or a type. Declarations also contain bodies which define the declaration. Declarations can be recycled and reused in different theories. PVS expressions provide the usual arithmetic and logical operators, function application, lambda abstraction, and quantifiers, within a natural syntax[8]. The manipulation of these operations will give permission for more detailed and difficult theories to be formed.

For example the theory sum below[5]:

```
sum: THEORY
BEGIN
n: VAR nat
f, g: VAR [nat -> nat]
sum(f,n): RECURSIVE nat =
(IF n = 0
    THEN 0
    ELSE f(n-1) + sum(f, n - 1)
ENDIF)
MEASURE n

sum_plus: LEMMA
sum((lambda n: f(n) + g(n)) , n)
= sum(f,n) + sum(g,n)
square(n): nat = n*n
```

```
sum_of_squares: LEMMA
 6 * sum(square, n + 1) = n * (n +1) * (2*n + 1)

END sum
```

This example demonstrates the definition of a sum theory. This theory is then reused in the proof in order to define a square theory. The square and sum theories are then adapted in order to define a sum of squares theory.

An extensive library of built-in theories provide hundreds of definitions and lemmas. A lemma is a logical proposition which is used as a stepping stone to a larger proof[18]. Lemmas must be proved to be true in order to prove the overall theory. Proof obligations are forced on constrained types which greatly increase the expressiveness of specifications. These obligations enforce the conservative extension of proofs. Proof obligations are handled automatically by the theorem prover[4].

2.3.3 PVS Theorem Prover

The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework[4]. The primitive inferences include propositional and quantified rules, induction, rewriting, and decision procedures for linear arithmetic.

Inferences are optimised for larger proof sets. Users can define procedures which have the ability to combine primitive inferences to produce high-level proof strategies. Proofs produce scripts that can be edited and joined with additional formulae, and then rerun. This encourages many similar theorems to be proved efficiently and also permits economical adjustment of proofs to follow changes in requirements or design. The development of readable proofs is also promoted. PVS theorems are proved interactively using Emacs.

2.3.4 GNU Emacs

GNU Emacs currently provides an interface for creating and editing PVS scripts. Interaction is command line based. Buffers separate the user's proofs from type checked proofs returned by PVS. Emacs is a popular interface for many other theorem provers.

An example of PVS running in Emacs is illustrated in the diagram below. There also exists an Emacs based extension which specialises in interacting with theorem provers. This interface is called Proof General.

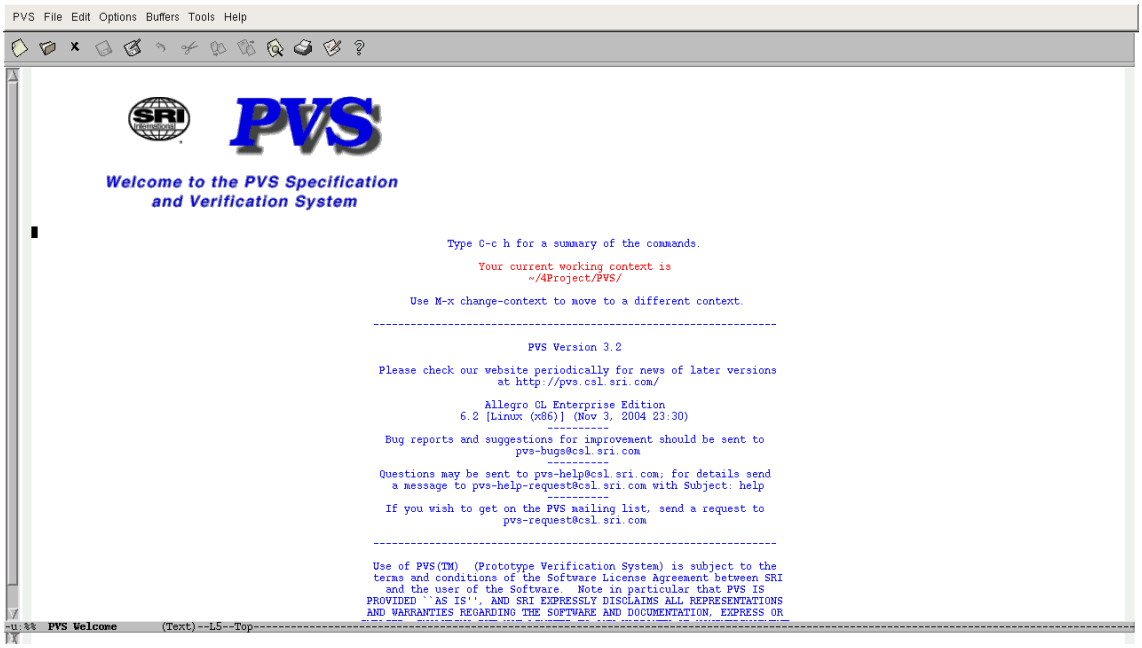

Figure 2.3: PVS in Emacs

2.4 Proof General

Proof General was created a specialised interface for theorem provers[12]. Proof General runs on top of GNU Emacs. The development environment is presented generically for all theorem provers. Proof General has a flexible windowing environment. This flexibility was designed for the purpose of increasing ease of use to users. Proof General contains many useful tools. Proof General is considered a powerful tool for interacting with theorem provers. Support is provided for debugging, browsing scripts, etc.

Proof General provides a rich graphically advanced interface within GNU Emacs. A Proof General plug-in for the Eclipse environment is currently being developed. This plug-in is called the Proof General Tool Kit[13]. Supported theorem provers include Isabelle and Coq.

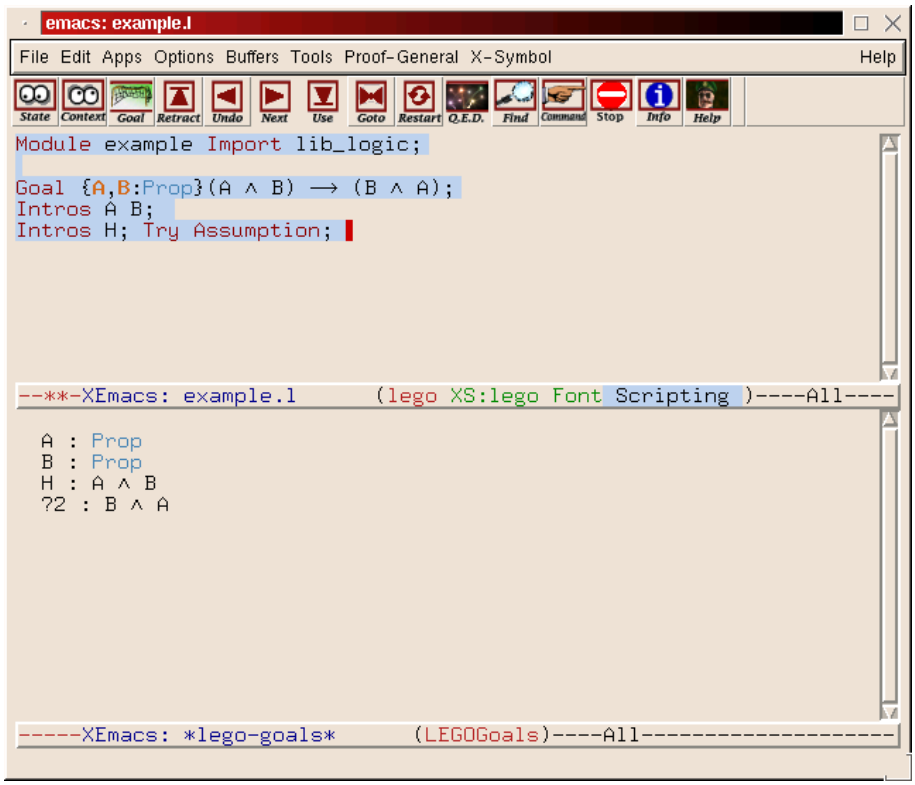

Figure 2.4: Proof General in Emacs

2.5 Isabelle/Coq

Isabelle and Coq are two prominent theorem provers which use Proof General as their front-end. Isabelle is a generic theorem proving environment. Developed at Cambridge and Munich it is a popular proof assistant based on higher-order unification[13].

Coq is a formal proof management system based on the Calculus of Inductive Constructions[10]. Coq was developed at INRIA in France. Both theorem provers are available under the open source license. Both theorem provers use GNU Emacs as a front-end in the same way that PVS does. Although, Coq and Isabelle have been integrated into Proof General[13] and are being used in Proof General running on top of GNU Emacs instead of the original GNU Emacs interface which PVS continues to use. Upon further research of these three theorem provers it was discovered that PVS and Coq shared many similarities. Coq includes higher-order functions and predicates, inductive and co-inductive data types and predicates, and a stratified hierarchy of sets.

Both the PVS and Coq theorem provers use homogeneous declarative methods for theories(known as theorems in Coq), identifiers, etc. For example, the definition of what exactly qualifies a legal identifier. Both languages state that a legal identifier must be a letter, digit or an underscore. PVS only differs in that is also defines '?' to be a legal character in an identifier. Recently, an Eclipse plug-in was designed and implemented for Coq.

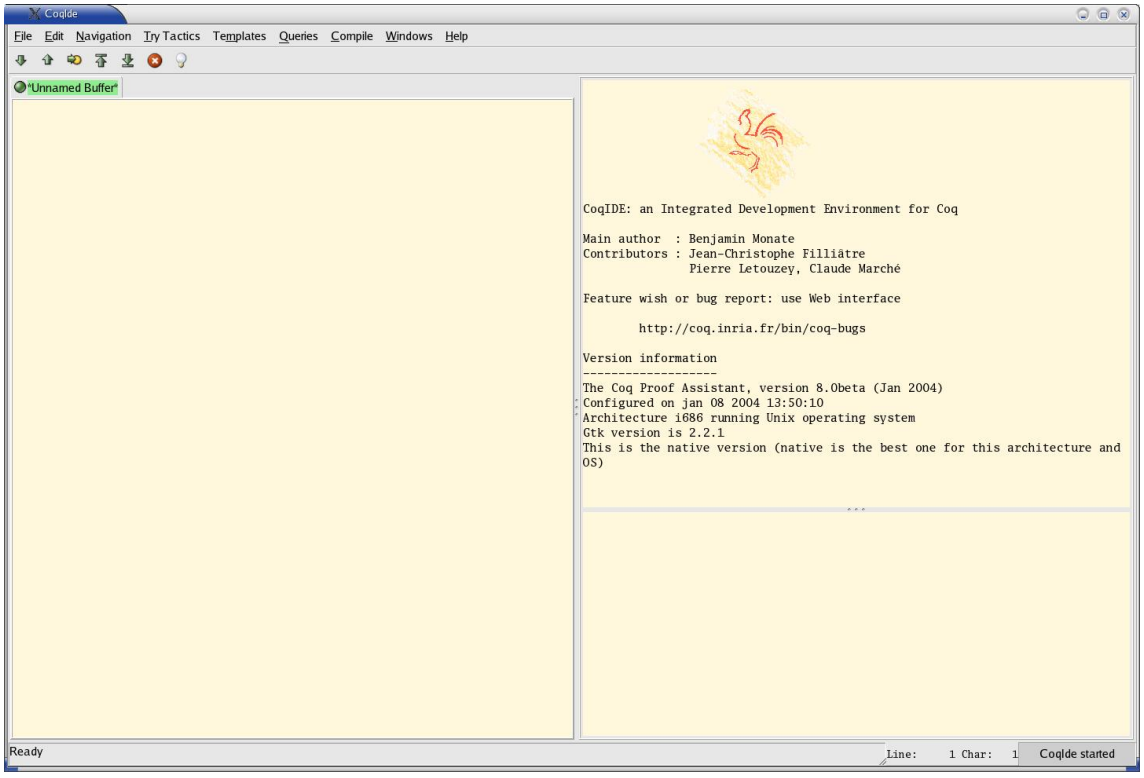

Figure 2.5: Coq in Emacs

2.6 Coq Plug-in

A Coq plug-in for Eclipse was developed by Mr. Julian Charles at INRIA. The Coq plug-in includes a documentation tool known as Coqdoc. Coq also includes its own dependency tools. The Coq plug-in enables full interaction with Coq. The graphical user interface includes a tool bar which contains buttons. Below Coq running as an Eclipse plug-in is illustrated.

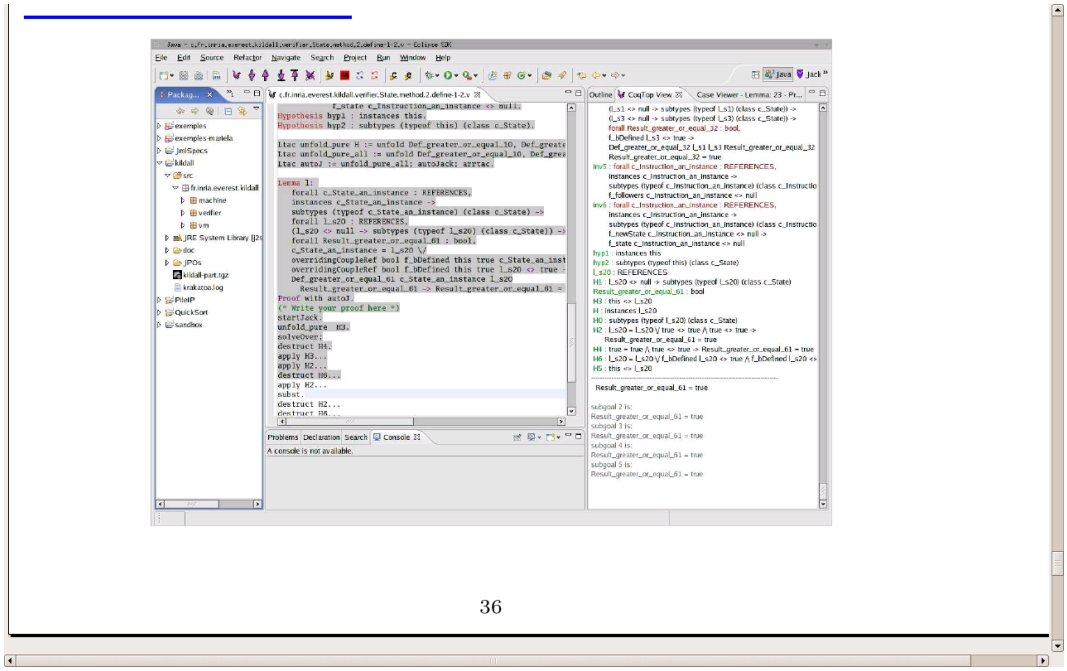

Figure 2.6: Coq Plug-in for Eclipse

When these buttons are pressed various commands such as:

- Reset - will reset all the current proofs.
- Begin Proof - will start a new proof by launching a new application.
- Evaluate Current - Proof checks progress in the current proof to be checked by Coq interactively.
- Evaluate All Proofs - checks progress in the current workspace and evaluates all proofs therein.

2.7 More on Coq

Similarities between Coq and PVS permit the Coq plug-in to be modified to a PVS plug-in. The buttons as described above which function with Coq have the ability to be mapped to buttons which interact with PVS.

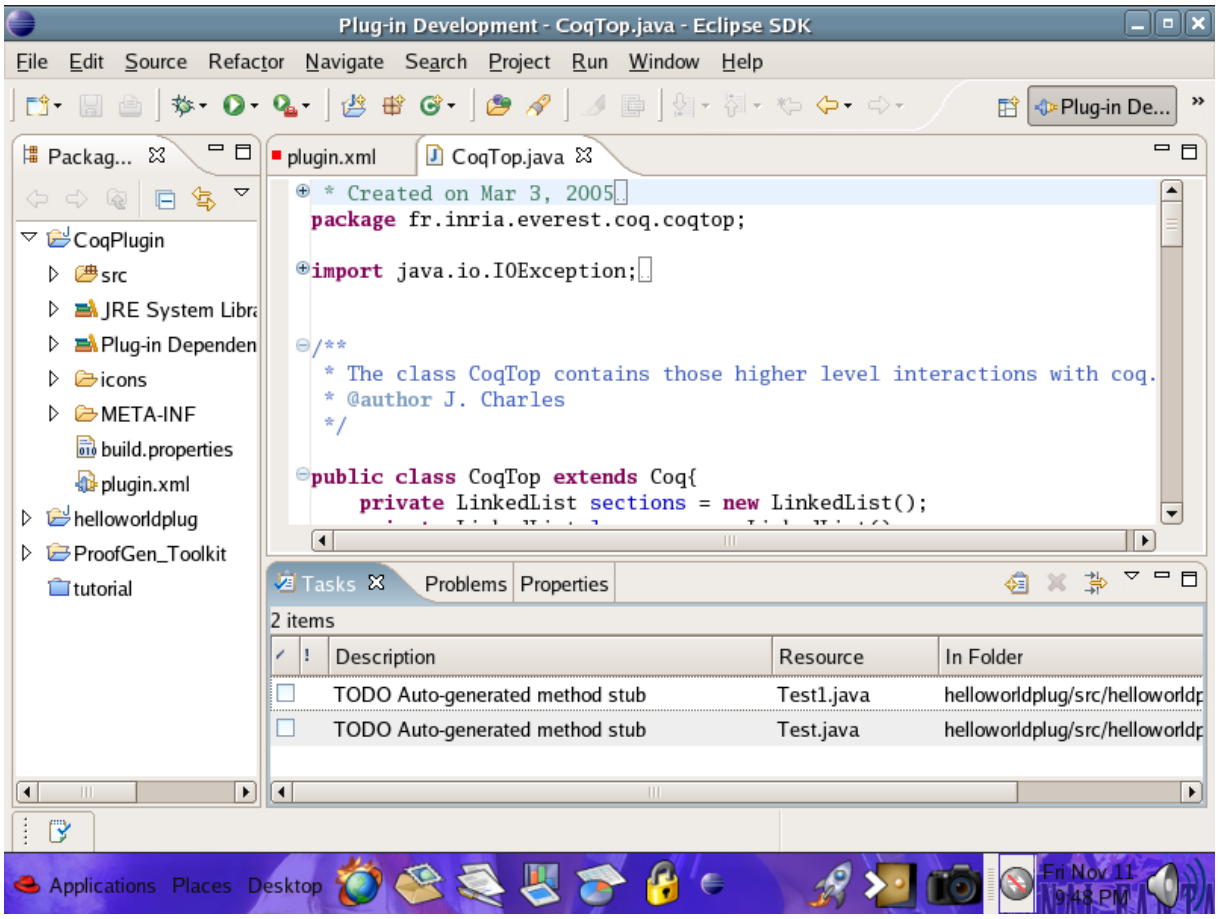

Figure 2.7: Coq in Eclipse

Chapter 3: **Design**

3.1 Introduction

The project specification provided for Eclipse to be the IDE in which to create the plug-in. However options still remained available as to how the plug-in was to be designed.

- To create a plug-in using the Proof-General Toolkit[13]
- Adapt the style and functionality provided by the Coq plug-in.

Both these options were evaluated for potential merits. It was decided not to use the Proof-General Toolkit due to the extreme difficulty in using the tool. The Coq plug-in seemed a more obvious choice for the following reasons.

- Coq and PVS are similar theorem provers.
- The existing plug-in had functionality desirable for the PVS plug-in.
- PVS plug-in would benefit from existing pre-defined classes.

3.2 Design Aims

The Coq plug-in supports the Coq language. This provides for an efficient way to edit Coq proof scripts and interact with Coq proofs. It was necessary to decide how a PVS plug-in would be implemented using the Coq plug-in as a foundation. Design aims needed to be outlined. The key design aims were identified and are listed below:

- Derive a PVS plug-in from existing Coq plug-in
- Support specialised identification for reserved words,symbols, etc.
- Support key PVS commands such as type-checking, changing context and start-proof.
- Make plug-in available all under an Open Source license.

Once design aims had been realised it was then necessary to implement them. The next section of the report will detail how this was carried out.

Chapter 4: **Implementation**

4.1 Introduction

To implement these design aims, it was first necessary to understand the Coq plug-in. This was extremely important as the Coq plug-in formed the basis for the PVS plug-in. Knowledge about the existing plug-in was attained through testing the application and studying each class and method. This helped to identify the features which could be re-used and/or adapted in order to produce the new plug-in. Methods, Variables and all Coq references were changed to reflect those of PVS. Key Coq functions and methods were identified. Parallel coding practises were adopted in order to provide equal levels of functionality that exist in Coq for PVS. For example, the feature which caused reserved words to be highlighted.

4.1.1 Reserved Words

Reserved words are identified to the user through coloured highlighting. Special highlighting was quickly identified as an important feature. It was felt important that users should be able to easily distinguish reserved words such as THEORY from a theory name theory. This was achieved by customising the existing IColorConstants class. This class recognized the definition of specific colours which were then applied to words.

In the PvsRuleScanner, Coq reserved words were reused and adapted to those in the PVS language. These reserved words were defined as an array of strings. Colours are applied e.g. THEORY colour is yellow, by looping through all reserved words belonging to THEORY and applying yellow to those words belonging to THEORY. Reserved words then appeared in the editor in this colour. The example details this, code adapted from the Coq plug-in[11]:

```
import org.eclipse.swt.graphics.Color;
import org.eclipse.swt.graphics.RGB;
import org.eclipse.swt.widgets.Display;

public interface IColorConstants {

// Adds colours for specific words.
public final static Color DEFAULT_TAG_COLOR = new Color(Display.getCurrent(),
new RGB(0,0,0));
public final static Color TAG_COLOR =
new Color(Display.getCurrent(), new RGB(100, 0, 100));
public final static Color STRING_COLOR =
new Color(Display.getCurrent(), new RGB(0, 0, 200));
public final static Color COMMENT_COLOR =
new Color(Display.getCurrent(), new RGB(0, 155, 0));
//Highlights a reserved word
public final static Color RESERVED_COLOR =
new Color(Display.getCurrent(), new RGB(155, 0, 0));
```

```
//Highlights a Theory
public final static Color THEORY_COLOR =
new Color(Display.getCurrent(), new RGB(155,155,0));
//Highlights a formula
public final static Color FORMULA_COLOR =
new Color(Display.getCurrent(), new RGB(255,165,0));

// Some background colors
public final static Color HILIT_COLOR =
new Color(Display.getCurrent(), new RGB(200, 200, 200));
public final static Color NORMAL_COLOR =
new Color(Display.getCurrent(), new RGB(255, 255, 255));

}



public class PvsRuleScanner extends RuleBasedScanner implements IColorConstants {

private int limit = 0;


public PvsRuleScanner() {
String [] reservedWords = { "AND", "ANDTHEN", "ARRAY", "ASSUMING", "AUTO-REWRITE","AUTO-R
"ELSE", "ELSEIF","ENDASSUMING", "ENDCASES", "ENDCOND", "ENDIF", "ENDTABLE", "EXISTS", "EX

};


/* A list of all the reserved symbols for the
 * refer to  page 13 of the PVS Documentation
 * Will be higlighted with a colour for symbols defined in colour class
 */


String [] specialSymbols = {  ")", "#]", "%" ,"&" ,"&&", "(", "(#", "(:", "(|",
"(||)" ,")", "*","**","+","++", "-","->" ,"//",
"/=", "/\\", "+",
" :" , ":)", "::", ":=",";" ,"<>", "==" ,"=>",
">","@","[","[]", "[|", "[||]",
"\\/", "^", "^^", "'", "{",
 "{|", "{||}", "|", "|)", "|->", "|=", "|>", "|[","|]","||","|}", "}", "~",
};

String [] theories = { "THEORY","BEGIN", "END",

};

IToken tag = new Token(new PvsTextAttribute(TAG_COLOR));
IToken comment = new Token(new PvsTextAttribute(COMMENT_COLOR));
IToken reserved = new Token(new PvsTextAttribute(RESERVED_COLOR));
IToken string = new Token(new PvsTextAttribute(STRING_COLOR));
IToken theory = new Token(new PvsTextAttribute(THEORY_COLOR));
IToken formula = new Token(new PvsTextAttribute(FORMULA_COLOR));
```

```
IToken def = new Token(new PvsTextAttribute(DEFAULT_TAG_COLOR));

WordRule wr = new WordRule(new PvsWordDetector(), def);

for (int i = 0; i < reservedWords.length; i++) {
wr.addWord(reservedWords[i], reserved);
}


for (int l =0; l< specialSymbols.length; l++){
wr.addWord(specialSymbols[l],string);
}

for (int t =0; t< theories.length; t++){
wr.addWord(theories[t],theory);
}

WordRule wexpr = new WordRule(new PvsExprDetector(), tag);
wexpr.addWord(".", tag);
IRule [] rules = {
```

4.1.2 Commands

Commands were felt to be an integral part of interacting with PVS. It was decided to adapt the Coq buttons to PVS buttons. This would also help reduce the level of difficulty in interacting with the theorem prover. Instead of having a command line based interaction like in Emacs users benefit from a helpful tool bar of clickable buttons. The main PVS were chosen and are outlined below:

- Type Checking - Saves all buffers, sends the current proof to PVS for type-checking and waits for a reply.
- Change of Context - Opens a directory forcing the workspace to be changed.
- Start-Proof - Stops all current proofs and starts a new proof.
- Reset - Resets all the proofs in the current workspace.

For example, the PVS command for type checking proofs.

4.1.3 Type Check

Type checking is the command which invokes PVS to check and prove the legitimacy the proof. This command was similar to the 'Evaluate All Commands' in Coq. The type checking command was adapted from it to reflect the requirements of PVS. When interacting with PVS through Emacs the command line argument for type checking is 'type-check PVS'. The command is still applicable when interacting through Eclipse. When the button is pressed this command is sent to PVS. Along with this command, all proofs within the current workstation are sent to PVS for type checking.

4.1.4 Change of Context

Change of Context is equivalent to the changing of a directory in Emacs. In Eclipse a window will pop up asking the user to choose a directory. Changing directory will license users to switch between projects, known in Eclipse as workspaces.The command is 'change context'. This command is adapted from the begin command in Coq. Once the command has cause a new application to launch i.e. provide a new workspace, a directory can be chosen to start the new proof project in.

4.1.5 Start Proof

Start Proof will halt all proofs in the current workspace and launch a new proof. This is also adapted from the begin command. The start proof will also invoke the stop command and call the begin command to start a new proof.

4.1.6 Reset

This will reset all proofs in the current workspace. The workspace is reset to the initial state when launched. The command does not vary greatly from that of Coq.

Providing this functionality for PVS on top of Eclipse did however provide a challenge. Part of this challenge included the problems encountered for the duration of the project.

Chapter 5: **Problems Encountered**

Problems were encountered during the course of implementing this project. They are described below.

5.1 Eclipse Plug-ins

Eclipse is a large environment. As such, plug-ins are very verbose and as such writing plug-ins is very verbose. This posed a significant challenge. Eclipse plug-ins are very detailed. Carrying out simple tasks proved extremely difficult. Many operations were needed to be carried out to execute even the simplest of tasks. For example, the text highlighting above.

Several classes all linked to each other needed to be implemented in order for the highlighting to be executed. The class IColorConstants needed to be declared. This was then implemented in the PvsRuleScanner class. PvsRuleScanner also extended another class RuleBasedScanner a class of Eclipse.

Eclipse is very expansive and a great familiarity is required to design plug-ins within it. A large amount of time is required to grow completely comfortable with its programming style. The amount of work required to implement such things as highlighted text is very high and can create errors in many different classes.

5.1.1 Runtime Errors

Run-time errors encountered were also difficult to debug due to the obscurity of error messages. Error messages pointed to a number of classes many of which have references in several other classes. Any one of which could have caused the initial error to occur. This was an unforeseen problem.

For example, a null-pointer exception can point initially to a specific line-number in a class but it will also reference every other class referencing that class. This can cause a loop effect. Large amounts of time are therefore attributed to debugging potentially simple errors.

The level of difficulty was also increased due to PVS. New unexplored areas proved troublesome and could be difficult to comprehend while adapting the Coq plug-in. For example , defining rules that recognize single or multi-line rules to be apart of the same proof or theory.

5.2 The Coq Plug-in

In order to come to grips with the plug-in it was crucial to understand the designer's programming methodology. This was not immediately visible, largely due to the lack of documentation provided. The lack of documentation for the Coq plug-in quickly proved problematic.

Efforts to debug code were hindered due to the lack of descriptions given for methods. These methods were defined in a manner that did not make their purpose immediately apparent. Alot of descriptions can be generated using the Javadoc tool.

5.2.1 Javadoc

Javadoc is a computer software tool from Sun Microsystems for generating API documentation into HTML format from Java source code. Javadoc is the industry standard for documenting Java classes[19]. Javadoc at times also lacked a proper indication as to what the method, class or variable defined and how it was implemented. Javadoc has the ability to reduce the level of difficulty in understanding another programmer's code.

Java doc usually is added to every method, variable and instance of a class. It will say what chief procedure it carries out and what chief classes it is implemented in. In the case of methods inputs required by the method are listed and types are given with a short description. A large amount of effort was spent trying to decipher code. Identifying those classes which were helpful and those that were not was further hindered.

Chapter 6: **Conclusions**

Many of the main aims were achieved. At present the plug-in is not fully functional but methods have been implemented for the highlighting of reserved words and sending commands to PVS through button presses. The PVS plug-in launches within Eclipse. New projects can be created and scripts can be edited showing full highlighting of reserved words.

The button presses are not functional however reserved word highlighting was accomplished successfully. Methods for all button events have been implemented. The source of the error is a Java runtime error which has proved difficult to debug. In light of this fact, further work is yet required to get full interaction with PVS.

The plug-in is being released under a FLOSS license. A mailing list is being created to kick start a community based on the development of the PVS plug-in. This mailing list will help to support PVS indefinitely. With the great potential of this project alot of future work is immediately recognizable.

Chapter 7: **Future Work**

At present, the PVS plug-in is incomplete. In order to get the system fully functional the errors need to be corrected. Given more time this issue has the feasibility have been resolved. Once this is achieved the plug-in has much potential for expansion.

Various plug-in extensions could have been implemented. For example an outline editor. An outline keeps track of all the proofs and theories which have been defined and stores them in a hierarchical manner. As the plug-in is now available a PVS-based community has been kick started. This will give those interested a chance to use and/or improve the plug-in. This open source community through collaboration can further improve the plug-in.

Since the plug-in was adapted from the Coq plug-in there is a possibility for further expansion of the project. Collaboration with INRIA has the potential to produce a "lightweight" generic theorem prover plug-in like Proof General. This new improved plug-in would serve both Coq and PVS in the same interface. Modifications could also help support other theorem provers such as Isabelle.

Bibliography

[1] *Eclipse* http:/www.eclipse.org/

[2] PVS Introduction- http://pvs.csl.sri.com/

[3] PVS Documentation - System Guide

[4] PVS Documentation - Language Reference

[5] PVS Documentation - Prover Guide

[6] PVS Documentation - Prelude

[7] PVS Documentation - Semantics

[8] PVS Documentation - Datatypes

[9] PVS Documentation - Theory Interpretations

[10] *Coq* - http://coq.inria.fr/

[11] *Coq Plug-in* - Mr. Julian Charles

[12] *Proof General* - http://proofgeneral.inf.ed.ac.uk/

[13] *Proof General Eclipse* - http://proofgeneral.inf.ed.ac.uk/cgi-bin/wiki.cgi

[14] *PVS* - http://en.wikipedia.org/PVS

[15] Eclipse Eric Clayberg Dan Rubel.*Building Commercial Quality Plug-Ins*

[16] PVS http://www-formal.stanford.edu/clt/ARS/Entries/pvs

[17] *Isabelle* http://en.wikipedia.org/wiki/Isabelle

[18] *Lemma* http://en.wikipedia.org/wiki/Lemma

[19] Javadoc http://en.wikipedia.org/wiki/Javadoc