The jSpider Project: a Prolog IDE in Java

Abstract. *The Integrated Development Environment is an essential ingredient for achieving increased productivity in writing applications. So far, such tools are rather scarce in the logic programming world and even the few ones that exist lack many of the required functionalities. Therefore we tried to develop such an environment and build it on top of one of the most powerful and robust Prolog nowadays – Sicstus Prolog [5]. In this paper, we describe the methodology we adopted in order to identify our product's requirements and we outline the main features of the resulted product in a comparative manner. The implementation is done entirely in Java and therefore benefits from the advantages inherent to the use of the Java programming language.*

Keywords: *Java, Integrated Development Environment, Prolog, Sicstus Prolog, IDE features.*

1 Introduction

It is already common knowledge that the most widely used tool for developing programs in virtually all programming languages is the integrated development environment (IDE), especially when programming-in-the-large is an issue. Alongside many other languages, Prolog has proven to be a strong candidate for such a tool as well. Although some Prolog IDEs are already out on the market, they are either commercial (usually an inconvenience for the academic environment) or too weak for certain purposes. To overcome these problems, we decided to take upon ourselves the task of developing a strong tool tailored for easy development of large Prolog applications.

Since we intended our project to be both open source and multiplatform the natural choice of implementation language was Java. Another requirement was that this IDE be used with Sicstus Prolog, a state-of-the-art, ISO standard compliant, Prolog development system, built around a high performance logic engine [5]. Hence the name for our Prolog IDE is jSpider: Sicstus Prolog Integrated Development EnviRonment in Java.

However, we tried to design it as 'interpreter free' as possible and to make it easily adaptable to other Prolog systems as well. We also had two other goals in mind: we wanted the product to be as user-friendly as possible and we wanted it to have a wide range of useful features.

The present paper examines the main features of this Prolog IDE, which was created as a response to our needs, and also the benefits of using the Java programming language to accomplish the task. We have chosen a comparative approach to our presentation, where we analyze the features of our application against those of other similar products, revealing both our strong and weak points.

The paper is structured as follows: section two deals with the methodology used to undertake the task of developing the Prolog IDE, considering both our implementation requirements and the facilities offered by other similar tools. Section three presents the features of our application, outlining their description, how they relate to those of similar products and also the benefits of programming them in Java. We end the discussion with some conclusions and possible developments in section four.

2 Fundamentals of the jSpider Prolog IDE

Already having a robust and efficient Prolog development system, the natural choice was to build an IDE around it. Since Sicstus Prolog is available for all major platforms and operating systems it was clear that we had to follow. The easiest way to do this was by using Java technology, which supplied us with all the right ingredients. What we obtained in the end was a powerful development environment linked to the interpreter through a communication module.

We began the development process with a wide analysis of related systems. This included both Prolog interpreters and Prolog IDEs. There were several types of inconveniences we identified in the research process. Some Prolog systems, including powerful and widely spread ones, only offered basic IDE features or even none whatsoever (GNU Prolog [15], SWI Prolog [16], Yap Prolog [7], , Ciao Prolog [8]), while others (Amzi! Prolog [13], Quintus Prolog [6] and Visual Prolog [12]) offered more advanced ones, but proved to be unsatisfactory on the whole (lack of code completion, poor project support).

A special case is worth mentioning here - the Emacs interface of Sicstus Prolog, which offers tight integration between Emacs and Sicstus. Developing Prolog applications then benefits from all the features provided by Emacs, the result being an advanced IDE for Sicstus offering a lot of unique useful features that were integrated in jSpider as well.

However, the tight integration with Emacs also has its disadvantages, the presence of Emacs in the Windows based systems being very scarce, and the interface is neither GUI-based nor intuitive enough. Some very useful features like remote predicate call [2], or object-oriented logic programming [3] support are also missing.

As a result we decided to build a stand alone GUI-based IDE that would have most of the advantages of the other studied IDEs and be very intuitive and easy to use. Based on our previous experience, we considered that an important issue in designing this application was to make it project oriented. Taking on from here, we crafted all of the subsystems in a manner that would easily allow us to provide such support. We made extensive use of the Façade design pattern [1] to achieve a high level of independence among our numerous subsystems.

3 Features of jSpider

As we mentioned above, the main part of this presentation deals with the description of the features of the jSpider Prolog IDE. Although most of these are features one can find in any typical IDE (syntax highlighting, code navigation etc.), we tried, and hopefully succeeded, to adapt them to the specific of the Prolog language.

The jEdit open source project [4] has been of great help by providing us with a foundation for our application. After a careful analysis of its features we have managed to identify those which would be suitable for our use and adopted them in our code. We have extended jEdit's code with our own so that it would be appropriate to the Prolog environment and integrated in our framework.

The Java platform, with its interfaces, listeners, adapters and the like, helped a lot here by allowing us to easily integrate jEdit's code into our project. It also provided us with the means of achieving increased productivity in the development work, which was of particular essence in this project. The productivity came from the fact that every time we were faced with a specific problem, Java seemed to offer a solution that fitted our needs. We will elaborate on specific solutions which we used during the sections that follow. Last but not least, Java allowed us to create a very extensible design, thus making our application suitable for further development.

3.1 Editing

Given the purpose of this application, it is clear that editing is a primary requirement. Starting with basic navigation and search and replace facilities and ending with more complex ones such as syntax highlighting and code completion, we tried to include everything a regular Prolog user might find useful in such an application. This process implied identifying a set of new useful features and selectively including existing ones from similar tools.

Basic editing features
Java's document model offered us the possibility of easily modeling operations such as document navigation, editing environment customization, undo/redo actions, selection, cutting and copying, pasting. Aside from this, using the regular expression package, we were able to quickly implement an advanced search and replace facility. From the environments we studied, the only one that had such a feature was Emacs. The rest only had basic search and replace.

Advanced code navigation
Inspired by Emacs, we included several powerful code navigation features in our IDE. Among these, go to next predicate, go to next clause, go to matching parenthesis and various other goto's. Such facilities cannot be found in other environments that we studied (except Emacs, which actually provides certain extra ones, too).

Syntax highlighting
Common to many environments syntax highlighting also appears in our application as a highly configurable feature. Java's XML support was used to provide users with a way to adapt the editor to their own Prolog system (in addition to our providing them with a default configuration file for Sicstus' syntax and library predicates).

Code generation, completion and reformatting
In order to allow the user to achieve increased productivity during the development process we created several code generation and completion capabilities in our environment. These allow quick access to things like predicate calls, definitions, arities, be they library or user defined.

From our experience with IDEs written for other programming languages, we also found it useful to present our users with the option of defining their own code reformatting engine. Once defined, this engine can be used to turn differently styled code into the desired form. While code generation tends to be available in other environments as well, our application tries to also extend the emphasis on code completion and indentation.

3.2 Working with files

The second group of features required by any IDE comprises of typical file operations. We turned to Java's graphical user interface, file input/output and multithreading facilities to complete this area of the project. Intending to supply the users with means of focusing on their business logic rather than file management, we tried to create a module with the following two main attributes: ease of use and robustness (multithreaded I/O).

Basic features
These include the standard *open*, *close* and *save* operations, available through an intuitive file browser (limited to the user's project space described below). Following jEdit's [4] example we designed all these operations as multithreaded ones. The purpose of this was to make sure that the application's graphical user interface remains active throughout these operations (the robustness goal stated above).

Project support
Many times during work with the Prolog language we, as well as others around us, felt the need of a better way of organizing our projects. Therefore we considered it useful to add such support to our IDE. This allows the user to easily work with all files relevant to her current work, including batch consulting them in the Prolog interpreter and concurrently accessing their contents. Java's I/O facilities were of particular help here since they enabled us to transparently access the supporting file system in a platform independent manner.

3.3 Prolog interaction

One of the purposes of this environment is to mediate a transparent communication between the user and the Prolog system. This imposed the requirement of a strong interaction with the supporting Prolog engine. Using Java we were able to completely take over the input and output of the interpreter and simply transforming it into a console for our application.

Prolog wrapper
To encapsulate the Sicstus Prolog interpreter in our environment we made use of Java's process launch and communication facilities. Not only did we provide the users with a simple Prolog console, but we allowed them to use a theoretically unlimited number of Prolog consoles at the same time. This was accomplished by implementing each Prolog wrapper as a separate thread and using a thread pool to provide these threads whenever necessary. Multiple Prolog engines running concurrently are particularly useful when developing distributed logic applications (e.g. client server, Linda based, etc). Communication is done transparently for the user, which means that both the queries and the answers appear to take place in an environment window, even though interprocess communication is actually used.

Run predicate
The purpose for writing a Prolog application is to consult it in an appropriate engine and then post queries. Therefore another feature the environment is supposed to have is that of loading parts of or even entire projects into the engine. By studying the way other applications do this (especially Emacs), we added capabilities like consult region, predicate, buffer or entire project.

Prolog Remote Predicate Call

By means of the Prolog Remote Predicate Call (PRPC) protocol introduced in [2], one can remotely execute his Prolog code or make use of other code hosted on a remote PRPC server. The mix of local and remote calls transparently supports distributed backtracking, opening the way for developing distributed logic applications in a very simple and straightforward manner.

Others

To make the Prolog interaction facility even more powerful, we have also provided support for easy access to various Sicstus built-in features. This includes a means to graphically manipulate all of its flags and an interface for using its library modules. Other environments generally don't support this in such a direct and user friendly manner.

3.4 Object oriented programming in Prolog

Better structuring of Prolog code for developing large applications (programming-in-the-large) naturally imposes the development of object oriented extensions for Prolog. Currently almost every major Prolog system has its own object oriented extension. Sicstus Prolog is no exception, in fact it has two dedicated extensions: Sicstus Objects [5] and LOOP [3]. In contrast with the Sicstus Objects approach which is prototype based, we developed LOOP, a class based object oriented extension of Sicstus Prolog centered on the view of objects as persistent mutable Prolog terms. LOOP support is a unique feature of the jSpider IDE which allows seamlessly integration of Prolog and LOOP code.

3.5 Help

Having the user in mind all along, we also implemented an advanced help system which is designed to be a quick reference to all the predicates' descriptions. Given its functionality, this feature can be safely designated as 'help on predicate'. The help message displayed on screen can be taken from either of the two following sources: the already existing documentation of Sicstus' library or built-in predicates and user-defined comments found in project files.

Relying on Java interfaces we managed to create a layer of abstraction which allows our help system to be independent of its source. Currently implemented is support for the *Texinfo* documentation system and Prolog source comments.

4 Conclusions

The IDE presented in this paper is considered to be a very useful tool for all of those which until now were constrained to rely only upon the simple interface provided by the default Sicstus Prolog development system. Furthermore, we think that our IDE succeeded in grouping together a collection of powerful features, thus bringing the benefits of integrated development, already available for many other programming languages, in the world of Prolog as well. Throughout the entire process we were able to save a lot of time by taking advantage of the large amount of existing Java support.

The table below offers a comprehensive feature comparison between jSpider and other Prolog systems considered in our study. One can see the unique set of features provided by jSpider for the development of Prolog applications.

Prolog System / Feature	Sicstus	Quintus	SWI	GNU	Ciao	XSB	Yap	IF	WinProlog	Strawbery	Amzi!	Visual	Emacs-SP	jSpider-SP
Basic editing	N	Y	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y
Code navigation	N	Y	N	N	N	N	N	N	N	N	N	N	Y	Y
Syntax highlighting	N	Y	N	N	N	N	N	N	Y	Y	Y	Y	N	Y
Code generation	N	N	N	N	N	N	N	N	N	N	N	Y	Y	Y
Code completion	N	N	N	N	N	N	N	N	N	N	N	N	N	Y
Basic file operations	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Project support	N	Y	N	N	N	N	N	N	N	N	Y	Y	N	Y
Prolog wrapper	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Multiple consoles	N	N	N	N	N	N	N	N	N	N	N	N	Y	Y
Local predicate call	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Remote predicate call	N	N	N	N	N	N	N	N	N	N	N	N	N	Y
LOOP support	N	N	N	N	N	N	N	N	N	N	N	N	N	Y

Table 1. Feature comparison table

We envision further development of jSpider in at least two main directions: adapting it to various other Prolog systems and thus helping a wider range of developers and progressively enhancing its functionality in order to meet increasing demands.

References

[1] Erich Gamma, Design Patterns, Addison-Wesley, 1995.

[2] Anonymous authors, *A Prolog Remote Predicate Call Protocol*, Proc. of ECIT 2002, Second European Conference on Intelligent Systems and Technologies, Iasi, Romania, July 17-20, 2002.

[3] Anonymous authors, *LOOP – A Language for LP-Based AI Applications*, Proc. of ICTAI'01, Thirteenth International Conference on Tools with Artificial Intelligence, IEEE Computer Society, Dallas, Texas, USA, November 7-9, 2001, pp.299-305.

[4] Internet reference, The jEdit project, http://www.jedit.org

[5] Internet reference, Sicstus Prolog - http://www.sics.se/isl/sicstus

[6] Internet reference, Quintus Prolog - http://www.sics.se/isl/quintus

[7] Internet reference, Yap Prolog - http://www.ncc.up.pt/~vsc/Yap

[8] Internet reference, Ciao Prolog - http://clip.dia.fi.upm.es/Software/Ciao

[9] Internet reference, XSB Prolog - http://www.cs.sunysb.edu/~sbprolog/xsb-page.html

[10] Internet reference, Strawbery Prolog - http://www.dobrev.com

[11] Internet reference, WinProlog - http://www.lpa.co.uk

[12] Internet reference, Visual Prolog - http://www.visual-prolog.com

[13] Internet reference, Amzi! Prolog - http://www.amzi.com

[14] Internet reference, IF/Prolog - http://www.ifcomputer.com/IFProlog

[15] Internet reference, GNU Prolog - http://gnu-prolog.inria.fr

[16] Internet reference, SWI Prolog - http://www.swi-prolog.org