DSL Designing And Evaluating For Ocean Models

Bachelor's Thesis

Serafim Simonov

September 30, 2020

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Dr-Ing. Reiner Jung

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 30. September 2020

Abstract

The development of ocean models requires knowledge from different domains. One aspect of the modeling is the model configuration that takes place in code files or parameter lists. The process of configuration of each ocean model is different and their users must know the differences. To make a configuration of the ocean models is easy we can implement a DSL that generates valid configuration files for each model. In this thesis we design and implement a such configuration DSL. Hereby we study the use cases scenarios involving model parametrization and one ocean model. Based on the findings we designed and implemented the DSL. Although the DSL does not generate all configuration files, the evaluation shows that the concept works.

Contents

Chapter 1

Introduction

1.1 Motivation

Ocean and climate scientist use mathematical models that describe real world. On their basis they can create computer simulations which, in their turn, help to visualize complex processes that can run for a long time in real life and make predictions about further development of ocean and climate. These simulations require computing capacities and specific programming languages, so that various process can be simulated efficiently. The development of such software is not a trivial task and by its nature requires the interdisciplinary knowledge.

On the one side you must have knowledge in mathematics and natural sciences, since a scientific model is based on mathematical formulas that describe various physical and chemical processes. On the other side you must have knowledge in the computer science, especially the software engineering. The software must be flexible and efficient. You must have an understanding of parallelization because simulations can run on multiple computers simultaneously.

There are many examples of such software. In general, a such software consists of various modules and each of them is responsible for some specific aspect of the simulation. On the top level there are configurations files that contain values for model parameters, information about simulation control and input files. The configuration files often have a complex structure and they are written in some programming language. So the user who only wants to configure a model, must have knowledge of the programming language in which the whole simulation software is written and the structure of that software. You must also take into consideration, that in reality you do not work with only one software, so the simple task of assigning initial parameters can become a tedious one.

The solution of this problem can be the development of a universal configuration Domain Specific Language (DSL), with which help we can automatically generate all affordable configuration files. This approach will make it easy for users with no or poor computer science knowledge to use many different ocean model simulations.

1.2 Goals

Our work can be divided in three stages.

Goal 1:Domain Analysis and DSL Design In the first stage it is necessary to collect all the information and gather knowledge about the domain for which we want to implement our DSL. This will help us to plan the further development and define required features of our DSL. Based on this knowledge we design the DSL syntax.

Goal 2: DSL Implementation On the basis of previous stage we will implement our DSL. The Goal of this stage is to provide a working example of the DSL that can produce valid output.

Goal 3: Evaluation In the last stage we evaluate the quality our DSL. The quality criteria concern the ability of the DSL to produce valid output configuration files and user experience.

Chapter 2

Foundations and Technologies

In the following chapter we discuss the Foundations and Technologies, that we use in the process of DSL development. In Section 2.1, we introduce XText Framework. We use it to develop the DSL. In Section 2.2 we discuss the concept of models and metamodels. These are used to create such aspects of the DSL like type system or declarations. To create file generators we partially utilize the GECO Approach. We discuss it in Section 2.3. We utilize Mernik's approach to create our DSL, which divides processes in multiple stages. It is discussed in detail in Section 2.4. In Section 2.5 we introduce Language Server Protocol that we use to deploy our DSL in JupyterLab. We present JupyterLab in Section 2.6.

2.1 XText Framework

XText is a framework for the programming language and DSL development [*XText*]. It provides sets of modern API and domain specific languages that can be used to specify a custom programming language. Based on your configurations it will provide automatically the implementation of your DSL that also can run on the Java Virtual Machine. XText covers such essential aspects of the language infrastructure as parser, linker, typechecker and compiler. The compiler components include abstract syntax tree (AST), the serializer, code formatter, scoping framework, code validation and code generator. In addition to that, XText provides support for Eclipse IDE out of box, making it easy to create custom Eclipse Plugin for your DSL. The core of XText framework is its grammar Language, that itself is a DSL and was designed for the description of textual languages. We use grammar language to specify language syntax defining rules. We can divide them into two main groups.

The first one are the terminal rules that also can be referred as token or lexer rules [*XText Documentation*]. In the first stage of parsing, called lexing, a character input is transfered into a sequence of tokens. The tokens are a strongly typed parts of the input sequence consisting of one or more characters. It is matched by a terminal rule and represent an atomic symbol.

After the lexing stage, the parser walks through the parser rules [*XText Documentation*]. In contrary to terminal rules, the parser rules produce the parser tree, called node model in XText, instead of tokens. So parser rules can be seen as building plans for semantic model.

2.2 Models and Metamodels

The process of any software design is accompanied by modeling. The DSL development is no exception, since any DSL itself can be seen as a model of its domain. As already mentioned in Section 2.1, Xtext generates semantic models out of the grammar specification. So it is important to be aware of the modeling concepts. In this section we introduce two main aspects of modeling.

A model is a representation of the structure, function or behavior of some system. A model can be seen as a set of instances with some attributes. These attributes can have primitive or enumeration values [Jung 2016]. The instances can also reference each other. The instances are defines by classes, data types or property declarations [UML15].

A metamodel should be seen as a model of a model. Meta Object Facility defines a metamodel as a set of classes with typed properties [Object Management Group (OMG) 2006]. Therefore, a class definition in a Java program is a metamodel and the instance of these class is a model, according to these definitions. In context of our thesis a DSL is a metamodel for the model, that has to be configured. In figure Figure 2.1 we illustrate the relationship between models and metamodels.

2.3 GECO Approach

The GECO approach targets primarily the construction and composition of code generators [Jung 2016]. It supports the construction, evolution and reuse of generators. The idea behind it consists of three aspects. First, the generators are divided into specific roles which leads to creation of multiple small generators called fragments. In the second step, these fragments are modularized along metamodel features. Finally, the output of generators is combined usind AOM and AOP technology. Therefore, GECO approach might play an important role during generator implementation.

2.4 DSL Development Process By Mernik

Mernik et al. [2005] propose a guide for DSL development. This approach contains five stages of the DSL development process. The first stage is the decision. The problem is analyzed to decide if a DSL development is necessary. The second stage, the Analysis, requires from the developers to identify the problem domain and acquire the domain knowledge. The third stage is the DSL Design. At this stage we must make a decision, whether it makes sense to reuse an existing DSL or create one from scratch. After the decision is made a developer must develop a DSL specification. The specification can be either formal or informal. The informal approach comprises a set of DSLs programs written in natural language. The formal design makes usage of regular expressions and grammar for syntax specifications. The fourth stage is the implementation of the DSL. In our case we

use XText as mentioned above. The last step is the deployment. In our case XText provides us automatically a language server, that we use with the JupyterLab.

The process of the DSL development is not linear but rather iterative. During the development we might return to previous stages, in the case we must reconsider some aspects of the design or problem analysis. The whole process is illustrated in Figure 2.2.

2.5 Language Server Protocol

The Language Server Protocol is an open JSON-RPS protocol use for communication between source code editors and servers that provide language-specific features [*Language Server Protocol*]. Originally any language was tied to a specific editor. To make it possible to work with any programming language in an editor, we must provide a specific implementation of the language services for every IDE. Language Server Protocol on contrary allows any programming language to be implemented and distributed for any editor that supports such feature.

2.6 JupyterLab

JuypterLab is a a web-based user interface for Project Jupyter [*JupyterLab Documentation*]. We use it in evaluation phase to deploy our DSL. Herby, we use language server protocol, discussed in the section above to activate editor functionalities like error highlighting for our DSL.

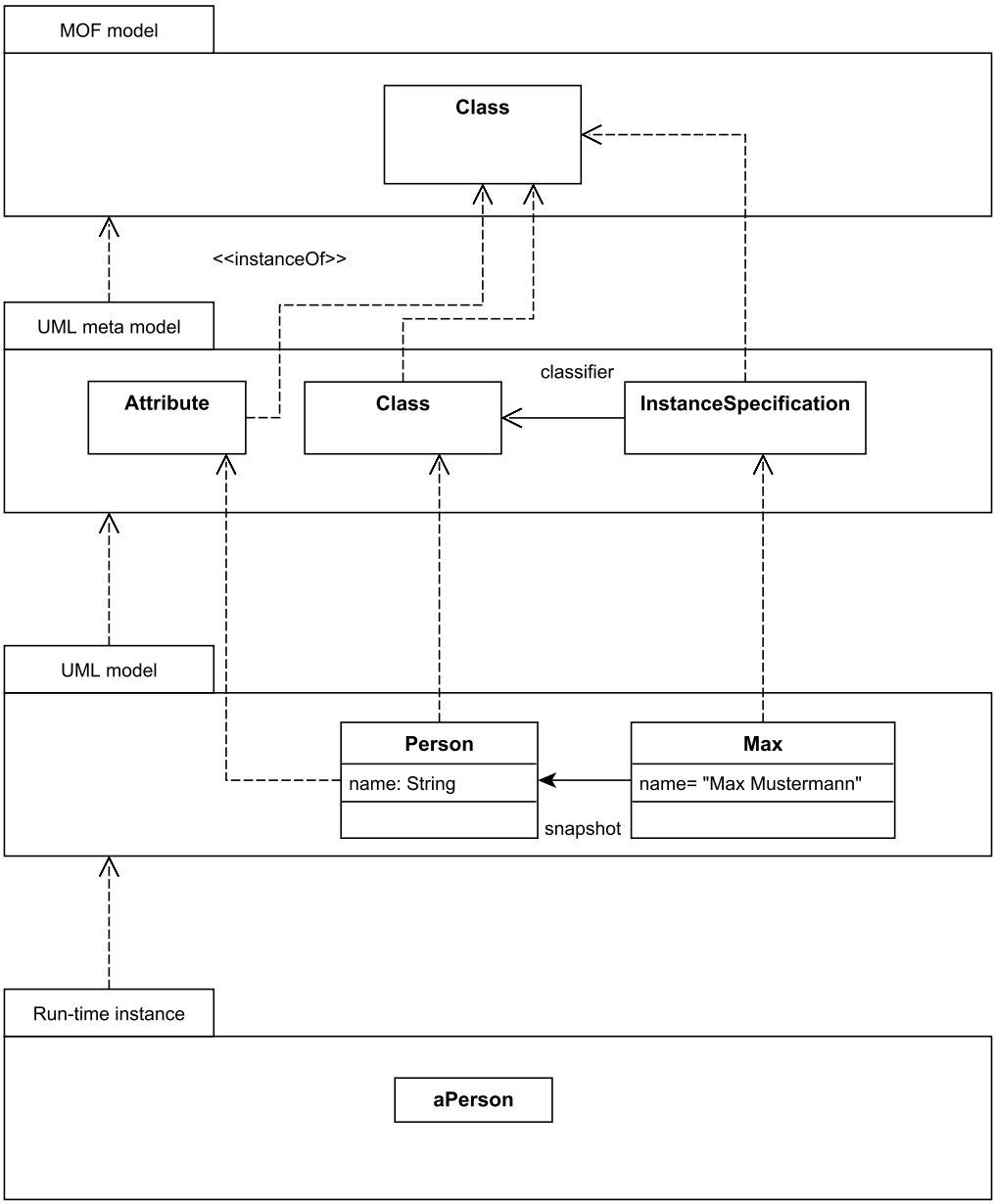

Figure 2.1. Illustration of relationship between models and metamodels on the basis of UML diagrams. Dashed lines represent "instanceOf" relationship [Bruegge and Dutoit 2009].

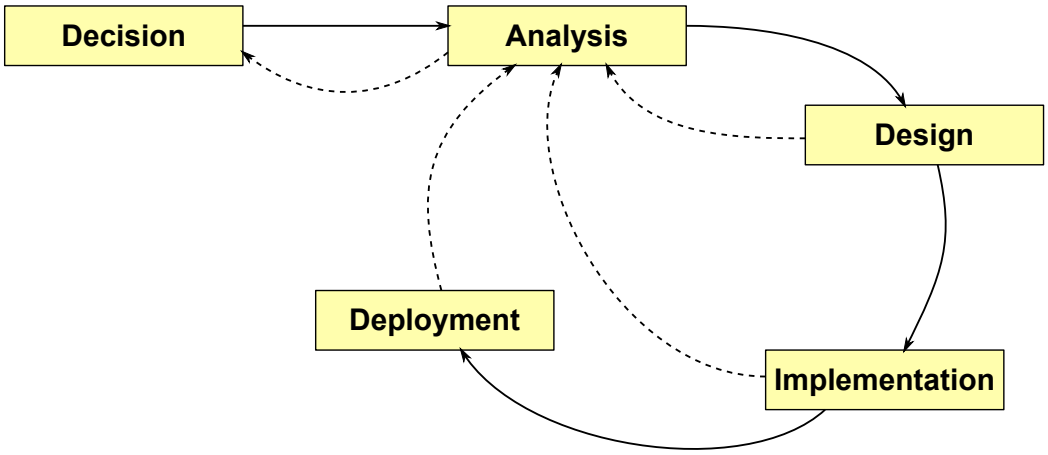

Figure 2.2. Visualization of DSL development process by Mernik et al. [2005].

Chapter 3

Domain Analysis and DSL Design

According to Mernik et al. the phases of DSL development are Decision, Domain Analysis, DSL design, implementation and the deployment. In this chapter, we discuss the domain analysis and DSL design. In Section 3.1, we will discuss the user behavior. This includes understanding of how scientists work, how and when the models are used and configured. At this place it is important to note, that since the main focus of this thesis lays in the configuration of an existing model, we take a look only at those use cases that covers the parametrization and not the development of a new model in a sense of a software. In Section 3.2, we proceed with the analysis of our case study model MITgcm [*MITgcm*]. The goal of the analysis is to identify how the model is configured, what exactly can be configured and what possible constraints and pitfalls have to be taken into consideration. Finally, in Section 3.3, we will formulate concrete requirements for our DSL, based on the information we gathered in previous sections.

3.1 Summary Of Model Run Workflows

In context of the OceanDSL projects interviews with scientists have been held. From these interviews we can derive two use cases that are relevant for the model parametrization.

The parametrization is required during the test of the model. For this purpose a model is set up and it should run test experiments. If a test run fails, the model must be reconfigured. At the end, if the test runs were successful, we should check if the whole model output is plausible, which can again result into a reconfiguration. Figure 3.1 visualizes this process.

Another process, that requires configuration of the model, is setting up of experiments. The parametrization process hereby can vary depending on what kind of experiment we are currently running. The first type of experiments, called sensitivity study, consist of running the same simulations which varies only in certain parameters. The goal of such model runs is to establish, how certain parameters affect the simulation outcome. Another type of experiments involves testing of some concrete scenario. The visualization of this use case is presented in Figure 3.2 visualizes this process.

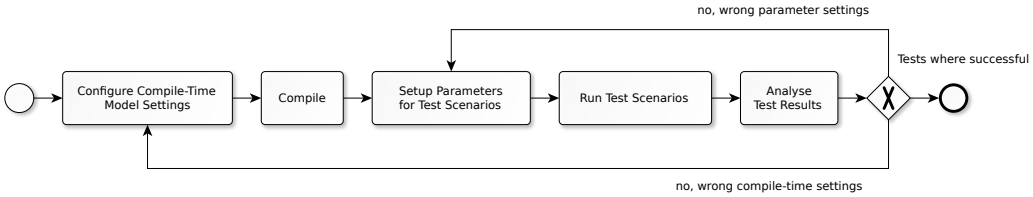

Figure 3.1. Graphical representation of the model test process

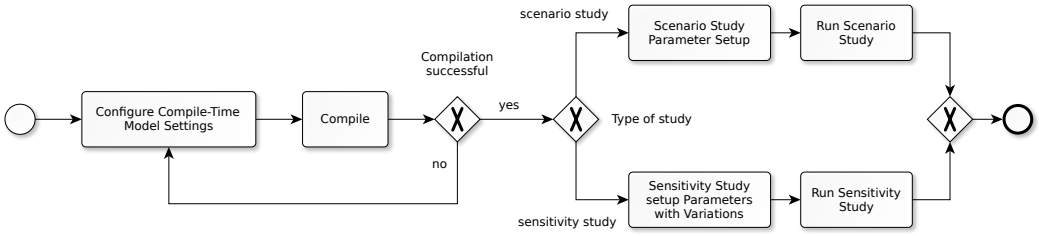

Figure 3.2. Graphical representation of the model run process

3.2 Massachusetts Institute of Technology General Circulation Model

MIT General Circulation Model is a numerical model that allows the study of the atmosphere, ocean and climate. As a software MITgcm model can be considered consisting of two parts. The first part is a set of internal code modules and each of them is responsible for some specific aspect of the computation. The second part can be viewed as an interface for end user, allowing him to specify values for different model parameters. The parameterization in MITgcm takes place in multiple files which can be found in different directories. Besides the actual parametrization, the user can provide some own input files that store specific data. In this section we discuss the structure of a typical experiment setup and the contents of each configuration file.

3.2.1 Structure Of Configuration Directory

The root directory contains several other sub-folders. The folder *code* contains experiment specific code. We can find the following two configuration files. The $SIZE.H$ file, contains the information about underlying computational grid. The file $packages.conf$ is optional and holds a list of optional packages that should be used in the current experiment.

The folder *input* contains, as its name suggests, the input data for the experiment. At least three following configuration files must be present in the experiment set up.

File *data* is used to specify default model parameters that are available direct out of box. Another important file is *eedata*, which is used to parameterize the execution environment. File *data.pkg* is optional and contains parameters relative to the packages used in the experiment. Additionally, the user can add some custom input files. If the user want to specify additional parameters, that are part of some package, he can do so in a *data.{packagename}* file. Additional folders are *result*, *build* and *run*, which however are not used in the parametrization process and are initially empty.

3.2.2 Structure And Contents Of SIZE.H

The structure of a typical *SIZE.H* file can be divided in three parts. At the beginning of every *SIZE.H* of each experiment provided in the MITgcm documentation, there is a big comment section that lists all variables and explains their meaning (see Listing 4.1). In general, all these variable are responsible for some sizing aspect of the simulation, as the file name suggests.

After the initial commentary we have a block where the parameters are declared and initialized (). There values for all parameters specified in *SIZE.H* must be positive integers. The parameters *Nx* and *Ny* are computed by the model. Therefore , in this configuration file *Nx* parameter is therefore set to *sNx*nSx*nPx* and *Ny* to *sNy*nSy*nPy*. Listing 3.2 contains example of parameter initialization with default values.

While the previous block can vary from the experiment to the experiment, the third pard of the *SIZE.h* contents remains constant through all experiments. This section is responsible for setting of of the maximum overlap region size of the array. Listing 3.3 shows this last section.

3.2.3 Structure And Contents Of *data* File

Other than *SIZE.h*, contents of the *data* file can vary a lot from the experiment to the experiment. However, the inner structure stay the same. The configuration parameters, are divided in five numerated groups with the prefix *PARM*. Each group has its own function. Under *PARM01* all parameters affecting the differential equations are listed, *PARM02* contains configurations regarding computing algorithms, *PARM03* contains time related data, *PARM04* specifies the grid dimensions and *PARM05* stores the paths to various input files.

The list of all parameters and the description of their properties can be found in the official documentation of the MITgcm [*MITgcm's user manual*]. Besides the affiliation of the parameters with the *PARM*-groups mentioned above, there are other important things to consider. The parameters can be one of these three types: boolean, numerical or a string.

By the boolean parameters you have to keep in mind, that some of them are exclusive among each other. For example , the description of the boolean parameters, that specify the grid type, implies that only one of them can be set to TRUE in one configuration. Further more, there are numerical parameters, that await different value input depending on the

Listing 3.1. Header of SIZE.H

```
1  CBOP
2  C    !ROUTINE: SIZE.h
3  C    !INTERFACE:
4  C    include SIZE.h
5  C    !DESCRIPTION: \bv
6  C     *==========================================================*
7  C     | SIZE.h Declare size of underlying computational grid.
8  C     *==========================================================*
9  C     | The design here supports a three-dimensional model grid
10 C     | with indices I,J and K. The three-dimensional domain
11 C     | is comprised of nPx*nSx blocks (or tiles) of size sNx
12 C     | along the first (left-most index) axis, nPy*nSy blocks
13 C     | of size sNy along the second axis and one block of size
14 C     | Nr along the vertical (third) axis.
15 C     | Blocks/tiles have overlap regions of size OLx and OLy
16 C     | along the dimensions that are subdivided.
17 C     *==========================================================*
18 C     \ev
19 C
20 C     Voodoo numbers controlling data layout:
21 C     sNx :: Number of X points in tile.
22 C     sNy :: Number of Y points in tile.
23 C     OLx :: Tile overlap extent in X.
24 C     OLy :: Tile overlap extent in Y.
25 C     nSx :: Number of tiles per process in X.
26 C     nSy :: Number of tiles per process in Y.
27 C     nPx :: Number of processes to use in X.
28 C     nPy :: Number of processes to use in Y.
29 C     Nx  :: Number of points in X for the full domain.
30 C     Ny  :: Number of points in Y for the full domain.
31 C     Nr  :: Number of points in vertical direction.
32 CEOP
```

Listing 3.2. Parameter specification in *SIZE.H*

```
      INTEGER sNx
      INTEGER sNy
      INTEGER OLx
      INTEGER OLy
      INTEGER nSx
      INTEGER nSy
      INTEGER nPx
      INTEGER nPy
      INTEGER Nx
      INTEGER Ny
      INTEGER Nr
      PARAMETER (
     &           sNx =  30,
     &           sNy =  15,
     &           OLx =   2,
     &           OLy =   2,
     &           nSx =   2,
     &           nSy =   4,
     &           nPx =   1,
     &           nPy =   1,
     &           Nx  = sNx*nSx*nPx,
     &           Ny  = sNy*nSy*nPy,
     &           Nr  =   4)
```

Listing 3.3. Maximum overlap region specification in *SIZE.H*

```
C     MAX_OLX :: Set to the maximum overlap region size of any array
C     MAX_OLY    that will be exchanged. Controls the sizing of exch
C                routine buffers.
      INTEGER MAX_OLX
      INTEGER MAX_OLY
      PARAMETER ( MAX_OLX = OLx,
     &            MAX_OLY = OLy )
```

Listing 3.4. File data as it can be found in example project

```
# Model parameters
# Continuous equation parameters
 &PARM01
 viscAh=4.E2,
 f0=1.E-4,
 beta=1.E-11,
 rhoConst=1000.,
 &
# Elliptic solver parameters
 &PARM02
 cg2dTargetResidual=1.E-7,
 cg2dMaxIters=1000,
 &
# Time stepping parameters
 &PARM03
 nIter0=0,
 nTimeSteps=10,
 deltaT=1200.0,
 pChkptFreq=31104000.0,
 chkptFreq=15552000.0,

# Gridding parameters
 &PARM04
 usingCartesianGrid=.TRUE.,
 delX=62*20.E3,
 delY=62*20.E3,
 &
# Input datasets
 &PARM05
 bathyFile='bathy.bin'
 zonalWindFile='windx_cosy.bin',
 &
```

grid type. The parameter *xgOrigin*, for instance, awaits a number of meters, if the chosen grid type is cartesian, or a value representing the latitude otherwise. The latter means in practice that the value is between -180 and 180.

By the numerical values we have to distinguish between integers and real number format. The integer format is often required by parameters, that are acting like some sort of "mode selector". An example for such parameter type is *rwWriteBinary*, that expects either an integer 32 or 64. By the parameters, that expect a real number, we must sometimes care for metrical units. For example, parameter gravity expects a value specified in m/s^2. So the user must ensure, that the value he wants to type in, is converted accordingly to these requirements by himself.

The string parameters can also be divided in two groups. One group contains those, parameters that act like a mode selector. There is a set of acceptable string values that describe some special mode in which the simulation must run. An example for such parameter is buoyancy relation which expects one of these three values: "OCEANIC", "OCEANICIP" or "ATHMOSPHERIC". Another parameter group are all the parameters belonging to *PARM05*. As already mentioned before, this parameters are placeholders to custom simulation input files. They simply expect the name of a file.

3.2.4 C Preprocessor Options

There are parameters that require us to provide additional information to the configuration set so that they can be used efficiently or even be used at all . Some parameters like for example *nonHydrostatic*, requires us to redefine C Preprocessor Options. By the default the preprocessor options are already set. However, we can redefine them in *CPP_OPTIONS.h* file which can be found in folder code.

3.2.5 Structure and Content of *eedata*

The file *eedata* has the least content among all mandatory configuration files we have already discussed. This file stores the information about the execution environment, in which the model must run. According to the official documentation there are only nine parameters that can be specified in this file. Six of them are boolean flags, that can be used to turn on and off some functionalities. Three other parameters are integers. For example *nTx* and *nTy* specify number of computation threads in the corresponding directions. The parameter *maxLengthPrt1D* is used to configure the maximal length of arrays printed to standard output.

3.2.6 Structure and Content of optional files

In the previous sections we examined the contents of files, which are mandatory for each experiment, that means, that they must be present at any model configuration. However, there are some optional configurations, that can be made. Besides the default parameters

that are available directly out of box, you can use specific configuration provided by several packages. However, one should abstain from looking at the packages like an external resource. According to the documentation of the MITgcm, these packages are components of the full MITgcm code.

If the user wants to use some additional package, he must declare it in *package.conf* and enable it in *data.pkg*. Further the parameters must be specified in distinct *SIZE.H* and *data* files, that typically get an extension corresponding to the name of the package. However, while some of these package specific are similar to their corresponding pedants, some other can have a different structure.

3.3 Summary of the Analysis of the MITgcm

In the previous sections we discussed the structure of the configurations files and use cases of the parametrization. Based on these findings we formulate following list of requirements:

1. The DSL must be designed for modelers and model users.
2. As modelers do not use classic IDEs, the DSL must be usable with and embedded in their standard tool chain, i.e., usable with Vi and Emacs, and integration in Jupyter.
3. The DSL must generate configuration files.
4. The DSL must support Generators for different models.
5. Since the models can be reused, the DSL must provide import functionality.
6. The DSL must not allow specification of invalid parameters
7. The DSL must support specification of units of measurement.

3.4 DSL Syntax Design

Listing 3.5 represents the desired syntax of our DSL. The lines 1-3 contain Header, which stores the information about the imported files. Lines 4-14 are the main block. The name of the this block should reference the name of the model experiment. After the colons it is required to specify which model must be used. In our case it is the MITgcm, but since our DSL must be universal, MITgcm should not be the only option. The main block contains ParameterBlocks, which names the user can specify freely. The names could be used as identifiers to import concrete blocks from other files as it is shown in line 13.

Listing 3.5. Draft of the desired syntax of the DSL

```
Header{
   import filename.ext
}
ExperimentName:nameOfUsedModel{
   ParameterBlockNameA{
            booleanParameter=true
            pathtoFileParameter="filepath"
                 }
   ParameterBlockNameB{
            parameterWithMeasurementUnit= 2.5 mg
            parameterWithoutUnit = 2
                }
   import ParameterBlockC from filename.ext
}
```

Chapter 4

Implementation of the DSL

In this chapter we discuss the implementation of our DSL. In Section 4.1 we explain the implementation of the syntax. To implement such functionalities like typing, parameter declaration and the support of units of measurement we designed special meta models. They are discussed in Section 4.2. The units of measurement are character sequences. We must implement a parser to transform these format in special *Unit* objects, that can be used in validation and generation phase. The implementation of the unit parser is covered in Section 4.3. The semantics checks are performed in a validator class, about which implementation we talk in Section 4.4. We also must create a generator that will transform the input written in our DSL into desired outputs required by MITgcm model. We cover this topic in Section 4.5.

4.1 Grammar

The configuration DSL grammar allows to set parameters for a specific model. It comprises of three aspects (cf. grammar file GCMDsl.xtext) header, semantic blocks and parameter specification which are discussed below.

Each XText grammar starts with a header where we declare the properties of the grammar. The contetns of header are presented in Listing 4.1. In line 1 of we declare the name of the language. This can be done through Java's class path mechanism. The name of the language must be the same as the filename in which we specify the grammar. In line 3 we order XText to generate an EPackage with the name *mitgcm/GcmDsl*. An EPackage is a part of the Ecore model. EPackage contains EClasses, EDataTypes and EEnums. The EModel is used to describe the structure of the instantiated objects [*XText Documentation*]. Line 3 imports the Declaration metamodel which we introduce in Section 4.2. The declaration models define which parameters exist for a specific scientific model.

Listing 4.1. Header of the grammar file.

```
1 grammar org.xtext.mitgcm.GcmDsl with org.eclipse.xtext.common.Terminals
2 generate gcmDsl "http://www.xtext.org/mitgcm/GcmDsl"
3 import "http://xtext.org/mitgcm/declaration" as declaration
```

Listing 4.2. Rules that specify the structure of Blocks.

```
1  Model:
2     mainblock=MainBody;
3
4  MainBody:
5     name=ID ':' model=[declaration::DeclarationModel|ID] '{'
6     (paramGroups+=ParamGroup)*
7     '}';
8     // Subgroups
9  ParamGroup:
10    name=ID '{'
11    (params+=Param)*
12    '}';
13
14 Param:
15    declaration=[declaration::ParamDeclaration|ID] '=' value=Literal (unit=Unit)?;
```

The main part of the grammar consists of rules to relate the configuration to a specific scientific model and setup the parameters. Listing 4.2 presents rules *Model*, *MainBody*, *ParamGroup* and *Param*.

The Model rule is the starter rule of the GCMDsl grammar. This rule states, that our model consists of one *MainBody* that is assigned to the feature called *mainblock*. After the starter rule we define the rule for *MainBody*. This rule states that *MainBody* has features *name*, *model* and *paramGroups*. The asterix sign indicates that we can assign multiple values to *paramGroups*. They also must be enclosed in curly braces.

The feature *name* is simply a character sequence. To describe this sequence in XText we used the predefined terminal rule ID. This rule states that a name can start with a letter or a underscore, followed by letters , underscore, digits, exponent symbol or a minus sign. To the feature model we assign an object of class *DeclarationModel* which is generated by the imported metamodel. The objects are referenced by their identifier and we use the terminal rule ID to parse it. To the feature *paramGroups* we add an arbitrary number of *ParamGroup* objects.

The definition of a *ParamGroup* rule is similar to the rule discussed above. It consists of a block enclosed in curly braces that can be identified by its *name*. *ParamGroup* contains an arbitrary number of the*Param* objects that are assigned to the feature *params*.

The rule *Param* specifies the syntax for the parameter specification. In this rule we declare three different features that store information about parameter name, its value and an optional unit of measurement. To the feature *declaration* we add a ParamDeclaration object. This object also originates form the imported model. The cross-reference is established through the terminal rule *ID*, like in the rule *MainBody*. The rule *ID* is used to specify an

Listing 4.3. Terminal rules and the hierarchy of *Literal*.

```
Literal:
   Primitive | Array;

Array:
   {Array} '{' elements+=Primitive* '}';

Primitive:
   Numeric | Text | Bool;

Unit:
   unit=ID;

Bool:
   value=BOOLEAN;

Text:
   value=STRING;

Numeric:
   (quant=DIGIT '*')?
   value=DIGIT ;

terminal BOOLEAN:
   'true' | 'false';

terminal DIGIT:
   ('-')? ('1'..'9') ('0'..'9')* ('.' ('0'..'9')*)?;

@Override
terminal ID:
   ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '^' | '-')
       *;
```

arbitrary character sequence. Feature *value* stores a *Literal*. This rule delegates to the rules *Primitive* or *Array*.

The hierarchy of *Literal* rule is presented in Listing 4.3. The rule primitive delegates further to three other rules: *Bool* , *Text* and *Numeric*. As the names of these rules can suggest, we use these rules to specify different kind of textual input. The rule *Bool* says that the feature *value* is either true or false. The rule *Text* means that value can be a string. To describe numerical literal we had to define a custom terminal rule named *DIGIT*. Since

numerical parameters can be either integers or float, we want our grammar be flexible in that aspect. A number can start with a minus sign, which in the DIGIT rule is achieved through using "?" operator. This operator means that this input is optional. A number should not start with "0" and can contain an optional floating part that starts with a dot. The rule *Numeric* utilizes *DIGIT* that is assigned ti features *quant* and *value*. Feature *quant* is optional and is used to describe how many identical *value* primitives are stored in an array. If *quant* is omitted, the literal is handled as a numerical primitive.

Arrays literal are simply a collection of primitive literals enclosed in curly braces.

4.2 Metamodels

When the user specifies the parameter we must ensure that this parameter exists in the used ocean model. We achieve this through implementation of a validator that scans the source file and checks the validity of parameter names and value types. This is, however,a tedious and not efficient way to ensure the parameter validity. We would need to implement a validator for each distinct ocean model that is supported by our DSL. Different models can have different parametrization. MITgcm alone has over 300 different parameters with different properties [*MITgcm's user manual*]. Directly adding parameter names to the validation is, therefore, tedious and hinders efficient maintenance and evolution of the DSL. To solve this problem we externalize the information in a separate models which hold all information on model parameters. Our idea is to create a special metamodel. An instance of that metamodel contains the description of the ocean model and its parameters. We Used Ecore Modeling Framework (EMF) to develop all meta models. [*Ecore Modelling Framework*].

4.2.1 Metamodel Overview

Before we discuss the structure and usage of all metamodels used in the development of our DSL, we must understand the relationships between all of them. In Section 4.1 we have discussed the grammar implementation. A grammar itself can actually be seen as a metamodel. It is a description of how a source file contents should be composed. A content of a source file is an instance of the grammar metamodel. XText creates a corresponding ECore model out of the specified grammar and generates appropriate java classes as the implementation of that ECore model. These classes are used internally to parse a textual input. In our grammar we also referenced another metamodel, that we call the Parameter Declaration metamodel. This metamodel is used to describe the parameters and their properties. To specify the properties type and unit of measurement, we also developed corresponding metamodels. Figure 4.1 shows the relationship between all these metamodels.

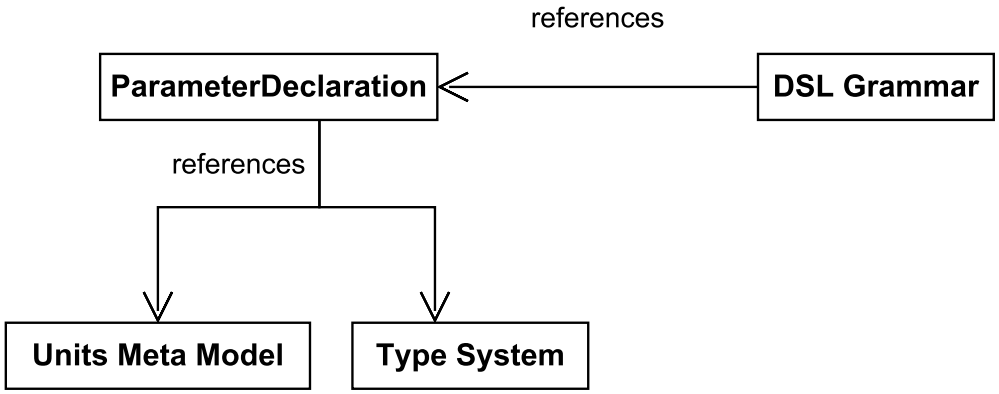

Figure 4.1. Relationship of the metamodels of the DSL grammar, Parameter Declaration, Units and Type System.

4.2.2 Type System Metamodel

In the article Introducing Type-Systems to Xtext-Languages, Jung et al. [2013b] have shown how to develop and use types in the DSL language created in XText. This approach also includes the development of an appropriate metamodel. Our DSL itself does not need the type system because we do not need any kind of variable declaration. However, we need to specify types of the parameters in our declaration meta model. Our type system comprises only of primitive types and arrays (see Figure 4.2). Furthermore, we must be able to reference types. To achieve this we reused the implementation of the Kieker instrumentation record language [Jung et al. 2013a]. This project also utilized the approach discussed above, and for this reasons it perfectly fits to our project. The idea of the implementation is to create an enumeration called *PrimitiveTypes*, where we list all primitive types that must be available in our type system. Than we create a class, named *PrimitiveTypeResource* that inherits from *ResourceImpl*. This class is used to store the instance of the type system metamodel. The next step is the implementation of the *PrimitiveTypeProvider*, that allows to query the *PrimitiveTypeResource* and its factory class.

4.2.3 Metamodel For Units

In our declaration model we need to specify SI units for our parameters. Our model should model following behavior [*The International System of Units*]:

- A unit consists of the name and a prefix part that is optional.
- We can also create composed units by multiplying or dividing them.

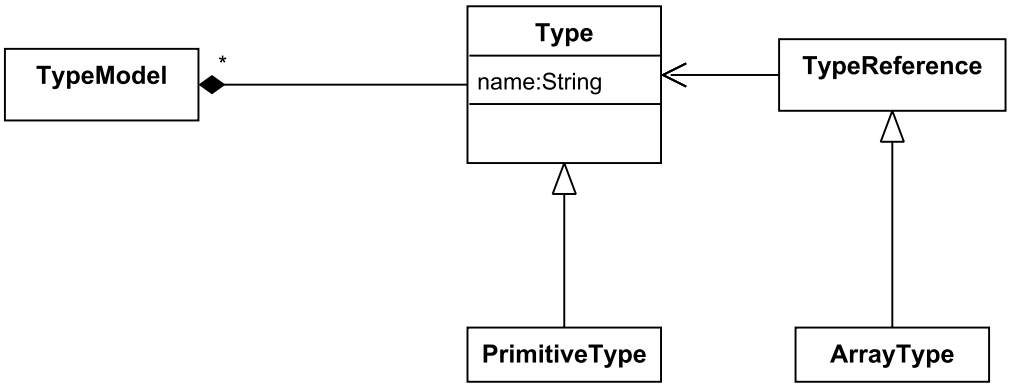

Figure 4.2. UML diagramm that visualizes the type system metamodel.

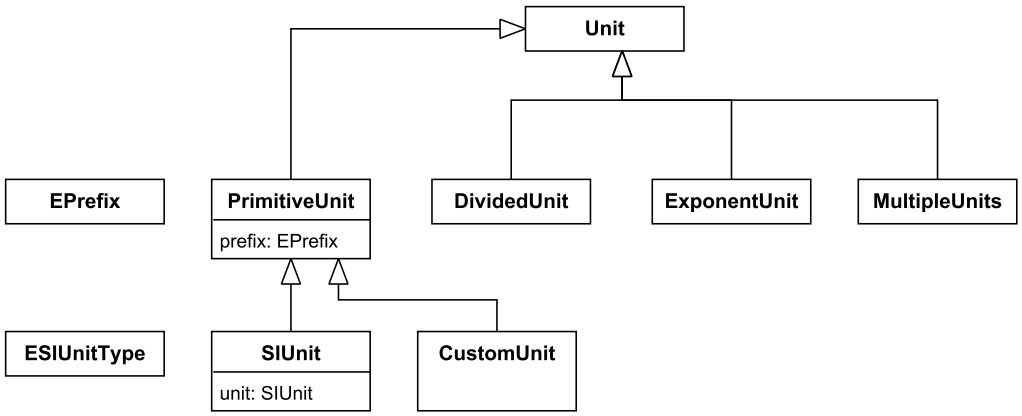

Figure 4.3. UML diagram that shows the structure of the unit metamodel.

▻ It must possible to specify an exponent for a unit.

Based on this information we developed a model as it is depictedin Figure 4.3.

4.2.4 Declaration Metamodel

We developed a contained metamodel to model parameter declaration. The containment property is important, since our model consists of several elements, which cannot exist on their own. In particular we want to express, that our declaration model consists of reference groups. These reference groups are used by the generator to identify which

parameter belongs to which configuration file. The reference groups, have parameters, that are associated with them. These parameters can also be characterized by their name, type and, optionally, unit of measurement. Figure 4.4 depicts the resulting metamodel.

4.2.5 Usage of Declaration Model

Above, we introduced several metamodels including Type System metamodel, Unit metamodel, and Parameter Declaration metamodel. To be able to use them in the DSL, we must provide an istance of the Declaration model.

For this purpose we created Java class called *MITgcmParameterSpecification*. This class offers a public method *getDeclarationModel()*, that returns an instance of the *Declaration model*. The model itself is created in private method *createDeclarationMode()*l. In this method we assemble the *Declaration model* using its containment property. At the very beginning we instantiate an empty *DeclarationModel* object that does not contain any reference group. Then we create distinct reference groups in separate private methods and add them to the model. The creation of a reference group involves specifying the parameters and their properties.

To make the DeclarationModel visible to XText, we follow the same pattern as with the type system integration. We implement a resource class (*ModelResource*) and the provider classes (*ModelProvider*). Additionally, we have to implement a *ModeGloballScope* class, that inherits from *AbstractScope*. This class is used to introduce a *Declaration Mode*l to the XText scope. In the last step we need to register ModelGlobalScope as a runtime module of our DSL in *MitgcmGlobalScopeProvider*.

4.3 Unit Parser

In our DSL-grammar units of measurement are seen as a character sequence. We need to implement a parser that can transform such character sequence into corresponding unit objects that are compatible with the unit metamodel. This will allow us to perform validator checks and units conversion during the generation process. The idea behind our implementation is to look at the parser as a finite state automaton (FSA). This FSA accepts the words that represent a valid unit. A word can start with characters that represent a prefix or a unit symbol. Other allowed characters are '*' and '/' that act like delimeter. The multiplying sign is used to denote a unit composed of unit multiplication. The slash sign is used to denote divided unit. A unit can be raised to the power by writing '^' sign followed by a number.

There are 20 prefixes and five base SI units symbols [*The International System of Units*]. In our case we also must be able to parse two derived units: Pascal and Joule. 16 prefix symbols out of 20 are not ambiguous. That means that they are represented with a single character and that character is not a starting symbol of any other prefix or unit. In particular prefixes 'm', 'P' 'c', 'd' and 'da' are ambiguous, since they do not have this property. Similar

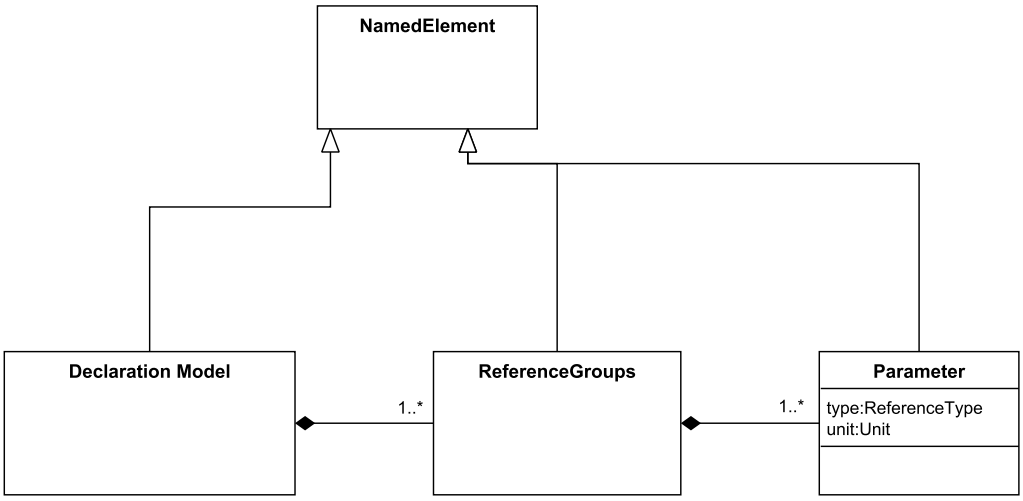

Figure 4.4. UML diagramm representing Declaration ECore Model.

to prefixes, we must take care of unit symbols 'm', 'ca', 'mol' and 'Pa', since their first character can also be a part of another unit or prefix.

Instead of designing one big state automaton, we split the task into multiple smaller automatons. We choose which automaton must run depending on what is the first character of the input. Figure 4.5 features a case, in that we read an unambiguous unit name symbol. Figure 4.6 is an automaton for units starting with an unambiguous prefix. Small letter 'm' is ambiguous , since it can be either represent the prefix 'mili' or be first letter of units meter and mole. The state automaton, that recognizes units starting with this letteri shown in Figure 4.7. Another cases, that we also must distinguish are units starting with letter 'c','d' and uppercase 'p'. Each of these cases are presented in Figure 4.8, Figure 4.9 and Figure 4.10.

After an automaton recognized a single unit, we check if it is followed by any delimiter ('*' or '/') or an exponent sign. If a delimeter is present, we restart our automata and process the next unit that follows after the delimiter sign. If the unit has an exponent, we read the number until the delimeter is read or we reach the end of the input. In case we read any invalid character at any stage of parsing process, we know that the whole input is not a valid unit.

We implemented these ideas in Java class *UnitParser*. This class has a public method *parse()* that executes the parsing process. This process runs in a while-loop until we read an invalid sign or reach the end of the input. Inside of this loop we have multiple if-else statements that are used to decide, which of the small automatons should process the input. The automatons are implemented in separate private methods with corresponding names.

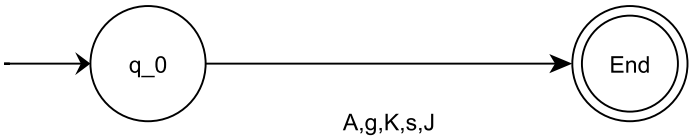

Figure 4.5. Automaton that accepts unambiguous units.

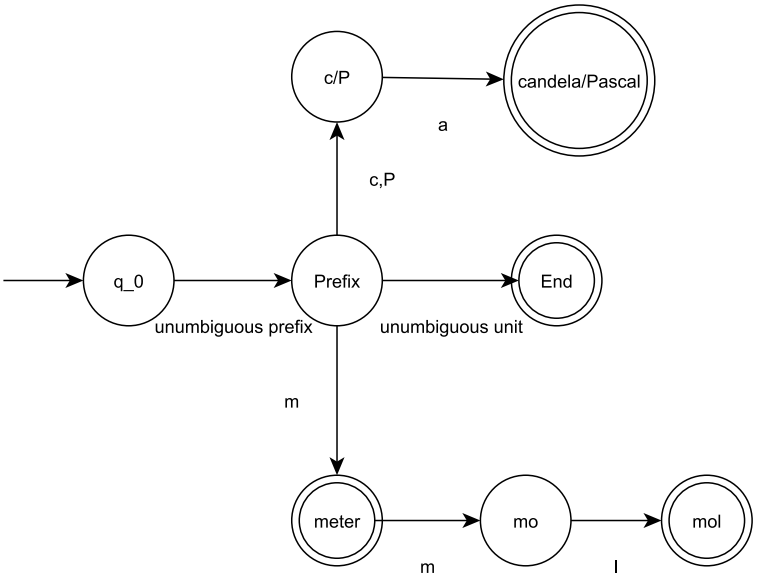

Figure 4.6. Automaton that accepts units starting with unambiguous prefix

To handle different delimiters and composed units we introduced two private lists in our class. At first, we add all units that are separated by "*" to the list *currentUnitStrea*. Then, if we read a '/ ' or the input is empty we create a corresponding unit object based on the length of *currentUnitStreak* and add it to *unitCol*. The *unitCol* is used at the end of the parsing process to create a final unit.

4.4 Validator

The grammar, described Section 4.1, defines the syntax of our DSL, but does not provide semantic checks. This task is addressed with a validator within an XText project. In our case, the validator is called MITGcmValidator. Listing 4.4 presents an excerpt of a semantical

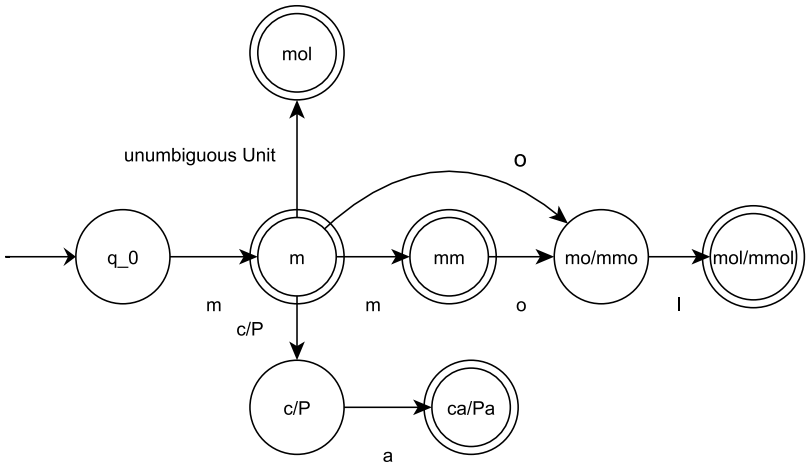

Figure 4.7. Automaton that accepts units starting with 'm.'

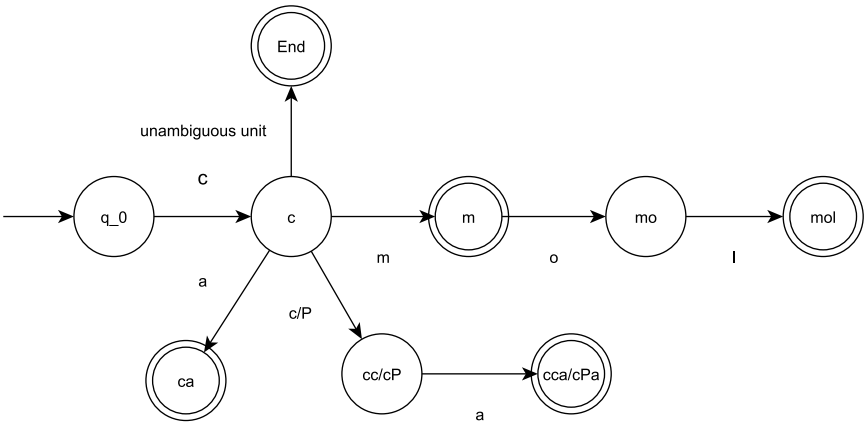

Figure 4.8. Automaton that accepts units starting with 'c'.

validation. In this case we want to check if the unit specified by the user is compatible to a required one. To notify XText that a method is a validator, we use the annotation "@Check". Since this check involves some property of a *Param* rule, we pass a corresponding object as a method parameter. To verify the unit, we check at first if a unit was specified at all. If this is the case, we parse the input to a Unit object using the parser we implemented before. In a Utility class we wrote a method that performs compatibility checks between two given units. We use this method to compare parsed unit with one specified in parameter

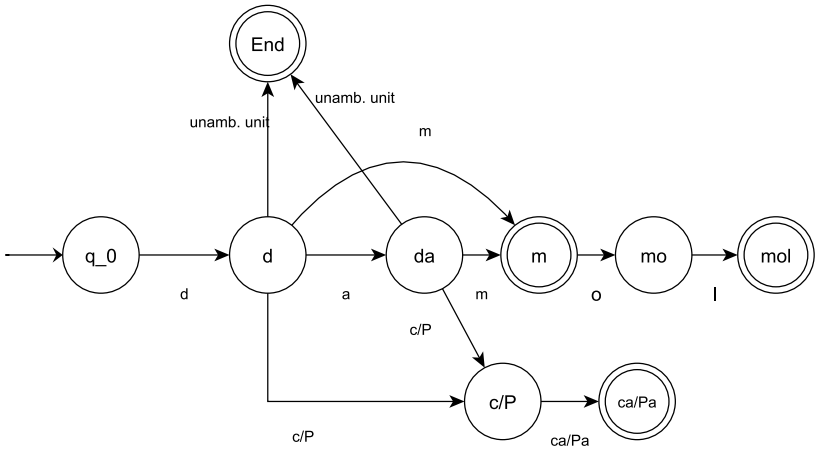

Figure 4.9. Automaton that accepts units starting with 'd'.

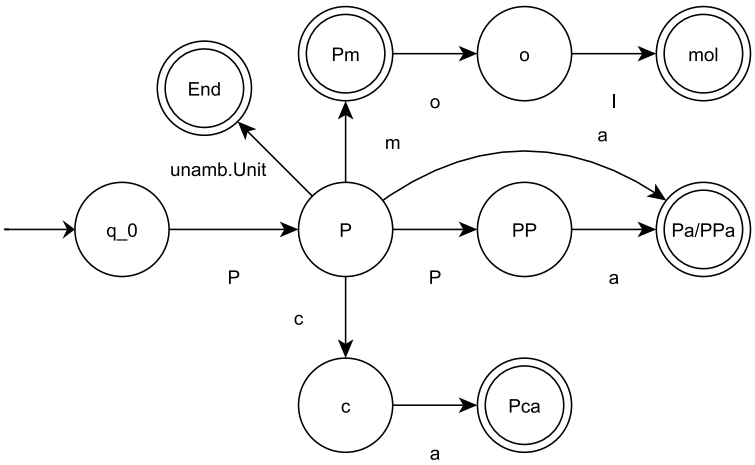

Figure 4.10. Automaton that accepts units starting with 'P.'

declaration. If this checks fails, an error is thrown that is displayed in the editor.

All semantic checks follows the same principle as in introduced excerpt. Beside unit verification, we perform type checks for parameter values. In Chapter 3 we learned that there are parameters that can have a specific values range. To ensure that the user can specify only allowed values, we also implemented validation checks.

Listing 4.4. This method checks whether specified unit is valid.

```
1  @Check
2     def checkUnitCompatibility(Param parameter){
3        if(parameter.unit!==null){
4           var x=new UnitParser2(parameter.unit.unit)
5           var res=x.parse()
6           if(!UnitsValidatorHelper.areCompatible(res,parameter.declaration.unit)){
7                 error("Specified unit is not compatible with the requiered",
                        GcmDslPackage.Literals.PARAM__UNIT,
8                 INVALID_NAME)
9           }
10       }
11    }
```

4.5 Generators

The last important aspect of our DSL is its generator. The generator must transform source code written in the DSL to a a valid MITgcm project. In Chapter 3 we learned that the minimal requirement for a MITgcm experiment run are following three files: *SIZE.H*, *data* and *eedata*. We implemented a distinct generator for each of these files. Hereby, we applied Strategy design pattern. Each generator implements a *MITGcm* interface and we call them subsequently in *GCMGeneratorContext*. Figure 4.11 shows UML diagram representing our generator design. In subsequent subsection we discuss the details of the generators implementation. We used XTend programming language, because it offers a built in template language. This functionality allows to describe the contents of the configuration files.

In general, the code generation process can be divided in a sequence of multiple input transformations [Jung 2016]. In our case we obtain the list of specified parameters, filter them depending on reference group, and convert the specified values to desired output.

We defined the interface called *MITGcm* with methods *generate()* and *getName()* before. Each file generator must implement this interface. The method *getName()*, returns in each case a name of the corresponding configuration file. The method *generate()* performs the input transformations.

The Listing 4.5 demonstrates the implementation of the *generate()* method in *EeData-Generator*. In line two we obtain the required parameters. For that purpose we wrote a method in separate utility class that queries the list of all parameters specified in DSL. In lines 4-7 we use Xtend template language to write the parameters in that file. The method *convertLitToString()*, is another method defined in the utility class and it simply prints the value of the Literal. The implementation of the same method in the class *SizeHGenerator* follows the same principle.

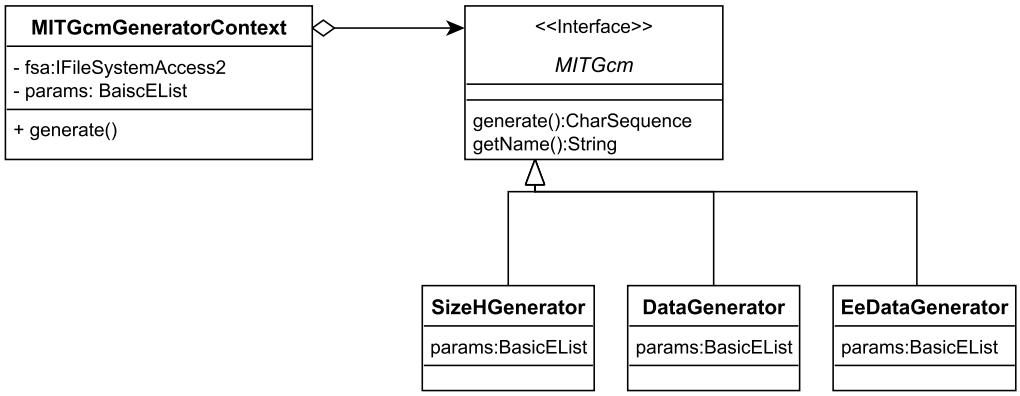

Figure 4.11. Generator design.

By the implementation of the generator for *data* we must take into account additional aspects like unit conversion and multiple reference groups. We wrote corresponding methods in the utility class *ModelQueryDataAndAggregation*. The method *extract()*, like in example above, is used to extract parameters of different reference groups. The method *formatOutput()* prints the formatted value of the parameter. Hereby it performs unit conversion by numerical values, if necessary. To convert the units, we defined a private method *convertValue()*. This method delegates internally the conversion to smaller methods depending on what kind of measurement unit must be converted.

We created a class *MITGcmGeneratoContex*t that can be used to run all generators. The class simply stores the generators in a list. We provided a method *addGenerator()*, to add a generator to the list. In the method generate we call *generateFile()* for each saved generator.

We use *GcmDslGenerator* as a client class. In the overridden method *doGenerate()* we instantitate an object of class *MITGcmGeneratorContext* and add the smaller generators. At the end we simply run the generation process by calling the *generate()* method of *MITGcmGeneratorContext*.

Listing 4.5. Implementation of *generate()* method for eedata configuration file.

```
1   override generate() {
2      var eeParameters = ModelQueryEeData.extract(parameters);
3      if (!eeParameters.isEmpty) {
4         '''
5         «" "»&EEPARMS
6         «FOR e : eeParameters»
7          «" "»«e.declaration.name»=«ModelQueryEeData.convertLitToString(e.value)
               »,
8         «ENDFOR»
9         «" "»&
10
11        '''
12     }else{
13        '''
14        «"  "»&EEPARMS
15        «"  "»&
16        '''
17     }
18  }
19 }
```

Chapter 5

Evaluation Of DSL

We developed a DSL that can be used for the configuration of ocean models. In particular our DSL is capable of generation configuration files for MITgcm. In following chapter we evaluate the result of our implementation. The evaluation is based on the experiment, that should demonstrate the abilities of the DSL.

In Section 5.1, we describe the technical setup for our experiment. This includes the configuration of the JupyterLab and MITgcm software. In Section 5.2, we discuss three test scenarios from MITGcm documentation that we chose for the experiment. Section 5.3 explains the execution of the experiment. After the experiment is performed, we discuss our finding in Section 5.4

5.1 Evaluation Setup

Configuration Of JupyterLab We test the usage of our DSL in JupyterLab. To configure JupyterLab we use Language Server Protocol (see Section 2.5). XText provides language server executable by the project compilation. The LSP allows DSL to be used in any editor that supports this technology. However, LSP does not provide support for code compilation. This functionality must be added manually in each editor.

We reuse a JupyterLab setup that was developed in OceanDSL project. This setup includes a docker build and a language kernel for the Jupyter written in Java. We must make some changes in this setup to enable language server usage and code generation for our specific case. In the kernel project we must adjust the method *eval()* in class *ExampleKernel*. In this method we call static method *generate()* from class named *Generator*. In *generate()* we call the generators we implemented for the DSL and write the result of the generation into files with corresponding names.

MITGcm Softwate [*MITgcm*] The project OceanDSL provides a Docker build that can be used to compile and run MITGcm models. In addition to that this project provides an octave script, that can be used to visualize the result of each model run.

5.2 Evaluation Scenarios

The official documentation of the MITgcm provides multiple example experiments [*MITgcm's user manual*]. During the process of evaluation we try to recreate the configuration files using our DSL. We picked up following MITgcm experiments described in the documentation.

1. Baratropic Gyre
2. Barotropic adjustment problem on latitude-longitude grid
3. Global ocean simulation in pressure coordinates

The first two experiments requires only those configuration files for which we have implemented the generator. The last experiment, requires additional configuration of *CPP_OPTIONS* file and *package.conf*. However, this experiment can also be used as a test scenario. We can verify if the files produced by our DSL can replace the the original parametrization.

5.3 Evaluation Plan

The goal of the evaluation experiment is to test the feasibility and practicability of the DSL. The criteria of the feasibility is the ability of our DSL to produce valid configuration files. The practicability includes the quality and intuitivity of the design and quality of produced configuration files.

Feasibility To evaluate the feasibility, we use our DSL to specify the configuration of the three scenarios discussed in Section 5.2. Hereby we perform the configuration in JupyterLab editor and produce the desired files. We manualy replace the original configuration files with those we produced. We run the scenarios and compare the output with the one produced using original configuration files.

Practicability To test the practicability we let a test person try to configure a model. For this purpose we provide JupyterLab setup. We also created a questionnaire that contains questions about demographics, professional background and experience with Ocean Models. The questionnaire contains also concrete tasks, that a test person must try to complete.

5.4 Result

Feasibility We could create the configuration files for all three use cases. It was also possible to use these files instead of their original counterparts.

At the end of the run of every scenario we could produce the same output as the original setup.

Practicability The content of the produced files are almost identical to that of the originals. However, there are some readability issues regarding the parameters that expect array values. The values are printed one per each line. There is also some minor formatting issue in the SIZE.h.

We discovered also one issue with the language server. It considers a Jupyter notebook file as a whole. We can specify two different models in one notebook, but language server doesn't provide adequate editor functionalities for the second model.

The test persons provided some feedback about the DSL using JupyterLab The creation of a DSL file was easy based on the knowledge of the DSL syntax.. One issue was lack of context help in editor, while creating new generation file. Summing up, it is required to pay more to attention to better user experience.

Chapter 6

Conclusion And Fututre Work

The development and evaluation of the Configuration DSL is a first step towards a usable tool for ocean scientists. In the following we will give a summary on the development and evaluation in Section 6.1 and layout the future development and features we intent to implement, in Section 6.2.

6.1 Summary

In the previous chapter we discussed the evaluation of the Configuration DSL in terms of feasibility and practicability. Evaluation of feasibility revealed that the DSL can generally be used to generate valid configuration output. However, at the same time in its current state the DSL can provide partial configuration for some MITgcm experiments.

The DSL is capable of recognizing valid parameter names and assign them to correct configuration files. Furthermore, the DSL provides support for specification of units of measurement. Currently, it is possible specify basic and derived SI units and combine them using arithmetic operations. However, we cannot specify complex units using parenthesizes. During the generation of configuration files we convert the values according the default unit of measurement. Our DSL can also check specified values of certain parameters preventing erroneous input.

We implemented generators for configuration files *SIZE.h, data* and *eedata*. Thus, it is possible to provide configuration for simple experiments. However, our DSL is not sufficient for experiments that require more complex parametrization using additional packages due to lack of adequate generators.

Using Language Server Protocol, we were able to configure JupyterLab to provide editor functionalities for our DSL. Received feedback from test persons indicates that in its current state DSL does not provide optimal user experience.

We also encountered a bug concerning language server that affects user expirience usin DSL in Jupyter. Language server does not provide correct IDE support for multiple model configurations.

6.2 Future Work

The future works must aim at improving both the feasibility and practicability of our DSL.

Feasibility To enhance feasibility we must implement generators for other configuration files of the MITgcm. The implementation of generators can be continued by following strategy pattern. The corresponding parameters must be also introduced in Declaration Model. Hereby, we must keep in mind that our DSL must be universal and it must be possible to specify and add support for other ocean models. We must provide an in interface that allows to add both Declaration Model and associated generator. We must consider to introduce new metamodel propertiesl to be able to specify the constraints of the value.

Practicability To be better suited for typical configuration usage scenarios, discussed in Section 3.1, our DSL must have an import functionality. The imports must be transitive, which means, that a configuration A imported by configuration B can be used by configuration C if it imports B.

Appendix A

Questionnaire About The User Experience With Configuration DSL In JupyterLab

A.1 Questionnaire Answers 1

```
# Evaluation of Practicability

## Prerequisites

Before setting up the prototype, it is necessary to install the
following set of software. These prerequisites are based on a Ubuntu
Linux installation. Thus, tasks might need to be installed differently
on other platforms.


- As the docker image is currently larger than 3 GB, you need at least
  10 GB on your harddrive to successfully run this tutorial.

- Java (OpenJDK 11) which will be automatically installed with Maven

  - ‘sudo apt install default-jdk‘

- Install Maven (3.6.3)

  - ‘sudo apt install maven‘

- Install git

  - ‘sudo apt install git‘

- Install Docker
```

```
  - ‘sudo apt install docker-compose‘ (1.25.0) This will automatically
     install docker (19.03.8 on Ubuntu 20.04)

  - Ensure that your user is member of the ’docker’ group. This can be
    done by modifying the ‘/etc/group‘ file, adding your user name to
    the end of the line of the docker group.

## Setup

You need at least 10 GB of free space on your Linux system for the docker
image and the build.

To setup Juypter do the following steps:
git clone https://git.se.informatik.uni-kiel.de/thesis/serafim-simonov-bsc.git
cd serafim-simonov-bsc
./build.sh

## Questionnaire

Material:

- MITgcm tutorial (for list of parameters)

- Basic DSL syntax

Demographics:

- Scientific background: Software Engineering, Research Software Engineer

- Expirence with ocean models: Limited

Task 1: Create a new configuration file in Jupyter
Q1: How difficult was it to create and a new configuration file?

- File creation was easy based on the knowledge of the syntax
Q2: Do you have any suggestions on how the experience could be improved?

- There is no context help available.

- Thus, I had to search for the correct name of the model "mitgcm" and
```

```
  I had to look up parameter in the MITgcm tutorial. This is
  sub-optimal.

Task 2: Edit a configuration file in Jupyter
Additional Material: Example file deepc.gdsl
Q1: How difficult was it to edit a source file?

- Easy
Q2: Do you have any suggestions on how the experience could be improved?

- Tooltips to show the valid range of values (in case there are one)

Task 3: Generate configuration files for Baratropic Ocean Gyre Example
Note: Our DSL, unlike Fortran, is case sensitive. Correct spelling for parameters
    in DSL
      can be looked up in MITGcm documentation in Section 3.8.
Q1: How difficult was it to generate configuration files for Baratropic Ocean Gyre
    ?

- Easy, as this is done automatically by Jupyter when I switch the cell
Q2: Do you have any suggestions on how the experience could be improved?

- When I used my self created configuration, it provided an error message, but
  maybe it can be made more detailed or tell which parameter are necessary (if the
  list is limited). Otherwise maybe provide a template?
```

A.2 Questionnaire Answers 2

```
Task 1:
# Evaluation of Practicability

## Prerequisites

Before setting up the prototype, it is necessary to install the
following set of software. These prerequisites are based on a Ubuntu
Linux installation. Thus, tasks might need to be installed differently
on other platforms.


- As the docker image is currently larger than 3 GB, you need at least
```

```
  10 GB on your harddrive to successfully run this tutorial.

- Java (OpenJDK 11) which will be automatically installed with Maven

  - ‘sudo apt install default-jdk‘

- Install Maven (3.6.3)

  - ‘sudo apt install maven‘

- Install git

  - ‘sudo apt install git‘

- Install Docker

  - ‘sudo apt install docker-compose‘ (1.25.0) This will automatically
     install docker (19.03.8 on Ubuntu 20.04)

  - Ensure that your user is member of the ’docker’ group. This can be
    done by modifying the ‘/etc/group‘ file, adding your user name to
    the end of the line of the docker group.

## Setup

You need at least 10 GB of free space on your Linux system for the docker
image and the build.

To setup Juypter do the following steps:
git clone https://git.se.informatik.uni-kiel.de/thesis/serafim-simonov-bsc.git
cd serafim-simonov-bsc
./build.sh

## Questionnaire

Material:

- MITgcm tutorial (for list of parameters)

- Basic DSL syntax
```

```
Demographics:

- Scientific background:

- Expirence with ocean models:

Task 1:
-Create a new configuration file
Material: MITGcm tutorial
Q1: How difficult was it to create and a new configuration file?
A1: Difficult.
  - The port range in the Jupyter configuration file (JupyterLsp/docker-compose.
      yml) is the wrong way round.
     So that the actual port is 18888.
  - I'm missing the materials.
Q2: Do you have any suggestions on how the experience could be improved?
A2: - Describe the basic DSL Syntax in the readme
    - Add a MITGcm tutorial link to the readme and mention it in the prerequisites
        . eg https://mitgcm.readthedocs.io/
    - Add a tutorial for basic tasks

Task 2:
-Edit a source file
Material: MITGcm tutorial
Q1: How difficult was it to edit a source file?
A1: Difficult. Is a configuration file meant by a source file? If so it was easy
    to access and edit.
Q2: Do you have any suggestions on how the experience could be improved?
A2: Add a tutorial for basic tasks or link it. eg. https://jupyterlab.readthedocs.
    io/en/stable/user/files.html

Task 3:
-Generate configuration files for Baratropic Ocean Gyre Example
Material: MITGcm tutorial
Note: Our DSL, unlike Fortran, is case sensitive. Correct spelling for parameters
    in DSL
      can be looked up in MITGcm documentation in Section 3.8.
Q1: How difficult was it to generate configuration files for Baratropic Ocean Gyre
    ?
A1: Difficult. Following Input resulted in fail and finish.
----
```

```
MyConfig:mitgcm {
    myygroup {
      sNx=90
      sNy=20
      OLx=2
      OLy=2
      nSx=1
      nSy=2
      nPx=1
      nPy=1
      Nr =15
   }

   data{
      startTime = 0.
      endTime   = 3110400000.
   }
}

----
>Problems. Your specified not a single parameter for data.
Generation failed
>Generation finished
----

Q2: Do you have any suggestions on how the experience could be improved?
A2: - Add example for required data parameters. Or return necessary parameters.
    - Add if clause to 'Generation finished' like
   boolean success = false;
   try {
       ...
       success = true;
   } catch (...) {
       ...
   }
   if (success) {
       return "Generation finished.";
   } else {
       return "There were issues. Generation finished with errors.";
   }
```

```
 or insert it in the try block

try {
    ...
    return "Generation finished.";
} catch (...) {
    ...
}

 - minor spelling mistakes
   - Generator.java:65 ... Your specified ...
   - Generator.java:88 ... Generationg failed ...
```

Bibliography

[Bruegge and Dutoit 2009] B. Bruegge and A. H. Dutoit. *Object-oriented software engineering using uml, patterns, and java*. 3rd. USA: Prentice Hall Press, 2009. (Cited on page 6)

[*Ecore Modelling Framework*]. *Ecore modelling framework*. eclipse.org/modeling/emf/. 2020. (Cited on page 22)

[Jung 2016] R. Jung. *Generator-composition for aspect-oriented domain-specific languages*. Kiel Computer Science Series 2016/4. Dissertation, Faculty of Engineering, Kiel University. Department of Computer Science, Kiel University, 2016. (Cited on pages 4, 30)

[Jung et al. 2013a] R. Jung, R. Heinrich, and E. Schmieders. Model-driven instrumentation with kieker and palladio to forecast dynamic applications. In: *Proceedings Symposium on Software Performance: Joint Kieker/Palladio Days 2013 (KPDAYS 2013)*. Volume 1083. CEUR Workshop Proceedings. CEUR, Nov. 2013, pages 99–108. URL: http://eprints.uni-kiel.de/22655/. (Cited on page 23)

[Jung et al. 2013b] R. Jung, C. Schneider, and W. Hasselbring. Type systems for domain-specific languages. In: *Software Engineering 2013 Workshopband*. Edited by S. Wagner and H. Lichter. Volume 215. Lecture Notes in Informatics. Bonn: Gesellschaft für Informatik e.V., Feb. 2013, pages 139–154. URL: http://eprints.uni-kiel.de/20641/. (Cited on page 23)

[*JupyterLab Documentation*]. *Jupyterlab documentation*. URL: https://jupyterlab.readthedocs.io/en/stable/. (Cited on page 5)

[*Language Server Protocol*]. *Language server protocol*. URL: https://microsoft.github.io/language-server-protocol/. (Cited on page 5)

[Mernik et al. 2005] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computer Survey* 37.4 (Dec. 2005), pages 316–344. (Cited on pages 4, 7)

[*MITgcm*]. *Mitgcm*. https://mitgcm.org. 2020. (Cited on pages 9, 33)

[*MITgcm's user manual*]. *Mitgcm's user manual*. https://mitgcm.readthedocs.io/en/latest/welcome-to-mitgcm-s-user-manual. 2020. (Cited on pages 11, 22, 34)

[Object Management Group (OMG) 2006] Object Management Group (OMG). *Meta Object Facility (MOF) Specification 2.0 Core*. formal/2006-01-01. Apr. 2006. (Cited on page 4)

[UML15]. *OMG unified modeling language*. Standard. Version 2.5. Object Management Group, Mar. 2015. URL: http://www.omg.org/spec/UML/2.5. (Cited on page 4)

[*The International System of Units*]. *The international system of units*. Bureau International des Poids et Mesures. URL: https://www.bipm.org/utils/common/pdf/si_brochure_8_en.pdf/. (Cited on pages 23, 25)

Bibliography

[*XText*]. *Xtext*. https://www.eclipse.org/Xtext/. 2009. (Cited on page 3)

[*XText Documentation*]. *Xtext documentation*. Bureau International des Poids et Mesures, Sept. 2014. URL: https://www.eclipse.org/Xtext/documentation/2.7.0/Xtext%20Documentation.pdf. (Cited on pages 3, 19)