

Generation of Text Editors for Custom Domain Specific Language on the Eclipse Platform

Master Thesis
submitted by
Paulus Sentosa
Matr. Nr.: 22946

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

Hamburg University of Technology

Abstract

IDEalize is the result of further development of Gymnast; from an Eclipse-based framework that provides support for tooling of DSLs, mainly for grammar specification and parsing, to become a framework for generating text editors with several utility features. Further components have been implemented, each providing additional functionalities which together make up the extended framework. This master thesis presents the framework in detail with main focus on the generator, a use case of the framework, and a comparison with other existing and under-development prototypes of IDE generator.

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, October 25, 2007
Paulus Sentosa

I would like to thank Prof. Dr. Ralf Möller for providing a very interesting topic and giving me the opportunity to work on this topic for my master thesis.

Further I would like to thank Prof. Dr. Dieter Gollmann for his willingness to be the second assessor of this 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.

And my thanks also goes to A. Jibran Shidqie, M.Sc, who has given me a lot of useful ideas during my work.

Contents

1	Introduction	7
1.1	Background	7
1.2	Objectives	8
1.3	Document Structure	9
2	IDEalize: Framework for Generating Eclipse-based Text Editors	10
2.1	Introduction	10
2.2	Input Model for Generator	13
2.2.1	Modifying Gymnast Metamodel	15
2.2.1.1	Partition	15
2.2.1.2	Keyword Highlighting / Token Coloring Service	15
2.2.1.3	Text Folding Service	16
2.2.1.4	Range Highlighting Service	16
2.2.2	.idegemodel Editor	17
2.3	The Generator	18
2.3.1	New Project Creation Wizard	18
2.3.2	Resource Generator	20
2.3.2.1	Setting Up Information for Generator	21
2.3.2.2	Core Plugin Generator	21
2.3.2.3	.idegenmodel Generator	23
2.3.2.4	UI Generator	24
2.3.2.5	The JET Templates	25
2.4	Usage of IDEalize	29
2.4.1	Using The Wizard	29
2.4.2	Generating and Modifying .idegenmodel	31
2.4.2.1	Partitioning	35
2.4.2.2	Token Coloring Service	35
2.4.2.3	Text Folding Service	35
2.4.2.4	Range Highlighting Service	36
2.4.3	Generating User Interface plug-in	36
3	Use Case: Text Editor for State Chart Language	37
3.1	State Chart Language	37
3.1.1	The Grammar	37
3.1.2	The Constraint	41
3.2	An Example : State Chart of Telephone Object	43
3.3	Generated Components for Supporting the Application of Constraint	44
3.4	How to Progress	46
3.4.1	Generating .ecore and the converter class	46

3.4.2	Adding .ocl file using the same name as the model in the same folder	46
3.4.3	Compiling the OCL statements	46
3.4.4	Generating the model codes out of OCL-augmented model	47
3.4.5	Importing the OCL plug-in	47
3.5	Some Screenshots	49
4	Other IDE Generators	54
4.1	xText	54
4.1.1	Introduction	54
4.1.2	How It Works	54
4.1.3	The Components	56
4.2	Textual Editing Framework (TEF)	56
4.2.1	Introduction	56
4.2.2	How It Works	57
5	Outlook	58
5.1	Summary	58
5.2	Future Work	59
A	State Chart Language	63
A.1	Grammar	63
A.2	Textual Ecore-based State Chart Grammar	64
A.3	Textual Representation of State Chart of Telephone Object	66

List of Figures

2.1	Components of IDEalize	10
2.2	Plug-in <code>org.eclipse.idealize.grammar2ecore</code>	11
2.3	Gymnast Metamodel	14
2.4	Additional Classes in Gymnast Metamodel	17
2.5	Gymnast Model Editor plug-ins	18
2.6	Plug-in for IDE Generation Project	18
2.7	Class Diagram Showing the Main Components of Generator	19
2.8	Extension Point <code>org.eclipse.ui.newWizards</code>	19
2.9	Core Wizard Classes	19
2.10	Code Generator for Core Plug-in, <code>.idegenmodel</code> and User Interface	20
2.11	JET Template Translation Process	20
2.12	<code>ResourceGenerator</code> -Class	21
2.13	<code>GenerationSetup</code> -Class	22
2.14	Templates Folder	26
2.15	Wizard for Generating Core Plug-in Project of Custom DSL	30
2.16	Filling in The Form	31
2.17	Resulting Core Plug-in	31
2.18	Resulting <code>.ast</code>	32
2.19	How to Associate Gymnast Model Editor to File with <code>.idegenmodel</code> Extension	33
2.20	Available Additional Features	34
2.21	Partitioning	35
2.22	Token Coloring Service	35
2.23	Text Folding Service	36
2.24	Range Highlighting Service	36
3.1	UML2 State Chart (State Machine)	38
3.2	Graphical Ecore-based State Chart Grammar	40
3.3	State Chart of Simple Telephone Object	43
3.4	Grammar Folder Structure and the OCL-Compiler Context Menu	47
3.5	OCL Compiler Preference Page	48
3.6	The Generated Plug-in Containing OCL-augmented Model Codes	49
3.7	Main Editor with Custom Syntax Highlighting	49
3.8	Content Assist	50
3.9	<code>AutoEditStrategy: SmartBrace</code>	50
3.10	Folding	50
3.11	Folded Text with Hover Showing The Content	51
3.12	Range Highlighting and Mark Occurrences	51
3.13	Matching Brackets and The corresponding Preference Page	52
3.14	Syntax Error Marker and Message on Editor	52

3.15	Validation Error Marker and Message on Editor	52
3.16	Outline View synchronized with Editor	53
4.1	Overview Diagram of oAW	55
4.2	Reconciliation Process in TEF	57
4.3	Template Classes in TEF	57

Chapter 1

Introduction

1.1 Background

Domain-specific languages(DSL), which are specifically designed to be used in their own domains, are gaining more and more preference in software development process. Because of their higher abstraction level, which can be traced back to the fact that DSLs follow the domain abstractions and semantic[Tol04], a DSL not only gives the same experience of working directly with the domain concepts to developers, but it also allows the rules of domain being applied into the language as constraints. Having these constrains enhances the well-formedness level of the language. Furthermore, the close alignment of language and problem domain, which contributes to more readable and understable specifications, thus easier to communicate with, results in better productivity.

The implementation of DSL has been mostly influenced by a kind of approach, which many developers tend to use in recent days, namely the model-driven approach. Having models to define concepts of a certain domain, since this is what model-driven is all about, the implementation will take advantages of the possibility to have conformation and validation checking on the model-based concepts. Abstract syntax of DSL can be defined in terms of metamodel, while its concrete syntax is represented through the association between the metamodel and its syntactic elements[JB06]. Both graphical and textual notation can be used in DSL. However the preference for using the latter may find its own justification[Spi03], on which the opt for considering the textual-based alternative for this project is based.

Despite the above-mentioned advantages of using DSLs, developers still have to cope with certain issues that come with the DSL itself, such as defining methodologies to derive DSL from the domain knowledge[TMC99]. One of the most important issues, which is the lack of appropriate supporting tools for DSL, has been partly solved through the existence of Gymnast[Dai05]. This framework provide supports for the definiton process of custom DSLs and the corresponding parsing infrastructure which applies to textual DSL. Moreover, Gymnast also provide an extension to Eclipse `JFace Text`, a User-Interface(UI) toolkit that comprises helper class to implement UI features, as part of the framework to help developers having a base for implementing full-featured text editor for DSLs on top of Gymnast. This extension is explained in Chapter 1 of [GS07] in detail.

However implementing a language-specific editor by providing specific exten-

sion to the framework for each DSL is nontrivial task. A solid know-how on the APIs of the framework and of Eclipse `JFace Text` is an indispensable requirement to be able to make progress. Given that some parts of the implementation are basically the same for every editor (e.g. base classes to be extended, interface methods to be implemented), repetitive implementation tasks are rather inevitable, if developers are to create editors for different languages. Still, certain adjustments are sometimes necessary to those parts, because of specific behaviours that should be implemented only in the editor for a specific DSL. These problems round up the other half of the issue which is not completely solved by `Gymnast` yet.

A common proposal to avoid sophisticated but partly repetitive coding tasks is to generate the codes automatically. A code generator would need an input (model), from which it gathers the required information for the code generation. The use of templates can be seen as an integral part of a generator. A generator should normally use the information from the input to make up the templates, which are left "incomplete" intentionally, so that the generated codes are variably depending on the input, thus universally applicable in the same context of usage.

Having the model-based specification of DSL as base for the generator, the model concepts can be mapped to resulting codes[Tol04]. In the simplest case, each modeling symbol produces certain fixed codes. But the generator is not restricted to producing codes only based on the information from the model. Instead it also has certain patterns of code generation which are preset in the templates. Combination of both sources determines how the resulting codes should look like. The implemented code generator for this project makes use of this characteristic exactly to accomplish the generation of the language-specific text editor. Using the predefined templates, information is extracted from the input, which is in this case the meta-model of the language, and embedded into the code generation process to produce the designated completely working codes.

1.2 Objectives

The project for this thesis is aiming at implementing a generator for Eclipse-based language-specific text editors with utility features, which users may be acquainted with from the default editor of Eclipse. Due to the fact that the generator should be built upon `Gymnast` with the significance of determining additional required components, the implementation steps should comprise configuring how the enhanced framework should be designed, and eventually implementing those missing components before merging them to a reliable framework.

Furthermore, a use case in generating text editor for State Chart Language will be given to illustrate how to make use of the framework. Besides all the common utility features, embedding the external OCL Compiler for model validation checking into the generated text editor will be made part of the use case. This serves the purpose of abiding the well-formedness of the language, the abstract syntax of which is defined as a metamodel.

Finally, by making some comparisons with existing prototypes of code generation framework users get the possibility to have a clear overview on what kind of tools are and will be on the market and how they differ from `IDEalize`.

This project has been carried out as part of Google Summer of Code 2007¹ under the supervision of Eclipse Foundation.

1.3 Document Structure

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

The `IDEalize` framework will be explained in detail in Chapter 2, which includes the design and implementation of the framework, with some references to other documents that cover certain parts of `IDEalize`.

Chapter 3 covers the complete use case of defining textual notation for State Chart and generating its text editor. A step-by-step explanation comprises: defining the grammar specification of the language, until getting the ready-to-use editor with all its features.

Some overview comparison with other code generation framework will be given in Chapter 4 which will be pointing out the main similarity and differences with `IDEalize`.

Chapter 5 will round up the discussion with some points of conclusion and possible future work.

¹<http://code.google.com/soc/2007/>

Chapter 2

IDEalize: Framework for Generating Eclipse-based Text Editors

This chapter describes the `IDEalize` framework, which is based on further development of `Gymnast` by putting additional components that contribute to the enhanced functionality, such as generating EMF-based resources and User Interface-related classes which make up the building blocks for a text editor. Having "implemented" the editor, the first step for a prospective development of a language-specific Integrated Development Environment (IDE) has been made. In further parts of this paper, the term "IDE" will be used mostly for referring to "text editor".

2.1 Introduction

`IDEalize` comprises components which are grouped in 5 different plug-ins. These are shown in Figure 2.1. Having a good understanding on how the underlying framework `Gymnast` works, which is explained in chapter 1: *Frameworks for text editors: JFace Text and Gymnast Runtime* of [GS07], gives a solid base to follow the discussion on `IDEalize`. This chapter provides a deeper look into the infrastructure of the IDE generation framework and how it works.

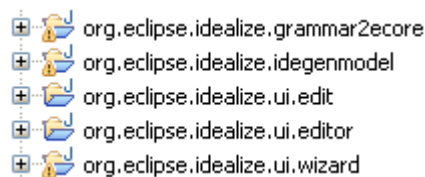


Figure 2.1: Components of `IDEalize`

The plug-in `org.eclipse.idealize.grammar2ecore` (Figure 2.2), which is referred before as `org.eclipse.idealize.generators.ast.ecore`, is covered in the chapter 2: *Generation of Parsing Infrastructure* from [GS07]. The following points are the most important things to understand about the plug-in `grammar2ecore`, before further explanation is given in the rest of this chapter.

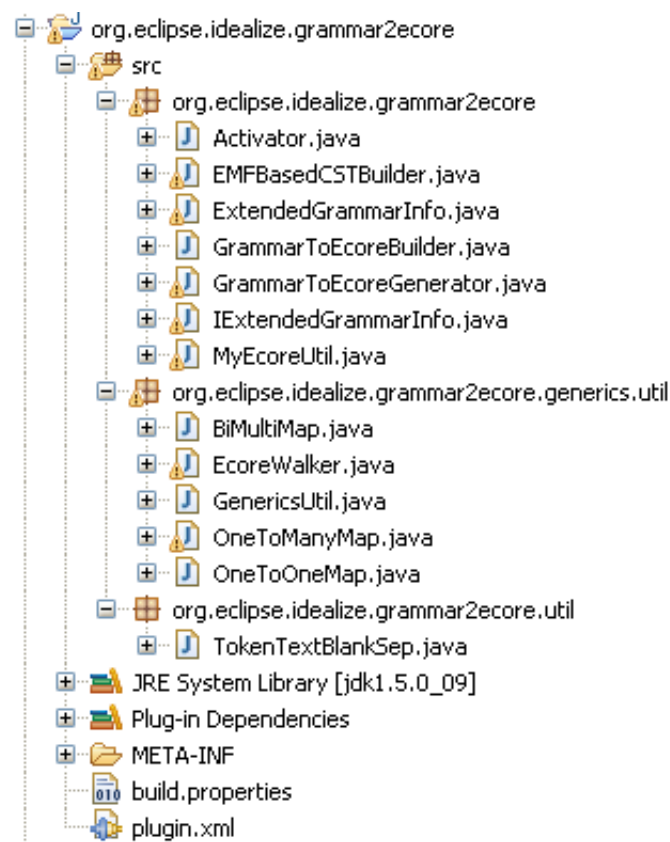


Figure 2.2: Plug-in `org.eclipse.idealize.grammar2ecore`

There are 2 basic steps which are inevitable for the IDE Generation. Firstly, the Grammar-to-Ecore transformation of Gymnast which provides base model for further customization needed to generate the IDE. And secondly, the generation of a POJO-based to EMF-based CST converter that will come in handy for the custom DSL-designers later to be able to specify how the IDE should be generated. Both steps are covered by the action **GenerateAST** from the context menu¹ of an `.ast` file with the option in the grammar specification set to:

```
astGenerator="org.eclipse.idealize.grammar2ecore"
```

The generated `.ecore` represents the AST of the custom DSL completely. But for generating a fully-featured IDE that supports the usage of this custom DSL, the current `.ecore` does not contain enough information, e.g the information on what keywords should have which coloring, which area within the editor showing certain language constructs are foldable, i.e. can be hidden to get a better overview of whole document, etc. are missing. And this is where the metamodel should be customized by adding some additional models that represent all the information above, i.e. models of the IDE features. Having set up the model, the corresponding EMF codes can now be generated; this comprises not only the codes of the model, but also of the graphical editor, with which instances of the model can be modified to complete the generated IDE with the desired features. Those instances are none other than the EMF-based CST of the custom DSL with the possibility for adding the features and they are serialized into a file with an extension `.idegenmodel`. The modification involves creating instances of the additional models that represent the IDE features and combining them with the existing instances of the language metamodel. The plug-ins `idegenmodel`, `edit` and `editor` are responsible for this activity.

All the components mentioned above make up the base, on which the generation process relies. For the process itself, a couple of code generators with some additional helpers are needed. Just like how an IDE is usually set up, the generated IDE should comprises core and user interface part, each of which is an own plug-in project. In between, another "generation" process is required, namely the generation of `.idegenmodel` file with an `.ast` file as its input (i.e. conversion from POJO-based to EMF-based CST of custom DSL). The `.idegenmodel` file will in turn serve as input for the IDE generator which results in the generation of the user interface part.

Throughout the different phases of generation process, *IDEalize* accomodates Eclipse M2T-JET technology² extensively, i.e. it makes use of JET code-templates (and its code generator technology) for every single file to be generated. The templates are basically oriented on the implementation classes of JFace Text and Gymnast Runtime explained in [GS07]. Depending on how the current likely-customized `.idegenmodel` of the custom DSL looks like, the templates use it as input model to generate the corresponding classes and files. Such an approach resembles the EMF code generation framework³. However, the requirement of such an input model exists only for the user interface part, since the core part has nothing to do with the model directly.

¹Eclipse Context Menu: a group of options, which is accessible by right-clicking on a certain file

²<http://www.eclipse.org/modeling/m2t/?project=jet#jet>

³<http://www.eclipse.org/modeling/emf/>

The core part, which is generated as result of using the `IDEalize` creation wizard, acts as an entry point to generation framework. By using the wizard the user is led into the first step of using the framework, in the way that a skeleton for a grammar file for the custom DSL is provided to be completed. This file is placed into a newly created `.core` plugin project named after the custom DSL. After completing the grammar, users should firstly progress just like they would when using `Gymnast`, namely to generate the POJO-based CST and the primordial parsing infrastructure, before they continue on executing further steps described above.

The IDE generation doesn't only mean the generation of classes implementing the editors and all its features, but it also involves the other parts of resource programming, such as creation of projects, folders, and non-Java files. Setting up a well-organized infrastructure of the generation process for different kind of resources makes a good base for an efficient process, and all this is covered by the `wizard` plug-in.

To summarize the steps required for providing infrastructure of IDE generator (steps described here are provided by `IDEalize` and EMF Framework):

- Generate EMF-based metamodel of `Gymnast` and POJO-EMF CST Converter
- Add new models to the metamodel that represent additional features
- Generate EMF-based model and the corresponding model codes and codes implementing the EMF editor
- Designer of custom DSL has to generate and customize `.idegenmodel` file as the result of the conversion from the custom `.ast` file using the EMF editor to provide an input and some additional information for IDE generator. The intermediate step for creating the custom `.ast` is done by making use of the provided creation wizard.

The next sections will give detail descriptions on the models and code generators which are required for the IDE generation process as a whole.

2.2 Input Model for Generator

From the introduction it is known, that the transformation from `Gymnast` grammar to its `.ecore` representation is necessary for IDE generation. Plug-in that is responsible for the transformation, the recipe behind and everything around the process are well documented in Chapter 2 of [GS07]. Using `gymnast.ast` as input, the corresponding `.ecore` file and additionally the forementioned CST converter are generated.

Figure 2.3 shows `gymnast.ecore`. Although `TopLevelClass` is not part of the grammar, it is also generated because the option `ecoreGenerateTopLevelClassForEMFEditor` is set to `true` due to the requirement explained before, namely as a "container" for adding children to the entry production. In Part 2 only some of the references contained in the container class `TopLevelClass` are shown due to lack of space. Except for `TopLevelClass`, `gymnast.ecore` contains the very same information as in the grammar file `gymnast.ast` itself.

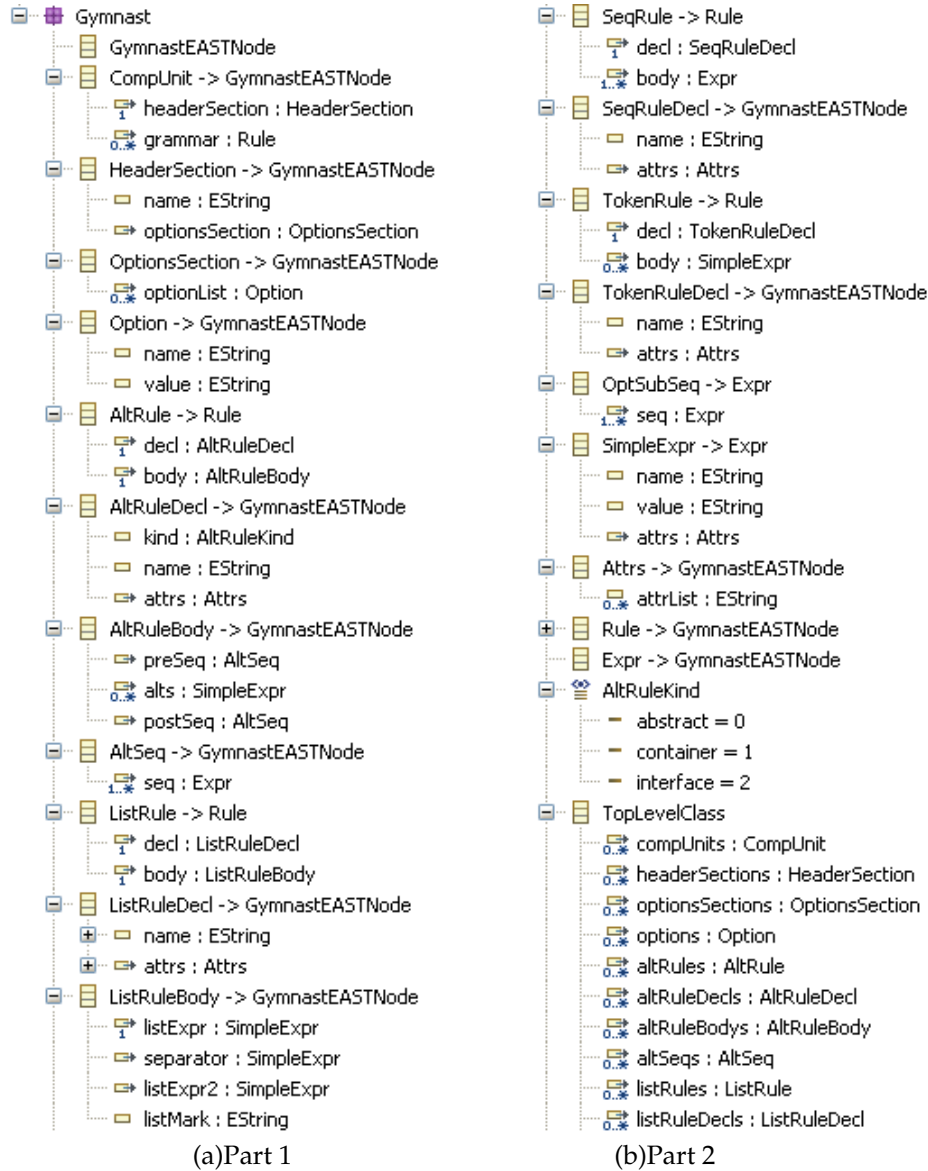


Figure 2.3: Gymnast Metamodel

2.2.1 Modifying Gymnast Metamodel

Due to the limitation of the Gymnast metamodel, it still can't be used to provide additional information for generating IDE. Therefore, a further modification is required by adding certain classes to the model that should serve as "placeholders" for the additional information. The number of such classes depends on what kind of information should be augmented into the model.

In Chapter 1 of [GS07] there are some code snippets showing how LDT-based text editor and its different features are implemented. The additional classes that will be integrated into the metamodel basically cover some of those implementation-related information, i.e. some features implementation can only be realized by fetching the information provided through the instantiation of those additional classes (which can be done by DSL designer by modifying the `.idegenmodel` file generated from the DSL grammar file). Thus, the resulting implementation is language-specific. In the next subsections, the classes are introduced in its Emfatic form, i.e. as snippets of `gymnast.emf` and grouped by the kind of features.

2.2.1.1 Partition

The concept of document partition is needed to implement certain features of the user interface. IDEalize provides implementation of the single default partition of type `IDocument.DEFAULT_CONTENT_TYPE`. But users will have the possibility to add custom partitions to meet their requirement on defining different areas within the document with specific behaviors. In order to be able to define partitions, following classes are introduced:

Listing 2.1: Partition-related Classes

```

1 class Partition {
2     val PartitionRule[+] rules;
3 } //No same partition rules
4
5 class PartitionRule {
6     attr String partitionName;
7     attr ScanningRule rule;
8     attr String startSequence;
9     attr String endSequence;
10 }
11
12 enum ScanningRule {
13     singleLine = 0;
14     multiLine = 1;
15     endOfLine = 2;
16     //(eventually self-created rule type, e.g. NonMatching-Rule)
17 }
```

Partitions are defined based on result of text scanning process. There are several scanning rules, each of which may have its start and end sequence. Completing the different rules makes up the partitions of a certain document recognized within the editor.

2.2.1.2 Keyword Highlighting / Token Coloring Service

An enumeration type holds the possible colors, which are already defined in class `LDTColorProvider` of Gymnast. The main class `TokenColoringService` is used to define the different coloring groups.

Listing 2.2: Token Coloring-related Classes

```

1
2 class TokenColoringService {
3     val TokenGroup[+] groups;
4 }
5
6 class TokenGroup {
7     attr String[1] name;
8     attr Color color;
9     ref SimpleExpr[+] listOfTokens;
10 }

```

Tokens/Keywords are grouped using different group name and each group has a certain color, e.g. normal keywords, special keywords, etc. Constraint that has to be applied for this class is that the list of tokens only contains instances of the production rule `simpleExp`, whose value starts with quotes `""` (every keyword starts with quotes, thus only tokens representing keywords can be given colors).

If no token coloring service is defined, the class extending `LDTCodeScanner` won't be generated. The editor's `SourceViewerConfiguration` that refers to the code scanner through its method `createCodeScanner()` will return null instead of instance of a code scanner. The `Keyword` class will simply contain the keywords, without specifying certain groups to which the keywords belong.

2.2.1.3 Text Folding Service

Listing 2.3: Text Folding-related Classes

```

1
2 class TextFoldingService {
3     ref Rule[+] rules;
4 }

```

Similar to partitioning, `IDEalize` also provides default implementation of text folding feature, which is assigned to every single language construct. However users may only want to fold certain areas. This can be done by defining the foldable areas. Requirement for this service is that the "foldable rules" are neither list nor token rules⁴. This user-defined text folding service is not additional to the default one, but rather an alternative, since the default won't be generated anymore as soon as class `TextFoldingService` is instantiated.

2.2.1.4 Range Highlighting Service

Listing 2.4: Range Highlighting-related Classes

```

1 class RangeHighlightingService {
2     ref Rule[+] rules;
3 }

```

⁴Rules are used in `Gymnast` for defining a grammar specification. More information on `Gymnast` rules can be found in [Dai05]

Similar to defining Text Folding Feature, user needs to choose some rules, whose range (i.e. lines over which the texts are located) should be highlighted on the editor side, when the cursor is currently placed within those area.

To implement the other features, such as Content Assist, Auto Edit, Annotation Hover, etc. no additional classes in the metamodel are needed.

The modified `gymnast.ecore` is shown in Figure 2.4.

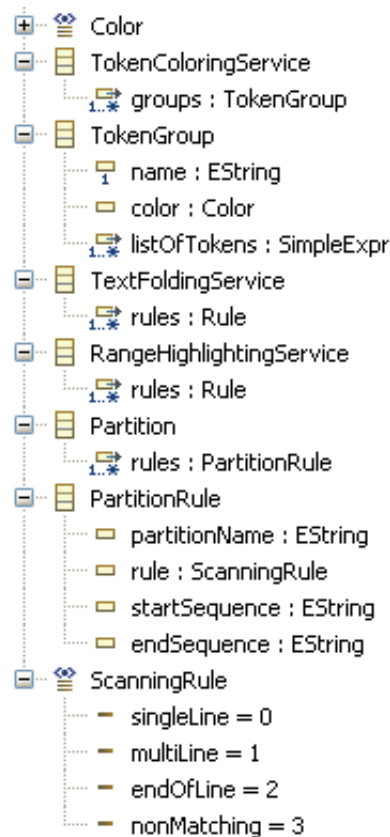


Figure 2.4: Additional Classes in Gymnast Metamodel

2.2.2 .idegenmodel Editor

Having modified `gymnast.ecore`, the next steps will be generating the model codes and its corresponding Edit and Editor plug-ins from the `gymnast.genmodel`. The generated EMF editor, usable on the second instance of Eclipse, is meant to be used for editing `.idegenmodel` of custom DSL. This is done by setting Gymnast Model Editor as default editor for any files with `.idegenmodel` extension. The plug-ins which are responsible for this Editor are shown in Figure 2.5. A further explanation on working with the editor will be given in Section 2.4 of this chapter.

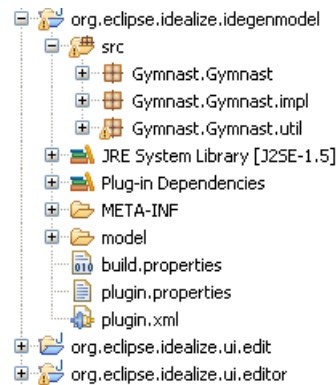


Figure 2.5: Gymnast Model Editor plug-ins

2.3 The Generator

This section deals with the classes which are responsible for the generation process. Those classes are parts of plug-in `org.eclipse.idealize.wizard` shown in Figure 2.6. Besides the different generators, the plug-in also includes a wizard implementation as a starting point of using the framework, which will provide the pilot project (i.e. core plug-in) for creating language-specific IDE. How the main components are related, is shown in the class diagram depicted in Figure 2.7, and the following subsections will give explanation on those classes shown in the diagram.

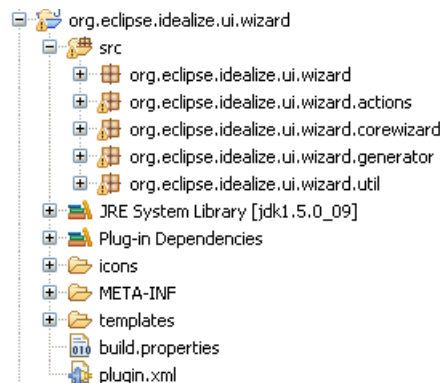


Figure 2.6: Plug-in for IDE Generation Project

2.3.1 New Project Creation Wizard

For the creation of wizard, the extension point `org.eclipse.ui.newWizards` should be extended (Figure 2.8). A category of wizard and the wizard name itself have to be defined. Figure 2.9 shows the package that is responsible for implementing the wizard.

By extending `org.eclipse.jface.wizard.Wizard`, a class called `NewCorePluginWizard` is the main wizard class and it also implements `INewWizard`. For the display of the wizard, an instance of `NewCorePluginWizardPage` is created, which basically provides fields and options that will be later set as initial content

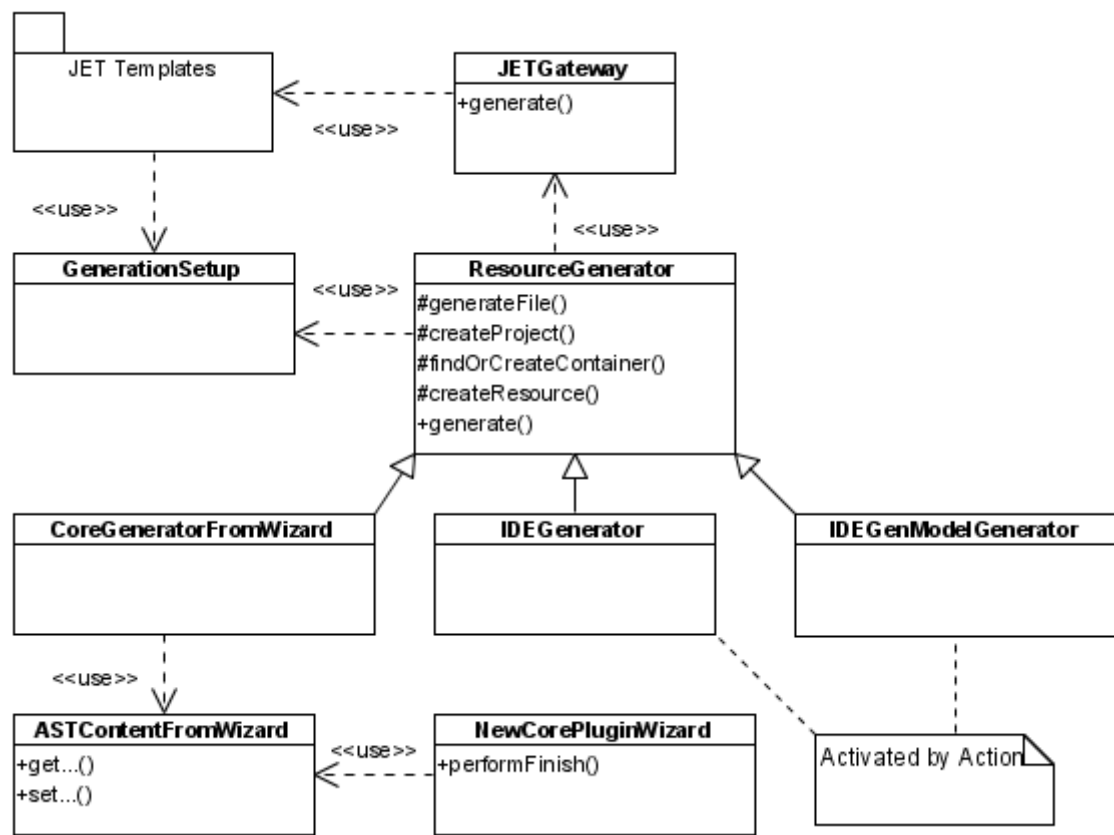


Figure 2.7: Class Diagram Showing the Main Components of Generator

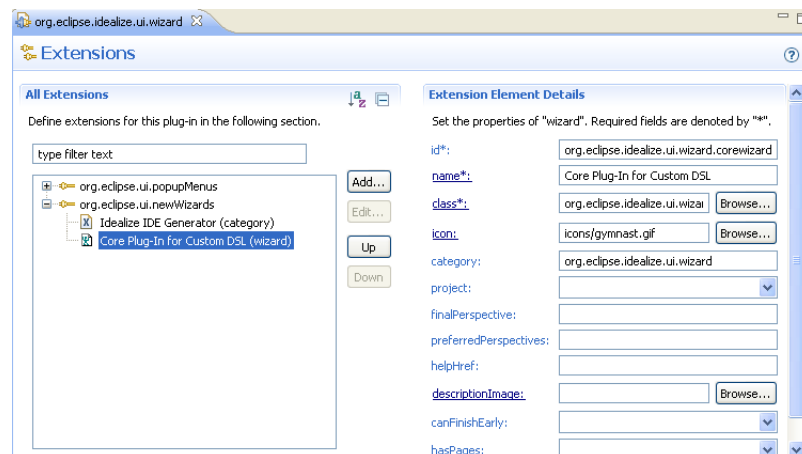
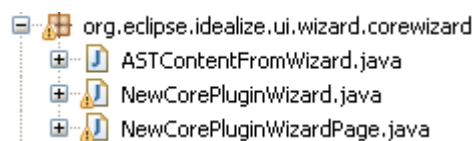
Figure 2.8: Extension Point `org.eclipse.ui.newWizards`

Figure 2.9: Core Wizard Classes

of the generated `.ast` file. `ASTContentFromWizard` serves as placeholder for the values input through the wizard page. As a result of the wizard, i.e. what is executed in the `performFinish()` method of `NewCorePluginWizard` when user activates the FINISH button of the wizard, is the instantiation of the `CorePluginGenerator` with the information kept by `ASTContentFromWizard` as arguments.

2.3.2 Resource Generator

Both packages `.generator` and `.util` are the core of this plug-in. They are shown in Figure 2.10.

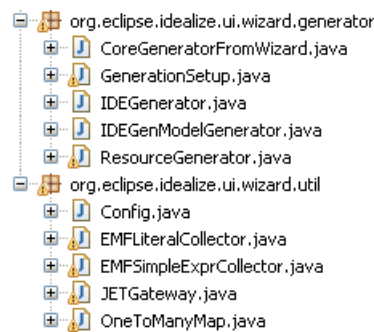


Figure 2.10: Code Generator for Core Plug-in, `.idegenmodel` and User Interface

The main technology behind the code generation framework used in `IDEalize` is Eclipse JET Technology[Fou07]. Among the important classes of JET, `JETEmitter` is the class, which the generator makes a direct use of. The idea is to use code templates which will be processed by `JETEmitter` (and several other JET classes) for each file or source code to be generated. Figure 2.11 shows a template translation process (image reproduced from [Pop04]). Within the templates, different kinds of JET tags should be used to dynamically generate codes in dependence on its input model, i.e. Java Objects. To ease the access to this technology, there are two classes that are introduced in [Pop04], which have been reused with slight modification to fit in the framework. Those classes are `Config` and `JETGateway`, with the latter calls the emitter class. Further explanation on JET technology, e.g. the translation process and its most important classes, and on both above-mentioned classes can be found in the article.

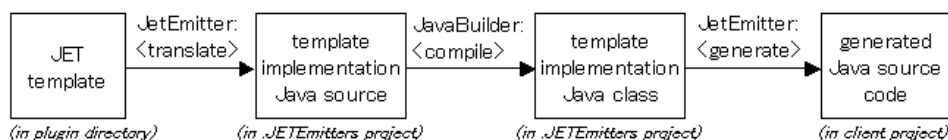


Figure 2.11: JET Template Translation Process

Using the generator framework, different kinds of resources are to be generated: besides the template-based non-Java and Java files, project and folders are also generated, where the generated files will reside. Due to the repetitive task of generating

resources, a class called `ResourceGenerator`, shown in Figure 2.12 provides different methods that deal with the generation process. A generator should extend this class and thereby override the method `generate()`. Depending on what the generator does, it may use the provided `generateFile()` for generating non-Java files, `generateJava` for generating Java codes, `createProject` for generating a plug-in project, and two helper methods, `getFilePath()` and `createResources`, that can be used to find the exact location of an input file (might be used if a generated file should be generated in the same location as its input) and to create a resource for holding an EMF serialization result (needed for `.idegenmodel`) respectively.

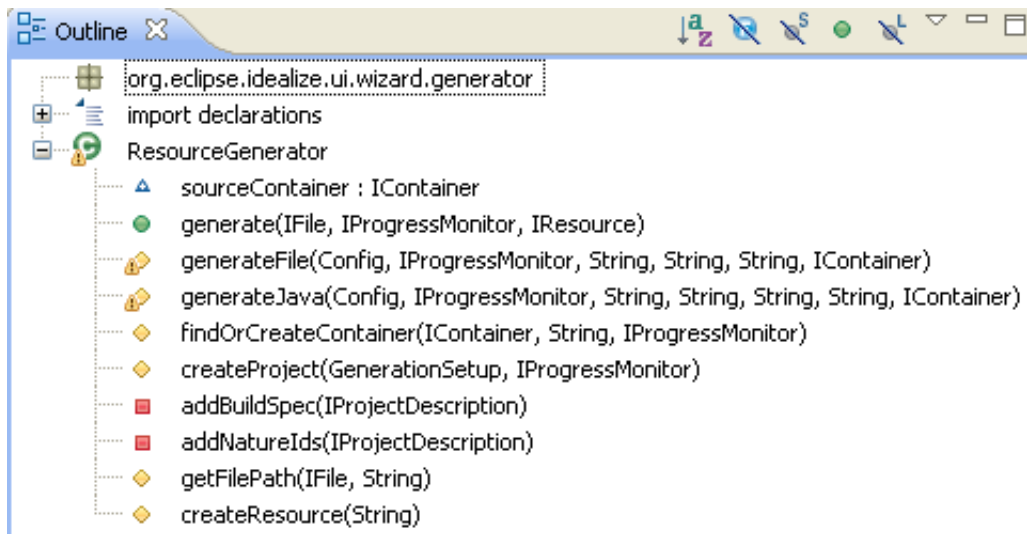


Figure 2.12: `ResourceGenerator`-Class

2.3.2.1 Setting Up Information for Generator

`GenerationSetup` (class members are shown in Figure 2.13) serves as centre for feeding the generators with the required information. Two constructors are defined; the first constructor which takes the name of DSL as argument, `GenerationSetup(String languageName)` is used by the Project Creation Wizard and needs no file input, while the second constructor takes files as its argument. The latter is used for generating `.idegenmodel` with `.ast` file as its input, as well as for generating the UI plug-in project with the `.idegenmodel` as information source.

As mentioned before, the code templates contain tags which will deliver content dynamically in dependence of the input. Those tags are basically referring to methods defined in `GenerationSetup`. Most of those methods are getter methods which returns some predefined names for packages and classes. By having them available on hand, the templates can simply use the names to complete certain parts of its codes, e.g. name of the generated class, imported classes with its qualified name, name of the package, etc.

2.3.2.2 Core Plugin Generator

Generating a core plug-in of an IDE for a custom DSL makes the first step of IDE generation. After getting the content from the wizard page, a `CoreGenerator`-

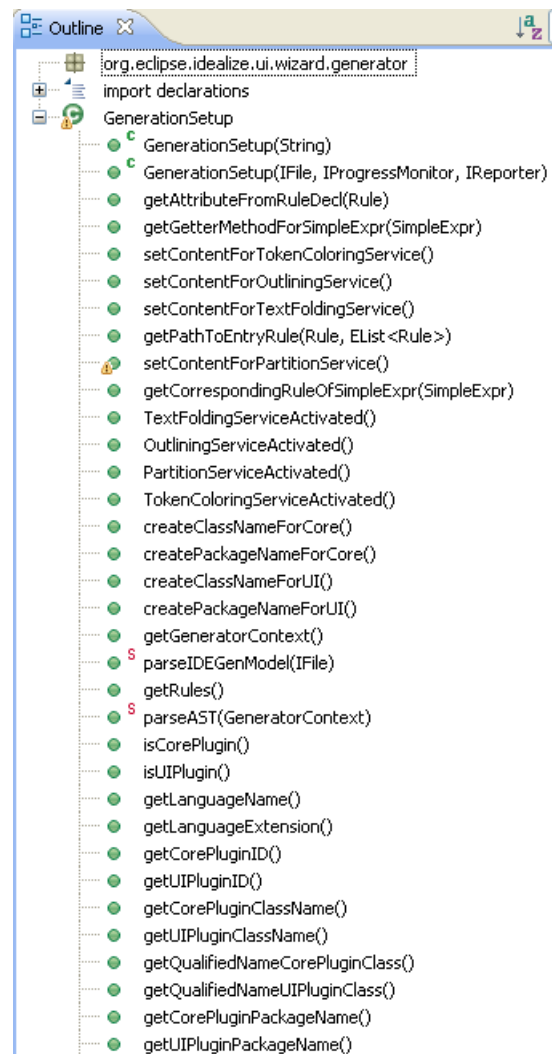


Figure 2.13: GenerationSetup-Class

FromWizard is instantiated. Consecutively a project, folders and some files are generated, having their names being set up within the `GenerationSetup` instance. The main result of the wizard besides the project itself is however a grammar file `<customDSLname>.ast`, with pre-filled options whose values come from the wizard page (i.e. instance of `ASTContentFromWizard`). User may now complete the grammar file of his/her custom DSL.

2.3.2.3 .idegenmodel Generator

For the generation of `.idegenmodel` the second constructor of `GenerationSetup` is used by passing the check on kind of the input file, i.e. extension of the file must be `.ast`. This constructor is called by the generator class `IDEGenModelGenerator` (Listing 2.6).

Listing 2.5: `GenerationSetup` for Generating `.idegenmodel`

```

1 public GenerationSetup(IFile file , IProgressMonitor monitor , IReporter
  reporter) {
2
3     try {
4         // Input is .ast file
5         if (file.getFileExtension().equalsIgnoreCase("ast")) {
6             context = new GeneratorContext(file , monitor , reporter);
7
8             compUnit = parseAST(context);
9             context.initCompUnit(compUnit);
10
11             grammarInfo = new GrammarInfo(context);
12             context.initGrammarInfo(grammarInfo);
13
14             languageName = grammarInfo.getLanguageName();
15             languageExtension = grammarInfo.getOptionValue("extension");
16
17         }
18         ....
19     }
20 }
21
```

Besides setting up some required information (e.g. language name, extension name, etc) the grammar file is parsed to get its entry rule (in Listing 2.5 the entry rule is `CompUnit`), which will be used by `POJO-to-EMF-based-Converter` as input for the conversion. The conversion itself is executed within the `generate()` method of `IDEGenModelGenerator`, as shown in following snippet:

Listing 2.6: `IDEGenModelGenerator`

```

1 public class IDEGenModelGenerator extends ResourceGenerator {
2
3     public void generate(IFile astFile , IProgressMonitor monitor , IReporter
      reporter) {
4
5         setup = new GenerationSetup(astFile , monitor , reporter);
6         compUnit = setup.getCompUnit();
7
8         ExtendedGrammarInfo extGrammarInfo = new ExtendedGrammarInfo(setup.
          getGrammarInfo());
9
10        GymnastAST2EMFConverter conv = new GymnastAST2EMFConverter();
11        Gymnast.Gymnast.CompUnit d = conv.convert(compUnit);
12

```

```

13     String filePath = getFilePath(astFile, ".idegenmodel");
14     System.out.println("Filepath:_" + filePath);
15     Resource convertedRes = createResource(filePath);
16     .....
17 }
18 ....
19 }

```

As result of the conversion, EMF-based CST classes of the grammar are gained (accessible through its entry rule `Gymnast.Gymnast.CompUnit`) and they are serialized into the file `<customDSLname>.idegenmodel`.

2.3.2.4 UI Generator

The generated `.idegenmodel`, used as input for the generator of UI plug-in, plays an important role in completing the generated code from the JET templates (Figure 2.14). In the constructor of `GenerationSetup` which is called by the class `IDEGenerator`, the `.idegenmodel` will pass the second check on the kind of input file. Then it will be parsed to get the root, i.e. entry point to the EMF-based model. There are two possibilities of what kind the root belongs to: either it is the `TopLevelClass` or the compilation unit `CompUnit` of the grammar, depending on whether the option `ecoreGenerateTopLevelClassForEMFEditor` is set to `true` or `false`. Afterwards, the entry rule of the grammar should be acquired, as many implementation aspects will refer to the entry rule. This is done by fetching the first rule from the list of grammar and checking the existence of an attribute called `entry`. Listing 2.7 shows the snippet.

Listing 2.7: GenerationSetup for Generating UI

```

1 public GenerationSetup(IFile file, IProgressMonitor monitor, IReporter
   reporter) {
2
3     try {
4         // Input is .ast file
5         if (file.getFileExtension().equalsIgnoreCase("ast")) {
6             .....
7         }
8         else { //Input is .idegenmodel file
9
10           EObject root = parseIDEGenModel(file);
11            if (root instanceof Gymnast.Gymnast.TopLevelClass) {
12                topLevelClass = (Gymnast.Gymnast.TopLevelClass) root;
13                emfCompUnit = topLevelClass.getCompUnits().get(0);
14            }
15            else { //CompUnit is root; no TopLevelClass
16                emfCompUnit = (Gymnast.Gymnast.CompUnit) root;
17
18                Rule entryRule = emfCompUnit.getGrammar().get(0);
19
20                Attrs attr = getAttributeFromRuleDecl(entryRule);
21                if (attr != null && attr.getAttrList().get(0).equalsIgnoreCase("entry"
22                    )) {
23                    customDSLEntry = entryRule;
24                    System.out.println("Entry_Rule:_" + customDSLEntry.getName());
25                }
26                else {
27                    System.out.println("Error: _No_entry_rule!");
28                }
29                .....
30            }
31        }
32    }
33    catch (Exception e) {
34        .....
35    }
36 }

```

```

30     }
31 }

```

After setting up some information, the constructor will also prepare a map called `parentChildrenTable`, which is a `BiMultiMap<Rule, String>` (shown in Listing 2.8). This map will hold information on relationships between production rules, i.e. a rule, whose definition contains further rules, thus a "parent-children" relationship (for the children rule, only the names are kept in the map, therefore a "String" value). `EMFSimpleExprCollector`, an extension of the generated EMF-based Switcher, is responsible for collecting the children rules of a rule. Within the production rule itself, the children are recognized as `SimpleExpr`, thus the collector will keep all `SimpleExpr` instances in the map. This map helps in finding out the "route" from the entry rule to a certain rule for the purpose of visiting the node representing that certain rule. More explanation on this in the section explaining the features.

Listing 2.8: `BiMultiMap` for Parent-Child Relationship between Rules

```

1
2  EList<Rule> rules = getRules();
3  for(Rule rule : rules){
4      EList<SimpleExpr> simpleExprs = new EMFSimpleExprCollector(rule).
5          getListOfSimpleExpr();
6      for(SimpleExpr simpleExpr : simpleExprs) {
7          parentChildrenTable.put(rule, simpleExpr.getValue());
8      }
9  }

```

In the `generate()` method of `IDEGenerator`, it progresses similarly to creating the core project: create the project, packages, and files. As the result, a new UI plug-in project is created with some implementation classes for the editor and its usability features.

2.3.2.5 The JET Templates

In order to understand the generated codes better, this subsection explains some of the templates in connection with the input model for the templates. Figure 2.14 shows that the templates are grouped based on the project, and furthermore on the kind of components or features of the UI they are implementing. In general, the naming of the templates make the base for the naming of the generated classes, e.g. from a template called `Editor.javajet`, a class with the name pattern `<customDSL>Editor.java` can be generated.

In relationship with `.idegenmodel`, `GenerationSetup` plays the similar role to `GenModelImpl` of Eclipse EMF in its relationship to a `.genmodel` file⁵. It provides numerous getter methods that deliver the required information, mostly (qualified) names of classes. However, as mentioned before, `GenerationSetup` also provides the additional information contained in `.idegenmodel` to generate the UI features.

Partitioning By default, there will be only one document partition that covers the entire document, which is of type `IDocument.DEFAULT_CONTENT_TYPE`. Having

⁵More information on EMF components can be found in [Fra07a]

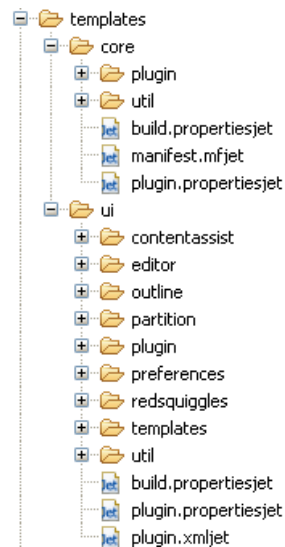


Figure 2.14: Templates Folder

only a single partition means that any implemented feature, that depends on the existence of document partition, will operate on the whole document too. To have more flexibility on determining which feature should be available to which part of the document, more specific partitions need to be defined.

The main class for the partitioning is the `PartitionScanner`. This is where the different scanning rules for the purpose of document partitioning are defined based on the information given by user through the `PartitionRule` instances of `.idegenmodel`. `PartitionRule`, as part of the class `Partition`, are useful for defining the partitions of a document precisely. Every partition which is defined here can then be referred from other classes for the purpose of activating features for the certain document partitions.

For every feature, `GenerationSetup` provides methods that check their existence. In this case, `PartitionServiceActivated()` is used for checking the existence of the additional partitions. Templates of classes whose generation directly related on partitioning are kept in the folder `templates.ui.partition`.

Token Coloring Service Token Coloring Service is created after instantiation of the `TokenColoringService` class with its `TokenGroup`'s (see class definition in Listing 2.2). Two templates for this feature are `CodeScanner.java` and `Keywords.java`.

The method `TokenColoringServiceActivated()` is used to check whether this feature is activated. Having confirmed the existence, following information may be obtained: list of keywords that should be placed into group(s), where each group has its own name and preferred color. Based on these groups, hashtables will be generated in `Keywords` class, each of which represents one keyword group. Within the `CodeScanner`, each keyword group will be then associated with the preferred color.

Text Folding Service The template `CSTChangeListener.java` deals with the implementation of this feature. Text Folding Service is implemented in two different ways. By default, i.e. when the class `TextFoldingService` isn't instantiated, folding is created by utilizing the `AST2EMFConverter` to get the EMF-based

CST which will be then traversed. Through each of the nodes, the corresponding POJO-based CST nodes can be found through the AST2CST map of the converter, and its positions can be determined, which are required to compute the folding range.

For the second alternative, which automatically avoids the generation of the default implementation of folder feature, the following points are necessary. Recall the class that models the text folding feature (shown in Listing 2.3). Just like for any other features, firstly the existence of the feature in `.idegenmodel` is checked through the method `TextFoldingServiceActivated()`. If it exist, it should contain the list of collapsable rules defined by the user.

For each of collapsable rules, a path to the entry rule can be retrieved using the method `GenerationSetup.getPathToEntryRule()`, which in turn traverses the `parentChildrenTable` defined before. The implementation of the text folding feature uses visitor pattern, as can be seen in `EmfaticCSTChangeListener`. The computed path is useful to determine the order of visiting the node hierarchy, starting from the entry rule down to the node representing the collapsable rules.

It is important to notice how the visitor code should be generated, i.e. whether `visit()` is required for getting the children node or it is enough to use the corresponding getter method. In case that a node is representing a `ListRule`, its children have to be visited. As the result, the code structure may show several level of visiting pattern, with the main `visit()` method is applied on the collapsable rule.

Content Assist Content assist feature is enabled for every custom DSL by default. The implementation is providing proposal for all defined keywords in the DSL, which is activated through button "Ctrl + Space". Templates related to this feature are placed in package `.contentassist`. In addition, the class `EditorMessage` and its corresponding properties file `EditorMessage.properties` that defines the activation short-cut, and implementation of some methods, e.g. `getContentAssist()` of `SourceViewerConfiguration` and `createActions` of `Editor` have to be generated as well. For code snippet refer to Chapter 1.3.9 of [GS07].

Auto Edit (Smart Brace, Automatic Closing Quote) A default implementation of auto edit strategy for the custom DSL includes smart brace and automatic closing quote. User can edit or add auto edit strategy by editing the method `customizeDocumentCommand()` of the generated `AutoEditStrategy` class. For codes on this and more information refer to Chapter 1.4.4 of [GS07].

Default Annotation Hover By implementing the method `SourceViewerConfiguration.getAnnotationHover()`, the information on errors and warnings are shown directly at the squiggles in addition to the display on error panel.

Mark Occurences A simple "Mark Occurences" feature is implemented in a way that the document is scanned for the same words, and these are highlighted, i.e. create an `Annotation` instance for the words, as soon as one of them is selected. Making use of existing components, this feature uses the Gymnast-generated `TokenNode` class to fetch the corresponding CST node of the selected text. The position of every selected word as result of the scanning process will be then associated with its `Annotation` instance to provide the highlighting.

The implementation of this feature in Emfatic Editor utilizes the existence of a `OneToManyMap<EObject, ASTNode>`, that keeps the relationship between the

EMF-based and POJO-based CST nodes, i.e. a declaration part and its usage parts of a certain rule. Hence a precise mark occurrence can be achieved.

Range Highlighting, Current Line Highlighting and other General Features In previous chapter, an additional class `RangeHighlightingService` is introduced, which should be instantiated in order to provide the range highlighting service. This contains the list of rules, by which the text area should be marked, i.e. the edge of the editor on that certain range is highlighted. The resulting implementation of the feature is placed in a `SelectionListener` class implementing the interface `ISelectionListener`, whose method `selectionChanged()` is responsible for giving a certain feedback in case of a selection is made on the editor. For this purpose, the listener must have been installed to the editor (in its `createPartControl()`-method).

The method `highlightRange()` is generated into this listener when the feature is activated and it will be called each time selection is made on the editor. Depending on which rules are selected to be highlighted for the instantiation of `RangeHighlightingService`, this method will contain several `instanceOf` checking to check whether the selected text is representing parts of the selected rules. If it is the case, the method `setHighlightRange()` of the editor will be called to provide the highlighting within the range of the rules.

Because of the possibility of different rule selections, in which some of the selected rules may be contained in the other selected rules, it is important to notice that the checking process may be overlapping, i.e. a certain smaller range is highlighted by the highlighter of a wider range. This could result in unwanted highlighting effect. Therefore, the user should be responsible to provide the correct order of selected rules, whose checking methods will be generated in the same order as well. Logically, the order should be in descending order of the scope size, so that smaller range will not be overlapped by the wider range.

Together with "Current Line Highlighting", "Show Line Numbers" and some other features, Range Highlighting can also be set, i.e. choose whether to turn the feature on or off, through the General Preference (**Preference > General > Editors > Text Editors**). User should not set any preference store (`IPreferenceStore`) to the editor, e.g. using the editor plug-in's own preference store through the method `setPreferenceStore()` during editor initialization time, to let the default preference store be used instead, i.e. `EditorsUI.getPreferenceStore()`.

Matching Bracket "Matching Bracket" is not included in the General Preference. Therefore, an additional preference page could be of use, where user can choose whether to activate the feature or not, and to choose the color of the matching brackets.

This feature is set by the `SourceViewerDecorationSupport`, which can be configured through the method `configureSourceViewerDecorationSupport` of `AbstractDecoratedTextEditor`. The support class will in turn activate the feature by setting a certain character pair matcher and preference keys, i.e. literal, for the feature itself and its color. A pair matcher, normally instance of `DefaultCharacterPairMatcher`, defines the characters for which its pair should be matched, and the partition in which the characters may exist.

For defining the preference keys mentioned before, a `PreferenceConstants` class containing those keys is created. Furthermore, a `PreferencePage` class is defined, whose fields' initial values e.g. a boolean field which activates the feature if it is set to `true` and a modifiable color field are defined in a `PreferenceInitial-`

izer class. Now that there are two Preference Pages which should be connected to the editor, the preference store of the editor needs to be set explicitly. This can be done by setting `ChainedPreferenceStore` as the editor's preference store, which contains both the default and the plug-in's own preference store.

Outline View Gymnast provides base classes for defining an outline view for the editor⁶. They are configured through the class `OutlineConfiguration`, that determines which `OutlineBuilder` and `ContentOutlinePage` to be used. An outline builder makes use of `OutlineNode` to define how each node of the (Tree-based) Outline View should look like, and what content it should display. A content outline page is the standard class for implementing an outline view by extending the `org.eclipse.ui.views.contentoutline.ContentOutlinePage`.

The implementation of Outline View also utilizes the `AST2EMFConverter`, just like the default implementation of Folding feature does. Because this requires instances of EMF `EObjects`, the `OutlineNode`, which by default consumes instances of class `ASTNode`, should be modified to use `EObject` instead. Hence the need of `EMFOutlineNode`.

The main idea of having an Outline View is to simplify the navigation over the content of the editor. Thus a connection between the editor and the view is needed, i.e. selection in the outline view has to be mirrored in the editor as well, and vice versa. For this purpose, a map which keeps the relationship between the `ASTNode`'s and the `OutlineNode`'s is created. An `ASTNode` may give the information about position of text in the editor, while an `OutlineNode` is representing a node in the Outline View. Recall that the converter required for building the outline also has a map for the EMF-POJO pairs, so that finding the relevant `ASTNode` from the `EObject` of `OutlineNode` will not be a problem, and vice versa.

Both viewers, i.e. the editor and the Outline View, need to react based on the selection within the counterpart. The editor has to install a selection listener that implements the methods of interface `ISelectionChangeListener`, which in turn calls the method `selectInOutline()` to find the corresponding part in the Outline View. A `ContentOutlinePage` also implements the same interface by default, which then forwards the selection event through the method `updateHighlight` to find the corresponding position within the editor.

2.4 Usage of **IDEalize**

Having seen the core of the framework, this section provides explanation on how to use the framework step-by-step.

2.4.1 Using The Wizard

To ease the usage of the framework, a New Project Creation Wizard has been implemented. This wizard is accessible through **File > New > Other ...**. A selection of wizards will be displayed, with **Idealize IDE Generator** being one of them, as shown in Figure 2.15.

Selecting **Core Plug-In for Custom DSL** will deliver the wizard form shown in Figure 2.16. The form provides fields, several of which, marked by "*", are compulsory fields. Values input by user into those fields will be set as the values of the

⁶See Gymnast documentation for more information

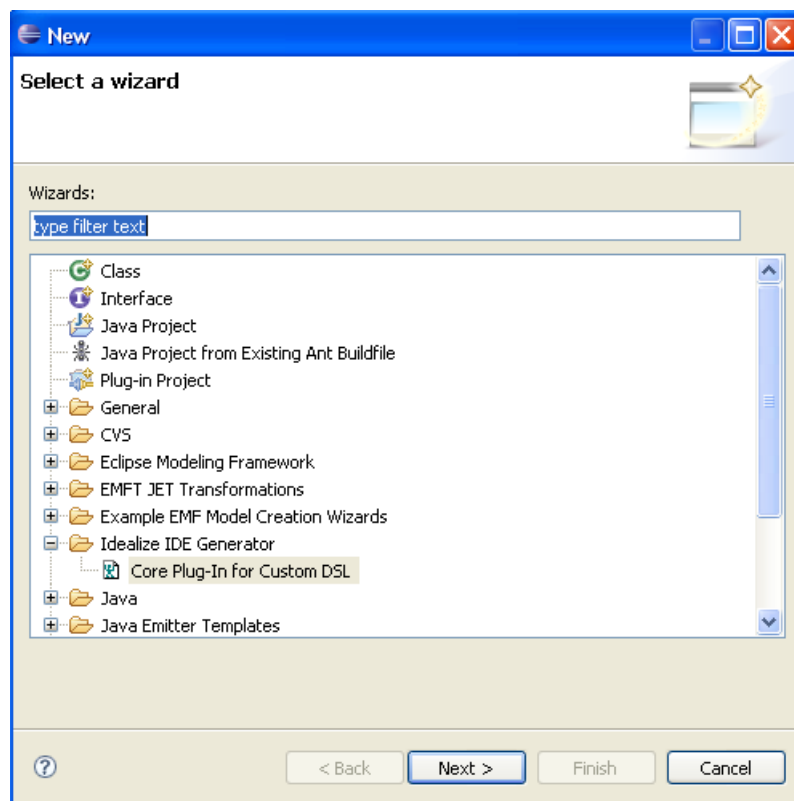


Figure 2.15: Wizard for Generating Core Plug-in Project of Custom DSL

options on the header of a grammar file. Two drop-down lists contain the predefined usable parser generator and AST generator. More information on the available options can be read in Chapter 2 of [GS07].

Figure 2.16: Filling in The Form

In the figure, an example of how to fill-in the form for a custom DSL called *StateMachine* is shown. The resulting *.ast* file, put in the *grammar* folder of the newly created core project (here *statemachine.core*, shown in Figure 2.17), will be automatically opened using *Gymnast Editor*. This is shown in Figure 2.18.

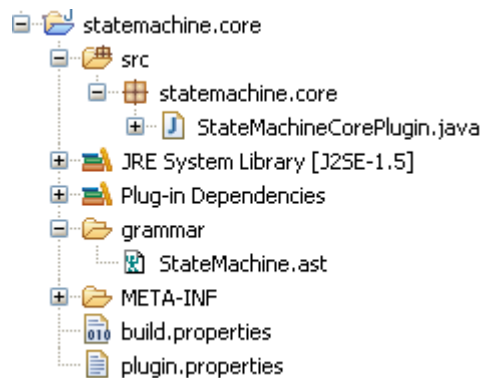


Figure 2.17: Resulting Core Plug-in

2.4.2 Generating and Modifying *.idegenmodel*

The next step will be generating the POJO-based CST classes. Notice that this generation requires the option *astGenerator* set to *primordial*. Using *org.eclipse.*

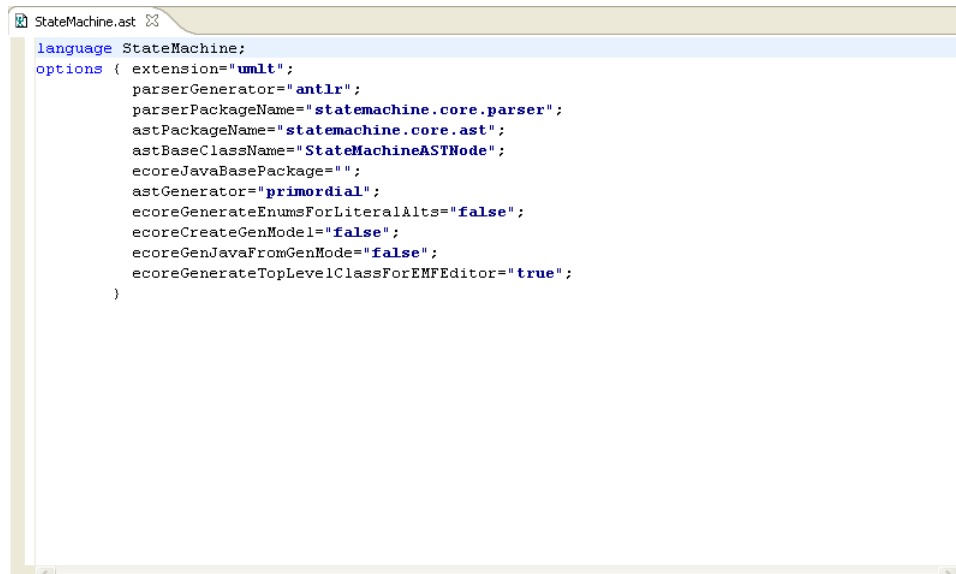


Figure 2.18: Resulting .ast

`idealize.grammar2ecore` will activate the generation of `.ecore` of the grammar and the corresponding CST converter. More on this can be found in Chapter 2 of [GS07].

To progress with the IDE generation, an `.idegenmodel` must be generated. The option to activate the `.idegenmodel` Generator is integrated as part of the context menu for files having the extension `.ast`. Right-clicking on the grammar file will provide the menu **Generate .idegenmodel**. A file with the same name as the grammar file with extension `.idegenmodel` will be generated and put into the same folder. For editing the file, the EMF generated Graphical Editor (provided by plug-in `org.eclipse.idealize.ui.edit` and `org.eclipse.idealize.ui.editor`) can be used. For regular use, the editor should be associated with the file with the extension `.idegenmodel`, and following steps are required, with the numbers corresponding to the numbers shown in Figure 2.19:

1. Open the Preference Page, choose the option **General > Editors > File Associations**.
2. Click the "Add" button besides the "File types" window, and add the extension `*.idegenmodel`.
3. Click the "Add" button besides the "Associated editors" (which is still empty by default).
4. Scroll down to choose "Gymnast Model Editor".

By now, every file which has `.idegenmodel` extension will be opened using the Gymnast Model Editor by default.

The `.idegenmodel` contains the very same information as the grammar file but it gives user the possibility to add certain modifications by creating instances of the feature-classes. These classes are accessible through **New Child** option of `TopLevelClass` as shown in Figure 2.20.

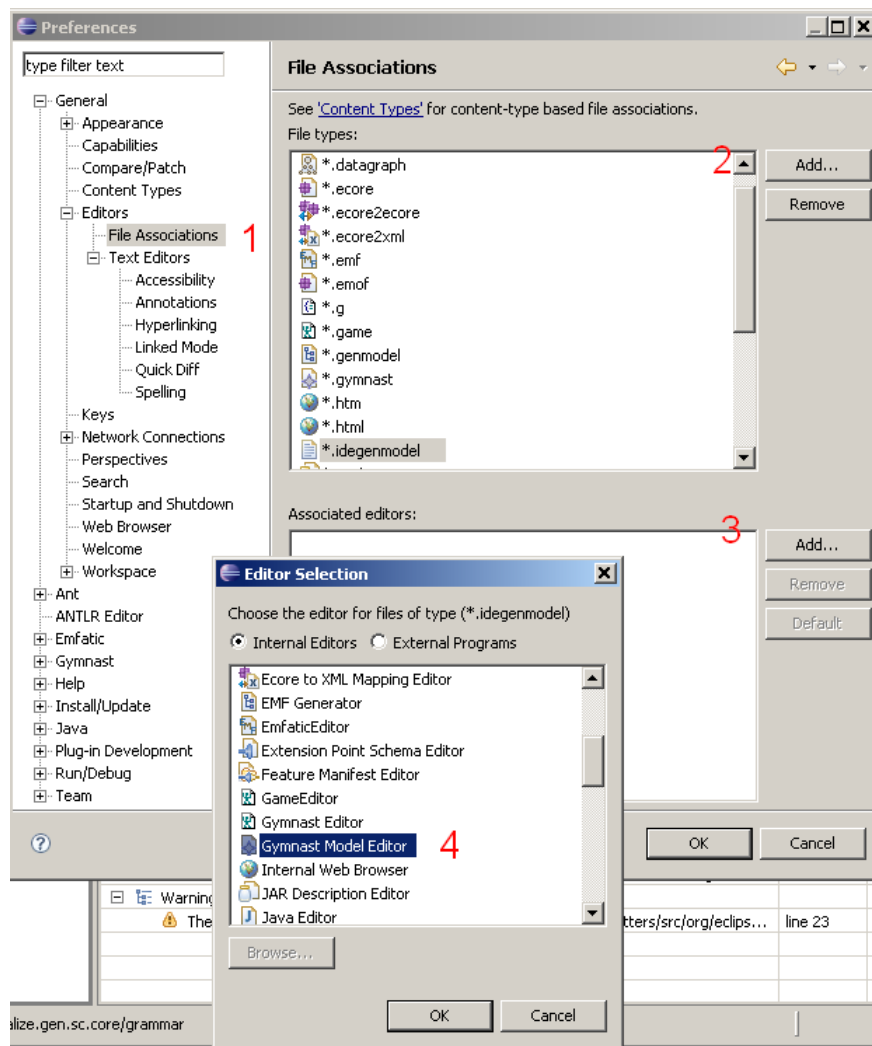


Figure 2.19: How to Associate Gymnast Model Editor to File with .idegenmodel Extension

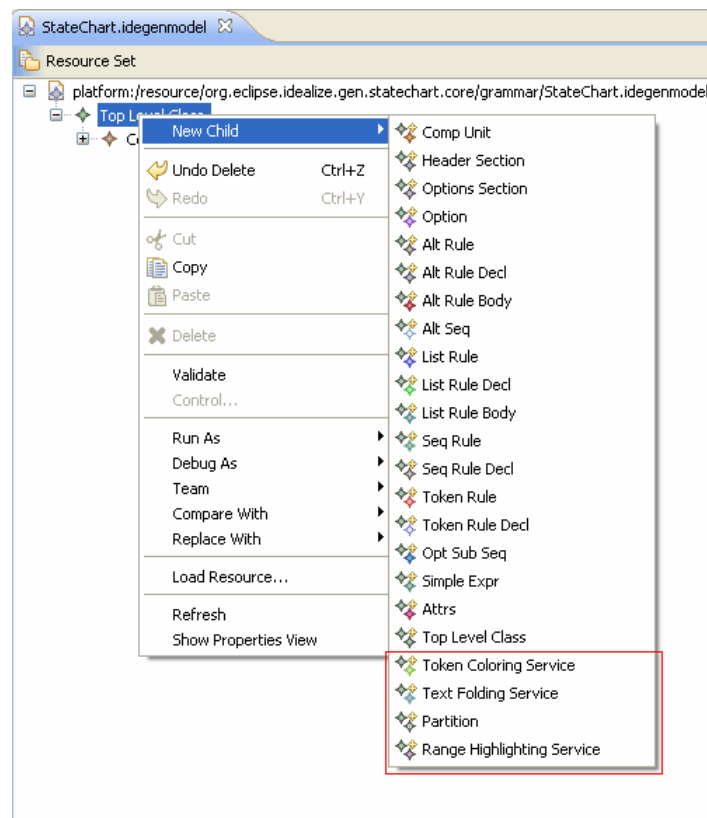


Figure 2.20: Available Additional Features

2.4.2.1 Partitioning

Document partitions are set by creating some partition rules under the `Partition` class. For each rule, the attributes can be modified using the Properties View. The field `Rule` contains some predefined kinds of `Rule` which are selectable from drop-down list. The newly created partitions will act as additional partitions to the single default partition.

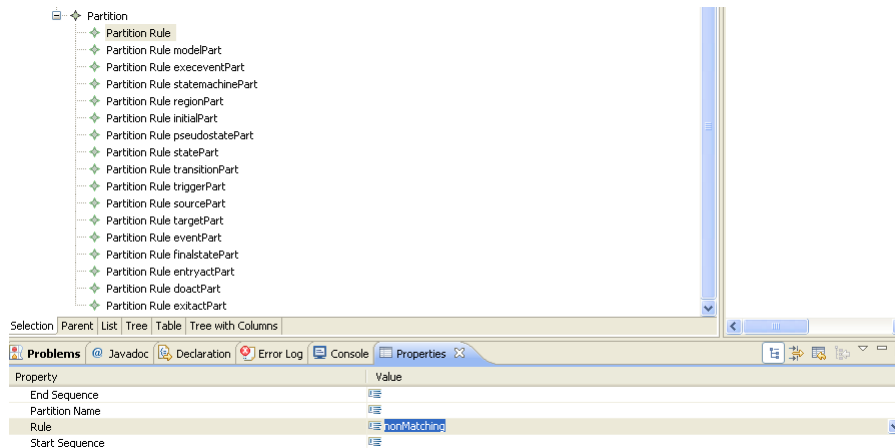


Figure 2.21: Partitioning

2.4.2.2 Token Coloring Service

Under Token Coloring Service, user may create different token groups, where each group should have its own name, color and list of tokens. The available colors are predefined, and selectable from a drop-down list. Adding tokens to the list is done by selecting the required tokens from a window that appear after clicking the button on the right side.

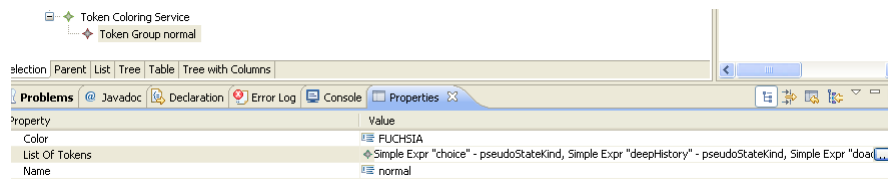


Figure 2.22: Token Coloring Service

2.4.2.3 Text Folding Service

For defining own Text folding Service, i.e. not using the default implementation of Text Folding, user only has to choose the rules which are collapsable on the text editor. The rules are available on a window as soon as the right-side button on the field `Rules` is activated. Activating this feature makes the code generator skip the generation of the default folding implementation, and vice versa.

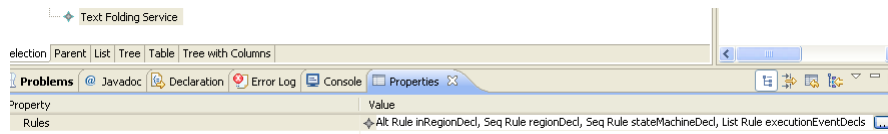


Figure 2.23: Text Folding Service

2.4.2.4 Range Highlighting Service

Similar to Text Folding, activating this feature requires the user to select the rules to be highlighted. As explained in the previous section, the selected rules should have descending order in the size of scope, so that the most inner rule still can be highlighted correctly.

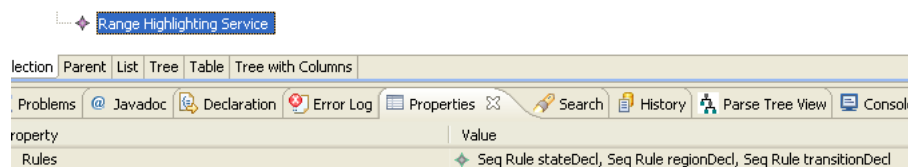


Figure 2.24: Range Highlighting Service

2.4.3 Generating User Interface plug-in

Without editing the generate `.idegenmodel`, user can already generate a standard text editor for the language. It depends on the users themselves, whether the features should be added, and which ones of them. Hence, the users have the full control on how the IDE later should look like.

Continuing on the example provided above, a fully-featured IDE should now be generated based on the modified `.idegenmodel`. Again, the IDE generator can be activated from the context menu of files having certain extension, i.e. the extension `.idegenmodel`. Depending on which features have been activated, corresponding classes are generated into specific packages, whose names are indicating which feature they are for. These classes are part of an own UI plug-in (here, `statemachine.ui`).

The Core and UI plug-in are related automatically as they are generated. A small modification needed to be done on the core is to add the generated packages in its list of *runtime libraries* in `plugin.xml`, i.e. `.parser` and `.ast` packages, and `.util` packages that are generated later (not at the time of core plug-in generation) because some of the classes are referred by classes of UI plug-in. Besides, additional plug-ins have to be imported afterwards depending on which parser generator is used before.

On the side of the UI plug-in, user can add an image for the editor. This can be put into the `icons` package. Afterwards, the `icon` entry in extension tab of `plugin.xml` has to be modified accordingly. This entry shouldn't be left empty, otherwise the editor can't be instantiated.

Chapter 3

Use Case: Text Editor for State Chart Language

Having seen the functionality of IDEalize-generated IDE based on custom DSL grammar specification, this chapter will show a use case of IDEalize by defining language for describing UML2 State Chart in full detail. A worth-mentioned addition to the generated IDE is that OCL Compiler has been integrated into the framework, so that the usability of the IDE is enhanced by well-formedness constraint checking feature at usage time, e.g. the correctness of applicable semantic.

3.1 State Chart Language

Figure 3.1 shows the metamodel of UML2 State Chart, out of which the grammar specification for the State Chart Language is defined. The grammar itself covers most of the classes introduced in this diagram.

3.1.1 The Grammar

The grammar specification of State Chart Language is written based on Gymnast syntax, an EBNF-like syntax, using Gymnast Editor. Therefore, the file has the extension `.ast`. The snippet is shown in Listing 3.1. The complete grammar can be found in Appendix A.1.

Listing 3.1: State Chart Grammar

```
1 sequence stateChartDecl : "statechart" name=ID LCURLY vertexDecls
2                               (finalStateDecl)? RCURLY ;
3
4 abstract vertexDecl : pseudoStateDecl | stateDecl ;
5
6 sequence pseudoStateDecl : kind=pseudoStateKind name=ID
7                               (LPAREN outgoing=transitionDecls RPAREN)? SEMI;
8
9 sequence stateDecl : "state" name=ID LCURLY inStateDecl
10                               (transitionDecls)? RCURLY ;
11
12 sequence compositeState : (entry=pseudoStateDecl)? regionDecls
13                               (exit=pseudoStateDecl)?;
14
15 sequence regionDecl : "region" name=ID LCURLY vertexDecls RCURLY ;
16
```

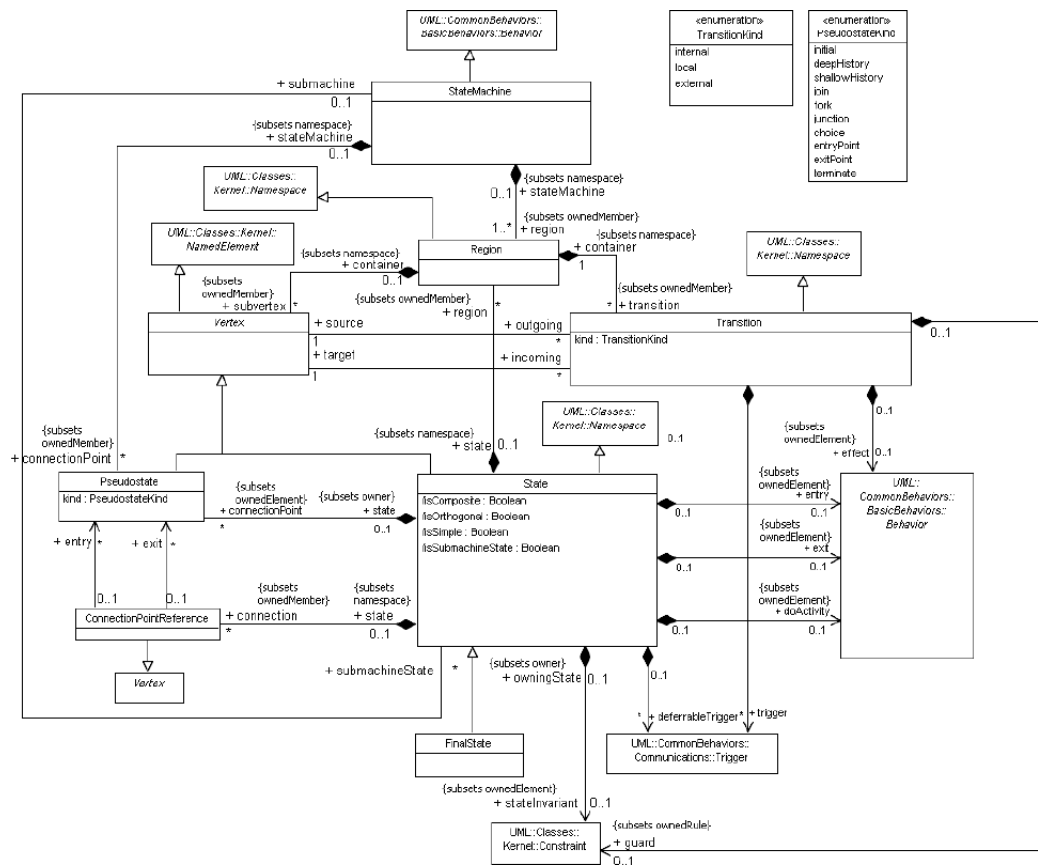


Figure 3.1: UML2 State Chart (State Machine)


```

17
18 sequence transitionDecl : " transition " kind=transitionKind
19                             name=ID LCURLY inTransitionDecl SEMI RCURLY ;
20 .....

```

Having set `grammar2ecore` as `astGenerator` for the grammar, an Ecore representation of the grammar file can be generated. This is shown in Figure 3.2.

Using Emfatic, a further transformation can be done, namely from Ecore to its Emfatic representation (i.e. `.emf`) that results in textual Ecore-based State Chart Grammar shown in Listing 3.2. The complete result can be found in Appendix A.2.

Listing 3.2: Textual Ecore-based State Chart Grammar

```

1
2 package StateChart;
3
4 abstract interface StateChartEASTNode {
5 }
6
7 class CompUnit extends StateChartEASTNode {
8     val ModelDecl[1] modelDecl;
9     val StateChartDecl[*] stateChartDecls;
10 }
11
12 class ModelDecl extends StateChartEASTNode {
13     attr String name;
14 }
15
16 class StateChartDecl extends StateChartEASTNode {
17     attr String name;
18     val VertexDecl[*] vertexDecls;
19     val FinalStateDecl finalStateDecl;
20 }
21
22 class PseudoStateDecl extends VertexDecl {
23     attr PseudoStateKind kind;
24     attr String name;
25     val TransitionDecl[*] outgoing;
26 }
27
28 class StateDecl extends VertexDecl {
29     attr String name;
30     val InStateDecl[1] inStateDecl;
31     val TransitionDecl[*] transitionDecls;
32 }
33
34 class CompositeState extends InStateDecl {
35     val PseudoStateDecl entry;
36     val RegionDecl[*] regionDecls;
37     val PseudoStateDecl exit;
38 }
39
40 class RegionDecl extends StateChartEASTNode {
41     attr String name;
42     val VertexDecl[*] vertexDecls;
43 }
44
45 .....

```

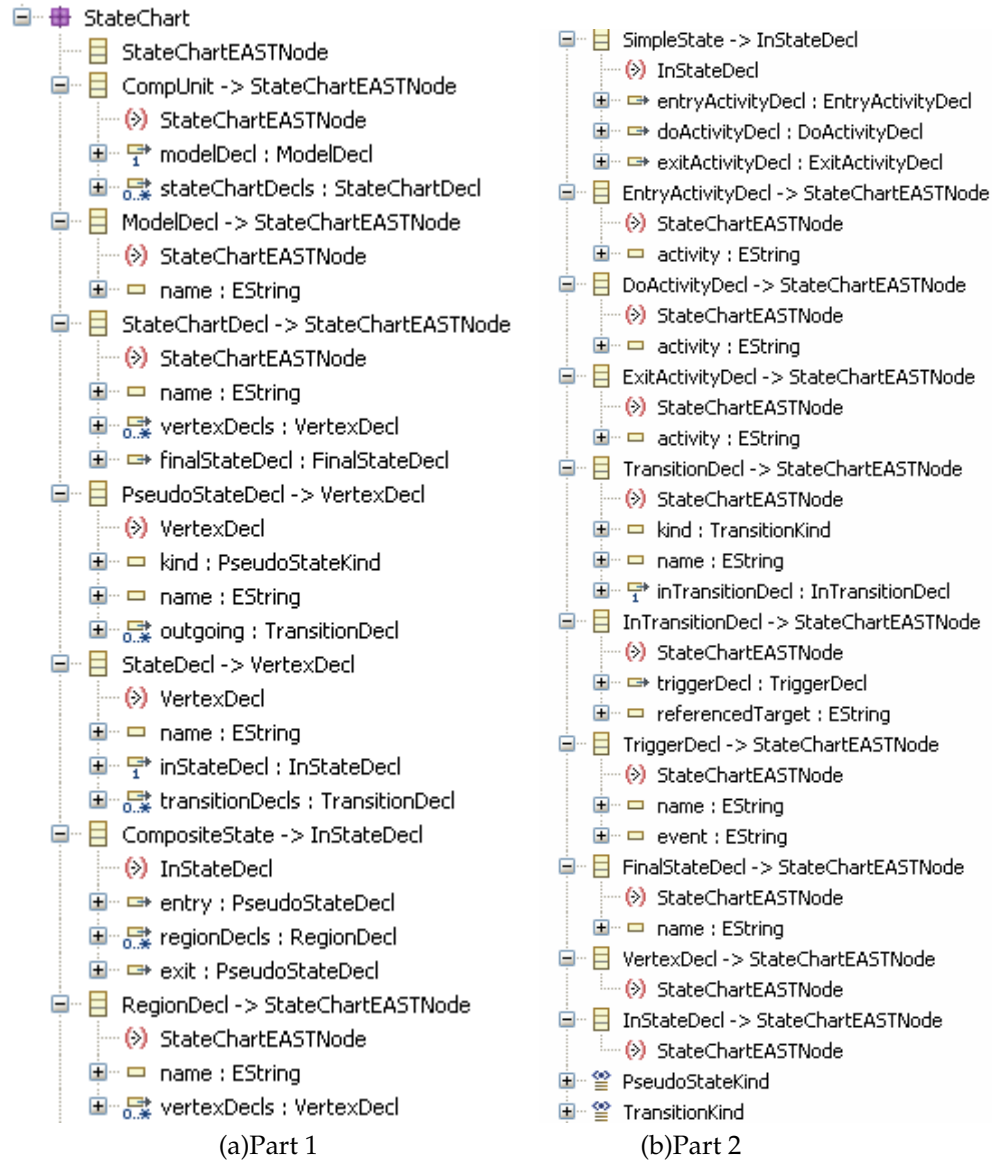


Figure 3.2: Graphical Ecore-based State Chart Grammar

3.1.2 The Constraint

In order to support textual description of model with higher precision and conformity, there are several OCL constraints which have been defined for the State Chart Metamodel. These are shown in Listing 3.3.

Listing 3.3: List of OCL Constraints

```

1 context PseudoStateDecl
2 inv singleOutgoingForInitial: self.kind = PseudoStateKind::initial implies
3   self.outgoing->size() <=1
4
5 context RegionDecl
6 inv singleInitialWithinRegion: self.vertexDecls->select(v | v.ocIsKindOf(
7   PseudoStateDecl))->
8   select(v | v.ocAsType(PseudoStateDecl).kind = PseudoStateKind::initial)->size
9   () <= 1
10
11 context RegionDecl
12 inv uniqueNameOfStateWithinRegion: self.vertexDecls->select(v | v.ocIsKindOf
13   (StateDecl))->isUnique(name)
14
15 context CompositeState
16 inv uniqueNameOfRegionWithinState: self.regionDecls->isUnique(name)
17
18 context StateDecl
19 inv selfTargetForInternal: self.transitionDecls <> null and self.
20   transitionDecls->exists(t | t.kind = TransitionKind::internal) implies
21   (self.transitionDecls->select(t | t.kind = TransitionKind::internal)->forAll(
22     t | t.inTransitionDecl.referencedTarget = self.name))
23
24 context CompositeState
25 inv entryOfComposite: self.entry <> null implies self.entry.kind =
26   PseudoStateKind::entryPoint
27
28 context CompositeState
29 inv exitOfComposite: self.exit <> null implies self.exit.kind =
30   PseudoStateKind::exitPoint

```

- `singleOutgoingForInitial`: a pseudostate of the kind "initial" can only have one instance of an outgoing transition, namely to an initial state.
- `singleInitialWithinRegion`: a region can only have one pseudostate of the kind "initial".
- `uniqueNameOfStateWithinRegion`: a region may have several states, each of which must be unique.
- `uniqueNameOfRegionWithinState`: a composite state consists of at least one region. In case that several regions exist, each of them must be unique.
- `selfTargetForInternal`: a transition of the kind "internal" must have itself as its transition target.
- `entryOfComposite`: an entry pseudostate of a composite state has to be an "entryPoint" pseudostate.
- `exitOfComposite`: an exit pseudostate of a composite state has to be an "exitPoint" pseudostate.

The OCL statements will be embedded into the corresponding model, which results in additional methods by the time the model codes are generated. Listings 3.4 and 3.5 show examples of those methods.

Listing 3.4: Constraint Checking Method for `inv: singleOutgoingForInitial`

```

1 public boolean singleOutgoingForInitial(DiagnosticChain diagnostics,
2     Map<Object, Object> context) {
3     String invName = "singleOutgoingForInitial";
4     /*
5      self.kind.=( StateChart :: PseudoStateKind :: initial ).implies( self.outgoing
6          ->size().<=(1))
7     */
8     Boolean equal1 = Boolean.valueOf(org.eclipse.ocl.util.ObjectUtil.equal(
9         this.getKind(), StateChart.PseudoStateKind.INITIAL));
10    Boolean implies2 = equal1;
11    if (!(implies2)) {
12        implies2 = Boolean.TRUE;
13    } else {
14        implies2 = ((new Integer(org.eclipse.ocl.util.CollectionUtil
15            .asOrderedSet(this.getOutgoing()).size())) <= 1);
16    }
17    if (!(implies2)) {
18        if (diagnostics != null) {
19            //calling the Validator and add the result of diagnose to Diagnostic
20                chain for reporting and marking purpose
21        }
22        return false;
23    }
24    return true;
25 }

```

Listing 3.5: Constraint Checking Method for `inv: entryOfComposite`

```

1 public boolean entryOfComposite(DiagnosticChain diagnostics,
2     Map<Object, Object> context) {
3     String invName = "entryOfComposite";
4     /*
5      self.entry.<>(null).implies( self.entry.kind.=( StateChart ::
6          PseudoStateKind :: entryPoint))
7     */
8     Boolean notEqual1 = !Boolean.valueOf(org.eclipse.ocl.util.ObjectUtil
9         .equal(this.getEntry(), null));
10    Boolean implies2 = notEqual1;
11    if (!(implies2)) {
12        implies2 = Boolean.TRUE;
13    } else {
14        Boolean equal3 = Boolean.valueOf(org.eclipse.ocl.util.ObjectUtil
15            .equal(this.getEntry().getKind(),
16                StateChart.PseudoStateKind.ENTRY_POINT));
17        implies2 = equal3;
18    }
19    if (!(implies2)) {
20        if (diagnostics != null) {
21            //calling the Validator and add the result of diagnose to Diagnostic
22                chain for reporting and marking purpose
23        }
24        return false;
25    }
26    return true;
27 }

```

28 | }

3.2 An Example : State Chart of Telephone Object

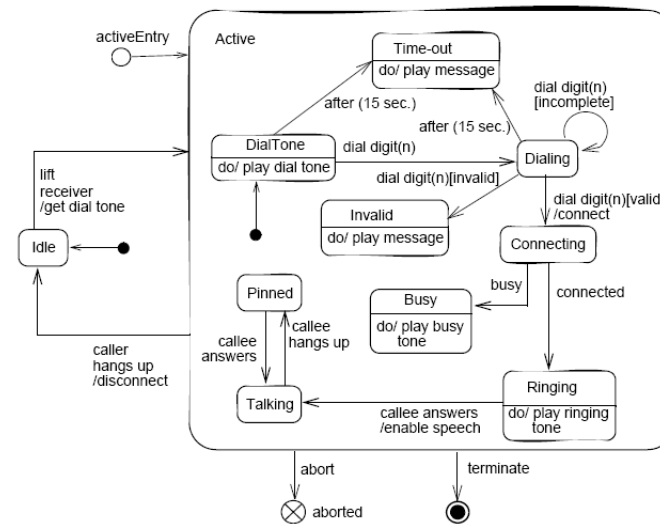


Figure 3.3: State Chart of Simple Telephone Object

A snippet showing parts of the textual representation of the telephone model is shown in Listing 3.6. The complete textual representation of this model can be found in Appendix A.3.

Listing 3.6: Textual Representation of State Chart of Telephone Object

```

1 model TelephoneObject;
2
3 statechart WorkingTelephone {
4
5     initial toIdle (
6         transition external toIdle {
7             target idle;
8         }
9     );
10
11     state idle {
12         transition external toActive {
13             trigger liftReceiver_getDialTone;
14             target active;
15         }
16     }
17
18     state active {
19         entryPoint activeEntry;
20
21         region active {
22             initial toDialTone (
23                 transition local toDialTone {
24                     target dialtone;
25                 }

```

```

26         );
27         .....
28     }
29
30     exitPoint activeExit;
31
32     transition external toIdle{
33         trigger callerHangsUp_disconnect;
34         target idle;
35     }
36     .....
37 }
38
39 finalstate terminated;
40 }

```

3.3 Generated Components for Supporting the Application of Constraint

If there are some constraints defined on the metamodel (i.e. .ocl file is available), the generator will generate slight different codes in class `CSTChangeListener`, compared to its generated code by default. `CSTChangeListener` is the class which is responsible for the validation of model described textually in the editor. Within its method `parseTreeChanged()`, which will be called whenever the parsed tree has changed (i.e. content of a document is modified), a validation method is executed on the model. The validation message will then be displayed on corresponding position in the text editor. Code snippet is shown in Listing 3.7.

This starts by getting the content of the actual document being edited, which is then parsed to get the `ParseContext`¹. Depending on the result of the parsing process, the instance of `ParseContext` may contain information describing the situation (i.e. it should contain error messages or warning if something goes wrong during the parsing, e.g. syntax error). Using the same context, the content will be then checked based on the defined OCL constraints, before some markers are placed where errors, if any, are found.

Listing 3.7: `parseTreeChanged()` -Method of `CSTChangeListener`

```

1
2 public void parseTreeChanged(ASTNode[] arg0) {
3
4     IFile file = _editor.getFile();
5     IDocument doc = _editor.getDocument();
6     String input = doc.get();
7     StringReader reader = new StringReader(input);
8     IParser parser = _editor.getParser();
9     ParseContext parseContext = parser.parse(reader);
10    addErrorsFromAST(parseContext); //validation here
11    MarkerUtil.placeMarkers(file, parseContext);
12
13    if (parseContext.getMessageCount() == 0) {
14        System.out.println("Validating_Text_OK!");
15    }
16    else {
17        ParseMessage[] msgs = parseContext.getMessages();
18        for (int i = 0; i < msgs.length; i++) {

```

¹More on `ParseContext` can be read in the documentation of `Gymnast`

```

19         System.err.println(msgs[i].getMessage());
20     }
21 }
22 }

```

In order for the validation process to be executable, the EMF-based CST classes of the custom DSL have to be generated, because the EMF Validation Framework is used for validating, which requires EMF-based objects as input. For this purpose, *IDEalize* has provided the possibility to generate a class that can be used for converting the POJO-based CST classes as result of parsing the input into its EMF counterparts, namely `<customDSL>AST2EMFConverter`. Having initialized the converter, the root of the parsed tree is used as input for the converter, the result of which, will in turn be used as argument of the `validate()` method of a *Diagnostician* object, a validity checker for *EObject* constraints.

It is important to notice, that the validation process done by *Diagnostician* is not the same as the basic validation process it used to do anymore, because the model has already been augmented with some constraints. Instead, the validation process is now done by invoking the constraint checking method embedded into the corresponding model codes, as the result of using OCL Compiler. Examples of constraint checking method are shown in Listing 3.4 and 3.5. More information on OCL compiler can be read from its documentation.

Similar to the parsing process described above, *Diagnostician* will also provide information about the result from validating the model in form of instances of *Diagnostic*. Recall that a converter provides a map `Map _ast2cst` that keeps the relationship between EMF-based CST classes and the POJO-based ones during the conversion process. This map is very useful to find the corresponding text and the position in the editor. Having this information in hand, whenever a validation error occurs, this can be marked on the related text in the editor with the corresponding validation error message. The complete method is shown in Listing 3.8.

Listing 3.8: Code for Validating Model

```

1 private void addErrorsFromAST(ParseContext parseContext) {
2
3     // warnings and errors computed not from the POJO-based CST but from the
4     // EMF-based CST
5     StateChartAST2EMFConverter conv = new StateChartAST2EMFConverter();
6
7     StateChart.CompUnit d = conv.convert((CompUnit)parseContext.getParseRoot()
8     );
9
10    if (d != null) { //to avoid checking even if text isn't complete yet
11        // invoke EcoreValidator
12        Diagnostician diagnostician = new Diagnostician();
13        final Diagnostic diagnostic = diagnostician.validate(d);
14        if (diagnostic.getSeverity() == Diagnostic.OK) {
15            System.out.println("Diagnose_Result_is_OK!");
16        }
17        for (Diagnostic childDiagnostic : diagnostic.getChildren()) {
18            String dMsg = childDiagnostic.getMessage();
19            if (childDiagnostic.getData().size() > 0) {
20                Object primarySourceOfProblem = childDiagnostic.getData().get(0)
21                ;
22                if (primarySourceOfProblem != null && primarySourceOfProblem
23                    instanceof EObject) {

```

```

20         ASTNode node = conv._ast2cst.get((EObject)
21             primarySourceOfProblem);
22         parseContext.addParseMessage(new ParseError(dMsg, node.
23             getRangeStart(), node.getRangeLength()));
24     }
25 }
26 }

```

3.4 How to Progress

The previous section has described how the constraint checking feature is integrated into the framework from the code perspective. This section will now give a short guide for users on what to do first to activate the feature.

In the previous chapter on IDEalize, the steps in using IDEalize are explained in detail, starting from making use the wizard up to generating the final IDE. Constraint checking feature is based on the model, which is "placed" in the core plug-in. Therefore, the steps described here should be done before generating the IDE. But it is not limited at this point of time, as the IDE can be regenerated after adding the feature.

3.4.1 Generating .ecore and the converter class

The plug-in `org.eclipse.idealize.grammar2ecore` provides the functionality of generating .ecore representation of the grammar and at the same time generating a converter class. Because the OCL compiler requires the existence of the base model in order to be executable, using this generator must be the first task to accomplish. The generated converter will possibly containing some class resolving problems, due to the fact the converter refers to model codes for its conversion result, which are not there yet. These problems can be solved as soon as the model codes are generated from the model.

3.4.2 Adding .ocl file using the same name as the model in the same folder

OCL statements which are already prepared should be put in a text file with the extension .ocl and is named after the model against which the constraints should be checked, i.e. the .ecore file.

3.4.3 Compiling the OCL statements

Right-clicking on the file in the presence of the homonymous .ocl file will provide the context menu **OCLCompiler > Compile**. If the compilation succeeds, a new folder (again with the same name as the model) will be generated, and it contains an .ecore file with its corresponding .genmodel file. The new .ecore file is basically containing the very same information as the source .ecore, besides that the OCL statements from the .ocl file are now integrated into the model. Figure 3.4 shows the structure of the folder and the context menu.

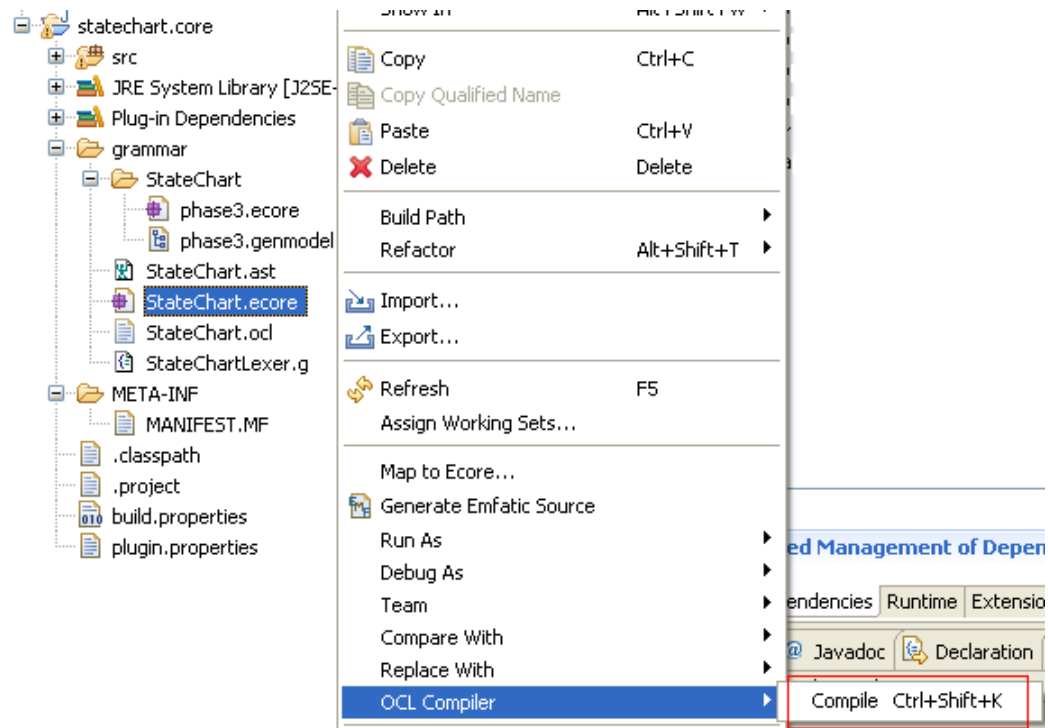


Figure 3.4: Grammar Folder Structure and the OCL-Compiler Context Menu

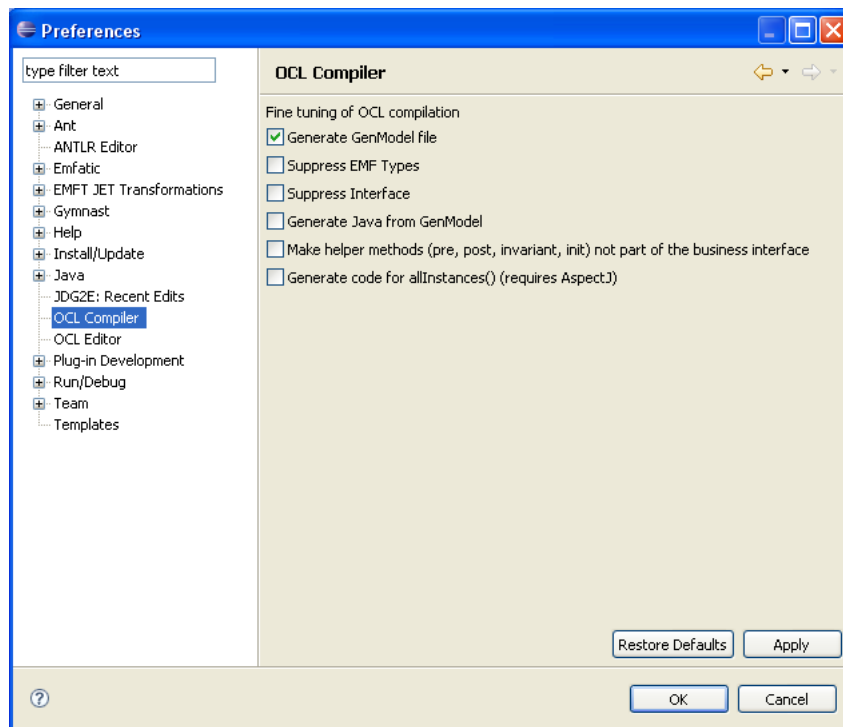
Important : to avoid that the result of OCL compilation is overriding any plug-in configuration and classes, the options "Generate Java from GenModel" and "Generate Code for allInstances()" in the preference page of OCL Compiler should be ticked-off. The option "Generate GenModel file" is the only one left as it is by default. The OCL Compiler Preference Page is shown in Figure 3.5.

3.4.4 Generating the model codes out of OCL-augmented model

This step is well-known from EMF tutorial on how to generate codes from an .ecore file. However, it is best to generate the model codes into a stand-alone plug-in (i.e. the model ID and the directories for the generated codes should be modified, e.g. an `statechart.ocl` plug-in) in order to avoid overriding of the existing configuration of the current plug-in (i.e. the core plug-in). This plug-in is shown in Figure 3.6. The additional codes (i.e. methods for constraint checking can be found on the respective classes, some of which are shown in Listing 3.4 and 3.5. Due to the existence of such methods, there is also an additional class generated into the `.util` package of the OCL Plug-in, which is called `<customDSLname>Validator`. This class will be called by the EMF validation framework (i.e. by `EcoreValidator`) for the validation process, which in turn call the corresponding constraint checking method of the classes.

3.4.5 Importing the OCL plug-in

The last step to be done is to import the generated OCL plug-in from other plug-ins needing this, i.e. the core plug-in (for the converter to clean the class resolving

**Figure 3.5:** OCL Compiler Preference Page

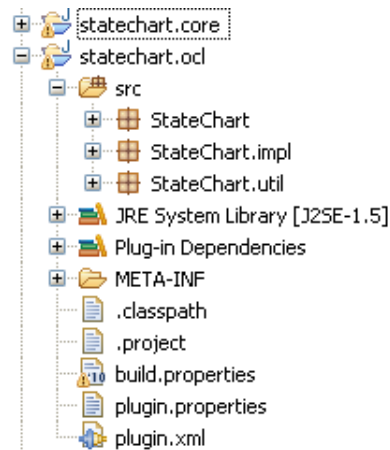


Figure 3.6: The Generated Plug-in Containing OCL-augmented Model Codes

problems) and the UI plug-in (to be able to refer to the conversion result). Every code regeneration will require the user to re-import the required OCL plug-in.

3.5 Some Screenshots

Following the steps described in IDEalize documentation and in the previous section, user may get the IDE for the custom DSL. The screenshots below shows the resulting IDE with some of its features.

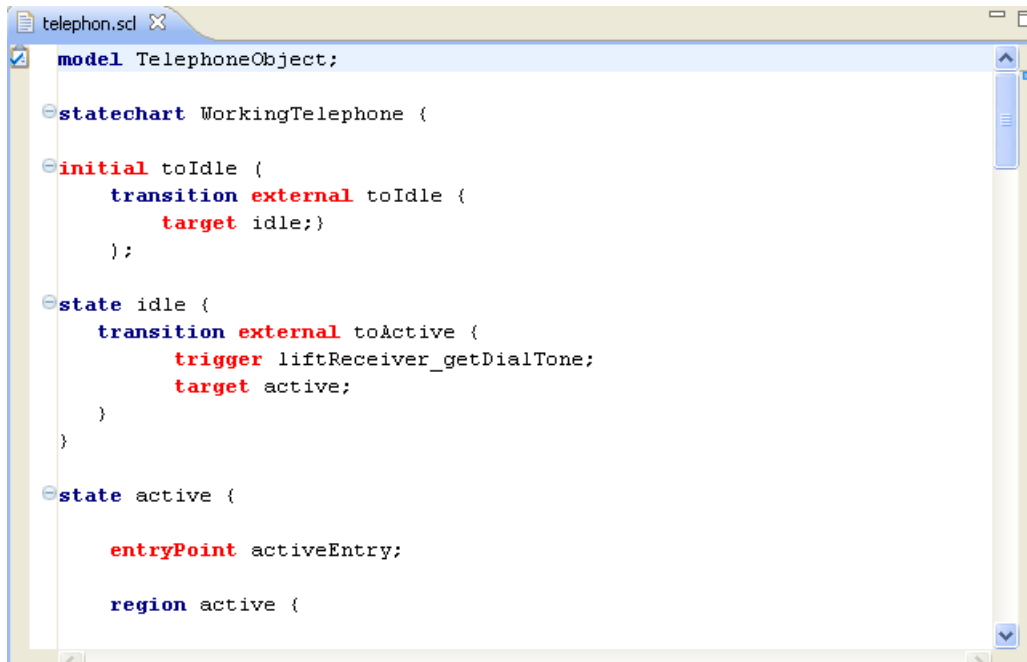


Figure 3.7: Main Editor with Custom Syntax Highlighting

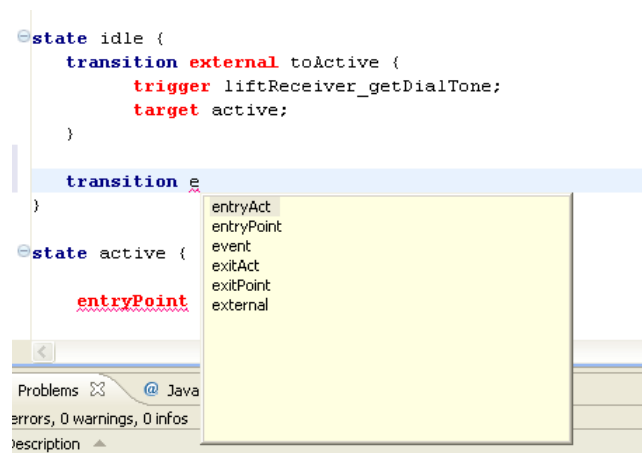
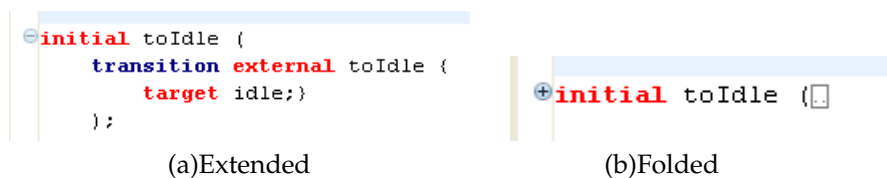


Figure 3.8: Content Assist

```
state active {
    entryPoint activeEntry;

    region active2 {
    }
}
```

Figure 3.9: AutoEditStrategy : SmartBrace



(a)Extended

(b)Folded

Figure 3.10: Folding

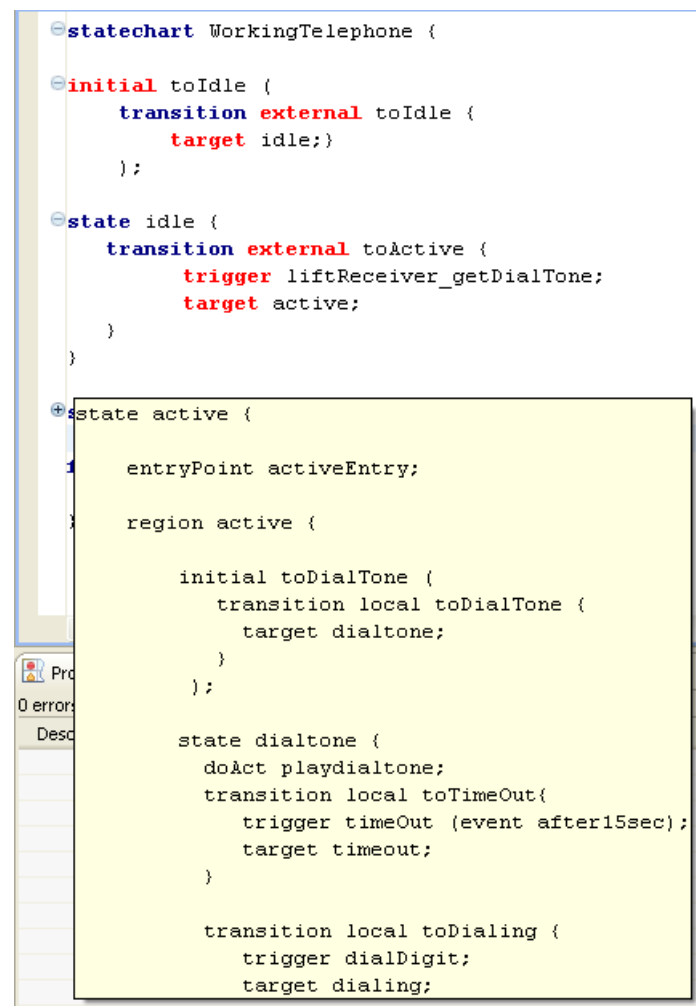


Figure 3.11: Folded Text with Hover Showing The Content

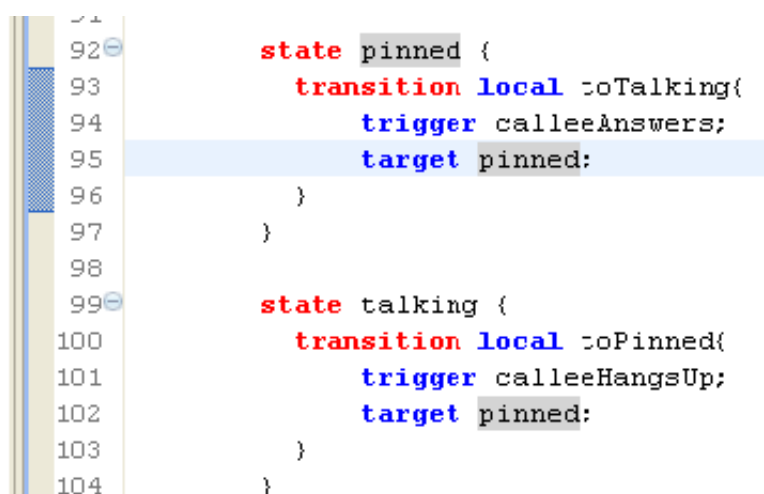


Figure 3.12: Range Highlighting and Mark Occurences

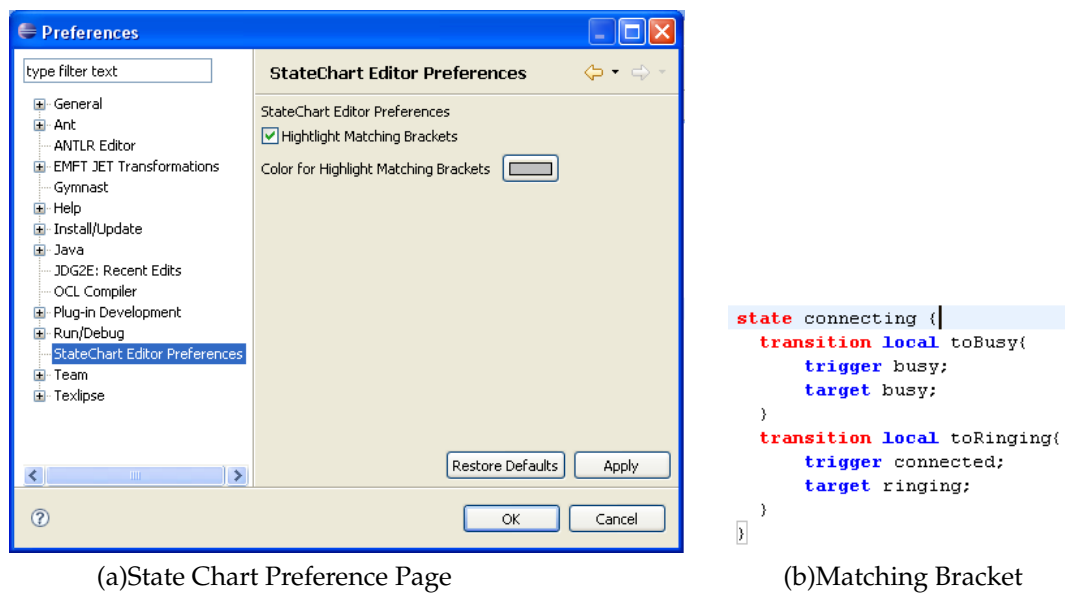


Figure 3.13: Matching Brackets and The corresponding Preference Page

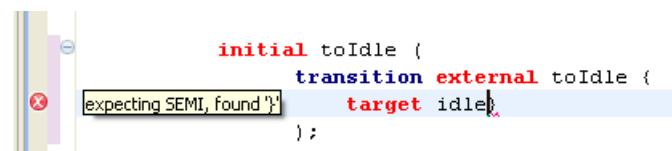


Figure 3.14: Syntax Error Marker and Message on Editor

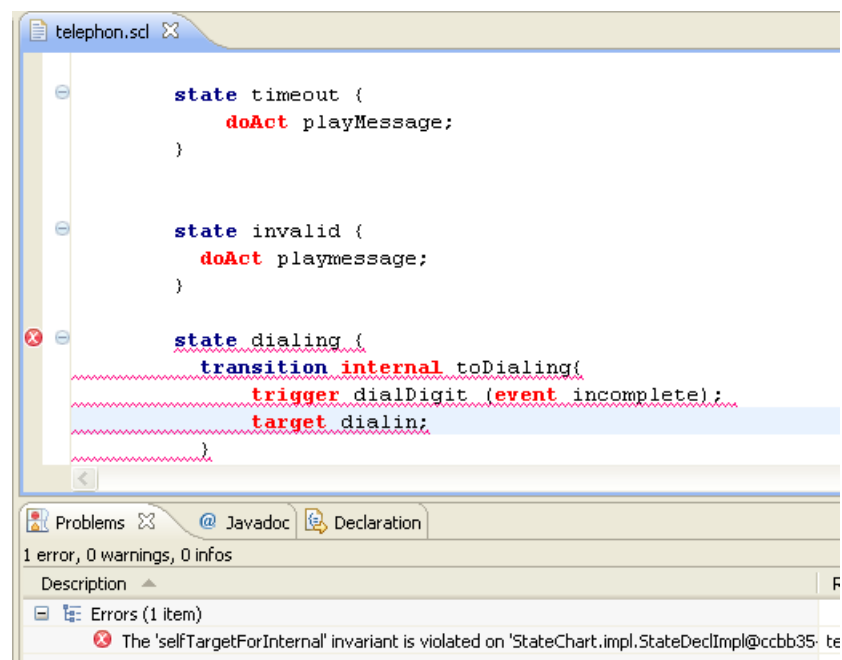


Figure 3.15: Validation Error Marker and Message on Editor

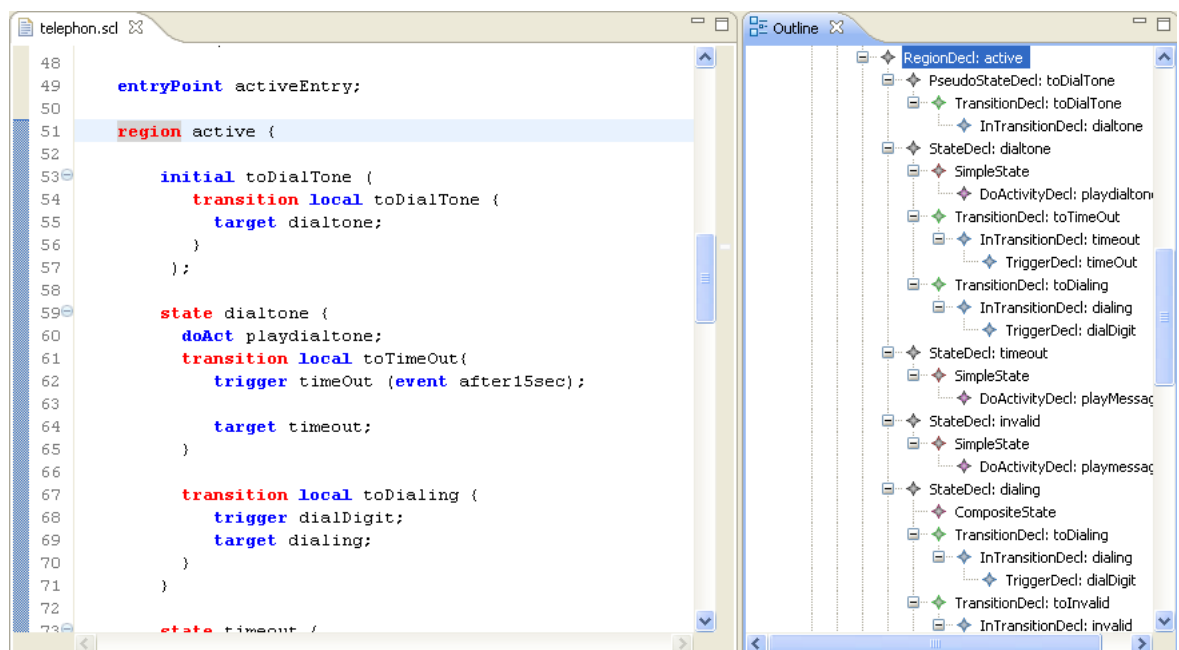


Figure 3.16: Outline View synchronized with Editor

Chapter 4

Other IDE Generators

IDEalize is not the first and only attempt to implement IDE Generator. There has been some prototypes existing on the market, each of which has its own advantages and disadvantages. This chapter tries to point out some aspects of the prototypes, which could be comparable to those of IDEalize.

4.1 xText

4.1.1 Introduction

xText is a textual DSL development framework, which is delivered as part of **openArchitectureWare**[War07], a supporting platform for model-driven software development. **oAW** is a "tool for building MDSD/MDA tools". At the core there is a workflow engine allowing the definition of transformation workflows as well as a number of prebuilt workflow components that can be used for reading and instantiating models, checking them for constraint violations, transforming them into other models and then finally, for generating code. Figure 4.1 shows the structure overview of **oAW**. The following information is based on the official reference documentation[EFH⁺07].

4.1.2 How It Works

Since xText is used for developing textual DSL, the first step of using xText is to specify the grammar of the language, which at the same time defining both its abstract syntax (i.e. metamodel) and its concrete syntax. The specification looks a lot like an extended Backus-Naur-Form just like in Gymnast, and also consists of different kind of *Rules*¹ for defining the language. Additionally there are some built-in lexer rules which can be used in the specification.

Based on the grammar, a so-called workflow file can now be executed, which will generate the language metamodel (i.e. `.ecore` file), parser, and some artifacts needed to implement and modify a text editor for the newly-defined DSL. At this point, user may already start using the generic text editor (i.e. by running a second instance of Eclipse) to describe models in the DSL. Having defined the DSL, a further useful step can be taken, namely to generate executable java codes, e.g. JavaBeans from the model entities. The codes are generated based on the defined model and

¹More information on Gymnast can be found in [Dai05]

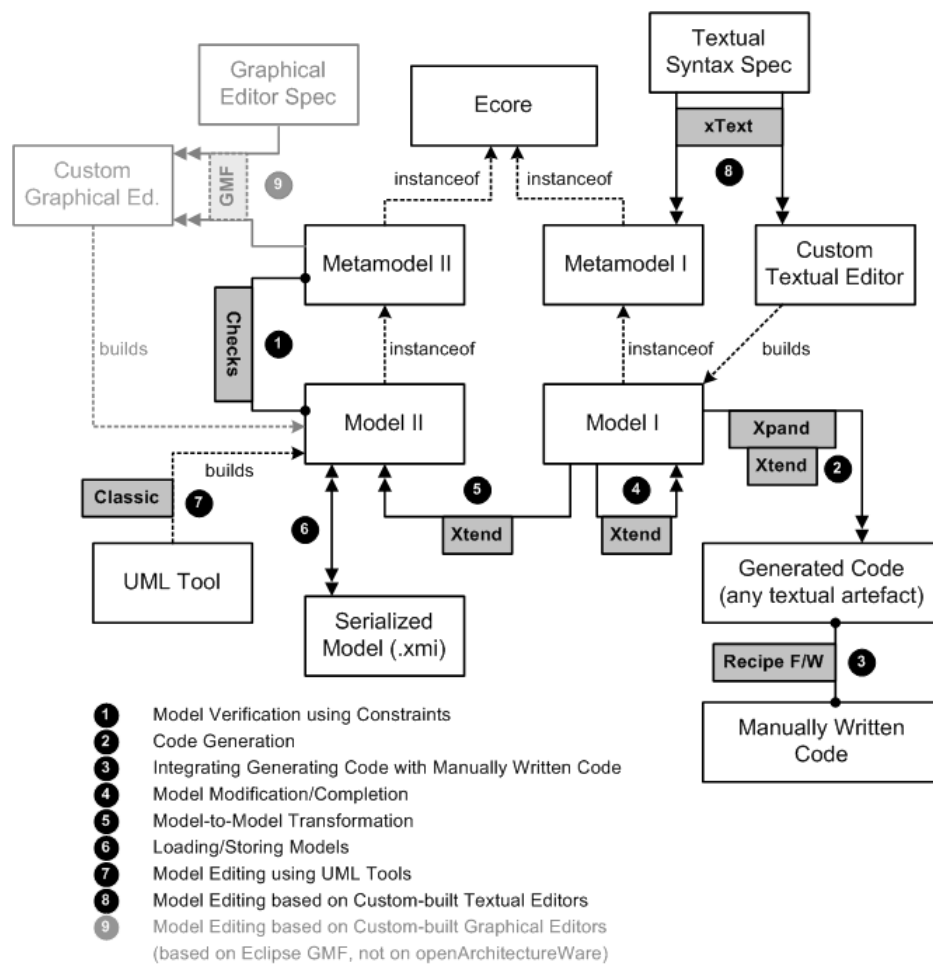


Figure 4.1: Overview Diagram of oAW

corresponding templates, that determines which information contained in the model should be generated.

4.1.3 The Components

A wizard should give a jump start for user to use `xText`, where user may define the name of the language, its extension, etc. This results in 3 different projects:

- **Main Project:** This project contains the main artifacts of the language, which include the grammar specification, the workflow file, generated check, extension and linking file, the metamodel derived from the grammar and Antlr parser artifacts[FLR07].
- **Editor Project:** Any text editor-specific information is contained in this project.
- **Generator Project:** The generator project is intended to contain the resources needed by the generator, such as the templates, etc.

Workflow File The workflow file serves as configurator for generator engine. It contains XML based configuration language which describes a number of so-called workflow components.

Check File A check file is written in **oAW** Check Language and contains declarative constraints for the model elements of a certain type. Having a check file, a validation of model written in the DSL can be checked.

Extension and linking File Extension and linking files are written in `XTend` language. `XTend` is mainly used for defining rich libraries of independent operations and no-invasive metamodel extensions based on either Java methods or **oAW** expressions. Those libraries can be referenced from all other textual languages, that are based on the expressions framework. Such an extension file is used e.g. for making customization on the implemented features of text editors and the linking file defines the linking semantic of the language.

Templates Templates are written in the language called `XPand`. By having templates, one can control the output of the generator.

4.2 Textual Editing Framework (TEF)

4.2.1 Introduction

The Textual Editing Framework (TEF) is a framework, which provides model editors based on a meta-model and a template based textual syntax description. TEF is developed as part of model driven approach for language development by Lehr- und Forschungseinheit Systemanalyse of Humboldt University of Berlin². TEF comes as an Eclipse plug-in and is designed to be used with Eclipse Modeling Framework (EMF)[Fra07a], where the EMF models are accessed through an abstract interface. Such an interface can be implemented for other technology as well. Information on TEF is taken from the official website[Fra07b].

²<http://www2.informatik.hu-berlin.de/sam/meta-tools/index.html>

4.2.2 How It Works

Unfortunately the documentation contains only information on certain aspects of how TEF works and hardly on the implementation part. Nevertheless, it explains about the so-called *Reconciliation Process*, which is principally similar to the concept of background parsing for validation checking in IDEalize. This process is shown in Figure 4.2. When user starts typing text, it will be parsed according to the defined language, which results in the tree-based representation of the text. The next step will be converting the tree into model conforming to the language metamodel, so that it can be checked against the constraints in the metamodel. Any problem which is encountered during the check will be reported back to user, i.e. red line with an error message hover under the corresponding position on the editor.

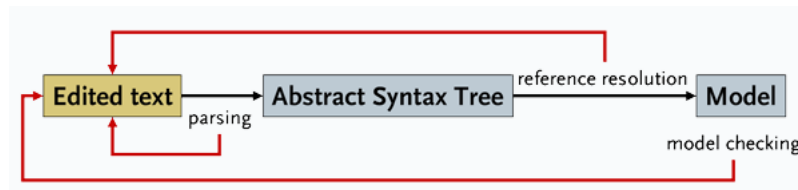


Figure 4.2: Reconciliation Process in TEF

Another worth-mentioned aspect of TEF is that TEF editors are created by describing the model notation as a set of templates. The templates, which are classified into different kinds of templates, define how model elements, their attributes, and the values of the attributes are represented. Templates also define all the semantic aspects that are necessary to create a semantic-rich editor. This means templates describe which items of the text will be highlighted, what proposals appear in a code completion (code assist), which parts can be folded, or how are the text indented. Templates are nested in each other to create a complete notation. Figure 4.3 shows the different classes of the templates.

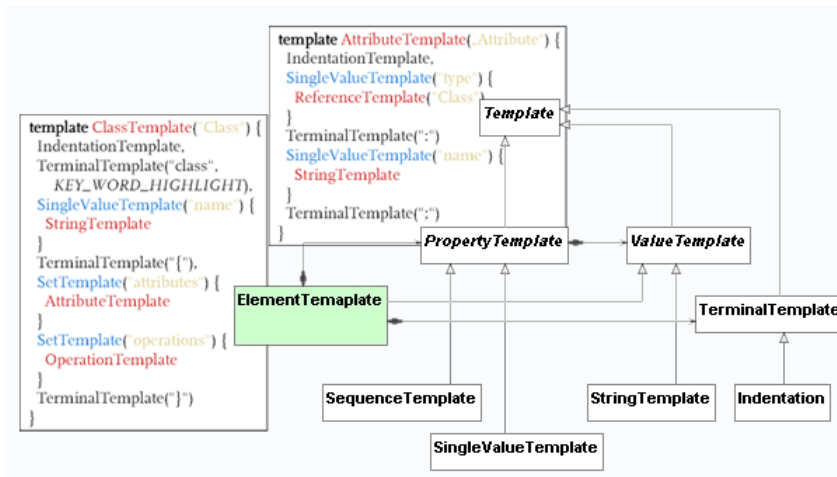


Figure 4.3: Template Classes in TEF

Chapter 5

Outlook

5.1 Summary

The main objective of this thesis is to develop Gymnast into a generator of language specific text-editors, based on the available infrastructure of Gymnast, i.e. for grammar specification and parsing purpose. Furthermore, the resulting generator should be directly applied in a case study for defining textual notation of UML2 State Chart Language and generating the corresponding editor to prove the usability of the generator.

After the topic is introduced in Chapter 1, Chapter 2 provides a detail explanation on *IDEalize*, the generator framework resulting from further development of Gymnast. Many important references to other relevant documents are given, in order for the reader not to miss the basic knowledge and follow the development process from the beginning. Afterwards step-by-step tutorial on how to use the generator is given.

Following the explanation in the previous chapter, Chapter 3 describes a use case of the framework in defining the textual notation of State Chart Language. Starting from the specification of the grammar, this chapter covers up to the steps of generating the text editor and enhancing it with different features. In this use case, an external OCL-Compiler has been integrated for the validation checking purpose using OCL which is embedded into the editor. Then some existing generator prototypes are presented in Chapter 4, which are comparable in certain aspects to *IDEalize*.

Without doubt having such a framework will boost the development process of a new DSL and help settling it into the hands of potential users, since the issue of lack of supporting tools for DSL is now solved to certain extent. DSL designers can now concentrate on defining a comprehensive DSL which should preferably cover the whole relevant aspects of the corresponding domain, without concerning about how the users should cope with the new language later. Furthermore, no additional knowledge is needed to provide a ready-to-use language-specific text editor by implementing it from the ground up since all this can be automatically generated, which means a time saving in the development process.

However, despite all the above-mentioned advantages, there are some drawbacks which can't be avoided completely yet. Every framework is implemented in a certain way, e.g. regarding the languages, workflow, requirements, etc. *IDEal-*

`ize` assumes that the grammar specification is written in an EBNF-like syntax, in order to be able to process it. Unfortunately not every existing domain of problems has an EBNF-like description; instead they are described using other syntaxes, so that an additional effort for the description in EBNF-like syntax is required, which is not always a trivial task, especially for a complex domain. Moreover, learning the framework-specific languages could be tedious, when they are completely different from the one users are accustomed to. A solution would be to have a syntax converter from alternative syntax to the required syntax, so that the additional effort of rewriting `syntax` can be omitted.

Another drawback, especially of `IDEalize`, is the absence of a DSL compiler. Having a text editor on which user may textually describe models is of big advantage, but the progress should not stop there. One could think of a simple compiler, which generates executable Java codes out of the models, so that the models could be in fact useful in the way they are planned to be.

5.2 Future Work

To minimize the drawbacks mentioned above, some possible future works can be done following this project. One of the plans in using `IDEalize` is to apply the generation process on other textual representations, e.g. Business Process Execution Language for Web Services (BPEL4WS), an XML-based language for the formal specification of business processes and business interaction protocols [IBM07]. Just like in other XML-based languages, the syntax of BPEL4WS is defined by using XML Schema Definition (XSD) [Con07].

Having mentioned about using a syntax converter before, a candidate to be considered is the technique applied by XSugar [fXL07] tool. As cited from its website, "XSugar makes it possible to manage dual syntax for XML languages. An XSugar specification is built around a context-free grammar that unifies the two syntaxes of a language. Given such a specification, the XSugar tool can translate from XML to alternative syntax and vice versa. Additionally, the tool statically checks that the transformations are reversible (i.e. bidirectional) and that all XML documents generated from the alternative syntax are valid according to a given XML schema". In [BMS07], an example is given on how to translate XML-based language to EBNF-based alternative, which is exactly the one `IDEalize` could make use of. The following lists reproduce the example by showing a DTD description and a corresponding valid document, the counterpart in a non-XML syntax, i.e. EBNF syntax, and an XSugar specification that specifies the connection between the two syntaxes concisely.

Listing 5.1: DTD

```

1 <!ELEMENT students (student*)>
2 <!ELEMENT student (name,email)>
3 <!--ATTLIST student sid CDATA #REQUIRED-->
4 <!--ELEMENT name (#PCDATA)>
5 <!--ELEMENT email (#PCDATA)>

```

Listing 5.2: A Valid Document based on 5.1

```

1 <students xmlns="http://studentsRus.org/">
2   <student sid="19701234">
3     <name>John Doe</name>
4     <email>john_doe@notmail.org</email>

```

```

5  </student>
6  <student sid="19785678">
7    <name>Jane Dow</name>
8    <email>dow@bmail.org</email>
9  </student>
10 </students>

```

Listing 5.3: An Alternative non-XML Syntax

```

1 John Doe (john_doe@notmail.org) 19701234
2 Jane Dow (dow@bmail.org) 19785678

```

Listing 5.4: XSugar Specification

```

1 xmlns = "http://studentsRus.org/"
2
3 Name = [a-zA-Z]+(\ [a-zA-Z]+)*
4 Email = [a-zA-Z._]+\@[a-zA-Z._]+
5 Id = [0-9]{8}
6 NL = \r\n|\r|\n
7
8 file : [persons p] = <students> [persons p] </>
9
10 persons : [person p] [NL] [persons more] =
11           [person p] [persons more]
12           : =
13
14 person : [Name name] _ "(" [Email email] ")" _ [Id id] =
15           <student sid=[Id id]>
16             <name> [Name name] </>
17             <email> [Email email] </>
18           </>

```

One of the question left regarding XSugar is how to integrate XSugar and *IDEalize*, to achieve the best of both tools. However, the usage of XSugar is restricted only to XML-based languages, so that a converter with a wider variety of inputs would be even more useful.

In term of compilation, *IDEalize* could generate a language-specific compiler, which in its simplest possibility produces JavaBeans codes of the model instances. This compiler should also be generated in dependence of the input model. A good approach in providing compiler has been introduced in **xText** of [War07], and a similar approach should fit into *IDEalize* as well. For example, based on the model defined in Appendix A.2, the Java class shown in Listing 5.6 could be generated from the model presented in Listing 5.5, where every instance of the class represents the instantiation of the model.

Listing 5.5: Textual Representation of A Model

```

1 class TransitionDecl extends StateChartEASTNode {
2   attr TransitionKind kind;
3   attr String name;
4   val InTransitionDecl[1] inTransitionDecl;
5 }

```

Listing 5.6: JavaBeans Code for Model

```

1 public class TransitionDecl extends StateChartEASTNode {
2
3   private String name;

```

```

4  private TransitionKind kind;
5  private InTransitionDecl inTransitionDecl;
6
7  public String getName() {
8      return name;
9  }
10 public void setName(String name) {
11     this.name = name;
12 }
13 public TransitionKind getTransitionKind() {
14     return kind;
15 }
16 public void setTransitionKind(TransitionKind kind) {
17     this.kind = kind;
18 }
19 public InTransitionDecl getInTransitionDecl() {
20     return inTransitionDecl;
21 }
22 public void setInTransitionDecl(InTransitionDecl inTransitionDecl) {
23     this.inTransitionDecl = inTransitionDecl;
24 }
25 }
26

```

There is a relevant work conducted by Jeffery Gray from University of Alabama at Birmingham, who introduces a DSL Debugger Framework called DSL Testing Studio[Gra07]. According to the introduction on the website, "the DSL Testing Studio assists in debugging and testing a program written in a DSL. It uses a grammar-driven automatic approach to generate the end-users DSL testing tools (e.g., debugger, unit test engine, and profiler) for various categories of DSLs (e.g., imperative, declarative, and hybrid DSL). It applies mapping technique for augmenting existing DSL grammars to generate the hooks needed to interface with a supporting infrastructure written for Eclipse that assists in debugging and testing a program written in a DSL". Combining *IDEalize* and DSL Testing Studio could boost the usability level of the framework, since both tools provide the essential functionalities which makes up a good and reliable IDE.

One of the common "problems" in using generator is the lack of possibility to make customization on the generator by third party. However, there are some suggestions made around this issue, e.g. in order to make the generator generating different resources, additional generator modules can be integrated. One way to do this is by providing open APIs through a comprehensive extension mechanism, e.g. the concept of extension point in Eclipse. Alternatively, one can think of more advanced code templates which can be combined with some configuration files for the generator to control the kinds of output to be generated. As a consequence of a better customization level, the applicability of the generator could be raised up to a certain extent.

Human-readability is also an important issue in language engineering. For example XML, one of the most used format for machine processable data interchange, is verbose, which makes it quite unsuitable for direct human-use. The Object Management Group (OMG)[Gro07] has tried to address this issue by introducing Human Usable Textual Notation (HUTN)[Gro04], which is based on the Meta-Object Facility (MOF), an OMG standard for the definition of information models and the subsequent mapping of these models to CORBA interfaces. HUTN specifies a generic textual notation which can be customized for any metamodel conforming to the

MOF[MH05]. An idea thereby is to apply the similar technique used in XSugar to ease the effort of introducing the HUTN-based specification, by having this translated from any XML-based language using XSugar tool. Bringing this idea a step forward, an extension for `IDEalize` to support HUTN could also be implemented to enhance its usability even further.

Appendix A

State Chart Language

A.1 Grammar

Listing A.1: State Chart Grammar

```
1 language StateChart;
2 options {
3     extension="stc";
4     parserGenerator="antlr";
5     parserPackageName="statechart.core.parser";
6     astPackageName="statechart.core.ast";
7     astBaseClassName="StateChartASTNode";
8    .ecoreGenerateTopLevelClassForEMFEditor="true";
9    .ecoreCreateGenModel="false";
10   .ecoreJavaBasePackage="";
11   .astGenerator="org.eclipse理想ize.grammar2ecore";
12 }
13
14 sequence compUnit [entry] : modelDecl stateChartDecls ;
15
16 sequence modelDecl : "model" name=ID SEMI;
17
18 list qualifiedID : id1=ID (qidSeparator idn=ID)* ;
19 token qidSeparator : DOT | DOLLAR ;
20
21 list stateChartDecls : stateChartDecl* ;
22 sequence stateChartDecl : "statechart" name=ID LCURLY vertexDecls
23     (finalStateDecl)? RCURLY ;
24
25 list vertexDecls : vertexDecl* ;
26 abstract vertexDecl : pseudoStateDecl | stateDecl ;
27
28 sequence pseudoStateDecl : kind=pseudoStateKind name=ID
29     (LPAREN outgoing=transitionDecls RPAREN)? SEMI;
30
31 token pseudoStateKind : "initial" | "deepHistory" | "shallowHistory" |
32     "join" | "fork" | "junction" | "choice" |
33     "entryPoint" | "exitPoint" | "terminate" ;
34
35
36 sequence stateDecl : "state" name=ID LCURLY inStateDecl
37     (transitionDecls)? RCURLY ;
38
39 abstract inStateDecl : compositeState | simpleState ;
40
41 sequence compositeState : (entry=pseudoStateDecl)? regionDecls
42     (exit=pseudoStateDecl)?;
```

```

43 |
44 | list regionDecls : regionDecl* ;
45 |
46 | sequence regionDecl : "region" name=ID LCURLY vertexDecls RCURLY ;
47 |
48 | sequence simpleState : (entryActivityDecl)?
49 |                         (doActivityDecl)?
50 |                         (exitActivityDecl)?;
51 |
52 | sequence entryActivityDecl : "entryAct" activity=ID SEMI;
53 | sequence doActivityDecl : "doAct" activity=ID SEMI;
54 | sequence exitActivityDecl : "exitAct" activity=ID SEMI;
55 |
56 | list transitionDecls : transitionDecl* ;
57 |
58 | sequence transitionDecl : "transition" DOT kind=transitionKind
59 |                         name=ID LCURLY inTransitionDecl SEMI RCURLY ;
60 | token transitionKind : "local" | "internal" | "external" ;
61 |
62 | sequence inTransitionDecl : (triggerDecl)?
63 |                         "target" referencedTarget=ID ;
64 |
65 | sequence triggerDecl : "trigger" name=ID
66 |                     (LPAREN "event" event=ID RPAREN)? SEMI;
67 |
68 | sequence finalStateDecl : "finalstate" name=ID SEMI;

```

A.2 Textual Ecore-based State Chart Grammar

Listing A.2: Textual Ecore-based Grammar

```

1 |
2 | @namespace(uri="StateChart", prefix="StateChart")
3 | package StateChart;
4 |
5 | abstract interface StateChartEASTNode {
6 | }
7 |
8 | class CompUnit extends StateChartEASTNode {
9 |     val ModelDecl[1] modelDecl;
10 |    val StateChartDecl[*] stateChartDecls;
11 | }
12 |
13 | class ModelDecl extends StateChartEASTNode {
14 |     attr String name;
15 | }
16 |
17 | class StateChartDecl extends StateChartEASTNode {
18 |     attr String name;
19 |     val VertexDecl[*] vertexDecls;
20 |     val FinalStateDecl finalStateDecl;
21 | }
22 |
23 | class PseudoStateDecl extends VertexDecl {
24 |     attr PseudoStateKind kind;
25 |     attr String name;
26 |     val TransitionDecl[*] outgoing;
27 | }
28 |
29 | class StateDecl extends VertexDecl {
30 |     attr String name;
31 |     val InStateDecl[1] inStateDecl;

```

```

32  val TransitionDecl[*] transitionDecls;
33  }
34
35  class CompositeState extends InStateDecl {
36    val PseudoStateDecl entry;
37    val RegionDecl[*] regionDecls;
38    val PseudoStateDecl exit;
39  }
40
41  class RegionDecl extends StateChartEASTNode {
42    attr String name;
43    val VertexDecl[*] vertexDecls;
44  }
45
46  class SimpleState extends InStateDecl {
47    val EntryActivityDecl entryActivityDecl;
48    val DoActivityDecl doActivityDecl;
49    val ExitActivityDecl exitActivityDecl;
50  }
51
52  class EntryActivityDecl extends StateChartEASTNode {
53    attr String activity ;
54  }
55
56  class DoActivityDecl extends StateChartEASTNode {
57    attr String activity ;
58  }
59
60  class ExitActivityDecl extends StateChartEASTNode {
61    attr String activity ;
62  }
63
64  class TransitionDecl extends StateChartEASTNode {
65    attr TransitionKind kind;
66    attr String name;
67    val InTransitionDecl[1] inTransitionDecl;
68  }
69
70  class InTransitionDecl extends StateChartEASTNode {
71    val TriggerDecl triggerDecl;
72    attr String referencedTarget;
73  }
74
75  class TriggerDecl extends StateChartEASTNode {
76    attr String name;
77    attr String event;
78  }
79
80  class FinalStateDecl extends StateChartEASTNode {
81    attr String name;
82  }
83
84  abstract interface VertexDecl extends StateChartEASTNode {
85  }
86
87  abstract interface InStateDecl extends StateChartEASTNode {
88  }
89
90  enum PseudoStateKind {
91    initial = 0;
92    deepHistory = 1;
93    shallowHistory = 2;
94    join = 3;
95    fork = 4;
96    junction = 5;

```

```

97 choice = 6;
98 entryPoint = 7;
99 exitPoint = 8;
100 terminate = 9;
101 }
102
103 enum TransitionKind {
104     local = 0;
105     internal = 1;
106     external = 2;}

```

A.3 Textual Representation of State Chart of Telephone Object

Listing A.3: Telephone Object

```

1 model TelephoneObject;
2
3 statechart WorkingTelephone {
4
5     initial toIdle (
6         transition external toIdle {
7             target idle;}
8     );
9
10    state idle {
11        transition external toActive {
12            trigger liftReceiver_getDialTone;
13            target active;
14        }
15    }
16
17    state active {
18
19        entryPoint activeEntry;
20
21        region active {
22
23            initial toDialTone (
24                transition local toDialTone {
25                    target dialtone;
26                }
27            );
28
29            state dialtone {
30                doAct playdialtone;
31                transition local toTimeOut{
32                    trigger timeOut (event after15sec);
33                    target timeout;
34                }
35
36                transition local toDialing {
37                    trigger dialDigit;
38                    target dialing;
39                }
40            }
41
42            state timeout {
43                doAct playMessage;
44            }
45

```

```

46
47     state invalid {
48         doAct playmessage;
49     }
50
51     state dialing {
52         transition internal toDialing{
53             trigger dialDigit (event incomplete);
54             target dialing;
55         }
56         transition local toInvalid{
57             trigger dialDigit (event invalid);
58             target invalid;
59         }
60         transition local toConnecting{
61             trigger dialDigit (event valid);
62             target connecting;
63         }
64         transition local toTimeOut{
65             trigger timeout (event after15sec);
66             target timeout;
67         }
68     }
69
70     state connecting {
71         transition local toBusy{
72             trigger busy;
73             target busy;
74         }
75         transition local toRinging{
76             trigger connected;
77             target ringing;
78         }
79     }
80
81     state busy {
82         doAct playbusytone;
83     }
84
85     state ringing {
86         doAct playringingtone;
87     }
88
89     state pinned {
90         transition local toTalking{
91             trigger calleeAnswers;
92             target talking;
93         }
94     }
95
96     state talking {
97         transition local toPinned{
98             trigger calleeHangsUp;
99             target pinned;
100         }
101     }
102 }
103
104 exitPoint activeExit;
105
106 transition external toIdle{
107     trigger callerHangsUp_disconnect;
108     target idle;
109 }
110

```

```
111     transition external toTerminated{  
112         trigger terminated;  
113         target terminated;  
114     }  
115 }  
116  
117 finalstate terminated;  
118  
119 }
```

Bibliography

- [BMS07] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual Syntax for XML Languages. 2007. <http://www.brics.dk/~amoeller/papers/xsugar/journal.pdf>.
- [Con07] World Wide Web Consortium. XML Schema. <http://www.w3.org/XML/Schema>, last accessed September 2007.
- [Dai05] Chris Daily. AST Framework Generation with Gymnast. 2005.
- [EFH⁺07] Sven Efftinge, Peter Friese, Arno Haase, Clemens Kadura, Bernd Kolb, Dieter Moroff, Karsten Thoms, and Markus Völter. *openArchitectureWare User Guide Version 4.2*, September 2007. <http://www.eclipse.org/gmt/oaw/doc/4.2/html/contents/index.html>.
- [fLR07] ANTLR: ANother Tool for Language Recognition. <http://www.antlr.org>, last accessed September 2007.
- [Fou07] Eclipse Foundation. Model To Text (M2T). <http://www.eclipse.org/modeling/m2t/?project=jet#jet>, last accessed September 2007.
- [Fra07a] Eclipse Modeling Framework. <http://www.eclipse.org/modeling/emf/>, last accessed September 2007.
- [Fra07b] Textual Editing Framework. <http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/index.html>, last accessed September 2007.
- [fXL07] XSugar: Dual Syntax for XML Languages. <http://www.brics.dk/xsugar/>, last accessed September 2007.
- [Gra07] Jeffrey G. Gray. Domain-Specific Language Testing Studio. <http://www.cis.uab.edu/gray/>, last accessed October 2007.
- [Gro04] Object Management Group. Human-Usable Textual Notation (HUTN) Specification. <http://www.omg.org/docs/formal/04-08-01.pdf>, August 2004.
- [Gro07] Object Management Group. <http://www.omg.org>, last accessed October 2007.
- [GS07] Miguel Garcia and Paulus Sentosa. Generation of Eclipse-based IDEs for Custom DSLs. Technical report, September 2007. <http://www.sts.tu-harburg.de/%7Emi.garcia/SoC2007/draftreport.pdf>.

- [IBM07] IBM. Business Process Execution Language for Web Services Version 1.1. <http://www.ibm.com/developerworks/library/specification/ws-bpel/>, last accessed September 2007.
- [JB06] Frédéric Jouault and Jean Bézin. On the Specification of Textual Syntaxes for Models. 2006.
- [MH05] Pierre-Alain Muller and Michel Hassenforder. HUTN as a Bridge between ModelWare and GrammarWare - An Experience Report. Technical report, October 2005. <http://planetmde.org/wisme-2005/HUTNAsABridgeBetweenModelWareAndGrammarWareAnExperienceReport.PDF>.
- [Pop04] Remko Popma. JET Tutorial Part 2 (Write Code that Writes Code). May 2004. http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.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.
- [Tol04] Juha-Pekka Tolvanen. Making Model-based Code Generation Work. *August/September*, 2004.
- [War07] Open Architecture Ware. <http://www.openarchitectureware.org>, <http://www.eclipse.org/gmt/oaw/>, last accessed September 2007.