Designing a Flexible Kernel Providing VDM++ Support for Eclipse

Jacob Porsborg Nielsen and Jens Kielsgaard Hansen

Informatics and Mathematical Modelling, Technical University of Denmark

Abstract. This paper describes the development of an Eclipse [2] based tool set supporting the VDM++ language. It outlines how the Eclipse framework is used to provide a flexible and easily extendible kernel. The basic functionality of the kernel is described as well as how the design is prepared for further development. It is a central point that new functionality can be added without modifying the kernel implementation.

1 Introduction

The project described in this paper is part of the Overture Project[12]. Overture is an open source project aiming at developing tools for VDM++.

Our project aims to provide the kernel for the Overture Tool Set. The project is carried out as a M.Sc. Thesis Project [1] at Technical University of Denmark.

other tools exist that support VDM++ development [11]. The motivation for the Overture Project is, however, to create a new open source tool, which can both be used as a development tool and for research purposes.

This paper will first introduce the functionality of the kernel. Then it summarizes our analysis of suitable tools and techniques for the kernel. The advantages of using Eclipse as a framework is presented followed by an overview of the chosen design. Some ideas for future development based on our project are then described. Finally we conclude the achievements of the project.

2 Functionality

This section will outline the functionality of the kernel.

As shown in Figure 1 the kernel provides the following facilities:

- Providing an Eclipse based editor.
- Parsing a VDM++ plain text specification to an Abstract Syntax Tree (AST). The AST supports use of Visitor Design Pattern [3] in order to make it easy for plug-ins to operate on the AST.
- Converting an AST to XML
- Converting XML to AST
- Pretty print from AST to VDM++ specification in plain text in the Editor.
- The kernel is implemented as plug-ins for Eclipse.
- It provides extension points so that new functionality can be added as Eclipse plug-ins.

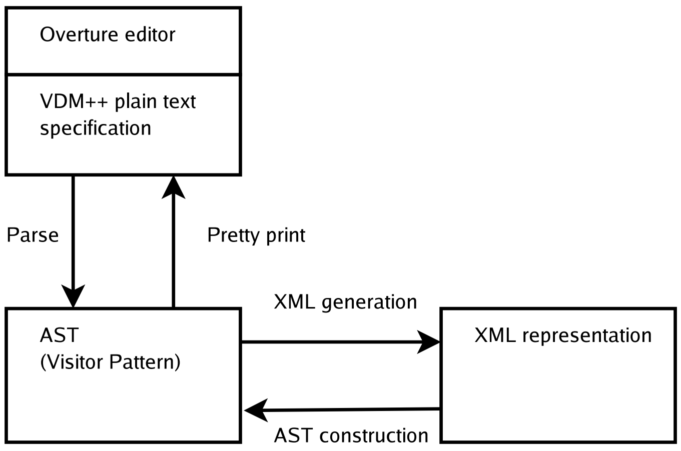

Fig. 1. Insert caption

3 Analysis

This chapter describes the choice of tools, techniques, and principles used in the kernel.

Requirements for the AST Classes

- Different AST classes must be created to represent the various language structures. If the AST classes are built with inheritance and the concept of abstract classes, common properties only need to be defined once.
- If there is an interface for each AST class, we can protect the tree from being modified by visitors. If a visitor needs to modify an AST, then this is still possible if it references the classes directly.
- The children of an AST class should be strongly typed. As there are different classes to represent different language structures, it will be easy to specify a type for each child, - possibly the type of one of it's super classes. This prevents errors when building trees and gives additional possibilities for methods operating on AST classes.
- The non abstract AST classes should have an accept method for different visitors. By enabling use of visitor design pattern, new functionality can be implemented without changing the AST structure.
- Visitor interfaces must be provided for some different types of visitors. Providing visitor interfaces handling arguments of generic types enables a very wide range of visitors to use the AST structure.
- The names used in the AST classes must be meaningful, as other developers should easily be able to use the AST classes in their plug-ins.

Creation of AST Classes Because of the large number of AST classes and corresponding interfaces, we have tested a selected tools for auto generation of these [1]. As none of them could fulfill all the requirements described in Section 3 we chose to write the AST classes by hand. This means writing about 350 classes and 350 interfaces.

Creation of the Parser After examining the EBNF specification for the VDM++ language[6] it was decided that a top down parser is sufficient for parsing the language. The approach for AST and parser construction is described in [4]. The parser generation tool called ANTLR [8] was chosen because it has good error handling, good documentation and a wide community.

Creation of the Pretty Print Visitor Pretty printing can be done by traversing an AST and continuously building a string. Positions and comments are stored in the AST and in the XML instance documents.

Working with XML The tool called JDOM [9] was chosen when working with XML. For importing an XML document JDOM can read an XML document, validate it according to an XML schema and build its own tree representation of the XML document. This can afterwards be converted to the AST representation. For exporting to XML, a visitor is used to traverse the AST and continuously build a corresponding JDOM tree. This can afterwards be exported to a formatted XML instance document.

4 Eclipse

Basically, Eclipse [2] is nothing but a framework intended to be extended by plug-ins. Though Eclipse is distributed with advanced support for programming in Java, the main purpose of Eclipse is to serve as a basic framework for tool plug-ins. In fact, all Java supporting tools in Eclipse, are ordinary plug-ins themselves. Plug-ins are based on extension points and extending those. If a plug-in provides an extension point, other plug-ins can interact with it by extending this extension point. Similarly, plug-ins can interact with the Eclipse framework by extending extension points provided by Eclipse. If a plug-in needs classes from another plug-in in order to work properly, one can specify a dependency.

Eclipse was chosen as the platform for the project, as it is a stable framework with extensive possibilities for adding additional plug-ins, but other environments exists. Some central features making Eclipse a good choice for the project are:

- Eclipse is a wide general framework, that can serve as a platform for many kinds of tools
- Eclipse is well documented and up-to-date supporting e.g. development with Java Generics
- The plug-in concept of Eclipse is powerful for the Overture project
- The framework provides many advanced features to Eclipse based editors

5 Design

This section describes the design of the different parts of the kernel. Because of the size of the kernel the focus is on showing principles in order to give an overview.

5.1 Design of the AST

This section gives an overview of the design principles for AST construction. Here is an example of how the AST is implemented:

Figure 2 shows that an AST class can inherit from one of the two abstract classes `InternalASTNode` or `InternalASTNodeWithComments`, depending on whether it can have associated comments or not. It was decided to store position information for each AST node. This information should e.g. be used for pretty printing or for making it easier to find the right line in the Overture editor using an outline view when working with large VDM++ specifications. The AST classes have set and get methods for each of their children and accept methods for visitors. They implement corresponding interfaces [5] showing only the get methods to the user. By having these interfaces the user can operate on the interfaces without being aware of the implemented AST classes. All AST classes inherits somehow from `InternalASTNode` or `InternalASTNodeWithComments`, either by inheriting from them directly or by inheriting from an abstract class that inherits from them. The abstract classes represents concepts of VDM++, like expressions, functions, operations, etc.

5.2 Design of the Parser

ANTLR generates a lexer and a parser unit. Furthermore, a filtering layer is placed between these two layers in order to handle comments. ANTLR has support for this as well. Precedence rules can be specified in the ANTLR grammar and grouping conventions are implemented when building the AST in the action code. With this solution it is possible to regenerate the parser without having to modify any files manually afterwards.

5.3 Design of the XML Schema

An XML schema is used to verify the correctness of an XML document. In the overture kernel a validation is performed every time an XML document is imported or exported. The schema is created using a tool called XML spy [10]. An XML instance document represents an AST and we have therefore chosen to follow the same structure when creating the XML schema.

5.4 Designing the Visitors

The different visitors should all implement an interface defining which nodes the visitor should visit. Furthermore, there should be a general visitor visiting all

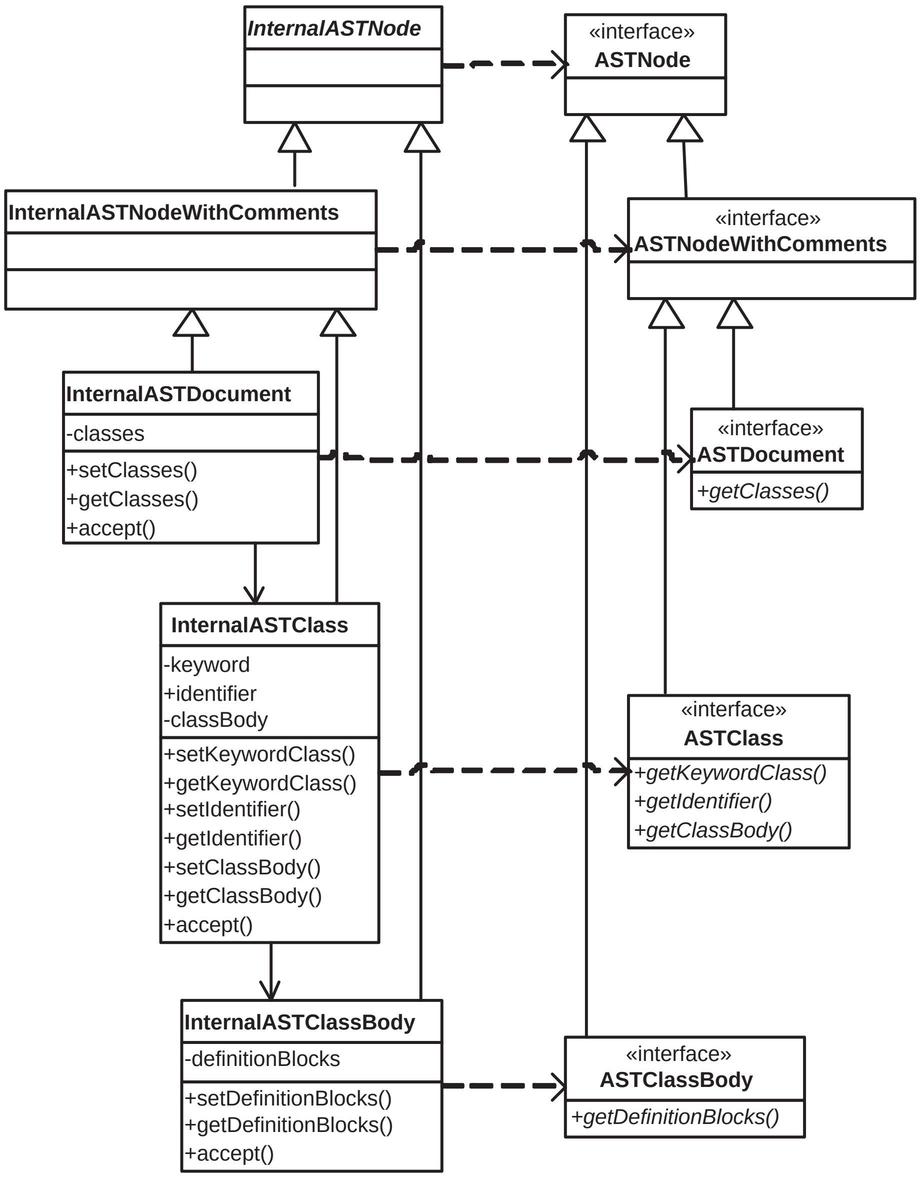

Fig. 2. Design of the AST classes

nodes. All visitors can then extend this general visitor. Section 3 briefly explained how JDOM can be used to export to XML using a visitor. The technique for pretty printing an AST to a VDM++ plain text specification is to use the position information stored in the AST classes for the identifier and keyword nodes to build a correctly formatted string.

5.5 Design of the XML to AST Parser

The XML to AST parser works similar to the parser described in Section 5.2. The difference is that it builds the AST from a JDOM document instead of getting the tokens from the lexer. No error handling is needed because the XML instance document is by definition well formed.

5.6 Design of the Editor

The Overture editor uses a large amount of Eclipse's functionality by extending the extension points defined by Eclipse. The implementation is therefore well integrated with the eclipse framework. It provides coloring of keywords as well as an outline view for giving overviews of large VDM++ specifications. Using the Overture menu provided it is possible to import and export XML.

5.7 Design of Extension Possibilities

The Overture editor is implemented in a plug-in providing a number of extension points in order to make the kernel as flexible as possible. The rest of the functionality of the kernel is implemented in other plug-ins extending the provided extension points. This functionality works so that the Overture editor is unaware of any parser or visitor implementation. Using this design other developers can replace our implementation of the parsers and visitors without having to modify any of the plug-ins. By implementing an interface for a type checker, anyone can extend the kernel with this facility.

6 Future work

There is a wide range of additional functionality that can be added to the kernel. This includes e.g. a type checker, a Java code generator, and import and export facilities to UML. These can all be added using the extension principles provided by the kernel.

For future development based on our kernel, we refer to [1]. A chapter of the master thesis report is dedicated to present how to extend the language or how to add additional plug-ins with new functionality. The report also documents the analysis, design, implementation, and test issues of the kernel development.

7 Conclusion

This paper has presented an overview of the functionality and the design of the kernel for the Overture tool set. The AST classes, parsers and the visitors have been implemented by following the theory [4] and selected guidelines [5]. The integration with Eclipse is done using plug-ins and the kernel has been constructed to be as flexible as possible.

8 Acknowledgements

We would like to thank our supervisors Anne E. Haxthausen, Hans Bruun and Peter Gorm Larsen for their enthusiasm for our project. We also thank the Overture Core Group for their input and good discussions.

References

1. M.Sc. Thesis Project, Development of an Overture/VDM++ Tool Set for Eclipse, by Jacob Porsborg Nielsen and Jens Kielsgaard Hansen, Informatics and Mathematical Modelling, Technical University of Denmark. Supervisors: Hans Bruun and Anne E. Haxthausen, External Supervisor: Peter Gorm Larsen.
   The project will be available ultimo August 2005 at http://www.imm.dtu.dk/English/Research/Search_publications.aspx
2. DAnjou et.al., Eclipse, The Java Developer's Guide to Eclipse, Addison-Wesley, second edition, 2005.
3. Erich Gamma et.al., Design Patterns, Addison-Wesley 1995.
4. David A. Watt et.al., Programming Language Processors in Java, Pearson Education Limited, 2000.
5. P. Mukherjee. A blueprint for development of language tools. Memo produced as input to Overture, 2004.
6. VDMTools language definitions, The VDM++ language, EBNF grammar and description of the language, http://www.vdmbook.com/langmanpp_a4_001.pdf
7. Eclipse, The Eclipse framework home page http://www.eclipse.org
8. ANTLR, ANTLR parser generator, http://www.antlr.org
9. JDOM, An XML framework for Java, http://www.jdom.org
10. Altova xmlspy, Graphical tool for generation of XML schemas, www.xmlspy.com
11. VDMTools, Tool set supporting VDM++ development. Was prior called IFAD VDMTools, now called CSK VDMTools. http://www.vdmbook.com/tools.php
12. Overture, The Overture project, http://www.overturetool.org