Framework for Text Editors of Domain Specific Languages in Eclipse

submitted by
Paulus Sentosa
Matr. Nr.: 22946

supervised by
Prof. Dr. Ralf Möller
Miguel Garcia, M.Sc.

Hamburg University of Science and Technology
Software Systems Institute (STS)

Abstract

Gymnast is an Eclipse-based framework that provides supports as domain-specific languages tooling. Using a grammar specification of a language as input, Gymnast generates classes that represent the language's concrete syntax and some utility classes. The generated classes can be used to build editors for the language. The long-term goal of Gymnast is however to be a full-featured language-specific IDE generator. This project work presents the framework in detail and some usage examples of the framework for defining languages and implementing their tools.

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, May 11th, 2007
Paulus Sentosa

I would like to thank Prof. Dr. Ralf Möller for providing a very interesting topic and giving me the oppotunity to work on this topic as my project work. I would also like to thank Miguel Garcia, M.Sc for his guidance, encouragement and endless patience during this work, especially in every single discussion session.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Nowadays language engineering plays an increasingly important role in computer science. Generally the resulting languages can be classified into two: General-Purpose Language (GPL) and Domain-Specific Language (DSL). GPL has a wide spectrum of usages and serves multi purposes for programming as well as for modeling tasks. Visual Basic, C, C#, C++, Java and UML are some examples of GPL. In contrast DSLs are defined to be specific only on their certain domains, which gives the possibility to cover every single aspect of the domains to be considered, in terms of presentation and implementation [Coo04][vDKJ00]. Moreover, DSLs are in certain ways more concise and therefore readable and may even be understood by non-programmer domain experts [JBC$^+$06]. Spreadsheet macros, Csound and GraphViz, and some other languages are classified into this group[1].

Programmers are looking more and more into defining and using the latter because they can be used specifically well to solve domain-specific problems which are within their field of interests. A certain productivity improvement can be achieved in defining specific solution using DSL, compared to using GPL. Each DSL provides built-in abstractions and notations to specify concepts and semantics focused on, and usually restricted to, a particular problem domain [WSTD05]. There are some works which confirm, that using DSLs is a main factor for gaining improvement in the productivity level of domain specific softawre implementation [KT00][ea96]. Not only academic, but also industrial and goverment communities make use of DSLs to describe, for example, 3D animations [Ell97], business rules [WTH03], insurance business logic [Weg04], software testing [Gro04] and military command and control [Wil03].

Despite all the advantages there are some issues that DSL designers are confrontated with. Prior to everything, methodologies are needed to derive a DSL from domain knowledge as proposed in [TMC99]. But the main issue which is often encountered during designing DSLs is the lack of appropriate tooling to support the process chain, which starts with defining the DSL itself up to providing usable tools to make use of the language in form of editors as a minimal requirement. Some frameworks which will be presented in the next chapters cope with this issue, including the framework with which this project work deals.

[1] http://en.wikipedia.org/wiki/Domain-specific_language

The implementation of DSLs is moving toward the use of model-driven approaches; that is, models are used as main artifacts (first class assets) in all stages of a development process. In this kind of approach, a metamodel defines all possible concepts and relations, which serve as basics for models to be defined. Having implemented this idea, it will be easier to check the conformity and validity of the models which are used to describe the domain-specific artifacts. In context of language engineering, the metamodel can be used to define the abstract syntax of the language; the concrete syntax, which represents the grammar of the language, is described as the definition of the association between the metamodel and syntactic elements [JB06].

While visual syntax fits well to describe a metamodel, model-based textual notation, also known as declarative modeling, may be favoured for describing the concrete syntax due to some considerations [Spi03]. Not only the high-level skills (the textual, abstract formalization of concrete concepts) but also the low-level skills (text manipulation using text-based editors) are perfectly matched by the model composition mechanism. Being closer to the program's representation, such a declarative notation also forces the designer to make differentiation between the model and the corresponding implementation and between essential part of the systems and its additional features. Visual syntax will also demand the capability of drawing the graphs nicely, thus using the drawing tools correctly, in order to get more or less readable and understandable description, which could be a tedious work. Furthermore, there are different tools available to support working with textual notation, starting from simple text processing editor, revisioning and versioning tools for management of source code and team work, up to automatization of text generation from even higher-level description using trivial scripts and tools operating on design process inputs. All advantages will sum up to provide higher overall productivity by using textual notation to describe the concrete syntax of the language.

Existing frameworks make use of the textual concrete syntax to provide designer with ready-to-use helper classes, such as parser and lexer, and classes representing nodes of the concrete syntax tree. By utilizing this classes an editor for the language can be built, whose input will serve in building abstract syntax tree out of the concrete part. Taking this possibility a step further, a full-featured integrated development environment (IDE) could also be generated automatically to help settling the newly defined DSLs into the hands of potential users.

1.2 Objectives

This project work will present some already existing frameworks that deal with the textual concrete syntax of DSLs. The main framework to be presented is however the Gymnast framework, on which Emfatic, a textual representation of Eclipse EMF Ecore model, is defined. Due to the lack of documentation about Gymanst and Emfatic by the time this project work is being carried out, they will be presented here in detail. Especially some improvements made on the graphical user interface (GUI) of Emfatic will be of interest, as they make the cornerstones of a full-featured IDE.

While Emfatic represents the Ecore Model of Eclipse EMF, another model-oriented textual notation representing UML2 State Machine will be defined. Some enhanced features that are developed for Emfatic GUI will be implemented for the new language as well. For this project work, every feature related to the GUI is to be implemented manually. However, as mentioned before, the work is moving toward the generation of the complete IDE, to which this work is oriented.

1.3 Document Structure

Chapter 1 gives introductory explanation on the topic of the project, including background information and the main objectives.

Some prototypes are then introduced in Chapter 2, giving the overview of the available tools at the moment of writing.

Entering the design phase of the development process of text editor for UML2 State machine in chapter 3 the main framework Gymnast is first introduced, by giving explanation on its components and manually implemented design patterns. This chapter also gives the overview of the Eclipse UML2 framework, using which the new textual notation will be implemented.

The following implementation phase consists of defining the textual notation by means of its grammar, generating and building the required classes for the implementation of editors and all its features. This will be given in Chapter 4.

Chapter 5 concludes and shows the possible future works.

Chapter 2

Existing Prototypes

This chapter gives an overview of available tools that support creation and use of DSLs.

2.1 Gymnast

All information on Gymnast and figures are cited and taken from [Dai05].

2.1.1 Overview

Gymnast is a framework for generating `ASTNode` class hierarchies similar to the JDT's `ASTNode` and its subclasses in the package `org.eclipse.jdt.core.dom`. The idea of generation is gained after examining the JDT's classes, which reveals that they contain repeating patterns of boiler plate code (e.g. for parent/child navigation, visitor pattern implementation, factory methods for parser tree construction) and that the codes are strongly tied to generated Java parsers (for each Java grammar rule an `ASTNode` class will be generated).

Based on a syntactic specification of a language, i.e. grammar of the language, Gymnast will generate the `ASTNode` classes, an `ASTVisitor` class and input to a parser generator. The resulted parser will build the corresponding abstract syntax tree using the generated `ASTNode` classes. Those classes inherit from common AST oriented classes allowing generic UI components (such as a parse tree view) to work for all languages generated by the framework. By processing the tree using the generated `ASTVisitor` all other works (e.g. semantic analysis, code generation) can be done.

2.1.2 Information Flow in Gymnast

Figure 2.1 shows the information flow in Gymnast by depicting the inputs, generated intermediate and end outputs and shows how they are related wihtin the generation process.

There are 3 kinds of inputs for Gymnast. The main input is the Gymnast grammar file with the `.ast` extension, in which the rules for a language are defined, whose kinds are presented in the next subsection. A lexer file for the parser generator is a prerequisite for the parser generation. As usual in code generation framework, Gymnast also needs predefined templates to define how the generated code for the `ASTNode` classes should look like.

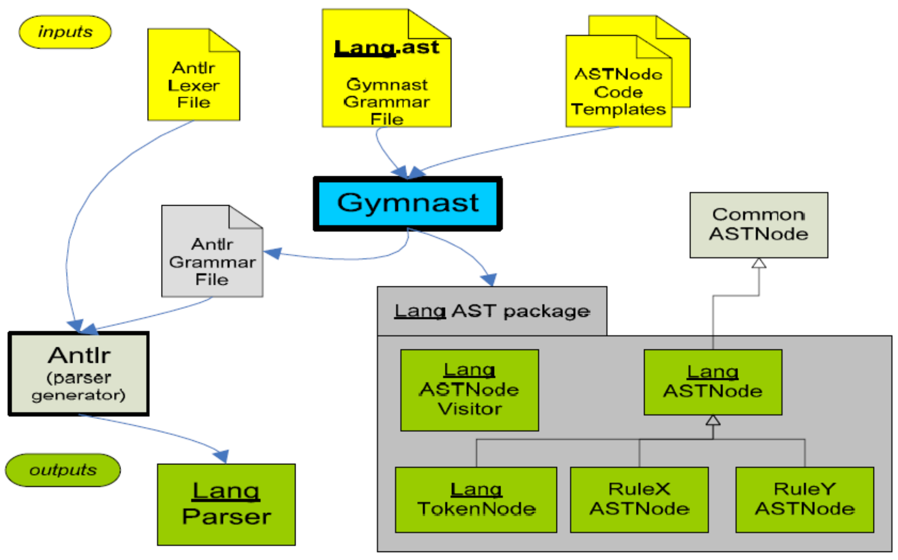


Figure 2.1: Gymnast information flow

The default parser generator which is used by Gymnast is `ANTLR`[1], as it can be seen on the figure. In later phase there will be more parser generators available to be used with Gymnast in order to advance its usability. Besides the lexer file, a parser generator also makes use of the corresponding grammar file for the parser generator which is generated out of the `.ast` file. Based on these inputs, a parser for the text being typed using the editor can be generated.

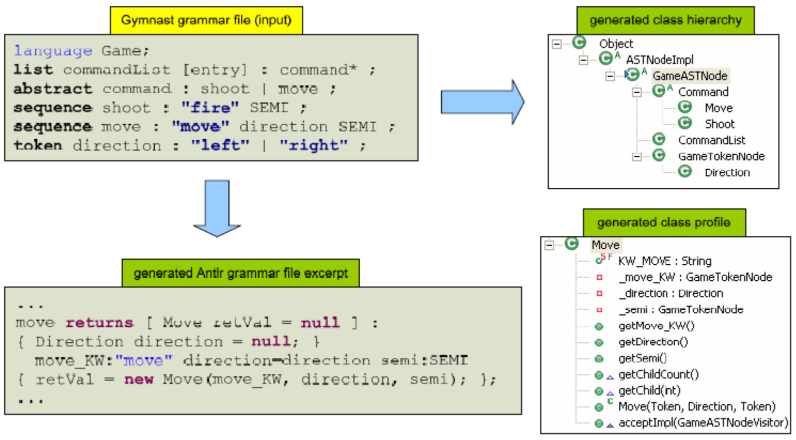
Each rule that is defined in the Gymnast grammar file has its corresponding class(es) being generated by Gymnast. All of them are derived from a common class called `ASTNode`, but only one class extends directly from it and represents the name of the new language. The other classes represent rules and tokens. As mentioned before, a visitor is also generated, which can be utilized to walk over and process the nodes of the trees, e.g. the generated classes. The following figure shows some excerpts from both Gymnast and the resulted parser generator (in this case `ANTLR`) grammar file and a generated class with its content and a tree view showing the position of the class in the hierarchy.

2.1.3 Rules

A syntactic specification of a language is defined as a group of rules. Rules define the keywords and the order in which text (source code) can be written. Predefined token are eventually used in rules. There are 4 kinds of rules defined in Gymnast:

- `token`
  A token defines a string literal or references another lexer token, e.g.:
  `token` modifier : `"public"` |`"private"` |`"protected"` ;
- `list`
  A list contains repetition of elements of same kind, e.g.:
  `list` elements : element* ;

[1] http://www.antlr.org


Gymnast grammar file (input)
generated class hierarchy
generated Antlr grammar file excerpt
generated class profile

```
language Game;
list commandList [entry] : command* ;
abstract command : shoot | move ;
sequence shoot : "fire" SEMI ;
sequence move : "move" direction SEMI ;
token direction : "left" | "right" ;
```

```
Object
  ASTNodeImpl
    GameASTNode
      Command
        Move
        Shoot
      CommandList
      GameTokenNode
        Direction
```

```
...
move returns [ Move retVal = null ] :
{ Direction direction = null; }
  move_KW:"move" direction=direction semi:SEMI
{ retVal = new Move(move_KW, direction, semi); };
...
```

```
Move
  KW_MOVE : String
  _move_KW : GameTokenNode
  _direction : Direction
  _semi : GameTokenNode
  getMove_KW()
  getDirection()
  getSemi()
  getChildCount()
  getChild(int)
  Move(Token, Direction, Token)
  acceptImpl(GameASTNodeVisitor)
```

Figure 2.2: Gymnast example grammar

- `sequence`
  A sequence defines the order in which the text (source code) is written. It may contain keywords, lexer tokens and other sequence, e.g.:
  `sequence` addition : number ADD number SEMI ;

- `abstract`
  An abstract produces inheritance pattern, e.g.:
  `abstract` type : class |interface ;

2.2 SAFARI

All information on SAFARI and figures are cited and taken from [CFJL07] and the official SAFARI website ([Res]).

2.2.1 Overview

SAFARI is an Eclipse-based meta-tooling framework that is intended to speed the creation of sophisticated development environments for new or existing programming languages by generating language-specific IDEs. To realize state-of-the-art functionality of an IDE, Eclipse uses Java Development Toolkit (JDT). The goal of SAFARI is to support all of the JDT's rich set of capabilities:

- editors with features like syntax highlighting, hover help, text folding, etc.
- different kind of views
- content assist and completion
- navigation
- incremental project building and dependency tracking
- debugging and refactoring

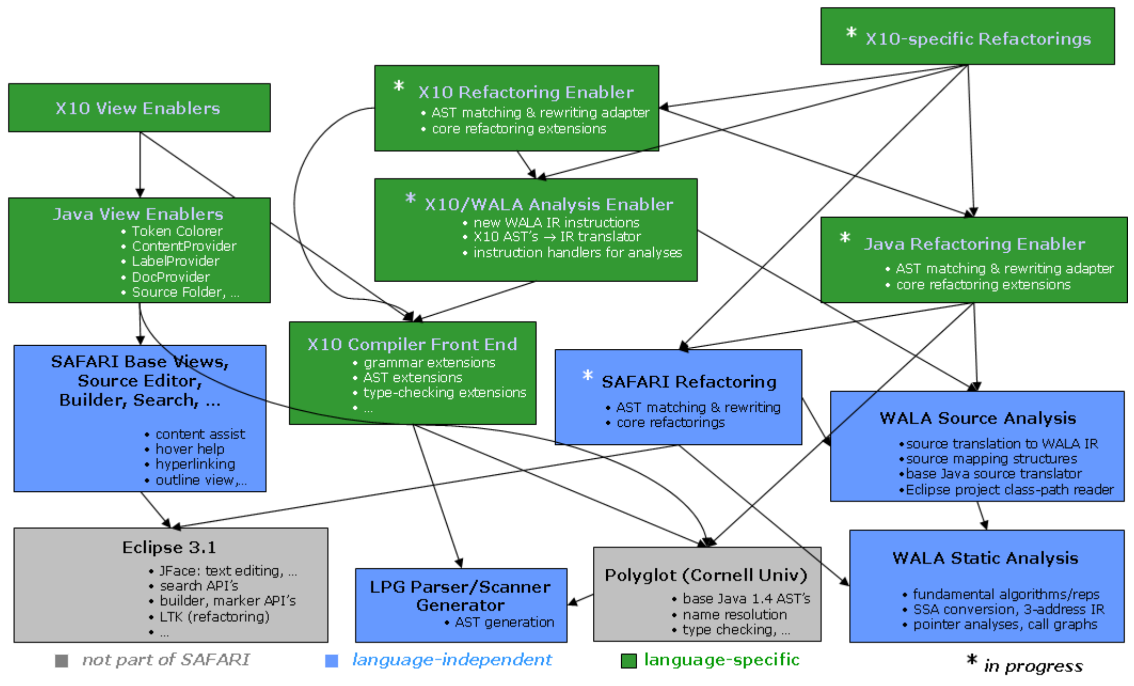

* X10-specific Refactorings
X10 View Enablers
* X10 Refactoring Enabler
• AST matching & rewriting adapter
• core refactoring extensions
Java View Enablers
• Token Colorer
• ContentProvider
• LabelProvider
• DocProvider
• Source Folder, ...
* X10/WALA Analysis Enabler
• new WALA IR instructions
• X10 AST's → IR translator
• instruction handlers for analyses
* Java Refactoring Enabler
• AST matching & rewriting adapter
• core refactoring extensions
SAFARI Base Views, Source Editor, Builder, Search, ...
• content assist
• hover help
• hyperlinking
• outline view,...
X10 Compiler Front End
• grammar extensions
• AST extensions
• type-checking extensions
• ...
* SAFARI Refactoring
• AST matching & rewriting
• core refactorings
WALA Source Analysis
• source translation to WALA IR
• source mapping structures
• base Java source translator
• Eclipse project class-path reader
Eclipse 3.1
• JFace: text editing, ...
• search API's
• builder, marker API's
• LTK (refactoring)
• ...
LPG Parser/Scanner Generator
• AST generation
Polyglot (Cornell Univ)
• base Java 1.4 AST's
• name resolution
• type checking, ...
WALA Static Analysis
• fundamental algorithms/reps
• SSA conversion, 3-address IR
• pointer analyses, call graphs
not part of SAFARI
language-independent
language-specific
* in progress

Figure 2.3: High level architecture of SAFARI

2.2.2 Architecture

Figure 2.3 depicts the high-level architecture of SAFARI. Extensibility of each component assures a higher degree of reuse across languages.

In Figure 2.4 the relationship between the user-visible services and the underlying analyses on which the services rely is shown. It can also be seen from the figure how SAFARI helps to accelerate the IDE development process. For example, the creation of an indexed search facility which is based on the use of AST patterns to specify the desired index entry types.

2.2.3 IDE Development Process Using SAFARI

Development of SAFARI IDE is an incremental process which is divided into several parts and described as follows:

1. **Language Descriptor.** Creating a language descriptor, which associates the language with unique identifier, set of file name extensions and a base language from which the language is derived

2. **Language Grammar and Parser.** Creating a grammar file which results in the generation of parser implementation

3. **IDE Appearance and Behavior.** Using the different implementation skeletons provided by the wizards, IDE developer can specify the implementation of the appearance and behavior of the IDE

4. **Project Building and Program Compilation.** SAFARI provides builder skeleton which can be implemented as a wrapper around existing compiler or as a new compiler starting from the resolved AST

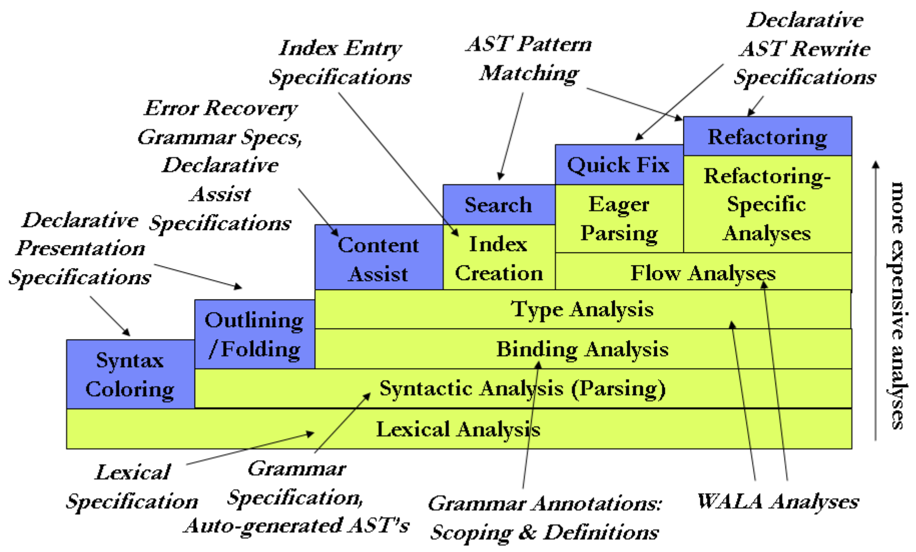

Index Entry Specifications
AST Pattern Matching
Declarative AST Rewrite Specifications
Error Recovery Grammar Specs, Declarative Assist Specifications
Declarative Presentation Specifications
Refactoring
Quick Fix
Search
Refactoring-Specific Analyses
Eager Parsing
Index Creation
Content Assist
Flow Analyses
Outlining /Folding
Type Analysis
Syntax Coloring
Binding Analysis
Syntactic Analysis (Parsing)
Lexical Analysis
more expensive analyses
Lexical Specification
Grammar Specification, Auto-generated AST's
Grammar Annotations: Scoping & Definitions
WALA Analyses

Figure 2.4: Relationship of user-visible language services to underlying analyses and SAFARI features that aid in creation of these services

5. **Refactoring and Analysis.** Refactoring is supported through the combination of a generic AST rewriting framework. The static analysis engine is provided through an exisiting open-source scalable analysis framework.

2.3 Textual Concrete Syntax (TCS)

All information on TCS and figures are cited and taken from [JBC$^+$06] , [JB06] and [JBK06].

2.3.1 Overview

TCS is a DSL for the context-free specification of textual concrete syntaxes which is developed as part of the model-based DSL building framework AMMA (ATLAS Model Management Architecture)[JBKV06]. From the specification, models can be serialized into their textual equivalent and text can be parsed into models. In other words, a TCS specification defines a bidirectional translation between a textual representation of a model and its internal representation. Such a feature is essential to the development of tools for text-based DSLs.

2.3.2 Architecture of the Framework

The core of the framework are formed by several DSLs, each of which is perceived as sets of models and used to define the components of other DSLs. Each of DSL has its abstract syntax, which is a conceptualization of a certain domain and captured in a metamodel called Domain Definiton MetaModel (DDMM), its concrete syntaxes and semantic definition. Some of the core DSLs are KM3 (Kernel MetaMetaModel) which is a language for describing metamodels, ATL (ATLAS Transformation Language) which is a model transformation language, and TCS itself. The abstract syntax of the DSLs, i.e. the DDMM, is defined in KM3, the concrete syntax is implemented with TCS and the semantic is written in ATL. Figure 2.5 shows the architecture of the framework.

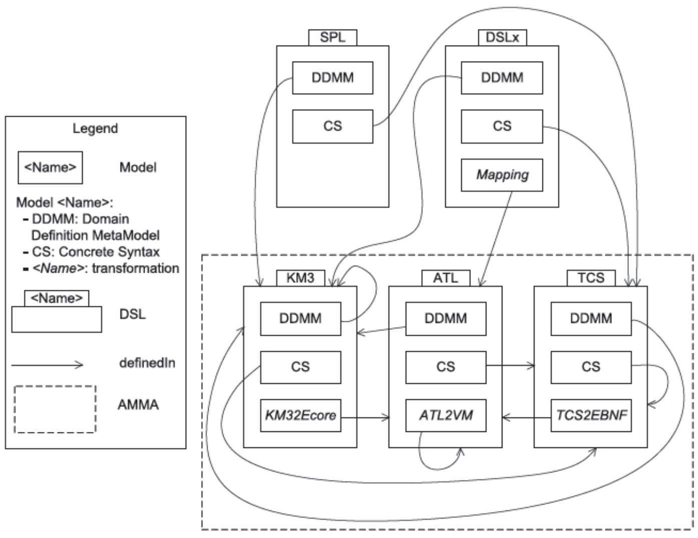


Figure 2.5: AMMA core DSLs

2.4 Textual Concrete Syntax Specification Language

All information on TCS and figures are cited and taken from [FSGM05].

2.4.1 Overview

TCSSL is a DSL which specifies bidirectional mapping, thus enables translation, between its concrete and abstract syntax via EBNF-like rules. Concrete syntaxes written in TCSSL may be used by compiler compilers to generate text analyzers that produce models as instances of metamodels, instead of merely concrete syntax trees. It should also be possible to generate, again from that concrete syntax specification, pretty printers, IDEs, or even incremental synchronizers that update the textual views representing a model, and symmetrically update a model when the textual representation changes. Figure 2.6 summarizes an example of usage of such specification.

2.4.2 Concepts of TCSSL

The main concept of TCSSL is presented in Figure 2.7.

2.4.3 Architecture of The Tool Chain

A prototype implementation of TCSSL has a tool chain that is divided into two main parts: a parser and a text generator. The parser parses the TCSSL specification, and generates the parser for its textual notation. The code generator is being developed to generate the TCSSL textual notation from the model repository.

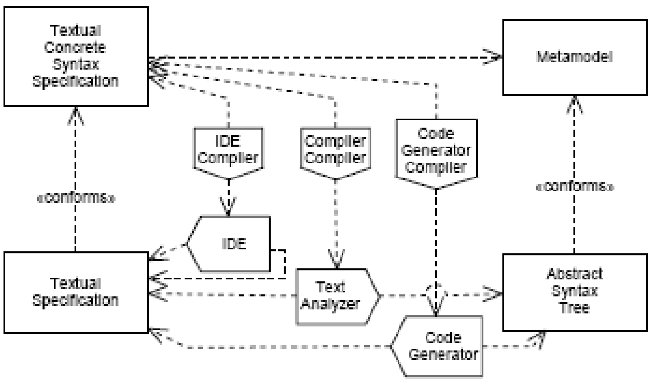


Figure 2.6: Usage of a Formal Textual Concrte Syntax Specification Example

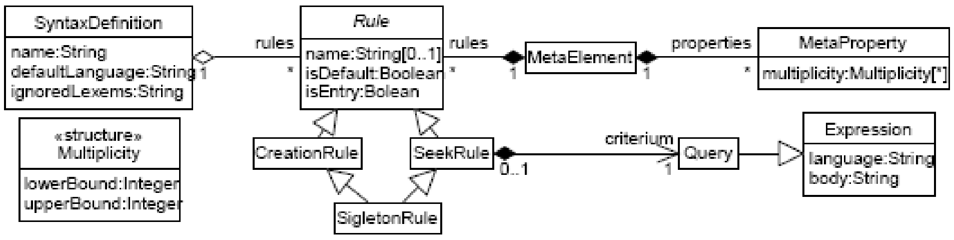


Figure 2.7: Main concept of TCSSL

For the parsing function, the content of the generic editor is used as input. The parser creates the corresponding elements into the model repository. ANTLR is used to simplify the parser generation. Rules in the ANLTR grammar are mapped one-to-one to the TCSSL rules. The ANTLR grammar file is the only element to produce for the transformation from concrete to abstract syntax. This grammar file embeds the interactions with the model. This grammar file produces the parser for the textual notation. Code generation from model is done using the generic template engine JET[2], which is part of Eclipse EMF Project. A template file is associated to each rule in TCSSL.

[2]http://www.eclipse.org/emft/projects/jet/#jet

Chapter 3

Textual Editor of UML2 State Machine : Design Phase

This chapter describes Gymnast, which serves as the core of the framework for DSL Tooling presented in this project, from developer's view by showing all its components and how they are related with each other, and gives a glance at the patterns that are used for implementing this framework.

3.1 Gymnast : Runtime and Generator

Gymnast comprise 2 main parts, `Runtime` and `Generator`, each of which is divided into 2 further parts, `Core` and `User Interface`. The `Runtime` part contains classes which are the very basic components of Gymnast. The core package defines the common shared classes and interfaces for all other generated classes, a class and an interface which will be used for building the Outline View feature of the user interface, classes providing facilities to define code templates and some helper classes. The User Interface related package is responsible for defining the basic editor and all its features as part of an IDE.

The core part of `Generator` takes care of the generation functionality of Gymnast, which result in generated DSL parser and the classes representing the nodes of the DSL's concrete syntax tree. This ability for generating classes is provided by making use of Eclipse extension-points mechanisms, through which a parser generator and a so-called AST generator are attached. A simple Gymnast editor and the selectable **Generate AST** function from context menu of the editor file are implemented in the User Interface package. Figure 3.1 shows the components of Gymnast and the dependency between them.

The following subsections give a deeper look into Gymnast by pointing out every component and how they are related.

3.1.1 Runtime - Core

The `runtime.core` package provides the base classes of Gymnast, from which classes for DSLs (concrete sytnax tree, parser, user interface) are generated or derived. Figure 3.2 shows the content of the package.

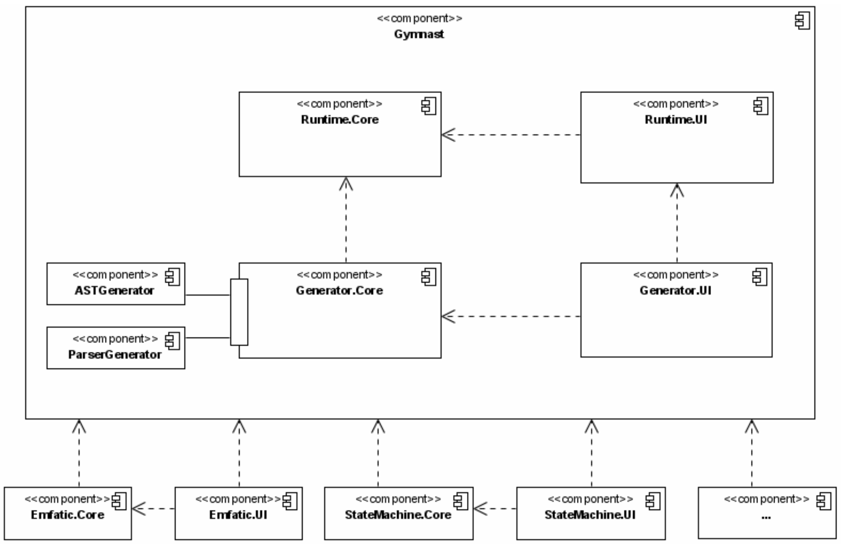


Figure 3.1: Component Diagram of Gymnast

AST

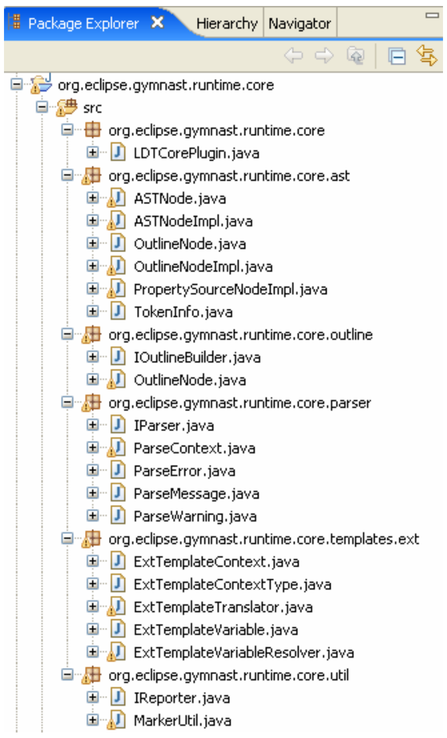
The package `org.eclipse.gymnast.runtime.core.ast` contains the following classes:

- `ASTNode`
- `ASTNodeImpl`
- `OutlineNode`
- `OutlineNodeImpl`
- `PropertySourceNodeImpl`
- `TokenInfo`

The main parts in this package are the interface `ASTNode` and its implementation class `ASTNodeImpl`. Classes representing nodes of the concrete syntax tree of languages, which are the core of every DSL, are derived from `ASTNodeImpl`. The interface's methods are used to enquire about the node itself.

`PropertySourceNodeImpl` extends from `ASTNodeImpl` and implements `IPropertySource` of Eclipse, which is an interface to an object that provide the properties of a selected element to be displayed in the standard Property View of Eclipse. A further derivation is the class `OutlineNodeImpl` that implements the method for dealing with outline nodes of an editor's content. Both `PropertySourceNodeImpl` and its derivation haven't been used so far. Instead, another class from the Outline package (see next subsection) is being used.

The class `TokenInfo`, as its name says, gives information about a token, i.e text, offset and type of the token. A token represents a word that is part of the grammar, e.g. a keyword, or a predefined symbol which can be used within the grammar, e.g. the word SEMI which represents the use of the symbol ";" in a syntax description.



```
org.eclipse.gymnast.runtime.core
  src
    org.eclipse.gymnast.runtime.core
      LDTCorePlugin.java
    org.eclipse.gymnast.runtime.core.ast
      ASTNode.java
      ASTNodeImpl.java
      OutlineNode.java
      OutlineNodeImpl.java
      PropertySourceNodeImpl.java
      TokenInfo.java
    org.eclipse.gymnast.runtime.core.outline
      IOutlineBuilder.java
      OutlineNode.java
    org.eclipse.gymnast.runtime.core.parser
      IParser.java
      ParseContext.java
      ParseError.java
      ParseMessage.java
      ParseWarning.java
    org.eclipse.gymnast.runtime.core.templates.ext
      ExtTemplateContext.java
      ExtTemplateContextType.java
      ExtTemplateTranslator.java
      ExtTemplateVariable.java
      ExtTemplateVariableResolver.java
    org.eclipse.gymnast.runtime.core.util
      IReporter.java
      MarkerUtil.java
```

Figure 3.2: Classes of Runtime - Core

Outline

The package `org.eclipse.gymnast.runtime.core.outline` contains the following classes:

- `IOutlineBuilder`
- `OutlineNode`

An important feature of a DSL tool is an outline of the text content written in a certain language to get a clear overview of content. `OutlineNode` is used to represent nodes in an outline tree. `IOutlineBuilder` provides the `buildOutline()` method which will be implemented by every class responsible for building the outline.

Parser

The package `org.eclipse.gymnast.runtime.core.parser` contains the following classes:

- `IParser`
- `ParseContext`
- `ParseError`
- `ParseMessage`
- `ParseWarning`

`ParserContext` delivers all informations needed to do parsing, i.e. the root of trees to be parsed, the parsed object any additional information (i.e. error and warning information, if there is any, which are represented by the classes `ParseError` and `ParseWarning` as derivation of `ParseMessage`). `IParser` gives the `parse()` method which will be used by any parser driver, i.e. a class which executes the parsing.

Templates

The package `org.eclipse.gymnast.runtime.core.templates.ext` contains the following classes:

- `ExtTemplateContext`
- `ExtTemplateContextType`
- `ExtTemplateTranslator`
- `ExtTemplateVariable`
- `ExtTemplateVariableResolver`

This package contains classes that define the template for code generation. This includes the template context, context type, translator, variable and variable resolver of the template. Template context is a context in which a certain template is being resolved, e.g. during the code generation, and this must have a certain type, which is defined as the context type. Templates translate its input string into a so-called template buffer. Content in the buffer which is valued as a variable is defined as a template variable using the template translator, which can later be resolved based on the context within which the variable exists.

Utility

The package `org.eclipse.gymnast.runtime.core.util` contains the following classes:

- `IReporter`
- `MarkerUtil`

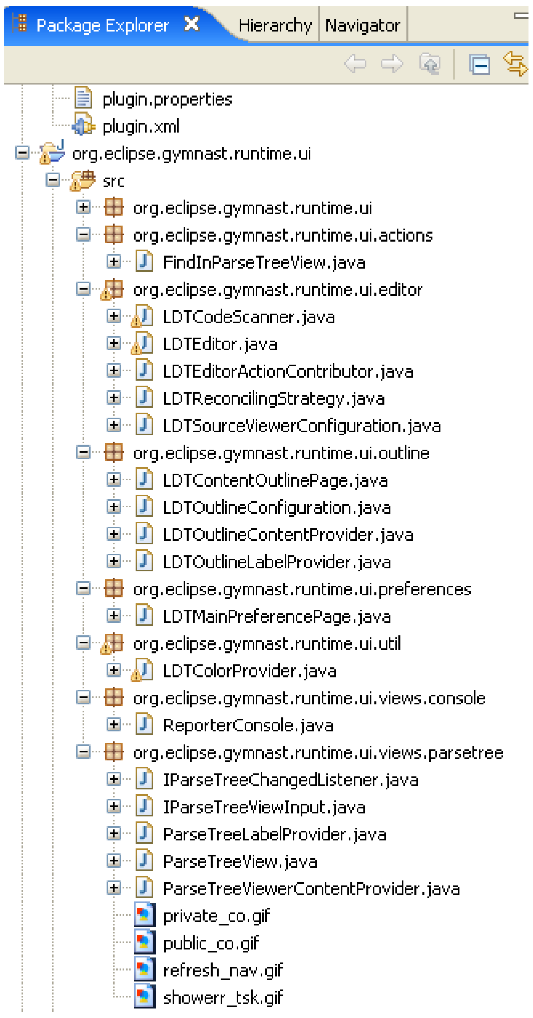
`IReporter` provides methods to deliver any (error or warning) messages which are implemented by the class `ReporterConsole` from the package `org.eclipse.gymnast.generator.runtime.ui`. `MarkerUtil` is used to put marker using the `IMarker` of Eclipse on any warning or error on the parser context based on the message contained in the context.

3.1.2 Runtime - User Interface

To implement the user interface of the DSL tool, Gymnast has provided several classes, from which user-interface classes for the corresponding DSLs may be derived . Figure 3.3 shows the content of the package.

Action

This package contains one single class, `FindInParseTreeView`, which defines an action that is chosenable from the context menu and executed from an active editor, and whose result is shown in another view. The action defined by this class is showing a chosen text from the editor (i.e. a chosen element) in a view called Parse Tree View.



```
    plugin.properties
    plugin.xml
org.eclipse.gymnast.runtime.ui
  src
    org.eclipse.gymnast.runtime.ui
    org.eclipse.gymnast.runtime.ui.actions
      FindInParseTreeView.java
    org.eclipse.gymnast.runtime.ui.editor
      LDTCodeScanner.java
      LDTEditor.java
      LDTEditorActionContributor.java
      LDTReconcilingStrategy.java
      LDTSourceViewerConfiguration.java
    org.eclipse.gymnast.runtime.ui.outline
      LDTContentOutlinePage.java
      LDTOutlineConfiguration.java
      LDTOutlineContentProvider.java
      LDTOutlineLabelProvider.java
    org.eclipse.gymnast.runtime.ui.preferences
      LDTMainPreferencePage.java
    org.eclipse.gymnast.runtime.ui.util
      LDTColorProvider.java
    org.eclipse.gymnast.runtime.ui.views.console
      ReporterConsole.java
    org.eclipse.gymnast.runtime.ui.views.parsetree
      IParseTreeChangedListener.java
      IParseTreeViewInput.java
      ParseTreeLabelProvider.java
      ParseTreeView.java
      ParseTreeViewerContentProvider.java
      private_co.gif
      public_co.gif
      refresh_nav.gif
      showerr_tsk.gif
```

Figure 3.3: Classes of Runtime - User Interface

Editors

The package `org.eclipse.gymnast.runtime.ui.editor` contains the following classes:

- `LDTCodeScanner`
- `LDTEditor`
- `LDTEditorActionContributor`
- `LDTReconcilingStrategy`
- `LDTSourceViewerConfiguration`

This package builds the core of the user interface by defining its main part, namely an editor for writing text (codes) using the DSL and out of which further processing can be made (executing code and building abstract syntx tree).

`LDTEditor` is the basis implementation of editor for every DSL developed using this framework. It extends `TextEditor` class of Eclipse and implements the interface `IParseTreeViewInput` from the package `parsetree` which provides method to interact with the input of the Parse Tree View.

Reconciling is needed to have synchronized document (i.e. content of the text viewer) and its resulted parsed tree of the document. This is an important issue if the document is being edited/changed. `LDTReconcilingStrategy` implements `IReconcilingStrategy`, an interface that provides methodes for reconciling, and does the reconciling step by simply re-parsing the documents to get the latest version of parsed tree and set this to the actual document. Further explanation on other classes of this package and can its extension can be seen directly on the specific implementation of each DSL tool (e.g. Emfatic User Interface).

Outline

The package `org.eclipse.gymnast.runtime.ui.outline` contains the following classes:

- `LDTContentOutlinePage`
- `LDTOutlineConfiguration`
- `LDTOutlineContentProvider`
- `LDTOutlineLabelProvider`

This package includes classes that are needed to build the outline view of an editor: the content and label provider and the content outline page itself. `LDTOutlineConfiguration` collects this information in a single instance and can be extended by every outline configurator of each DSL. More information on use of content-outline-page-related classes can be found in [DFK$^+$05].

Preference

The class `LDTMainPreferencePage` is used to define the content of general preference (**Windows** >**Preferences ...**) page of this tool.

Utility

As its name says, `LDTColorProvider` gives color to keywords of a DSL and is used by classes that scan the text for these keywords and by class `ReporterConsole`.

Console

The class `ReporterConsole` used the above mentioned interface `IReport` to report different kind of messages (e.g. error message, warning message) by making use of a `MessageConsole` instance of Eclipse. `MessageConsole` is a console that used in Eclipse to display task-related messages.

Parse Tree

The package `org.eclipse.gymnast.runtime.ui.views.parsetree` contains the following classes:

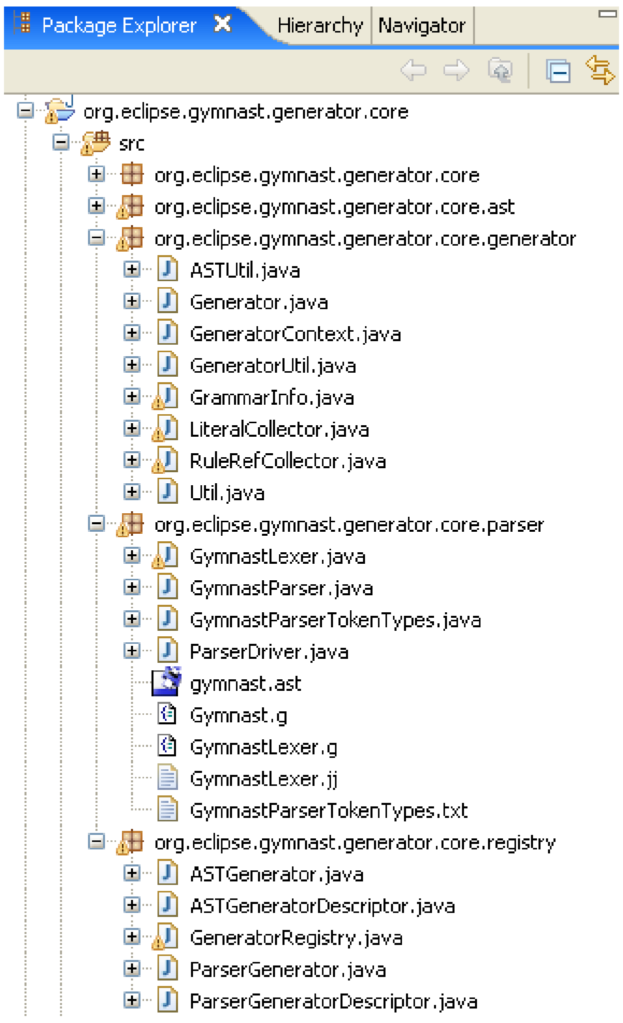
- `IParseTreeChangedListener`

- `IParseTreeViewInput`
- `ParseTreeLabelProvider`
- `ParseTreeView`
- `ParseTreeViewerContentProvider`

`ParseTreeView` defines a view that is used to display the abstract syntax tree which is resulted by parsing the text content of an editor. Other classes are implementation of the content and label provider, listener and input for this view. More information on classes needed for defining a view can be found in [DFK+05].

3.1.3 Generator - Core

The generator.core package contains classes which provide the generation mechanisms of Gymnast. Figure 3.4 shows the content of the package.



```
org.eclipse.gymnast.generator.core
  src
    org.eclipse.gymnast.generator.core
    org.eclipse.gymnast.generator.core.ast
    org.eclipse.gymnast.generator.core.generator
      ASTUtil.java
      Generator.java
      GeneratorContext.java
      GeneratorUtil.java
      GrammarInfo.java
      LiteralCollector.java
      RuleRefCollector.java
      Util.java
    org.eclipse.gymnast.generator.core.parser
      GymnastLexer.java
      GymnastParser.java
      GymnastParserTokenTypes.java
      ParserDriver.java
      gymnast.ast
      Gymnast.g
      GymnastLexer.g
      GymnastLexer.jj
      GymnastParserTokenTypes.txt
    org.eclipse.gymnast.generator.core.registry
      ASTGenerator.java
      ASTGeneratorDescriptor.java
      GeneratorRegistry.java
      ParserGenerator.java
      ParserGeneratorDescriptor.java
```

Figure 3.4: Classes of Generator - Core

AST

Content of this package is representing the nodes of the concrete syntax tree of the DSL (i.e Gymnast) which are generated based on its grammar definition, a visitor implementation for visiting those classes and a class representing tokens being used in the grammar. As mentioned before, the classes are derived from the `ASTNodeImpl` class. <LanguageName>`ASTNode`, which is directly derived from `ASTNodeImpl`, is the base class of other classes representing the node classes. The hierarchy between classes are formed in accordance with the defined grammar, e.g. the `abstract` rule of Gymnast results in generation of an abstract parent class with its children class(es).

Generator

The package `org.eclipse.gymnast.generator.core.generator` contains the following classes:

- `ASTUtil`
- `Generator`
- `GeneratorContext`
- `GeneratorUtil`
- `GrammarInfo`
- `LiteralCollector`
- `RuleRefCollector`
- `Util`

`ASTUtil` provides methods to inspect rules and expressions that are defined as Gymnast grammar (i.e. rules written in `.ast` file). Inspection is done by checking the name and the type of the rules.

`Generator` class plays the main role in generation mechanism. It has the method `generate()`, which in turn uses an instance of `GeneratorContext` (defining the context of generation, e.g. timestamp of generation, IDs of AST and parser generator being used, packages' names to be used, lexer, visitor, token class to be used, etc.) , `GeneratorUtil` (defining some helper methods to be used within the generation process, e.g. name setter, package creator, string manipluation by using the class `Util`, etc.) and `GrammarInfo` (an extension of `GymnastASTNodeVisitor` that provides some helper methods to process the Gymnast grammar) to execute the code generation process.

Both classes, `LiteralCollector` and `RuleRefCollector`, are derivation of `GymnastASTNodeVisitor`, which are used to collect keyword literals and rules respectively that are defined within a grammar file. Collected literals and rules are used in the generation of node classes. Additionaly, generating parser also needs the knowledge of the existing rules to correctly parse the text.

Parser

The package `org.eclipse.gymnast.generator.core.parser` contains the following classes:

- `GymnastLexer`
- `GymnastParser`
- `GymnastParserTokenTypes`
- `ParserDriver`

This package contains the classes which are needed for the parsing. This includes the grammar file(s), generated parser, lexer and the list of useable tokens, and a parser driver, which implements the `parse()`-method of interface `IParser`. Thus, a parse driver is responsible for the activation of parsing-related classes, e.g. parser and lexer class, and other actions, e.g. creating parse error messages. At this current stand, the parser classes of Gymnast are generated by using ANTLR.

Registry

The package `org.eclipse.gymnast.generator.core.registry` contains the following classes:

- `ASTGenerator`
- `ASTGeneratorDescriptor`
- `GeneratorRegistry`
- `ParserGenerator`
- `ParserGeneratorDescriptor`

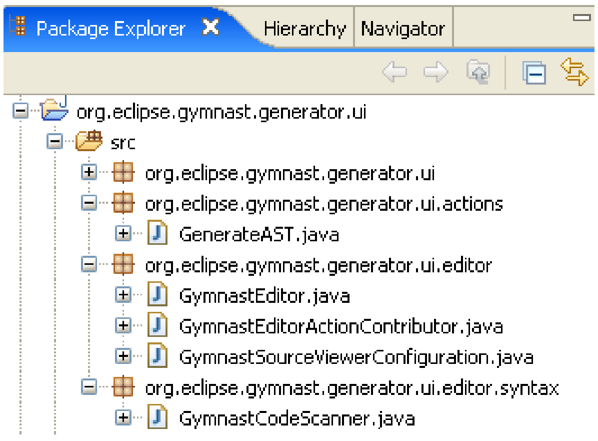
The package `Registry` is responsible for the management of the AST generator(s) and parser generator(s) being used. For this purpose, the class `GeneratorRegistry` has the task to initialize the generators based on the IDs given to it, keep the information about existing generators and its descriptor using maps, and deliver the information on enquiry. While each implementation of parser generator that is used by Gymnast needs to extend the abstract class `ParserGenerator`, creating an AST generator means extending `ASTGenrator`. Every generator owns a descriptor that gives information (e.g. name, description) about the generator and through which the generator can be fetched.

3.1.4 Generator - UI

Everything, which is related with the user interface of Gymnast, is implemented by classes contained in this package. Figure 3.5 shows the content of the package.

Action

The class `GenerateAST` defines the context menu **Generate AST**, which is made available and selectable on every Gymnast grammar file. An activation of this menu creates an instance of class `Generator`, which in turn triggers the initialization of AST and parser generator through its `generate()` method.



```
org.eclipse.gymnast.generator.ui
  src
    org.eclipse.gymnast.generator.ui
    org.eclipse.gymnast.generator.ui.actions
      GenerateAST.java
    org.eclipse.gymnast.generator.ui.editor
      GymnastEditor.java
      GymnastEditorActionContributor.java
      GymnastSourceViewerConfiguration.java
    org.eclipse.gymnast.generator.ui.editor.syntax
      GymnastCodeScanner.java
```

Figure 3.5: Classes of Generator - User Interface

Editor

The package `org.eclipse.gymnast.generator.ui.editor` contains the following classes:

- `GymnastEditor`
- `GymnastEditorActionContributor`
- `GymnastSourceViewerConfiguration`

This package contains the editor class, which is an extension of `LDTEditor`, and its `ActionContributor` and `SourceViewerConfiguration` classes. These classes don't define any additional methods compared to the `LDT`-classes.

Syntax

`GymnastCodeScanner` is an extension of `LDTCodeScanner`, which adds the keywords to be used in defining Gymnast grammar.

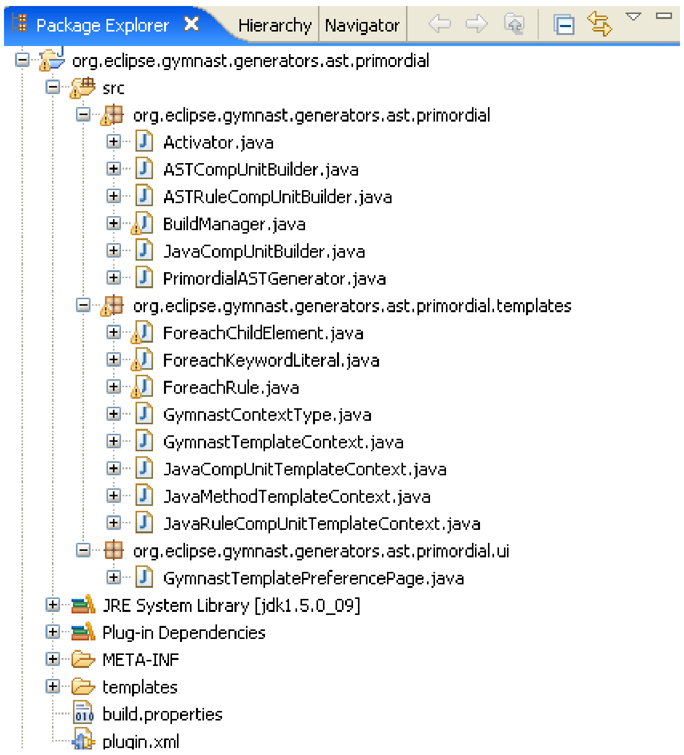
3.1.5 Extension Point: astGenerator

The plug-in `org.eclipse.gymnast.generator.core` provides 2 kinds of extension points: `parseGenerators` and `astGenerators`. Through these extension points, developers can provide their own generators. At this point of development, the AST Generator being used is shown in Figure 3.6.

The Package

The class `PrimordialASTGenerator` extends the abstract class `ASTGenerator`. It makes use of an instance of `BuildManager` to do the building task[1] by giving the actual context of generation. Through its method `createBuilders()`, a `BuildManager` add builders for the base class (i.e. class that extends `ASTNodeImpl` directly, is generally named after the language name concatenated with `ASTNode`), token class, and visitor class, each of which is kind of `ASTCompUnitBuilder`, a derivation of abstract class `JavaCompUnitBuilder`. The latter provides the method `build()` that produces

[1] Code generation based on predefined templates



```
org.eclipse.gymnast.generators.ast.primordial
  src
    org.eclipse.gymnast.generators.ast.primordial
      Activator.java
      ASTCompUnitBuilder.java
      ASTRuleCompUnitBuilder.java
      BuildManager.java
      JavaCompUnitBuilder.java
      PrimordialASTGenerator.java
    org.eclipse.gymnast.generators.ast.primordial.templates
      ForeachChildElement.java
      ForeachKeywordLiteral.java
      ForeachRule.java
      GymnastContextType.java
      GymnastTemplateContext.java
      JavaCompUnitTemplateContext.java
      JavaMethodTemplateContext.java
      JavaRuleCompUnitTemplateContext.java
    org.eclipse.gymnast.generators.ast.primordial.ui
      GymnastTemplatePreferencePage.java
  JRE System Library [jdk1.5.0_09]
  Plug-in Dependencies
  META-INF
  templates
  build.properties
  plugin.xml
```

Figure 3.6: Classes of ASTGenerator - Primordial

the class files. Another builder is the `ASTRuleCompUnitBuilder`, which extends `AST-CompUnitBuilder` and is responsible for determining the kind of the rules defined in grammar and building them (i.e. generating classes representing the rules. When the **Generate AST** option is chosen, the builders start the generation process.

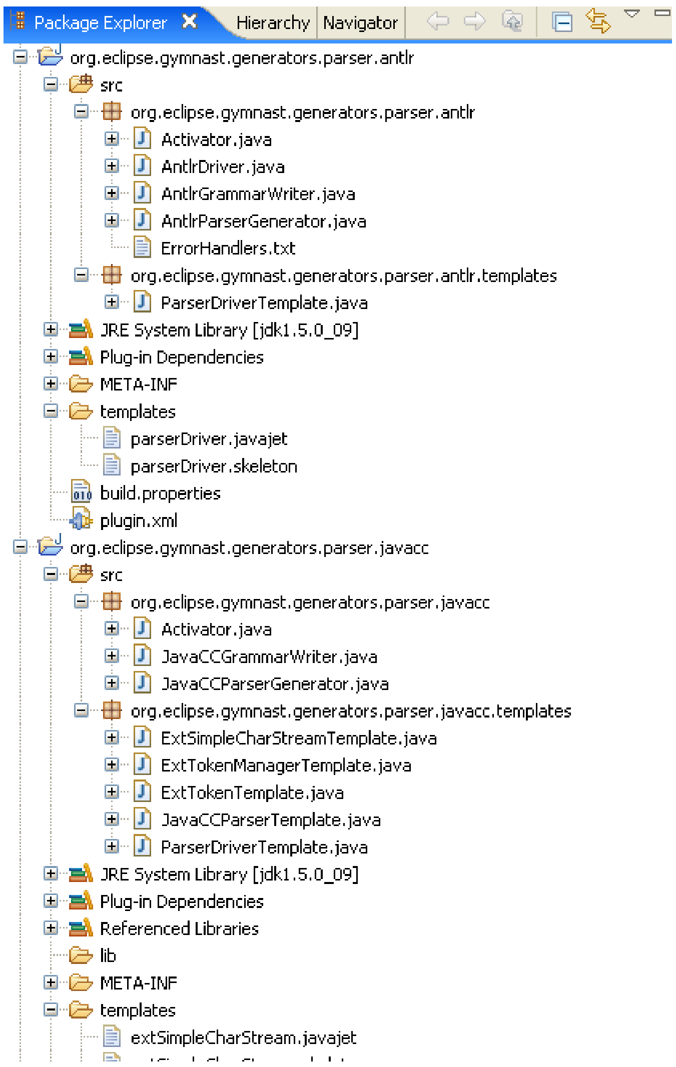
Templates

The pre-defined templates define the structure of the generated codes. The main classes in this package are `GymnastTemplateContext` and `GymnastContextType`, which extends `ExtTemplateContext` and `ExtContextType` respectively from the `template` package of `Runtime - Core`. `GymnastTemplateContext` provides some general name resolvers, e.g. for package name, base class name, etc. This class is extended by three other template context classes (`JavaCompUnitTextualContext`, `JavaRuleCom-pUnitTextualContent`, `JavaMethodTextualContent`), each of which defines some additional resolvers, e.g. for method name, type name, extension name, etc. All resolvers are initialized at the time the base class is created and this is done by calling `addResolvers()` method of `GymnastContextType`.

The `Foreach`-classes define further resolvers. `ForeachRule`-class is used to resolve the rules' name for the visiting purpose by the visitor class. Thus the visitor knows what kind of rules (i.e. CST nodes) it has to expect. `ForeachKeywordLiteral`-class is responsible for resolving the define keywords within the rules and non-keyword elements of a rule is resolved by `ForeachChildElement`-class.

3.1.6 Extension Point: parserGenerator

There are 3 parser generators provided to be used by Gymnast. For two of them, ANTLR and JavaCC, supporting implementation are already provided. Another parser generator in plan is LPG. Figure 3.7 shows the packages.



```
org.eclipse.gymnast.generators.parser.antlr
    src
        org.eclipse.gymnast.generators.parser.antlr
            Activator.java
            AntlrDriver.java
            AntlrGrammarWriter.java
            AntlrParserGenerator.java
            ErrorHandlers.txt
        org.eclipse.gymnast.generators.parser.antlr.templates
            ParserDriverTemplate.java
    JRE System Library [jdk1.5.0_09]
    Plug-in Dependencies
    META-INF
    templates
        parserDriver.javajet
        parserDriver.skeleton
    build.properties
    plugin.xml
org.eclipse.gymnast.generators.parser.javacc
    src
        org.eclipse.gymnast.generators.parser.javacc
            Activator.java
            JavaCCGrammarWriter.java
            JavaCCParserGenerator.java
        org.eclipse.gymnast.generators.parser.javacc.templates
            ExtSimpleCharStreamTemplate.java
            ExtTokenManagerTemplate.java
            ExtTokenTemplate.java
            JavaCCParserTemplate.java
            ParserDriverTemplate.java
    JRE System Library [jdk1.5.0_09]
    Plug-in Dependencies
    Referenced Libraries
    lib
    META-INF
    templates
        extSimpleCharStream.javajet
```

Figure 3.7: Classes of Parser Generator

Each parser generator supporting implementation, as it can be seen from Figure 3.7, has a parser generator class, e.g. `AntlrParserGenerator` which extends the class `ParserGenerator` of package `org.eclipse.gymnast.generator.core.registry`. A parser generator class makes use of a grammar writer class that outputs a grammar file, which in turn is used as an input for methods invoking the classes (i.e. part of the original libraries)of the parser generator being used. The invocation is implemented in e.g. `AntlrDriver`-class, or within the parser generator class itself (e.g. `JavaCCParserGenerator`.

As in code generation of the concrete syntax tree classes, templates are also used for the parser class, as well as for the parser driver class, as can be seen in package `javacc.templates` Furthermore, there are also templates in form of `JET` templates, which can be alternatively used for generating those classes. These can be found in folder `template`.

3.2 Design Patterns

In implementing the framework, some design patterns are used to ease the development process. Coming along those patterns, implementation of EMF classes also occupies certain patterns. Some of them are explained below:

3.2.1 Template Pattern

By defining a parent class with some of its methods left unimplemented, the template pattern is used. The parent class just provides the templates, which will be then used by the derived classes through their implementation of the methods in a specific way according to their needs. This is realized within the framework, as many classes are declared as `abstract` with some of its methods left unimplemented and are extended by several other classes, which then define their own usage of the methods.

3.2.2 Visitor Pattern

One of the generated classes is a visitor class, which is used to do processing on different node classes by visiting them. On the nodes, the processing is done within the visitor's method, e.g. `beginVisit()`, whose implementation depends on the kind of the visited node. This method is called after a node accepts the visit of the visitor through method `accept()`. Figure 3.8 shows the diagram of this pattern.

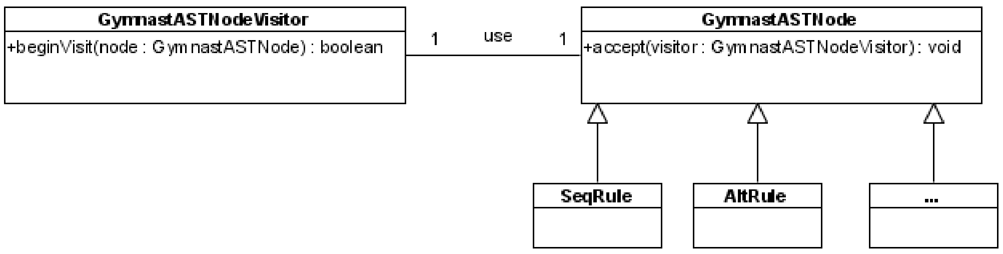

GymnastASTNodeVisitor
+beginVisit(node : GymnastASTNode) : boolean
1
use
1
GymnastASTNode
+accept(visitor : GymnastASTNodeVisitor) : void
SeqRule
AltRule
...

Figure 3.8: Visitor Pattern

3.2.3 Decorator Pattern

Using decorator pattern gives the flexibility to have additional functionalities only on certain subclasses. An `LDTEditor` is such a decorator class, which extends Eclipse `TextEditor`. It introduces some new methods, of which all its concrete derived classes make use. Those derived classes represents the editors for the different DSLs created using this framework. Figure 3.9 shows the diagram of this pattern.

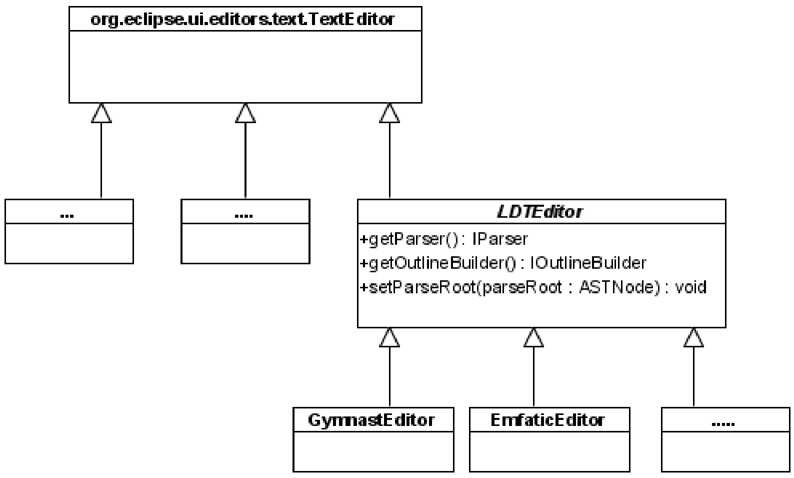

org.eclipse.ui.editors.text.TextEditor
...
....
LDTEditor
+getParser(): IParser
+getOutlineBuilder() : IOutlineBuilder
+setParseRoot(parseRoot : ASTNode) : void
GymnastEditor
EmfaticEditor
.....

Figure 3.9: Decorator Pattern

3.2.4 Factory Pattern

Factory pattern provides an interface for creating instances. Classes needed to get the instances call the interface method implementation provided by a concrete factory class. Eclipse EMF (and thus Eclipse UML2 Implementation) make intensive use of the factory pattern. It creates its model instances using factory instance, as shown in following snippet:

Listing 3.1: Factory Pattern in EMF

```
private void buildClass(ClassDecl classDecl, EPackage ePackage) {
 final EClass eClass = EcoreFactory.eINSTANCE.createEClass();
 ...
 ...
 EAttribute eAttr = EcoreFactory.eINSTANCE.createEAttribute();
 ...
 ...
 EReference eRef = EcoreFactory.eINSTANCE.createEReference();
 ...
 ...
}
```

3.2.5 Observer Pattern

The components that build the user interface of the framework rely on the implemented observer pattern in Eclipse as well as own observer[2]. Observer pattern encapsulates the dependency between objects, so that a change happened on one object can be known by another object, and this reacts correspondingly. Such a dependency can be found often in the context of user interface, e.g. an editor, whose contents should also be displayed in another view. When the content (i.e. text) of the editor is changed, the content of the view should be changed as well. To be able to react on the change, this view must register itself as an observer (i.e. implements the observer interface) and the editor as event provider (i.e. add the corresponding listener). Eclipse provides different kinds of observers which are based on different kinds of events.

[2]Observer is also known as listener

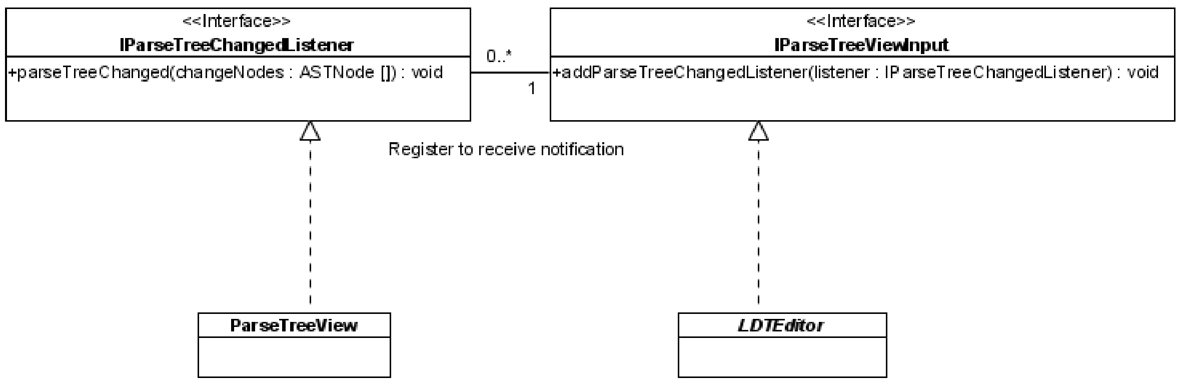

<<Interface>>
IParseTreeChangedListener
+parseTreeChanged(changeNodes : ASTNode []) : void
0..*
1
<<Interface>>
IParseTreeViewInput
+addParseTreeChangedListener(listener : IParseTreeChangedListener) : void
Register to receive notification
ParseTreeView
LDTEditor

Figure 3.10: Observer Pattern

Chapter 4

Textual Editor of UML2 State Machine : Implementation Phase

After the basic framework is described in the previous chapter, this chapter introduces an example of developing new DSL and its tool. Emfatic is the first DSL which has been developed by using Gymnast. The new DSL will focus on describing one of the UML2 subsets, namely State Machine.

4.1 Textual Notation of UML2 State Machine : The Grammar

The very first step in developing a new DSL with its tooling by using Gymnast is to define the grammar of the language. A Gymnast grammar file has an `.ast` extension. There are some rules on how to define such grammar, which have been shortly introduced in Chapter 2.

While Emfatic is the textual representation EMF Metamodel Ecore, the new DSL will describe a State Machine Metamodel of Eclipse UML2 Implementation. According to the explanation on Eclipse website, UML2 is an EMF-based implementation of the UML metamodel for the Eclipse platform. The objectives of this subcomponent are to provide a useable implementation of the metamodel to support the development of modeling tools, a common XMI schema to facilitate interchange of semantic models, test cases as a means of validating the specification, and validation rules as a means of defining and enforcing levels of compliance [Ecl07].

The base for defining grammar for language of State Machine is the State Machine Metamodel of Object Management Group [Gro04]. Figure 4.1 shows the metamodel which is taken from [Gro06]. A grammar file which more or less covers the important concepts of the State Machine is shown in Listing 4.1.

Listing 4.1: State Machine Grammar

```
language StateMachine;

options { k=3;
```

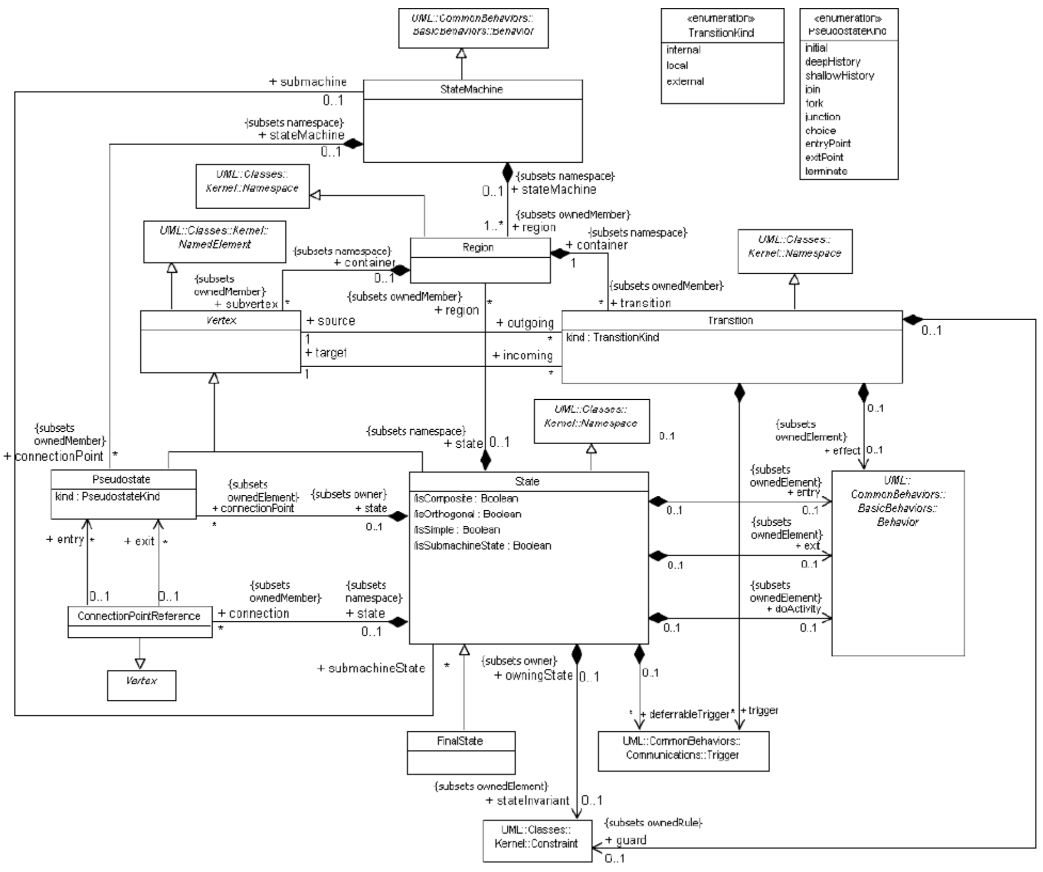

UML::CommonBehaviors::BasicBehaviors::Behavior
«enumeration» TransitionKind
internal
local
external
«enumeration» PseudostateKind
initial
deepHistory
shallowHistory
join
fork
junction
choice
entryPoint
exitPoint
terminate
StateMachine
+ submachine 0..1
{subsets namespace} + stateMachine 0..1
UML::Classes::Kernel::Namespace
UML::Classes::Kernel::NamedElement
{subsets namespace} + stateMachine 0..1
{subsets ownedMember} + region 1..*
Region
{subsets namespace} + container 0..1
{subsets namespace} + container 1
{subsets ownedMember} + subvertex *
{subsets ownedMember} + region *
{subsets ownedMember} + transition *
Vertex
+ source 1
+ target 1
+ outgoing *
+ incoming *
Transition
kind : TransitionKind
{subsets ownedMember} + connectionPoint *
{subsets namespace} + state 0..1
Pseudostate
kind : PseudostateKind
{subsets ownedElement} + connectionPoint *
{subsets owner} + state 0..1
State
/isComposite : Boolean
/isOrthogonal : Boolean
/isSimple : Boolean
/isSubmachineState : Boolean
{subsets ownedElement} + effect 0..1
{subsets ownedElement} + entry 0..1
{subsets ownedElement} + exit 0..1
{subsets ownedElement} + doActivity 0..1
UML::CommonBehaviors::BasicBehaviors::Behavior
+ entry *
+ exit *
ConnectionPointReference
{subsets ownedMember} + connection *
{subsets namespace} + state 0..1
Vertex
+ submachineState *
{subsets owner} + owningState 0..1
FinalState
+ deferrableTrigger *
+ trigger *
UML::CommonBehaviors::Communications::Trigger
{subsets ownedElement} + stateInvariant 0..1
UML::Classes::Kernel::Constraint
{subsets ownedRule} + guard 0..1

Figure 4.1: Metamodel of State machine

```
            parserPackageName="de.tuhh.sts.statemachine.core.parser";
            astPackageName="de.tuhh.sts.statemachine.core.ast";
            astBaseClassName="StateMachineASTNode"; }

sequence compUnit [entry] : modelDecl executionEventDecls
                            stateMachineDecls;

sequence modelDecl : "model" name=qualifiedID SEMI;

list executionEventDecls : executionEventDecl*;
sequence executionEventDecl : "execevent" name=ID SEMI;

list stateMachineDecls : stateMachineDecl* ;
sequence stateMachineDecl : "statemachine" name=ID
                            LCURLY regionDecls RCURLY;

list regionDecls : regionDecl* ;
sequence regionDecl : "region" name=ID
                      LCURLY inRegionDecls (finalStateDecl)? RCURLY;

list inRegionDecls : inRegionDecl* ;
abstract inRegionDecl : pseudoStateDecl | stateDecl | transitionDecl;

```

```
sequence pseudoStateDecl : "pseudostate" kind=pseudoStateKind
                           name=ID SEMI;

token pseudoStateKind : "initial" | "deepHistory" | "shallowHistory" |
                        "join" | "fork" | "junction" | "choice" |
                        "entryPoint" | "exitPoint" | "terminate";

sequence stateDecl : "state" name=ID LCURLY inStateDecl RCURLY;

sequence inStateDecl : (entryActivity=entryActivityDecl)?
                       (doActivity=doActivityDecl)?
                       (exitActivity=exitActivityDecl)?;

sequence transitionDecl : "transition" name=ID
                          LCURLY inTransitionDecl SEMI RCURLY ;


sequence inTransitionDecl : triggerDecl
                            "source" referencedSource=qualifiedID SEMI
                            "target" referencedTarget=qualifiedID ;

sequence triggerDecl : "trigger" name=ID
                       LPAREN "event" referencedEvent=qualifiedID
                       RPAREN SEMI;

sequence finalStateDecl : "finalstate" name=ID SEMI;

sequence entryActivityDecl : "entryact" activity=ID SEMI;
sequence doActivityDecl : "doact" activity=ID SEMI;
sequence exitActivityDecl : "exitact" activity=ID SEMI;


list qualifiedID : id1=ID (qidSeparator idn=ID)*;
token qidSeparator : DOT | DOLLAR ;
```

4.2 An Example of State Machine

To make use of the grammar, a simple example describing the states of an operating microwave will be implemented. The microwave and all its states and the transitions between and operation within them are shown in Figure 4.2.

A textual representation of a state machine, which fulfills the grammar described above, has an `.umlt` extension. The textual representation of the microwave model is shown in Listing 4.2.

Listing 4.2: Textual representation of the state machine model of a microwave

```
model Microwave;

execevent doorOpened;
execevent doorClosed;
execevent buttonPressed;
execevent timersTimesOut;
execevent eventCome;
execevent eventGo;

```

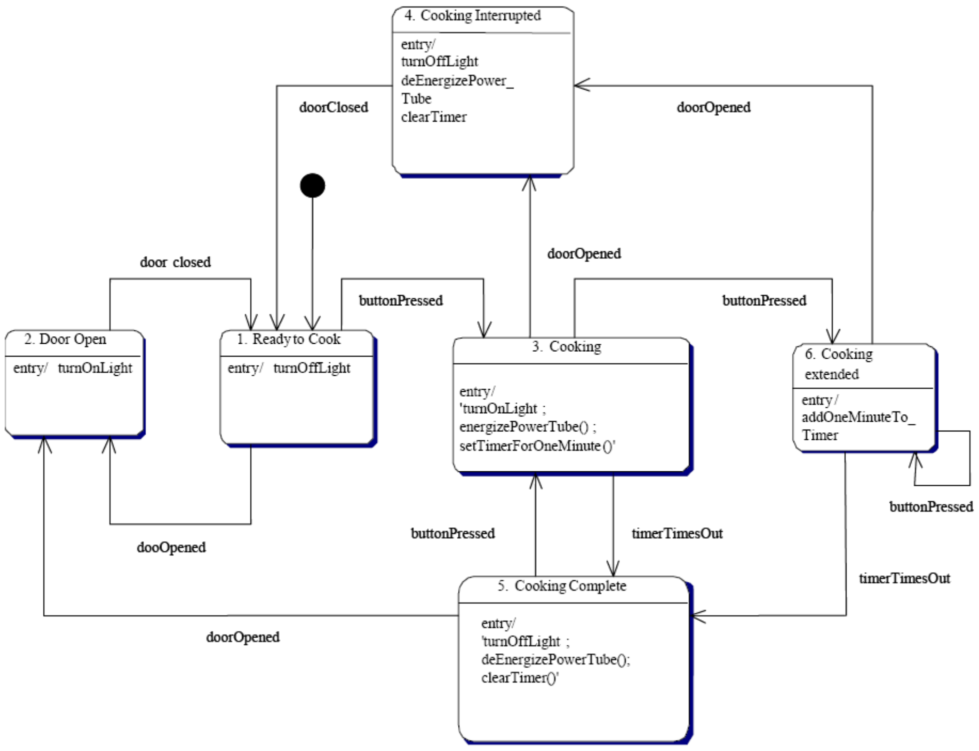


Figure 4.2: State Machine of Microwave

```
statemachine statesOfMicrowave {

region OperatingMicrowave {

pseudostate initial start;

state ReadyToCook_1 {
        entryact turnOffLight;
}

state DoorOpen_2 {
        entryact turnOnLight;
}

state Cooking_3 {
        entryact turnOnLight;
        doact energizePowerTube;
}

state CookingInterrupted_4 {
        entryact turnOffLight;
        doact deenergizePowerTube;
}

state CookingCompleted_5 {
        entryact turnOffLight;
        doact deenergizePowerTube;
}

state CookingExtended_6 {
        entryact addOneMinuteToTimer;
}

transition doorOpened1_2 {
        trigger pullDoor (event doorOpened);
        source ReadyToCook_1;
        target DoorOpen_2 ;
}

transition doorOpened5_2 {
        trigger pullDoor (event doorOpened);
        source CookingCompleted_5;
        target DoorOpen_2;
}

transition doorOpened3_4 {
        trigger pullDoor (event doorOpened);
        source Cooking_3;
        target CookingInterrupted_4;
}

transition doorOpened6_4 {
        trigger pullDoor (event doorOpened);
        source CookingExtended_6;
        target CookingInterrupted_4;
}

transition doorClosed2_1 {
        trigger pushDoor (event doorClosed);
        source DoorOpen_2;
        target ReadyToCook_1;
}

transition doorClosed4_1 {
        trigger pushDoor (event doorClosed);
        source CookingInterrupted_4;
        target ReadyToCook_1;
}

transition buttonPressed1_3 {
        trigger pressButton (event buttonPressed);
        source ReadyToCook_1;
        target Cooking_3;
}

transition buttonPressed6_6 {
        trigger pressButton (event buttonPressed);
        source CookingExtended_6;
        target CookingExtended_6;
}

transition buttonPressed5_3 {
        trigger pressButton (event buttonPressed);
        source CookingCompleted_5;
        target Cooking_3;
}
}
}
```

4.3 Building the Abstract Syntax Tree

Basically, classes used for implementing the framework for textual representation of UML2 State Machine model are almost the same as the classes being used to implement Emfatic. Nevertherless, the concrete syntax tree of the languages are different, although the generated classes are of the same sorts: base classes representing the nodes, visitor class, and token class. Because of the similarity, this section will present a part of Emfatic documentation which covers the core implementation, which is in fact a "superset" of the State Machine implementation.

This plug-in `org.eclipse.emf.emfatic.core` makes the core of Emfatic. It mainly defines the generation process from text to model and vice versa. Figure 4.3 shows all the packages in the Package Explorer of Eclipse.

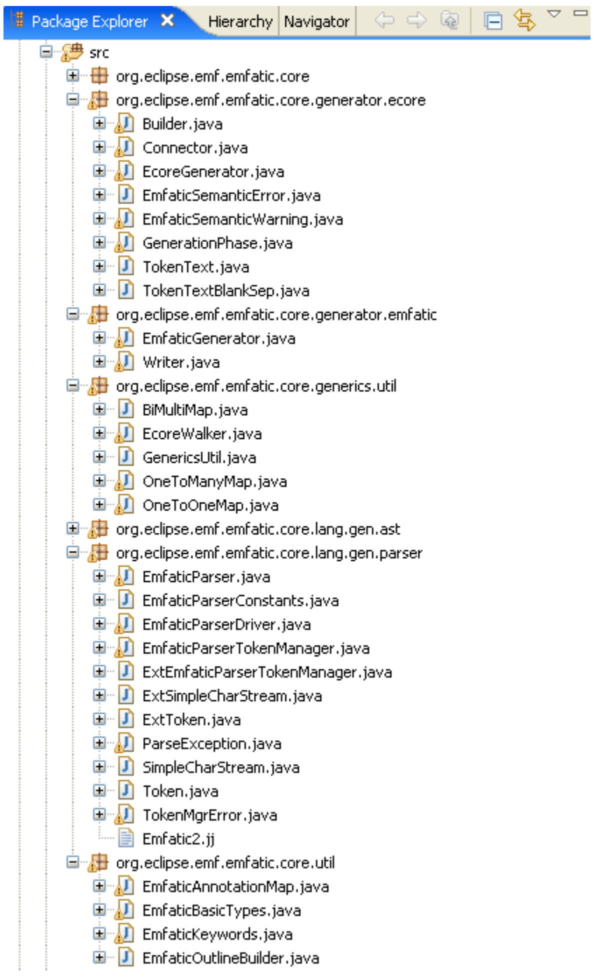
Ecore

The package `org.eclipse.emf.emfatic.core.generator.ecore` contains the following classes:

- `Builder`
- `Connector`
- `EcoreGenerator`
- `EmfaticSemanticError`
- `EmfaticSemanticWarning`
- `GenerationPhase`
- `TokenText`
- `TokenTextBlankSep`

Classes in this package are responsible for the generation of EMF Ecore model out of its text representation (i.e. Emfatic). `Builder` and `Connector` are 2 classes which instantiate the model and put the parts together based on the input text. Both classes are derived from `GenerationPhase` that defines methods to deal with the model being created. They are invoked through the class `EcoreGenerator`, which is activated when the context menu **Generate Ecore** is chosen on an Emfatic file, and creates the corresponding resource.

`EmfaticSemanticError` and `EmfaticSemanticWarning` are derivation of the above mentioned class `ParseError` and define different error and warning messages, which will be output if the corresponding condition occures. `TokenText` and `TokenText-BlankSep` are variations of the default visitor `EmfaticASTNodeVisitor` which specialize in getting certain part of text (token).



```
src
  org.eclipse.emf.emfatic.core
  org.eclipse.emf.emfatic.core.generator.ecore
    Builder.java
    Connector.java
    EcoreGenerator.java
    EmfaticSemanticError.java
    EmfaticSemanticWarning.java
    GenerationPhase.java
    TokenText.java
    TokenTextBlankSep.java
  org.eclipse.emf.emfatic.core.generator.emfatic
    EmfaticGenerator.java
    Writer.java
  org.eclipse.emf.emfatic.core.generics.util
    BiMultiMap.java
    EcoreWalker.java
    GenericsUtil.java
    OneToManyMap.java
    OneToOneMap.java
  org.eclipse.emf.emfatic.core.lang.gen.ast
  org.eclipse.emf.emfatic.core.lang.gen.parser
    EmfaticParser.java
    EmfaticParserConstants.java
    EmfaticParserDriver.java
    EmfaticParserTokenManager.java
    ExtEmfaticParserTokenManager.java
    ExtSimpleCharStream.java
    ExtToken.java
    ParseException.java
    SimpleCharStream.java
    Token.java
    TokenMgrError.java
    Emfatic2.jj
  org.eclipse.emf.emfatic.core.util
    EmfaticAnnotationMap.java
    EmfaticBasicTypes.java
    EmfaticKeywords.java
    EmfaticOutlineBuilder.java
```

Figure 4.3: Classes of Emfatic Core

Emfatic

The package `org.eclipse.emf.emfatic.core.generator.emfatic` contains the following classes:

- `EmfaticGenerator`
- `Writer`

The counter part of the `Ecore` package is `Emfatic` package, whose classes are responsible for generating the textual representation of an Ecore model. `EmfaticGenerator` uses the `Writer` class to generate the text out of an input model.

Generics Utility

The package `org.eclipse.emf.emfatic.core.generics.util` contains the following classes:

- `BiMultiMap`
- `EcoreWalker`

- `GenericsUtil`
- `OneToManyMap`
- `OneToOneMap`

`GenericsUtil` provides methods to support EMF Generics for both Ecore model and textual representation generation. In order to keep the connection between model declaration and its usages, two kinds of maps are defined: `OneToManyMap`, that maps a model declaration and its usage(s), and `OneToOneMap` that maps a certain usage of model its declaration. These maps are filled in by the time of model building and can be very useful for different features of the user interface, such as hyperlink. `BiMultiMap` combines both maps. An `EcoreWalker` navigates the full Ecore Abstract Syntax Tree (AST). It implements the `accept()` method of the Visitor pattern for an `EModelElement` object. In this way the `accept()` method need not be part of `EModelElement`.

AST

The `AST` package contains classes that are generated from the language grammar and represent the nodes of concrete syntax tree of the language.

Parser

The package `org.eclipse.emf.emfatic.core.lang.gen.parser` contains the following classes:

- `EmfaticParser`
- `EmfaticParserConstants`
- `EmfaticParserDriver`
- `EmfaticParserTokenManager`
- `ExtEmfaticParserTokenManager`
- `ExtSimpleCharStream`
- `ExtToken`
- `ParseException`
- `SimpleCharStream`
- `Token`
- `TokenMgrError`

All parsing-related classes in this package are generated by Gymnast using JavaCC parser generator. The default `ParserDriver` is generated and has been extended to do a pre-checking of errors on the abstract syntax tree. The generation of the classes are based on available templates used by the parser generator.

Utility

The package `org.eclipse.emf.emfatic.core.util` contains following classes:

- `EmfaticAnnotationMap`
- `EmfaticBasisTypes`
- `EmfaticKeywords`
- `EmfaticOutlineBuilder`

Some utility classes are defined, such as `EmfaticAnnotationMap` that keeps information about the annotations of model, `EmfaticBasicTypes` and `EmfaticKeyWords` which lists the basic types of Ecore and the usable Emfatic keywords respectively, and `EmfaticOutlineBuilder` that implements the interface `IOutlineBuilder`.

4.4 Implementation of the Enhanced Features of Text Editor

Just like the core implementation, the additional features of text editor for UML2 textual representation are also "subset" of the ones implemented for Emfatic editor. Again, Emfatic documentation on User Interface improvements will be presented.

The main motivation behind the User Interface improvements was adding support for EMF Generics to the original Emfatic [Gar07]. Some of these features are enhanced outline view, mark occurrences, folding, hovers, live problem and warning markers, navigable hyperlinks, new file wizard, auto edits such as smart brace, user-provided templates and syntax-aware content assist.

The plug-in that is responsible for the User Interface (editor and views) and all its features is `org.eclipse.emf.emfatic.ui`. It contains classes, which are grouped by their functions into packages. Figure 4.4 shows the complete packages of the plug-in.

One class is not included in any of those packages (it is placed in the base package `org.eclipse.emf.emfatic.ui`, which can be seen on the figure), namely `EmfaticUIPlugin`, which extends `AbstractUIPlugin`. In every plug-in project such a class is created as default to allow every plug-in being integrated into Eclipse platform UI. For Emfatic UI this class needed to be extended by following functionalities:

- Instantiating instance representing listener for text resource change which causes error and is compensated by showing red squiggle for each error on the editor. This is done by calling the constructor of `EmfaticRedSquiggler`.
- Getting the plug-in's Emfatic template store through `getEmfaticTemplateStore()` method.
- Getting the plug-in's Latex context type registry through `getEmfaticContextTypeRegistry()` method.
- Displaying error message in error log (also used by project creation wizard) through `log()` method.
- Getting a(n) (cached) image for the plug-in from the icon-directory and its descriptor through `get(Cached)Image()` and `getImageDescriptor()`.

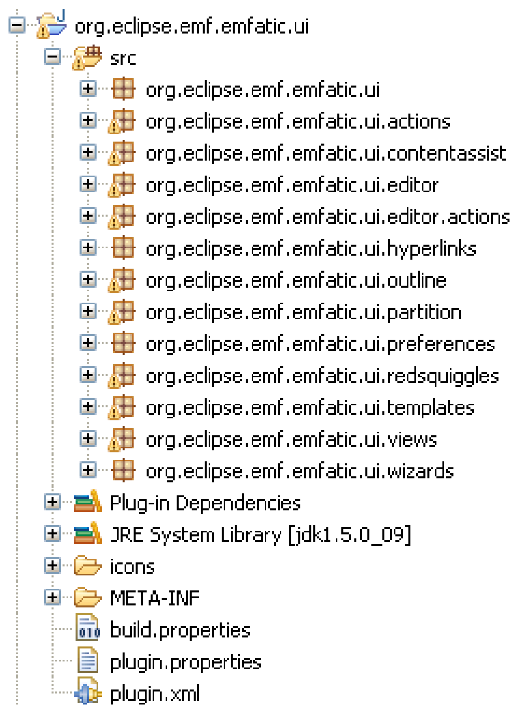

org.eclipse.emf.emfatic.ui
src
org.eclipse.emf.emfatic.ui
org.eclipse.emf.emfatic.ui.actions
org.eclipse.emf.emfatic.ui.contentassist
org.eclipse.emf.emfatic.ui.editor
org.eclipse.emf.emfatic.ui.editor.actions
org.eclipse.emf.emfatic.ui.hyperlinks
org.eclipse.emf.emfatic.ui.outline
org.eclipse.emf.emfatic.ui.partition
org.eclipse.emf.emfatic.ui.preferences
org.eclipse.emf.emfatic.ui.redsquiggles
org.eclipse.emf.emfatic.ui.templates
org.eclipse.emf.emfatic.ui.views
org.eclipse.emf.emfatic.ui.wizards
Plug-in Dependencies
JRE System Library [jdk1.5.0_09]
icons
META-INF
build.properties
plugin.properties
plugin.xml

Figure 4.4: Packages of Emfatic User Interface

The complete methods contained in EmfaticUIPlugin are shown in its outline page, as seen in Figure 4.5.

Following are the explanation on each of the packages.

Action

The package `org.eclipse.emf.emfatic.ui.actions` contains following classes:

- `GenerateEcore`
- `GenerateEmfatic`

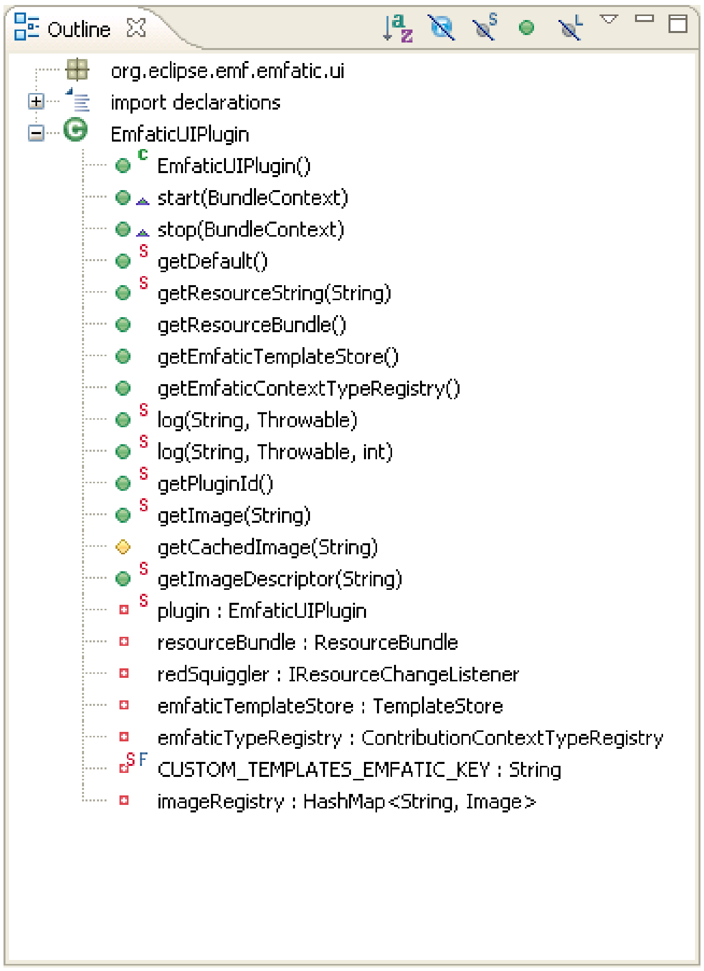
Both classes define context menus, which are visible by clicking right on the corresponding files, in this case `*.emf` and `*.ecore` respectively. Both classes are defined in `plugin.xml` as extension to `ui.popupMenus`-extension point.

The class `GenerateEcore` is left as its default implementation. In `GenerateEmfatic` there are some new methods introduced. ecoreValidate() uses the class `Diagnostician` as validity checker for basic EObject constraints. `HandleDiagnostic()`, `createMarkers()`, getMarkerID(), `deleteMarkers()` are methods used to set or remove markers on or from an Emfatic file, depending on how the validation's results look like

Content Assist

The package `org.eclipse.emf.emfatic.ui.contentassist` contains following classes:

- `CascadedContentAssistProcessor`


Outline

```
org.eclipse.emf.emfatic.ui
import declarations
EmfaticUIPlugin
    EmfaticUIPlugin()
    start(BundleContext)
    stop(BundleContext)
    getDefault()
    getResourceString(String)
    getResourceBundle()
    getEmfaticTemplateStore()
    getEmfaticContextTypeRegistry()
    log(String, Throwable)
    log(String, Throwable, int)
    getPluginId()
    getImage(String)
    getCachedImage(String)
    getImageDescriptor(String)
    plugin : EmfaticUIPlugin
    resourceBundle : ResourceBundle
    redSquiggler : IResourceChangeListener
    emfaticTemplateStore : TemplateStore
    emfaticTypeRegistry : ContributionContextTypeRegistry
    CUSTOM_TEMPLATES_EMFATIC_KEY : String
    imageRegistry : HashMap<String, Image>
```

Figure 4.5: Profile of `EmfaticUIPlugin` class

- `EmfaticContentAssistProcessor`
- `EmfaticKeywordContentAssistProcessor`
- `WordPartProcessor`
- `ProposalComparator`

A content assist processor proposes completion and computes context information for a particular context type. There are three classes of this kind, and each of them has to implement the interface `IContentAssistProcessor` of JFace content assist package. `CascadedContentAssistProcessor` includes both other processor (`EmfaticContentAssistProcessor` and `EmfaticKeywordContentAssistProcessor`) and computes the proposed completion. `ProposalComparator` and `WordPartProcessor` are two helper classes which compare the completion proposal (used to define the order in `sort()` function of Array) and detect word parts needed to be completed respectively.

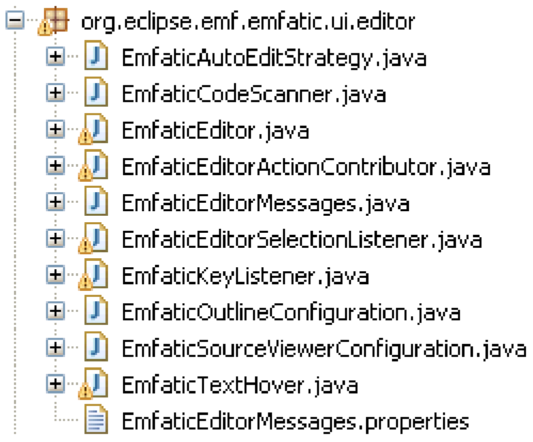

org.eclipse.emf.emfatic.ui.editor
EmfaticAutoEditStrategy.java
EmfaticCodeScanner.java
EmfaticEditor.java
EmfaticEditorActionContributor.java
EmfaticEditorMessages.java
EmfaticEditorSelectionListener.java
EmfaticKeyListener.java
EmfaticOutlineConfiguration.java
EmfaticSourceViewerConfiguration.java
EmfaticTextHover.java
EmfaticEditorMessages.properties

Figure 4.6: Classes of `emfatic.ui.editor`

Content assist is configured by overriding one of the methods in the source viewer configuration class of editor, namely the method `getContentAssistant()`. More information on implementing content assist feature can be found on [Zoi06].

Editor

The package `org.eclipse.emf.emfatic.ui.editor` contains following classes:

- `EmfaticAutoEditStrategy`
- `EmfaticCodeScanner`
- `EmfaticEditor`
- `EmfaticEditorActionContributor`
- `EmfaticEditorMessages`
- `EmfaticEditorSelectionListener`
- `EmfaticKeyListener`
- `EmfaticOutlineConfiguration`
- `EmfaticSourceViewerConfigurationtexttt`
- `EmfaticTextHover`

Figure 4.6 shows the classes of the editor package shown in Package Explorer of Eclipse.

The core of this package is the `EmfaticEditor` class, which extends `LDTEditor` of Gymnastic. This editor implements two interfaces, `IShowInTargetList` and `IShowInSource`, which enable the **Navigate** >**Show In** option for an active document. More information on this can be found in [DFK$^+$05]. A target view, which will show the location of this document after the Show-In choice, has to implement the corresponding `IShowInTarget` interface or adapt to this interface by overriding the `getAdapter()` method.

Initiating an `EmfaticEditor` consists of adding listener for changes on a parsed tree and document provider. Each editor must have a source viewer configuration in order to add customization, such as content assist, syntax highlighting, etc. Most of value added features of the editor are introduced within the class `EmfaticSource-Viewerconfiguration`. `OutlineConfiguration` class provides implementation to deliver a content outline of an opened editor.

For supporting folding feature in the editor, the methods `createPartControl()` of `TextEditor` and `createSourceViewer()` of `AbstractTextEditor` have to be overridden, in which a `ProjectionSupport` instance is installed on a `ProjectionViewer` and this viewer is defined as the source viewer respectively. Defining collapsible regions is done by the method `updateFoldingStructure()` which adds the `Projec-tionAnnotation` instance to the `ProjectionViewer` . A more detailed explanation on implementing folding feature can be found on [Dev05].

Further methods are defined to implement the hyperlink functionality within the editor. For this purpose, two maps are defined: a one-to-many map to keep the mapping between the declaration of an object (in its metamodel) and its concrete usages, and a one-to-one map for the mapping between the concrete declaration and its counterpart in the metamodel.

`EmfaticCodeScanner` is used for syntax highlighting. It defines different color application for different kinds of keyword. Defining syntax highlighting is enabled through the `getPresentationReconciler()` method of the source viewer configuration.

Hover popup will also appear over text. The class `EmfaticHoverText` is responsible for this feature by implementing the interface `ITextHover`.

Editor Actions

The package `org.eclipse.emf.emfatic.ui.editor.actions` contains following classes:

- `EmfaticEditorActionMessages`
- `OpenDeclarationAction`

`OpenDeclarationAction` class is needed for the context menu **Open Declaration**. Text, tool tip and declaration of this action are returned by class `EmfaticEd-itorActionMessages`.

Hyperlinks

The package `org.eclipse.emf.emfatic.ui.hyperlinks` contains following classes:

- `EmfaticHyperlink`
- `EmfaticHyperlinkDetector`

Hyperlink in text viewer is represented by the class `EmfaticHyperlink`. In order to find this hyperlink in a given region within the viewer, a hyperlink detector is being used. During the detection process a method `getLandingPlace()` is used to find out where the cursor should be moved to, based on the type of the CST node being hyperlinked.

Outline

The package `org.eclipse.emf.emfatic.ui.outline` contains following classes:

- `AnnFilterAction`
- `AnnotationFilter`
- `AttrFilter`
- `AttrFilterAction`
- `EmfaticContentOutlinePage`
- `OpFilter`
- `OpFilterAction`
- `RefFilter`
- `RefFilterAction`
- `TypeParamFilter`
- `TypeParamFilterAction`

`EmfaticContentOutlinePage` defines the outline page of a text viewer by extending the class `org.eclipse.ui.views.contentoutline.ContentOutlinePage` of Eclipse. It consists of different filters, which determine which elements may be shown in the outline viewer. Corresponding action classes are defined as well to toggle between activation and deactivation of the filter and are shown as button on top of the viewer. Context menu available to the outline page is for showing the type hierarchy of element being selected on the outline viewer. This is defined within the `populateContextMenu()` method. The `selectFromEditor()` method is responsible for showing the element on the outline viewer, which is chosen from an editor. For this purpose, a selection listener must be installed on the editor, that is, `EmfaticEditorSelectionListener` (from editor package).

Partition

The package `org.eclipse.emf.emfatic.ui.partition` contains following classes:

- `DebugPartitioner`
- `EmfaticDocumentProvider`
- `EmfaticPartitionScanner`
- `HTMLAnnotationHover`
- `HTMLPrinter`
- `NonMatchingRule`

Every JFace Text based editor is partition-aware, that is, its text content can be divided into non-overlapping regions of text. This is a feature which is important for the editor for supporting other additional features, such as error marking, content assist, etc. In `EmfaticDocumentProvider`, by extending `FileDocumentProvider`, creating document instance and loading documents from the file system abilities are made available.

To set the partition scheme for editor, the class DebugPartitioner is created, which is an instance of `IDocumentPartitioner` needed for partitioning and in turn uses an `EmfaticPartitionScanner` instance to define word scanning rules based on the different content type (one of which is a `NonMatchingRule` for undefined words) and find tokens corresponding to individual document partitions. More information on implementing partitioning can be found on [Zoi06].

Preferences

The package `org.eclipse.emf.emfatic.ui.preferences` contains following classes:

- `EmfaticPreferencePage`
- `EmfaticTemplatesPreferencePage`
- `PreferenceConstants`
- `PreferenceInitializer`

All classes of this package are used to define the preference page of Emfatic. Both `EmfaticPreferencePage` and `EmfaticTemplatePreferencePage` are extension to `org.eclipse.ui.preferencePage`-extension point. `EmfaticPreferencePage` is a class representing a general preference page contributed to the Preferences dialog. `EmfaticTemplatePreferencePage` is shown in the Preferences dialog as well, but used for defining templates for editor. To define the default value of the preference page at run time an extension to `org.eclipse.core.runtime.preference`-extension point must be made. And this is realized by `PreferenceInitializer`. `PreferenceConstants` define some constants for the plug-in preferences.

Red Squiggles

The package `org.eclipse.emf.emfatic.ui.redsquiggles` contains following classes:

- `EmfaticCSTChangeListener`
- `EmfaticRedSquiggler`
- `EmfaticRedSquigglerDeltaVisitor`

The main idea of these classes is to install resource change listener. This is done by implementing the interface `IResourceChangeListener`. A resource change event will be passed to the listener when a change occurs, and its delta (set of changes) is visited by `DeltaVisitor`, which processes all the changes. `EmfaticRedSquiggler` is a resource change listener and delta object of events passed to it will be visited by the `EmfaticRedSquigglerDeltaVisitor`. More information on resource change listener can be found on [Art02].

Templates

The package `org.eclipse.emf.emfatic.ui.templates` contains following classes:

- `EmfaticContextType`
- `EmfaticTemplateCompletionProcessor`

`EmfaticTemplateCompletionProcessor` is a kind of content assist processor[1] which takes a `TemplateContextType` class as its argument. A `TemplateContext-Type` defines a context within which templates are resolved. `EmfaticContextType` is derived from `TemplateContextType` and enables all possible variable resolvers defined by Eclipse.

Views

The package `org.eclipse.emf.emfatic.ui.views` contains following classes:

- `MethodsViewContentProvider`
- `MethodsViewLabelProvider`
- `TypeHierarchyMessages`
- `TypesView`
- `TypesViewContentProvider`
- `TypesViewDoubleClick`
- `TypesViewLabelProvider`

In relation with the Show-In feature of the editor, `TypesView` is a target view which can be chosen for the navigation purpose of a document. Moreover `TypesView` is used to show the type hierarchy within a document through `TypeViewer`. Another view (a table viewer) is used for showing the defined methods within a document, which is called `MethodViewer()`. Both of the viewers have its own content and label provider. An action class (`TypesViewDoubleClick`) is created to enable the double-click action on certain text element which leads to its declaration within the same document just as how the hyperlink feature works.

Wizards

The package `org.eclipse.emf.emfatic.ui.wizards` contains following classes:

- `EmfaticNewFileCreationPage`
- `EmfaticNewWizard`

In order to create new wizard, `org.eclipse.jface.wizard.Wizard` must be extended. Pages of the wizards, i.e. steps described in the wizard, are created by extending `org.eclipse.jface.wizard.WizardPage`. In order to get the wizard running, some methods of both classes have to be implemented properly. Just like other features, wizards are made available through extension point. The extension point for defining wizard is `org.eclipse.ui.newWizard`.

[1] Please refer to package describing content assist for more information

Chapter 5

Outlook

5.1 Summary

The objectives of this project work were to present some frameworks that deal with textual concrete syntax of DSLs, with the documentation of Gymnast as the main purpose, and to describe the use of Gymnast to develop DSLs and its tool (i.e. text editor) by representing Emfatic, a textual representation of EMF Ecore Model, and developing a new DSL for Eclipse implementation of UML2 State Machine model.

After introduction to the topic in Chapter 1 is given, Chapter 2 provides the overview of some frameworks that already exist at the time of writing. For each framework, the main concept has been introcuded and the architecture is described roughly to provide an insight into the framework. In Chapter 3, Gymnast framework is explained in detail. All packages and the contained classes are introduced by explaining their functionalities and how they are related to each other, to build the base of Gymnast.

On top of Gymnast, Emfatic and the textual representation for UML2 State Machine model are developed. These are discussed in Chapter 4. For every DSL development, a grammar must be defined based on the metamodel to be realized in textual representation. The State Machine metamodel and its corresponding grammar are introduced at the beginning of the chapter, before the implementation aspects of the framework is shown in the following sections. As the end result, UML2 State Machine models can now be described textually using the specific editor with enhanced features, whose implementation is derived from the Emfatic Editor.

At this phase, Gymnast only generates the classes which serve as infrastructure. Further steps (i.e. classes) which are needed to build the tools must be manually coded by DSL-developers. The manual coding demands additional knowledge from developers beside the domain-specific knowledge already owned, e.g. knowledge about EMF and UML2 in case of Emfatic and State Machine language, and knowledge about Eclipse API to build the user interface elements of IDE. A more preferable condition is to have a ready-to-use tool in hand, so that the developers can fully concentrate on defining the DSL.

5.2 Future Work

The idea for the future work, which was also mentioned in Chapter 1, is to fully develop Gymnast into an IDE generator. This idea has been proposed and accepted as part of Google Summer of Code (GSoC)[1], so that the concrete work will be progressing very soon. Technical aims of this work, as taken from the proposal for GSoc submitted by Mr. Miguel Garcia, is to extend Gymnast for:

- generation of EMF-compliant classes for concrete syntax tree nodes that enables well-formedness checking of CSTs at an earlier phase.
- generation of visitor for CST unparsing purpose which results in pretty-printed grammar
- generation of a text editor with all the additional features already coded manually

Bringing the ideas one step further, case studies will be undertaken to validate the approach in developing the IDE generator by applying the IDE generation process on further textual representation, e.g. an executable UML-style language but EMF-based, EJB3QL for metamodel, a human-readable notation for UML.

[1] http://code.google.com/soc/

Bibliography

[Art02] John Arthorne. How You've Changed! *Eclipse Corner Article*, August 2002.

[CFJL07] Philippe Charles, Robert M. Fuhrer, Stanley M. Sutton Jr., and Chris Laffra. SAFARI: A Meta-Tooling Platform for Creating Language-Specific IDEs. 2007.

[Coo04] S. Cook. Domain-Specific Modeling and Model-driven Architecture. *D. Frankel and J.Parodi (ed.), The MDA Journal: Model Driven Architecture Straight from The Masters*, December 2004.

[Dai05] Chris Daily. AST Framework Generation with Gymnast. 2005.

[Dev05] Prashant Deva. Folding in Eclipse Text Editors. *Eclipse Corner Article*, March 2005.

[DFK$^+$05] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer's Guide to Eclipse*. Addison Wesley, 2nd edition, 2005.

[ea96] R. Kieburtz et al. A Software Engineering Experiment om Software Component Generation. *Proc. of 18th IEE International Conference on Software Engineering*, 1996.

[Ecl07] Eclipse. Model Development Tools, 2007. http://www.eclipse.org/uml2.

[Ell97] C. Elliot. Modeling Interactive 3D and Multimedia Animation with an embedded Language. *Proc. of First USENIX Conference on Domain-specific Languages*, October 1997.

[FSGM05] Frédéric Fondement, Rémi Schnekenburger, Sébastien Gérard, and Pierre-Alain Muller. Metamodel-Aware textual Concrete Syntax Specification. 2005.

[Gar07] Miguel Garcia. Improvements To The Emfatic Editor. February 2007.

[Gro04] Object Management Group. UML Testing Profile, 2004. http://www.omg.org.

[Gro06] Object Management Group. *Unified Modeling Language : Superstructure*, April 2006.

[JB06] Frédéric Jouault and Jean Bézivin. On the Specification of Textual Syntaxes for Models. 2006.

[JBC$^+$06] Frédéric Jouault, Jean Bézivin, Charles Consel, Ivan Kurtev, and Fabien Latry. Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. 2006.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. 2006.

[JBKV06] Frédéric Jouault, Jean Bézivin, Ivan Kurtev, and P. Valduriez. Model-based DSL Frameworks. 2006. submitted for publication.

[KT00] S. Kelly and J. Tolvanen. Visual Domain-specific Modeling: Benefits and Experiences of Using MetaCASE Tools. *Proc. of International workshop on Model Engineering, ECOOP 2000*, 2000.

[Res] IBM Watson Research. http://domino.research.ibm.com/comm/research-projects.nsf/pages/safari.Introduction.html.

[Spi03] Diomidis Spinellis. On the Declarative Specification of Models. *IEEE Software*, 20(2):94–96, March, April 2003.

[TMC99] S. Thibault, R. Marlet, and C. Consel. Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation. *Software Engineering*, 25(3):363–377, 199.

[vDKJ00] A. van Deursen, P. Klint, and J.Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[Weg04] H. Wegener. Balancing Simplicity and Expressiveness: Designing Domain-specific Models for the Reinsurance Industry. *Proc. of the 4th OOPSLA Workshop on Domain-specific Modeling*, October 2004.

[Wil03] D. Wile. Lessons Learned from Real DSL Experiments. *Proc. of the 36th Hawaii International Conference on System Sciences*, 2003.

[WSTD05] H. Wada, J. Suzuki, S. Takada, and N. Doi. Leveraging Metamodeling and Attribute-Oriented Programming to Build a Model-driven Framewrok for Domain Specific Languages. 2005.

[WTH03] G. Wagner, S. Tabet, and H.Boley. MOF-RuleML: The Abstract Syntax of RuleML as a MOF Model. *Proc. of Integrate 2003*, October 2003.

[Zoi06] Phil Zoio. Building an Eclipse Text Editor with JFace Text. April 2006. http://www.realsolve.co.uk/site/tech/jface-text.php.