



TECHNISCHE UNIVERSITEIT EINDHOVEN
ASML

**Design and Validation of a
Model-Driven Engineering
Environment for the Specification and
Transformation of T-ReCS models**

Thomas Delissen

Author:
Thomas Delissen, BSc
0528738
Computer Science and Engineering

Supervisors:
Prof. dr. M.G.J. van den Brand
m.g.j.v.d.brand@tue.nl
Dr. Ir. Istvan Nagy
istvan.nagy@asml.com
Ir. Sven Weber
sven.weber@asml.com

February 12, 2011

Abstract

Within ASML, a domain specific language is being evaluated for supervisory machine control. This language is called T-ReCS (Task Resource Control System). This thesis explores the use of model driven engineering techniques to support the evolution and maintenance of T-ReCS. This is done by defining an abstract and concrete syntax for the language, as well as a prototype model driven engineering environment based on the Eclipse Modeling Framework and the former openArchitectureWare language workbench. This prototype is validated by performing a number of case studies, from both a language users and a language developers perspective.

Contents

1	Introduction	1
1.1	Problem domain	1
1.2	Project goal	2
1.3	Approach	3
1.4	Outline	4
2	Language engineering methodology	5
2.1	Definitions	5
2.2	ASML prerequisites	8
2.3	Eclipse modeling framework	9
2.4	openArchitectureWare	11
2.5	Language design approach	14
2.6	Summary	18
3	Abstract Syntax design	19
3.1	DAS for RaPTR	19
3.2	FAS for RaPTR	22
3.3	Example model: Coffee vending machine	24
3.4	Summary	26
4	Concrete Syntax design	27
4.1	Syntax requirements	27
4.2	Selecting a concrete syntax style	30
4.3	Concrete syntax implementation in oAW	36
4.4	Example model in concrete syntax	39
4.5	Summary	40
5	MDEE design	41
5.1	MDEE flow	42
5.2	Editor aspects in Xtext	43
5.3	Model to model transformations	46
5.4	Model to text transformation	51

5.5	Summary	53
6	MDEE Validation	54
6.1	MDEE extensions	54
6.2	Language user perspective: Use cases	58
6.3	Language developers perspective: Change cases	62
6.4	Summary	69
7	Conclusions and recommendations	71
7.1	Conclusions	71
7.2	Recommendations	74
7.3	Summary	77
	Bibliography	79

Chapter 1

Introduction

This report describes the results of the master project called *Design and Validation of a Model-Driven Engineering Environment for the Specification and Transformation of T-ReCS models*. The master project has been carried out at ASML Netherlands B.V., located in Veldhoven. The main topic of this master thesis is the investigation of a suitable model-driven engineering environment for the T-ReCS language[1], including a transformation to a formal verification language.

1.1 Problem domain

ASML is the worlds leading manufacturer of lithographic wafer scanners. The primary manufacturing process of a wafer scanner is the exposure of an IC (integrated circuit) pattern onto a wafer.

The wafer scanners, such as those from the TWINSCAN series [2], are complex manufacturing machines. They contain a large number of mechatronic components that execute different tasks in parallel as well as sequential. Figure 1.1 shows an architecture of complex manufacturing machines which is also applicable to the wafer scanners. At the lowest level, actuators and sensors (also known as the transducers) control the actual hardware of the system. Messages from the sensors flow back to the higher level components of the system. The resource control level is responsible for the execution of tasks on specific components, which can be considered as mechatronic subsystems. The research in this report focusses on the scheduling of tasks on resources (over time).

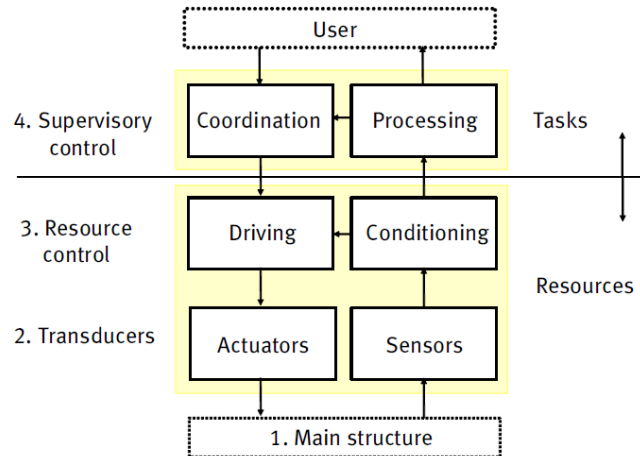


Figure 1.1: Manufacturing machines structure [3]

The scheduling of tasks to be executed is done through controllers, which are software components at the supervisory control level of the systems control architecture. The main role of a controller component is to map the requests it receives from higher level components (input side), such as the user interface, to lower level controllers or resource drivers (output side).

How the controller component responds to its inputs is specified in its software implementation. At ASML, several controller domains within the TWINSCAN architecture could use the T-ReCS paradigm to perform the mapping from client side to server side. T-ReCS stands for *Task Resource Control System* [1], and a first prototype of T-ReCS was implemented at ASML a couple of years ago. As an example of a controller domain, one could think about the waferstage of a machine, which moves the wafers into position and aligns them in an appropriate way. The wafer stage receives requests from higher level controller domains (for example: ‘process 10 wafers’) and translates these into lower level tasks (for example: ‘move wafer from position A to position B’).

The T-ReCS language has been evolving over the past few years. Many concepts have been added to the original set of features and are being added frequently (every 3 to 4 months), to evaluate its useability.

1.2 Project goal

The research question for this project is the following:

Using model-driven engineering techniques, is it possible to create and maintain a complete model driven engineering environment for the development and use of an industrial

size language, including transformation support?

Defining an abstract syntax for T-ReCS provides a basis for supporting the maintainability and the evolution of the language. By using model driven engineering techniques such as language workbenches (see Section 2.1.4), it will be possible to create an appropriate concrete syntax for the abstract syntax of the language. Furthermore, transformations to other representations using this abstract syntax definition as a starting point will be defined. Language workbenches are a relatively new technology and it is not sure whether it is feasible to use them for a very large domain specific language such as T-ReCS. Therefore, the research in this project will follow a proof of concept approach.

1.3 Approach

A prototype implementation will be made for the abstract syntax of T-ReCS in a language workbench. A number of textual concrete syntax alternatives will be designed, of which one will be implemented in the language workbench based on preferences by the domain experts at ASML. In a related research project the T-ReCS language is formalized and a mapping to a process algebraic language is defined to enable verification. This transformation will also be incorporated into this prototype.

The prototype model driven engineering environment will be used to implement a T-ReCS model and running it through the transformation chain. Based on this validation, an answer to the research question will be formulated.

The approach for this assignment can be divided in a number of phases.

1. Research related work on language engineering and define a language engineering approach for this project.
2. Analyze the T-ReCS language and prototype implementation and define an abstract syntax for it.
3. Define several concrete syntax alternatives for T-ReCS and validate the useability of these concrete syntaxes. Based on this evaluation, select one for further development.
4. Create a model-driven engineering environment for T-ReCS, based on the abstract syntax, the concrete syntax and the static semantics of the language.
5. Create an automatic transformation to the process algebraic verification language, using the abstract syntax model as a basis.
6. Validate the model driven engineering environment and provide recommendations for future development.

1.4 Outline

The remainder of this thesis will be organized as follows. Chapter 2 will discuss the related work on language engineering technologies, and will elaborate on the methodology of this project. It will also contain a description of the constraints put on the project by the chosen technology. Chapter 3 will present the design of the abstract syntax, based on an extensive domain analysis. Chapter 4 contains the rationale behind the design of our concrete syntax alternatives and elaborates on the design of one of those alternatives, as preferred by ASML. Chapter 5 contains the specifics on the model driven engineering environment design. This includes scoping rules, static semantic validations and a number of transformations, such as the transformation to an algebraic verification language. Chapter 6 describes the validation done on the model driven engineering environment. Finally, Chapter 7 presents the conclusion on this thesis and the recommendations for future work.

Chapter 2

Language engineering methodology

In this chapter the different aspects of language engineering are explored, such as domain specific languages, model driven engineering and language workbenches. Based on these technologies, the methodology of the language engineering approach for this project is presented. This methodology refers to the methods and techniques used to create the abstract syntax, the concrete syntax and the model driven engineering environment for the language.

2.1 Definitions

2.1.1 Syntax, semantics and pragmatics

As described by Slonneger and Kurtz [4], a language definition usually consists of three parts: its syntax, its semantics and its pragmatics. Syntax describes *how* to define models in the language (the form of models) while semantics describes what the models *mean* (the meaning of models). Pragmatics describe the *way* the language is used (or ought to be used).

The syntax can be divided in abstract syntax and concrete syntax. The abstract syntax describes the high-level structure of language elements and their relation. The concrete syntax defines the actual (textual or graphical) representation of the models. The abstract syntax is also referred to as the meta-model of the language.

2.1.2 Domain specific languages

In this thesis the *domain specific language* (DSL) is an important concept. For this report we use the definition of Mernik et al [5], which describes a DSL as a ‘programming language tailored to a particular application domain’. This is the opposite of a *generic programming language* (GPL), which can be used to solve any kind of problem.

A domain specific language trades generality for expressiveness in a particular domain. The main advantage of using a DSL over using a GPL for a particular domain is an increase in productivity: it takes less effort for the developers to express domain specific constructs, which increases productivity. Furthermore, domain specific languages are easier to understand for domain experts and are easy to learn for *new* developers [6]. It is less likely to make errors in a DSL, which also increases productivity.

However, a disadvantage of DSLs is that they need to be defined in terms of syntax, semantics and pragmatics. General programming languages usually come with editor and debugger support, extended libraries, abundant documentation, a fully fledged user community and can be used immediately. Furthermore, DSLs always need to be learned, while developers might already know general purpose programming languages. The benefit of using a DSL should outweigh the effort required to define, implement and learn to use it.

2.1.3 Model driven engineering

Model driven engineering (MDE) is a software engineering paradigm in which models play a central role. A *model* can be anything from a graphical to a textual representation of a system, as long as the model is written in a well-defined language [7]. A *meta-model* is a model that defines this language. Since the meta-model is a model itself, it also has to be written in a well-defined language, which can be referred to as the *meta-meta-model*.

A second important aspect of MDE are model transformations. *Model transformations* are automatic transformations from a model defined in one meta-model to a model defined in another meta-model. To be able to perform a model transformation, a *mapping* should exist from the source meta-model to the target meta-model. Code generation can also be seen as a model transformation, in which the target meta-model is the grammar of a concrete syntax. Figure 2.1 gives an overview of these concepts and their relations.

In this definition we have chosen a situation in which both meta-models are described in the same meta-meta-model, which does not always need to be the case. If meta-models conform to different meta-meta-models, a mapping should exist between those meta-meta-models before the mapping between the meta-models can be defined. More information on MDE (and its relation to OMGs MDA) can be found in Kent [8].

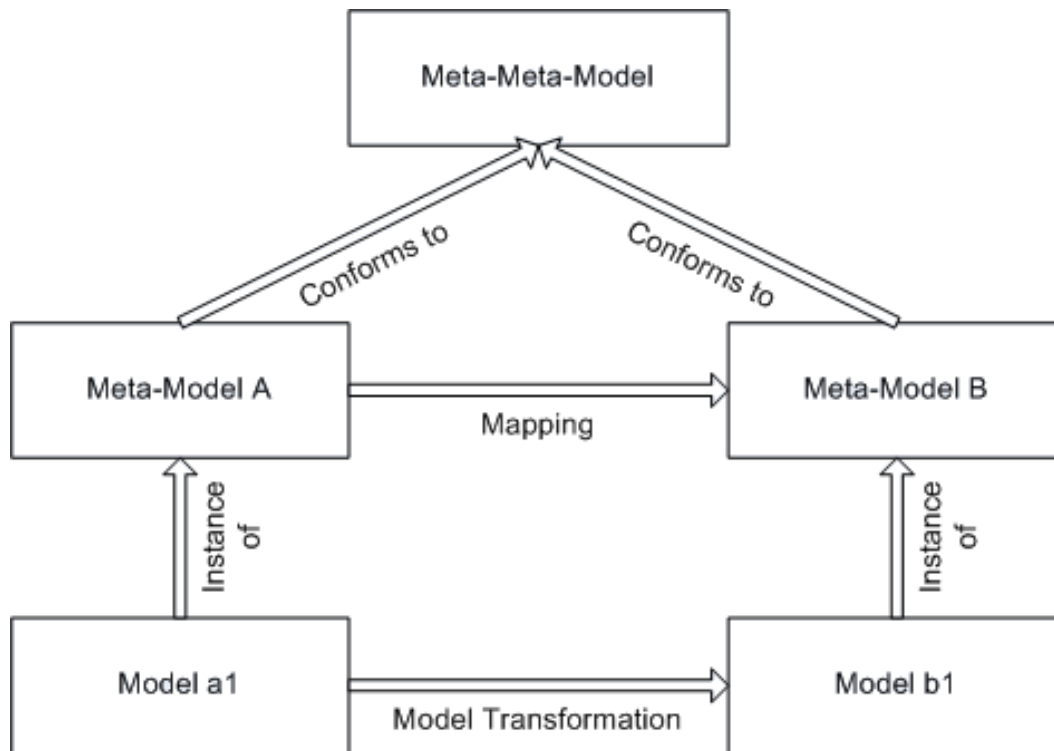


Figure 2.1: Overview of MDE concepts

2.1.4 Language workbenches

Tool support is an important aspect of the development of domain specific programming languages. In the past, a simple text editor would have been sufficient as tool support. Nowadays, programmers are used to integrated programming environments, with context highlighting, type suggestions and debugging facilities.

Thus, for the language to be used, we also need a state of the art integrated development environment (IDE). Also, when using MDE techniques, an environment is needed in which it is easy to modify the language and work with transformations to other languages. Such an environment is called a *model driven engineering environment* (MDEE). As was stated in the introduction of this chapter, it is very hard to create such a MDEE for a newly designed language, although different IDEs have many features in common. This is where language workbenches come in.

A *language workbench* is an integrated development environment which supports language developers in creating a (domain specific) language and its transformations in accordance with the MDE paradigm. Martin Fowler [9] defines a language workbench as a tool which has the following characteristics:

- Users can freely define new languages which are fully integrated with each other
- The primary source of information is a persistent abstract representation
- Language designers define a DSL in three main parts: schema (grammar), editor(s) and generator(s)
- A language workbench can persist incomplete or contradictory information in its abstract representation

The main feature of language workbenches is that they automatically generate important artifacts for language engineering, such as an editor, based on limited input. The way in which a language workbench aids the language engineer and how extensive the support is varies for each language workbench.

2.2 ASML prerequisites

For this project, parts of the Eclipse Modeling Framework (EMF) [10] and the former openArchitectureWare (oAW) [11] tooling are used. EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model[10]. It is an Eclipse based project consisting of several plugins which can be used for Model Driven Engineering. This workbench is used because it is a prerequisite of ASML. However, good reasons exist for choosing this language workbench.

First of all, it is widely adopted, which makes sure there will be enough support for using the language workbench.

Secondly, it is an open source project, which means it can be used free of charge.

Third, the source code is available for customization and editable to accommodate for new features. This reduces the threat of a vendor lock-in by using this workbench. Finally, multiple other Eclipse plugins are based on the EMF representation, which provides additional tool support. Future features based on this EMF representation can be integrated seamlessly.

2.3 Eclipse modeling framework

Tools within EMF use a common meta-model representation, called Ecore. In this Ecore representation, the meta-model for a language can be defined.

The basic meta-meta-model for Ecore is depicted in Figure 2.2.

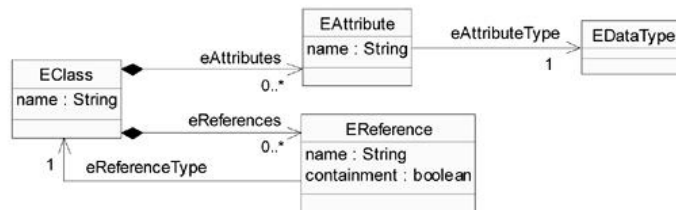


Figure 2.2: Ecore meta-meta-model

Within this meta-meta-model, four main classes are defined:

1. **EClass**: These are the main classes, as they are modeled in Ecore. Each concept in the language usually is a class, with a name (identifier), and a number of attributes and references.
2. **EReference**: These are references an EClass has to other EClasses. The references can be associations or composition relations.
3. **EAttribute**: EClasses can have attributes, which are of a certain EDatatype.
4. **EDatatype**: These are the simple datatypes within Ecore, such as String, Boolean and Integer.

With these four classes, it is possible to define the entire abstract syntax model (or meta-model) for a language, although usually the subclass association is also used. With the subclass association, it is possible to define superclasses. The subclass contains all associations and attributes of the superclass and may contain more information.

Two representations will be used to define the meta-model of a language. The first is by giving its *signature*, which is a mathematical representation of the production rules which are possible in the meta-model.

The signature of a language consists of production rules, which have a function on the left side of the arrow and a label on the right side of the arrow. Functions have a number of attributes, which are written between the brackets. Uppercase names are rule labels and quoted lowercase names are terminals. The cardinality of rules is specified by tokens next to the rule labels. They mean the following:

- ‘ ’ : No cardinality, this means that exactly one element should be entered at that position.

- ‘?’ : Optional cardinality, there should be either zero or one of the specified element.
- ‘*’ : Zero or more elements should be specified.
- ‘+’ : One or more elements should be specified.
- RULE ‘|’ RULE** : Either the left hand side or the right hand side of the bar is taken.
- ‘a-z’ : Range operator, every element in the range from ‘a’ until ‘z’ is allowed.
- ‘^ RULE’ : Pointer operator, this is a reference to the rule instead of a complete definition of the rule.

Besides this mathematical notation, a graphical notation will be used, which is an adapted UML class diagram notation. In fact, this notation has already been used to describe the Ecore meta-model in Figure 2.2. A legend for the Ecore graphical notation is provided in Figure 2.3.

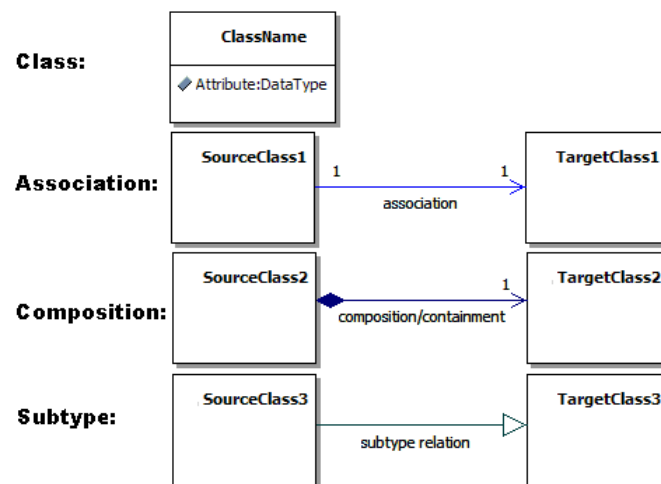


Figure 2.3: Ecore abstract syntax model legend

A class is depicted as a box, with the name on top. Attributes are listed inside the box, with the name of the attribute first, and the datatype of the attribute after the colon.

The association relation specifies that there is a reference from the source class to the target class. A composition relation (indicated by the black diamond at the source end of the relation) specifies that the class at the target end of the relation is part of the class at the source of the relation. The ends of relations can have a cardinality associated with them. These represent the number of classes a relation can be associated with.

The subclass association defines the class at the source end of the association as the subclass of the class at the target end of the association. This means that the subclass has all the relations and attributes of the superclass, possibly extended with some more relations. An example of these representations is provided in Example 2.1.

Example 2.1

Traffic junction signature and graphical meta-model.

As an example, we will model a simplified *traffic junction*. The traffic junction consists of a number of *streets* and a number of *lightposts*. The lightposts contain an arbitrary number of traffic *lights*, which can be red, yellow or green. Furthermore, a lightpost is located at a street, and streets are either a one-way street or not.

The signature for this meta-model:

```

junctn(ID,STREET+,
        LIGHTPOST+)-> JUNCTION

street(ID,ISONEWAY)  -> STREET
lightpost(ID,LIGHT*,
           ^STREET)  -> LIGHTPOST

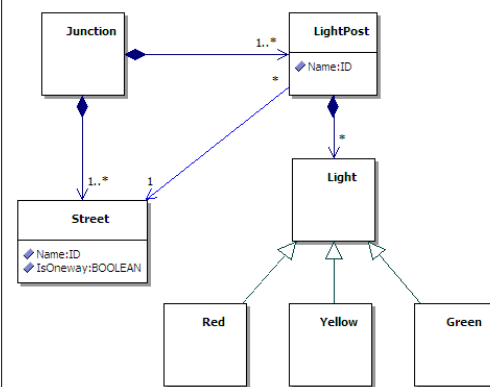
red('red')           -> LIGHT
yellow('yellow')     -> LIGHT
green('green')       -> LIGHT

isOneWay('true'|'false')-> ISONEWAY
([a-z]|[A-Z])
([a-z]|[A-Z]|'_'|[0-9])*-> ID

```

□

A graphical representation of this meta-model:



The signature and the Ecore graphical representation will be used to specify the abstract syntax in Chapter 3. As motivated in Section 2.2, the abstract syntax will be implemented in EMF.

2.4 openArchitectureWare

The main tools within EMF that have been used for this project are part of the former openArchitectureWare (oAW). These tools are:

Xtext: Tool used for defining textual syntaxes and generating an Ecore model from this textual syntax.

Xtend: Model to model transformation language, used to transform models of one Ecore model into models from another Ecore model.

Xpand: Model to text transformation language, used to transform models defined in an Ecore model to text (code).

Check: Validation language used to define extra constraints on the syntax.

In the remainder of this document, this set of tools will be referred to as openArchitectureWare (oAW).

Xtext is the most important tool for this project. It is designed as a framework for developing textual DSLs [12]. The main advantage of Xtext is that the concrete syntax and the abstract syntax are defined together in the same grammar file. This makes it possible to quickly create a concrete syntax for your language, which can be used by all the other tools in EMF.

Xtend and Xpand are languages used for transformation and generation from the Ecore model. Xtend is a functional language (with side effects) which is primarily used for model to model transformations. Xpand is a template based transformation language. When a certain transformation is not very straightforward, Xpand templates can use supporting functions, defined in Xtend. The syntax of Xpand and Xtend, including an example transformation are explained in Chapter 5.

Check is a language which can be used in combination with Xtext, to put additional constraints on the concrete syntax. In this project, it is not used. Instead, the validations are done by Java functions, which allow for more advanced constructs and provide better performance.

2.4.1 Xtext

Xtext grammars consist of a set of rules which generate Ecore classes according to their specification. In Example 2.2, ‘Model’, ‘Street’ and ‘Lightpost’ are examples of rules. The first rule in an Xtext grammar is always the root element. If in the abstract syntax, multiple root elements exist, this is solved by introducing a governing root class called ‘Model’.

Rules consist of three kinds of elements: keywords, attribute assignments and delegations to other rules. Keywords are written between quotes, and are automatically highlighted in blue. Attributes are specified by a name and an assignment operation. Delegations to other rules are rule names with an optional cardinality. The types of cardinalities are the same as those which are defined for the signature in the previous section: ‘?’, ‘+’, ‘*’. In the Xtext grammar, three kind of assignments can be identified.

- ‘=’ : Simple assignment of a single value to a attribute, for example in line 5.
- ‘+=’ : List assignment. The attribute on the left side of the assignment is a list, and multiple values can be added to it.
- ‘?=’ : Boolean assignment. Typically used in combination with an optional keyword: If the keyword is entered, the boolean attribute is set to true.

References between rules can be specified as done in line 14 of Example 2.2. Here, a 'Lightpost' refers to 'Street' by a name identifier of the form 'name=ID'. Note that ID is an Xtext specific identifier.

Example 2.2

Traffic junction Xtext grammar.

This example contains a grammar for the traffic junction meta-model introduced in Example 2.1. The root rule of this example is 'Model' (line 4). The specification of a model begins with the keyword 'Junction', followed by an identifier for the Model class. A common pattern to represent comma separated lists can be seen in lines 6 and 7. A lightpost can have one or more lights of a certain color. To force Xtext to generate classes for constant rules like 'Red', 'Yellow' and 'Green', the annotations {Red} {Green} and {Yellow} are added.

```

1 grammar an.Examplegrammar with org.eclipse.xtext.common.Terminals
2 generate examplegrammar "http://www.Examplegrammar.an"
3
4 Model:
5     'Junction' name=ID ':'
6     'Streets' streets+=Street (',' streets+=Street)* ';'
7     'LightPosts' lightposts+=Lightpost (',' lightposts+=Lightpost)* ';'
8 ;
9 Street:
10    (isOneWay?='one-way')? name=ID
11 ;
12 Lightpost:
13    'Post' name=ID ':' colors+=Color+
14    'location' street=[Street|ID]
15 ;
16 Color:
17    Red | Yellow | Green
18 ;
19 Red: {Red} 'Red' ;
20 Yellow: {Yellow} 'Yellow' ;
21 Green: {Green} 'Green' ;

```

□

Besides the definition of the grammar, which results in the syntax of the language, the editor can also be modified. Xtext generates a default editor for the defined language, with code-completion and syntax highlighting. Additional features can be added by overriding certain functions of the framework. In this way, validations can be added, with which developers can implement static semantic checking. Other editor functionality, such as a custom scoping rules or quickfix functionality can also be implemented in this way. In theory it is possible to modify the entire editing environment, but in practice this requires quite extensive Java skills and in depth knowledge of the Xtext environment.

2.5 Language design approach

The most important part of the language design is the specification of the language. This is split in syntax and semantics. However, before the syntax and semantics can be defined, the requirements for the language need to be gathered. A structured approach for language design is proposed by Mauw [13]:

1. Gather requirements/ Domain analysis
2. Design syntax and semantics
3. Implement compiler or interpreter
4. Create editor environment for language

This can be described as the classical approach, which does not take into account language workbenches. With language workbenches, you usually get a basic editor environment for free when the syntax is defined. However, in most cases, this still needs to be improved, so these steps are still considered relevant for this project.

2.5.1 Language design using oAW

For this project, oAW and EMF are used to support the language design process. The requirements gathering and the domain analysis is not influenced by the tooling. Tooling becomes relevant at the moment the concrete syntax and the abstract syntax are defined. In this project, Xtext is the first tool which influences the way of working.

As explained in Section 2.4, Xtext is designed as a tool to define a concrete syntax and *generate* an abstract syntax. However, in the classical language engineering approach, it is preferred to first define the abstract syntax and derive a concrete syntax for that. In Xtext, this is possible in theory, as one can import existing Ecore models and create a concrete syntax for that. Figure 2.4 depicts this way of working, in which a domain abstract syntax (DAS) is derived from the domain analysis, and a concrete syntax (CS) is designed for that.

A drawback of this approach is that the structure of the concrete syntax is severely influenced by the shape of the abstract syntax. The main restriction of importing existing Ecore models is that the concrete syntax should produce the exact classes of the abstract syntax, including the associations and attributes that belong to these classes. There are many situations in which this is not desirable. Language developers should have full freedom in defining the concrete syntax, as it is the part of the language the users will interact with most.

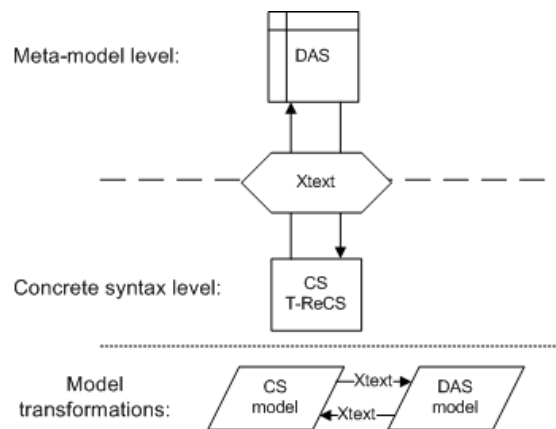


Figure 2.4: Envisaged model transformation flow using Xtext

Example 2.3

Traffic junctions alternative grammar representation

This example illustrates why it is not possible to create an *arbitrary* concrete syntax for the DAS. When looking at the abstract syntax of Example 1, the choice could be made to use an alternative concrete syntax representation of the Lightpost definition. Instead of giving a list of colors, the grammar could look as follows:

```
Lightpost:
  'Lightpost' name=ID
    'has' redNumber=Integer 'redlights'
    'and' yellowNumber=Integer 'yellowlights'
    'and' greenNumber=Integer 'greenlights'
;
```

This would allow a much more concise notation in the case the Lightpost have many lights. However, when importing the example Ecore model, this is not allowed, since the concrete syntax should follow the abstract syntax, which specifically requires a list of lights of a certain color.

□

The alternative is to define the DAS in Ecore, define the CS as a grammar in Xtext (which can express all the elements in the DAS) and *generate* an Ecore meta-model from the grammar. This generated Ecore meta-model is an abstract syntax which is specific for that CS, for which the term Concrete Abstract Syntax will be introduced (CAS). To connect the CS with the DAS, a model to model transformation will be made from the CAS to the DAS. This approach, which is depicted in Figure 2.5 is followed for this project.

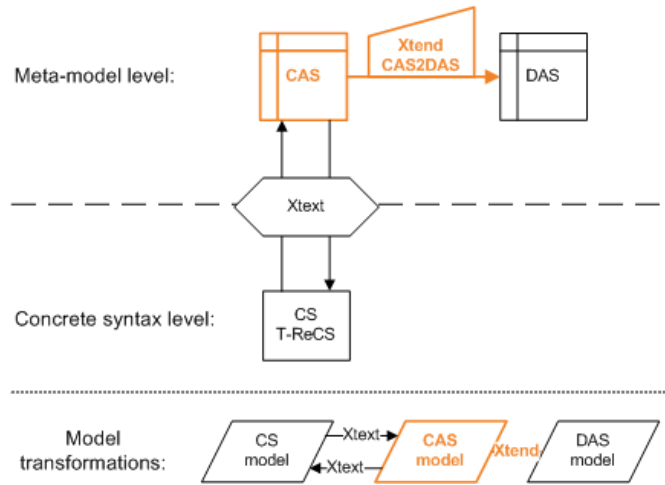


Figure 2.5: Model transformation flow with generated CAS (orange color depicts new elements)

2.5.2 Iterative semantics based approach

In a related KWR project, the semantics for T-ReCS are being defined. The definition of the semantics contains a mathematical signature, which can be used to create the abstract syntax model. As part of this KWR project, the language is also mapped to a formal verification language, which allows the verification of models written in T-ReCS. The implementation is designed in such a way that the transformation of models represented in the semantic definition to the verification language concrete syntax is relatively straightforward.

As a first approach, the semantic definition could be used as a starting point to define the language. The complete semantic definition of the language would contain a mathematical signature from which an abstract syntax could be derived. This will be called the *formal abstract syntax* (FAS). After this, a transformation from this FAS to the verification concrete syntax could be implemented.

The question rises whether the FAS could replace the DAS. With respect to information, the FAS and the DAS should be the same, otherwise the DAS has too much information (language constructs are present which have no semantics), or the FAS is not complete yet (for some language elements, no semantics have been defined yet). However, even if they contain the same information, they may be *modeled* in different ways. Reasons for this can be the perspective at which the language developers are looking at the language, or the purposes for which the DAS or FAS is used.

One of the issues with selecting the FAS as the main language definition is the fact the formal semantics for the language were still being developed in the course of this project. Furthermore, the structure of the FAS is different from the structure derived from the

domain analysis. The FAS is very useful for the transformation to the verification language, but it could be hard to transform the concrete syntax to this representation. The main reason for this is that the language user will not think of the language in a formal semantics oriented way, so the structure of the concrete syntax designed for a language user will be different.

Thus, the choice was made to maintain both meta-models. Using this approach, we will have two different abstract syntaxes. First of all the formal abstract syntax (FAS), based on the semantics, and secondly, the domain analysis oriented abstract syntax (DAS), based on the domain analysis. The FAS will be the basis for transforming to the verification language concrete syntax. Figure 2.6 gives an overview of the different parts of the model generation flow.

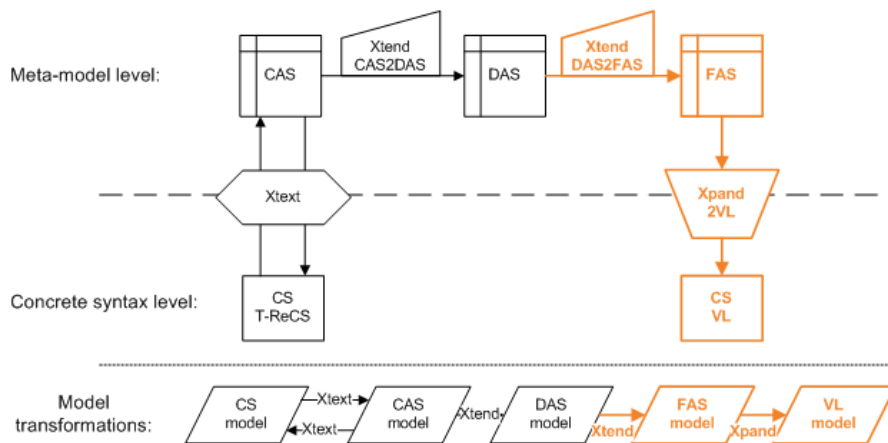


Figure 2.6: Decoupling the domain analysis and formal semantics based meta-models

The complete cycle is build in an incremental way. The FAS will contain the elements defined by the formal semantics, while the DAS will contain the elements which are derived from the domain analysis. In this project, the FAS will be a subset of the DAS, although in the future both versions should be made equal.

In the current situation, the DAS may be omitted and replaced by an Xtend transformation. However, the approach containing the DAS splits a large transformation from CAS to FAS into two transformations, which are smaller and easier to maintain. When the concrete syntax is modified (which is the most likely change case), only the concrete syntax and its transformation to the DAS need to be changed, and not the DAS, the FAS or the transformation from DAS to FAS. Furthermore, the DAS may be a useful alternative to the FAS for future transformations.

2.6 Summary

In this chapter the basic concepts and the approach related to the language engineering part of this project has been described. In the next chapter the definition of an abstract syntax model for the T-ReCS language is discussed, which is a result of the domain analysis.

Chapter 3

Abstract Syntax design

This chapter contains a description of (a subset of) the domain abstract syntax (DAS) of T-ReCS as it was derived from the domain analysis, and the formal abstract syntax (FAS) as it was derived from the semantics definition of the language. The CAS is not discussed, as this is generated from the concrete syntax which is introduced in Chapter 4.

For confidentiality reasons we will use a subset of T-ReCS for the main text of this thesis. This subset of the language will be called RaPTR (Reduced and Public T-ReCS), and will be used to explain all relevant concepts of the language.

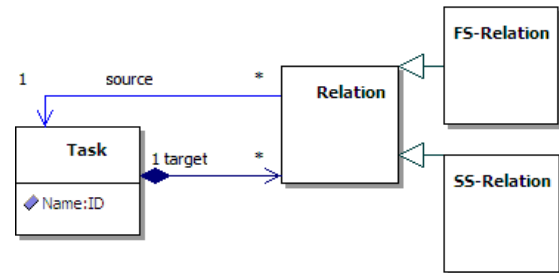
3.1 DAS for RaPTR

As a basis for designing the DAS, the existing system was an important source of information, as it contained all the information that should be available in the language. This information was scattered and sometimes unclear and incomplete, so the clarifications from the language and domain experts proved to be an important source of information. The main requirement for the DAS was that it should contain all information of the old language definition, so that it will be backwards compatible. Other requirements, such as scalability and usability are of more relevance to the concrete syntax, which will be discussed in Chapter 4.

The design of the DAS is done in an incremental process, which consists of a number of steps. These steps will be briefly discussed, to show the design process of RaPTR. For each step, the classes that are added are colored orange.

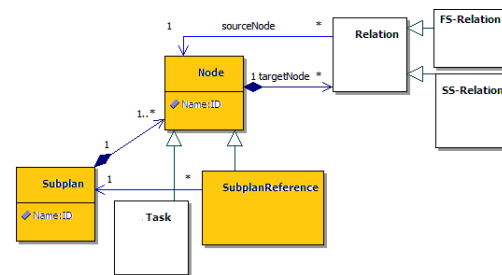
Step 1: Tasks and precedence relations

The main concern of RaPTR is to define a mapping from a client-side request to a sequence of tasks (see Section 1.1). When a request is made, a number of *tasks* are scheduled to be executed. A task has a number of relations. Such a relation can be a finish-start relation or a start-start relation with another task. A *finish-start relation* indicates that the task can start its execution once the source task is finished. A *start-start relation* indicates that the task can start its execution once the source task is started.



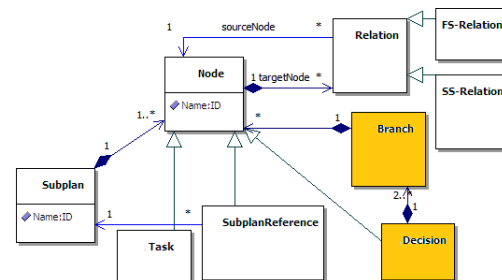
Step 2: Subplans and subplan references

A set of interrelated tasks is contained in a subplan. A *subplan* can be referenced from within another subplan, which introduces the concept of *subplan reference*. Just as tasks, a subplan reference can have FS-relations and SS-relations with tasks or other subplan references. To prevent redundant relations, we introduce the abstract concept of *node*, which can be either a task or a subplan reference.



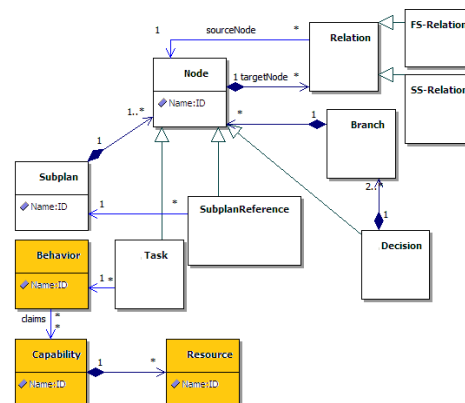
Step 3: Decisions and branches

Sometimes the execution of tasks is dependent on a choice. Therefore, a decision node is introduced. *Decisions* contain a number of branches, of which one will be processed once the decision is made. *Branches* are containers, which in turn contain a number of nodes.

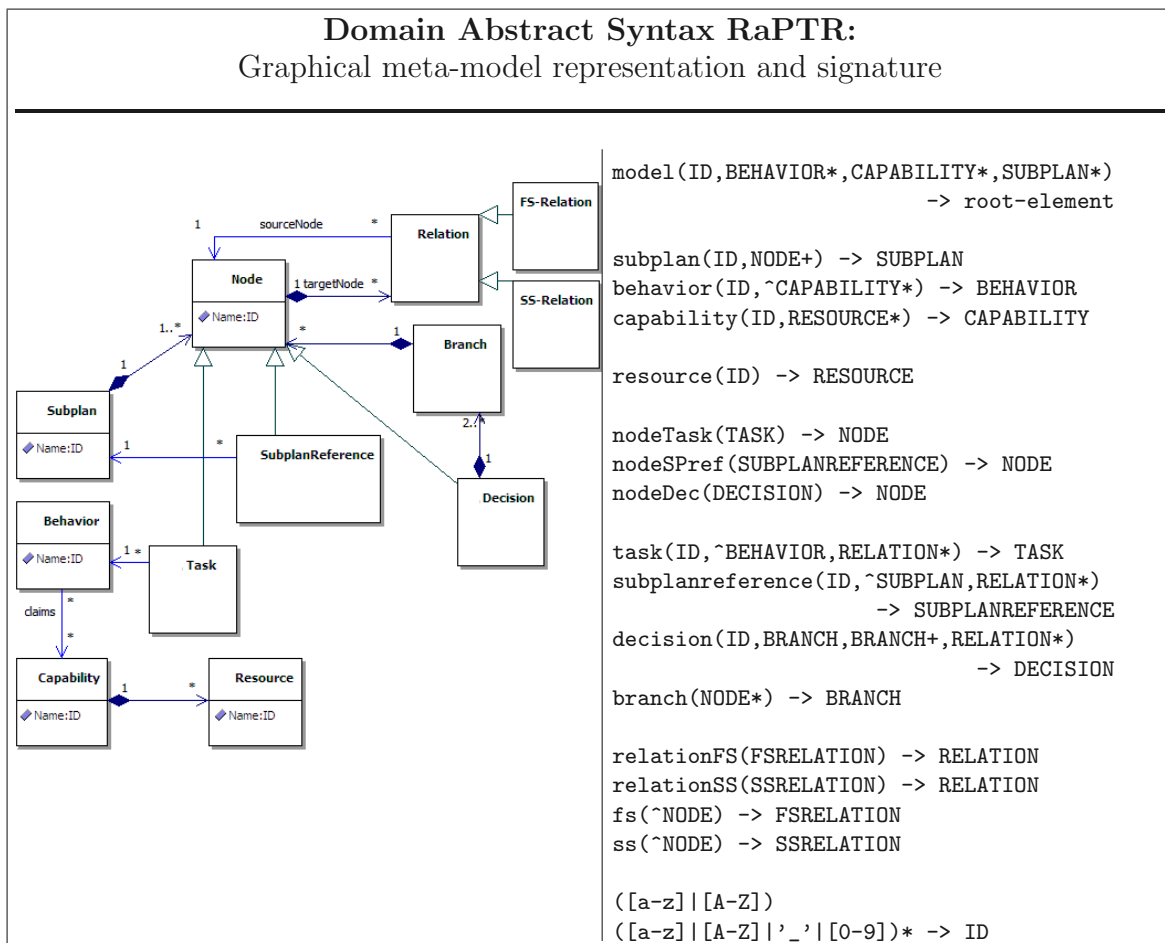


Step 4: Behaviors and capabilities

A final addition is the mapping of tasks to resources. Tasks can claim certain resources they need for execution. Sometimes multiple resources of the same type exist, and a task of a certain type can claim either of these resources. To model this, the *behavior* is introduced, which is a task type. A behavior can claim a *capability*, which is a resource type. Resources are modeled as a composition within capabilities, although they still need an identifier.



This results in the following DAS for RaPTR:



3.1.1 Additional constraints

Additional design constraints on the meta-model which are not immediately obvious from the abstract syntax model are described in Table 3.1. These constraints are static semantic constraints, which could be covered by either the concrete syntax editor (through validations) or by the transformation from CAS to DAS.

Nr	Class	Constraint
1	Relation	The relational ‘chain’ may not contain cycles, as this would result in deadlocks.
2	Branch	Nodes in a branch can only have a relation with nodes within the same branch.
3	Subplan	Nodes in a subplan can only have a relation with the top level nodes of that subplan, not with nodes in branches or in different subplans.
4	SubplanReference	A subplan reference cannot refer to a subplan which (transitively) refers back to the original subplan, as this would result in infinite sequences.
5	Node	A node cannot have two relations with the same sourcenode.
6	Behavior	Behaviors can only claim capabilities which have at least one resource.

Table 3.1: Additional design constraints for the DAS

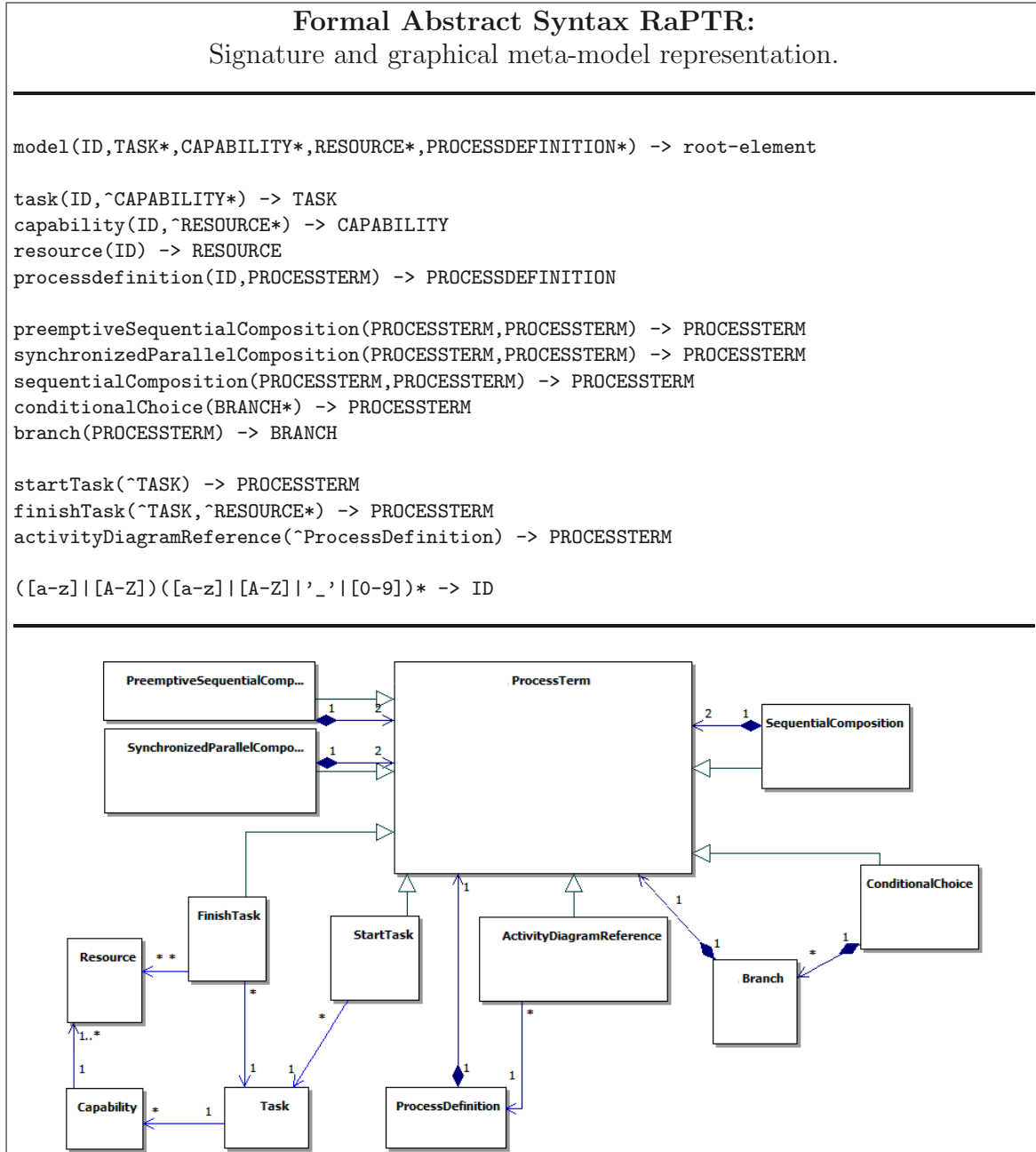
3.2 FAS for RaPTR

The FAS has been derived from the formal semantic definition in the related research project. All notions which occur in the DAS for RaPTR, *can* be expressed in the FAS.

The main notion in the FAS is the *process term*. The process term is an abstract class, so it can be a number of concrete classes, including classes which contain other process terms. In this way a tree-structure is built which contains tasks and relations between them. For example, a process term can be a start-task, which is a task which still needs to start. Two start-tasks can be composed with eachother by means of a *synchronized parallel composition*, which means the tasks can be executed in parallel. Another composition is the sequential composition, which means that two task can only execute in sequence (which is the FS-Relation in the DAS).

The complete reasoning behind the structure of this model will not be explained in this thesis, as it is part of the semantics definition in the related research project. For this project it is only important to demonstrate that a mapping is possible from the DAS to

FAS and that the FAS representation can be easily transformed into a concrete syntax representation of the verification language. A good illustration on the issues that can occur with such a transformation can be found in [14].



3.3 Example model: Coffee vending machine

For the remainder of this thesis we will use an example model, which describes a fictitious system for a simple coffee vending machine. In this section, the coffee vending machine will be informally introduced, and the model is described in terms of the Ecore implementations of the DAS and FAS of RaPTR.

The coffee vending machine can receive a choice from a client. At the time the choice is received, a plastic cup is provided. In parallel to this, the choice is processed, and a decision is made based on input from the choice. If coffee was selected, the machine makes coffee, otherwise it makes tea. After either tea or coffee has been made, sugar is added and the process is finished. This process is graphically depicted in the activity diagram as shown in Figure 3.1.

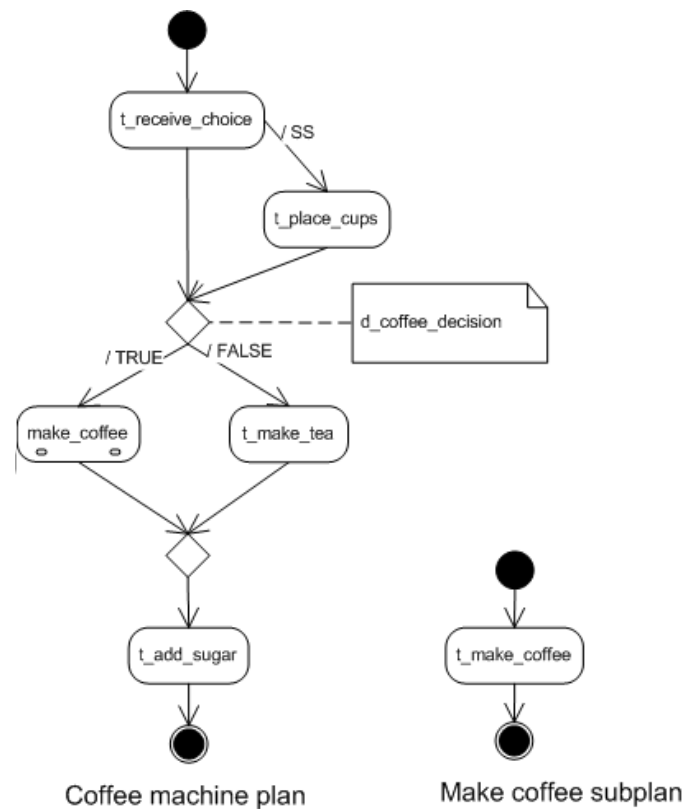


Figure 3.1: Activity diagram for simple coffee vending machine

When modeling this coffee vending machine in RaPTR, some extra information is added in order to show all the functionality that is available. For example, a number of capabilities and resources have been added. The Subplan to represent the ‘make coffee’ action

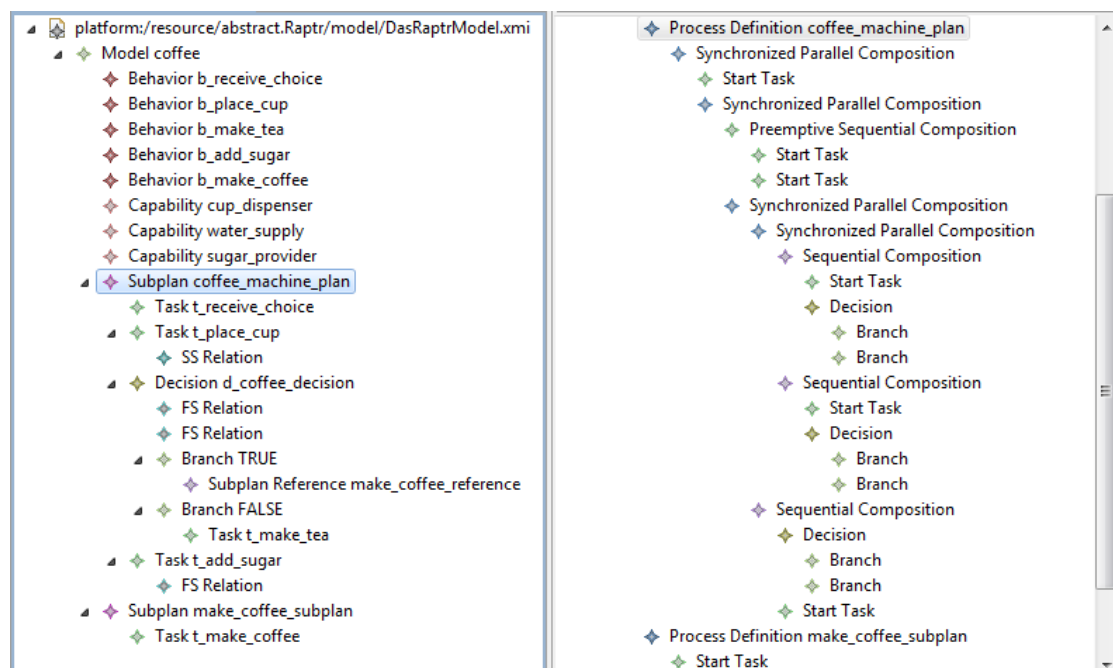
has been introduced as a subplanreference, although it could also have been modeled as a simple task within the original subplan.

Three capabilities have been added: *cup_dispenser*, *water_supply* and *sugar_provider*. There are two cup dispensers in the system, which means Cup_dispenser has two resources. Each task in the activity diagram is of a specific behavior, with the same name as the task. The behavior of t_place_cups claims cup_dispenser, the behaviors of t_make_tea and t_make_coffee claim water_supply and the behavior of t_add_sugar claims the sugar_provider.

With this information, it is possible to implement the example coffee model in the Ecore models of the DAS and the FAS. They can be viewed through the Sample Reflective Ecore Model Editor, which is the standard interface of EMF. Example 3.3 gives the view of the coffee model, which is implemented in both Ecore meta-models.

Example 3.1

Example model implemented in both the DAS (left) and the FAS (right)



□

3.4 Summary

In this chapter we have given the results from our domain analysis, which we have consolidated as an abstract syntax model of the RaPTR language. This model will be the basis for our concrete syntax design and an important input for the implementation of the model driven engineering environment for the language.

Chapter 4

Concrete Syntax design

This chapter will elaborate on the design of the concrete syntax for RaPTR. The concrete syntax is a very important aspect of the language, as it will be the main interface for language users to define models.

4.1 Syntax requirements

There are a number of general best practices with respect to concrete syntax design. Kleppe [7] defines the following:

- Aim for simplicity
- Be consistent
- Give the user flexibility
- Show related concepts in the same way
- Show unrelated concepts in different ways
- Use color sparsely
- Test your design on language users

During the design of the concrete syntax we will keep these guidelines in mind. Some will be very applicable in certain design decisions, while others will have a more general influence on the design. Besides these general guidelines, there are some additional requirements defined by ASML.

The main requirement of the concrete syntax is of course that it should adhere to the abstract syntax (DAS) defined in Chapter 3: it should be possible to express everything which is part of the RaPTR language in the concrete syntax. Besides this functional requirement, a number of non-functional requirements or constraints exist, which have

been gathered from discussions with the domain experts or where defined in the project plan. These requirements have an influence on the way the concrete syntax is designed.

4.1.1 The concrete syntax must be textual

According to Groenniger et al [15], there are a number of benefits to using a textual concrete syntax instead of a graphical concrete syntax, both for the user of the language as well as for the developer of the language.

For the language user the following advantages of textual syntax over graphical are distinguished:

Information content: Textual models (usually) need less space to display the same amount of information as graphical models. This is especially important for large models, where it is hard to get a complete overview of graphical information.

Speed of creation: For experienced users, with today's tools, writing text is a lot more efficient than drawing graphical models.

Integration of languages: Integrating graphical languages with each other or with textual languages is very hard. It is much easier to integrate textual models, which improves understandability of these integrated models.

Speed and quality of formatting: With graphical languages, a lot of time is lost in the placement of elements in the model, while in textual languages this is much easier.

Platform and tool independency: There are textual editors on each operating system, but for graphical models, custom editors are always needed.

Version control: Version control is very important in modern software development. However, the current tools which are used the most (such as SVN and CVS) are textual based. For graphical models it is much harder to do proper version control.

For textual languages a lot more technical support and experience is available than for graphical models. Graphical models tend to be more intuitive than textual models, but this does usually not outweigh the advantages of textual languages as described above.

4.1.2 The concrete syntax must be usable

Useability aspects will play a role in the high level design. Questions such as how the language is composed have influence on the learnability of the language. Also, the design of the low level concrete syntax is important, as it might influence the efficiency with which users can write models, as well as the overview the language provides. Finally, the concrete syntax should be similar to what the language developers are used to. This doesn't mean we have to remodel the C language, but for common data structures it does not make sense to use a completely different notation than C or Java do.

4.1.3 The concrete syntax must be extendable

This requirement plays a role in the structuring of data. It is strongly related to the compositionality of the language. If the language is composed of several disconnected parts, these can easily be modified. Furthermore, in a language which is build in a compositional manner, it is easy to extend the language by adding new (orthogonal) components.

4.1.4 The concrete syntax must be composable

To be compositional on the language level means that the concrete syntax can be constructed from several loosely coupled parts. The benefit of high compositionality is that components can be easily added or removed (which makes it extendable). Compositionality on the level of model files improves scalability.

For the concrete syntax, this requirements means that models can be spread over multiple files. This makes it easier for multiple developers to work on the same model. For this, we will need to be able to create references between files and elements within these files. Furthermore, we will try to cut the language in a number of separate parts, each of which can be created separately (although there may be references between the parts).

4.1.5 The concrete syntax must be scalable

The concrete syntax should be scalable in the sense that it must be possible to describe models in a very concise manner. If the code is concise, larger models can still be viewed in total. A symbolic notation could facilitate this. The downside of a concise or symbolic syntax is that it is much harder to learn than for example a verbose syntax. Since the concrete syntax should also be usable, a tradeoff to these requirements should probably be made.

Locality of change is another important aspect of a scalable concrete syntax. This means that changes in a model should not effect the entire model, but only a single file (or as few lines as possible) in the file.

4.2 Selecting a concrete syntax style

The programming paradigm used influences the concrete syntax notation of a language extensively. A *paradigm* is the methodology on which the language is based. Examples of paradigms are imperative programming, functional programming and logic programming.

Also, the symbol style of the language can vary. With *symbol style*, the way the language uses keywords and delimiters is meant. For example, a language can employ a verbose style, with many large keywords, which is very easy to read and understand. On the other hand, a language could employ a mathematical style, in which keywords are minimized as much as possible, to allow a very concise notation of the models.

Depending on what the language user (programmer) is used to and what kind of problem needs to be solved, some styles and paradigms work better than others. As a preliminary step for implementing a concrete syntax, a number of different design alternatives have been explored. In this section, the paradigms will be described by writing the example coffee model in them and discussing the merits of each approach.

Three notations have been evaluated:

1. Imperative style: A mostly object-oriented style, with a Java-like syntax
2. Functional style: Functional programming style, with a Lisp-like syntax
3. Logic style: Logic programming style, with a Prolog-like syntax

Based on the evaluation of these notations, a design is made for the concrete syntax of RaPTR.

4.2.1 Imperative programming

The imperative style of programming is a widely used style in modern industry. Its main way of describing computation is in terms of statements that are executed in sequence. The Java language is a good example of this, and has been used as a guideline for the design of this concrete syntax prototype. The following code gives an imperative notation for the example model (see section 3.3).

```

Model coffee {
vars:
    cup_dispenser = new Capability(2); // The argument is the number of resources
    water_supply = new Capability(1);
    sugar_provider = new Capability(1);
    b_receive_choice = new Behavior();
    b_place_cup = new Behavior(cup_dispenser); // The argument is the claimed capability
    b_make_tea = new Behavior(water_supply);
    b_make_coffee = new Behavior(water_supply);
    b_add_sugar = new Behavior(sugar_provider);
    make_coffee_subplan = new Subplan();
    coffee_machine_plan = new Subplan();
begin:
    coffee_machine_plan {
        vars:
            Task t_receive_choice = new Task(b_receive_choice);
            Task t_place_cup = new Task(b_place_cup);
            Task t_add_sugar = new Task(b_make_sugar);
            Decision d_selected_coffee = new Decision([TRUE,FALSE]);
        begin:
            t_receive_choice;
            t_place_cup.addStartStartRelation(t_receive_choice);
            if (d_selected_coffee.TRUE)
            then {
                vars: SubplanReference make_coffee_reference =
                    new SubplanReference(make_coffee_subplan);
                begin: make_coffee_reference;
            }
            else {
                vars: Task t_make_tea = new Task(b_make_tea);
                begin: t_make_tea;
            }
            d_selected_coffee.addFinishStartRelation(t_place_cup);
            d_selected_coffee.addFinishStartRelation(t_receive_choice);
            t_add_sugar.addFinishStartRelation(d_selected_coffee);
        }
    }
    make_coffee_subplan {
        vars:
            Task t_make_coffee = new(b_make_coffee);
        begin:
            t_make_coffee;
    }
}

```

Listing 4.1: Example model in imperative programming notation

In this imperative concrete syntax, blocks are defined as a variable declaration part which is used in the imperative instruction part. The variables (Behaviors, Capabilities, Tasks, Decisions) that occur in the concrete syntax are directly derived from the abstract syntax. Resources are modeled as an integer argument of capabilities, which is different from the abstract syntax.

The main advantage of using this style is that it could easily be embedded in Java as an internal DSL. An internal DSL is a DSL which can be represented within a host language [6], which is in this case Java. Classes for each object can be generated and methods can be set for the contents of the language. Language users familiar with Java would be able to use this language with little effort.

A disadvantage of this prototype is that it is not very concise. Each variable has to be instantiated, placed within a container, and relations need to be defined for that. This results in a very large model. It seems like a lot of extra overhead is needed to define this model in an iterative way.

4.2.2 Functional programming

In a functional programming style, functions are the primary elements of the language. The entire program to be executed is a function, which gets a number of arguments. When translating the abstract syntax to a functional style, it seems logical to follow the tree structure as it is defined in the abstract syntax.

In this case, the toplevel function *'model'* will get four arguments: An identifier, a list of Capabilities, a list of Behaviors and a list of Subplans. Lists will be also depicted as functions with a certain amount of arguments, which are the contents of the list. This means that the function *'capabilityList'* will return a list with its arguments as contents. The same holds for functions such as *'behaviorList'* and *'subplanList'*. The function *'subplan'* takes five arguments: An identifier, a tasklist, a decisionlist, a subplan referencelist and a process, which describes the relations between the elements.

We will use a Lisp-like syntax, which means function arguments are enclosed by brackets and separated by commas. The example model in the functional style is defined by the following code.

```

model(coffee,
  capabilityList((cup_dispenser,2),(water_supply,1),(sugar_provider,1)),
  behaviorList((b_receive_choice,()),(b_place_cup,(cup_dispenser)),
              (b_make_tea,(water_supply)),(b_make_coffee,(water_supply)),
              (b_add_sugar,(sugar_provider))),
  subplanList(
    subplan(make_coffee_subplan,(
      taskList(t_make_coffee(b_make_coffee))
      decisionList(),
      subplanReferenceList(),
      process(t_make_coffee()))),
    subplan(coffee_machine_plan,(
      taskList(t_receive_choice(b_receive_choice),
              t_place_cup(b_place_cup),
              t_add_sugar(b_add_sugar))
      decisionList(
        d_selected_coffee(
          TRUE(taskList(),
              decisionList(),
              subplanReferenceList(
                make_coffee_reference(make_coffee_subplan)),
              process(make_coffee_reference())),
          FALSE(taskList(t_make_tea(b_make_tea))
              decisionList(),
              subplanReferenceList(),
              process(t_make_tea()))
        ))
      subplanReferenceList(),
      process(t_add_sugar(FS d_selected_coffee(
        FS t_receive_choice(),
        FS t_place_cup(SS t_receive_choice)) ))))
  )

```

Listing 4.2: Example model in functional programming notation

The functional style is more concise than the object-oriented style, since it contains less overhead in keywords. Its structure is also very simple, because the language consists of functions with arguments. However, because of this concise structure, it is also very hard to understand what a model written in this notation means. The nesting of functions can be very deep, which results in a large number of closing brackets after each other. In the example, extra structure is added by indenting the functions, but this is not as clear as it is with the object oriented style. It is possible to modify the layout of the model in order to gain a better overview, but this would result in a much larger model.

4.2.3 Logical programming

In logic programming, a model is a list of rules, which have a declarative part and a body, which contains the constraints on the declared part. An example logic programming language is Prolog, which has rules of the form ‘declaration :- constraints’. The following example describes the example model in a logic programming style.

```

Model(coffee).

Capability(cup_dispenser) :- resources(2).
Capability(water_supply) :- resources(1).
Capability(sugar_provider) :- resources(1).

Behavior(b_receive_choice).
Behavior(b_place_cup) :- claims(cup_dispenser).
Behavior(b_make_tea) :- claims(water_supply).
Behavior(b_make_coffee) :- claims(water_supply).
Behavior(b_add_sugar) :- claims(sugar_provider).

Subplan(coffee_machine_plan) :- {
    Task(t_receive_choice) :- type(b_receive_choice).
    Task(t_place_cup) :- type(b_place_cup), started(t_receive_choice).
    Decision(d_select_coffee) :- finished(t_receive_choice, t_place_cup finished),
        branches(
            TRUE (
                SubplanReference(make_coffee_reference) :- type(make_coffee_subplan).
            ),
            FALSE (
                Task(t_make_tea) :- type(b_make_tea).
            )).
    Task(t_add_sugar) :- type(b_add_sugar), finished(d_select_coffee).
}.

Subplan(make_coffee_subplan) :- {
    Task(t_make_coffee) :- type(b_make_coffee).
}.

```

Listing 4.3: example model in Logical programming notation

The main advantage of this approach is that it follows the way language users create models in RaPTR. A typical use case consists of adding a task to a subplan and defining the constraints for it. This way of working follows the logic programming paradigm. A downside of this notation is the fact that the notation is not very readable. It is hard to understand what the constraints on some of the declarations mean, especially if the language user is not familiar with the logic programming paradigm.

4.2.4 Style choice

Each paradigm has his own advantages and disadvantages, so no attempt is made to select the ‘best’ paradigm. It is still useful to compare the paradigms on some simple metrics, to get an idea of the scalability and understandability of the paradigms. The chosen metrics are the *maximum nesting depth*, the *number of references* and the *keywords or symbol overhead*.

The *maximum nesting depth* gives an indication on how much nesting occurs in a paradigm. Nesting is introduced by classes which contain other classes, but also by

the paradigm. The nesting depth should be low, because deep nesting will reduce the understandability of the model.

The *number of references* could give an indication on the conciseness of the prototype. The less references are needed, the better.

With *keywords or symbol overhead* the number of keywords and symbols in a model is divided by the number of useful information, which is anything besides the keywords. A low overhead results in very concise models, which is good for scalability, but may be hard to read. A large number of keywords may clutter the model, although they can also provide a model which is very readable. Bracket pairs are counted as one keyword.

The following table gives the metrics for the example model written in the three styles.

Metric	Imperative	Functional	Logical
Maximum nesting depth	4	6	4
Number of references	14	17	14
Keyword overhead	112/43=2,60	100/39=2,56	69/35=1,9

Table 4.1: Metrics for the models in different notational styles

As is to be expected, the functional notation has the deepest nesting depth. It is hard to see whether enough closing parentheses are present, because the nesting tends to be very deep. The number of references is about the same size for all three models. The functional prototype has a slightly higher reference count than the other prototypes, but this is due to the fact that the task declaration is separated from the process definition, which is not the case in the other models. The most interesting metric is the keyword (or symbol) overhead, which clearly indicates that the logical notation provides the least overhead, although it is still clearly structured.

When looking at these metrics, the logical notation seems the best paradigm to base the concrete syntax design on. In discussions with the domain experts, the logic programming style was indeed chosen to implement the language on. It has a number of other benefits, most importantly that it allows the language user to model in terms of ‘business rules’, which is the preferred way of working. However, for some parts of the concrete syntax, a mixture of paradigms is also possible.

Even though the base for the language will be logical programming, the symbolic style described in Listing 4.3 may not be the best for this case. Instead, a verbose style is chosen, in which the rules can be written down as if they are correct English sentences. This makes understanding what the rules mean much easier. The downside of this approach is that the models will probably not be as concise as when using a symbolic style, but this can be solved by introducing a transformation to a graphical representation. How this verbose logical programming concrete syntax is implemented in oAW is discussed in the next section.

4.3 Concrete syntax implementation in oAW

At the top level of the language, two separate parts within RaPTR can be distinguished, the subplans and the behavior/capability definition. Both parts can be designed as separate packages.

4.3.1 Behaviors and capabilities

The grammar in Xtext format for Capabilities and Behaviors has been defined as follows:

```
Capability:
    'Capability' name=ID ('has' resourcenumber=INT 'instances')?
    ;

Behavior:
    'Behavior' name=ID ('claims' capabilities+=[Capability]
        ('and' capabilities+=[Capability])* )?
    ;
```

The information that has to be provided for behaviors and capabilities is derived from the DAS. Behaviors need a reference to a number of Capabilities. As a part of Capabilities resources need to be defined. In the DAS, resources need an identifier. For the concrete syntax, these identifiers do not need to be specified by the language user, only the number of resources a capability has need to be specified. The identifiers will be generated at the time the concrete syntax is transformed into the DAS. This is in accordance with the 'aim for simplicity' guideline.

Furthermore, the order in which Behaviors and Capabilities are defined could have been restricted. For example, allowing Behaviors to be defined after *all* Capabilities are defined. However, since flexibility is considered an important aspect of the concrete syntax design, different elements of the language can be written down in an arbitrary order. A downside of this may be that models can become very badly structured once they grow larger. When they need to be maintained, it may become hard to find a specific rule, as it can be defined anywhere. To overcome this, the language users might agree upon using a particular convention (i.e. pragmatics). In this way, the language will still be flexible, while structure is imposed in the way the language users use the language.

This is also the main reason for leaving the largest part of the definitions optional. When for capabilities no resource number is set, the transformation should generate a resource number 1. This allows a concise way of defining a Capability with a default resource number of 1. The information needs to be added in the transformation, but it improves the usability of the language. A thing to keep in mind with this definition is that the language user could define 0 for the resource number. Although this might be a

legal construction, problems would arise if a behavior claims such a capability. A static semantic check is introduced to avoid this situation.

4.3.2 Subplans

The Xtext grammar for writing down subplans in the verbose logical programming style is defined as follows:

```

Subplan:
  'Subplan' name=ID '{'
  (subplanRules+=Rule)+
  '}'
;
Rule:
  TaskRule | SubplanReferenceRule | DecisionRule
;
TaskRule:
  'Task' name=ID 'of_type' behavior=[Behavior]
  ('requires' relations+=Relation ('and' relations+=Relation)*)?
;
SubplanReferenceRule:
  'SubplanReference' name=ID 'of_type' subplanname=[Subplan]
  ('requires' relations+=Relation ('and' relations+=Relation)*)?
;
DecisionRule:
  'Decision' name=ID
  ('requires' relations+=Relation ('and' relations+=Relation)*)?
  'branches' branches+=Branch branches+=Branch (branches+=Branch)*
;
Branch:
  label=STRING '{'
  (branchRules+=Rule)*
  '}'
;
Relation:
  sourceNode=[Rule] (isSS?='started'|isFS?='finished')
;

```

As described in Section 3.1, Subplans consist of nodes with relations. In the verbose logic programming style, this means that rules declare the nodes and the relations are written down as the constraints on these nodes. The decision node is different from the other nodes, because it contains branches, which again can contain an arbitrary number of nodes. The way to define a node containment is by enclosing it in curly brackets ('{' and '}'). Any kind of keyword could be chosen for this, but the curly bracket is a variation which is familiar to most language users, as it is used in other languages (such as C and Java) as well.

The ordering of nodes in the subplan is arbitrary, again to allow maximum flexibility in the definition of the language. As with Capabilities and Behaviors, this could result in poorly structured models, if language users do not take care where they define their nodes. So as a convention, language users who are defining a node could choose to only create references to nodes which were declared above them (textually). In this way, it is much easier to read the ‘flow’ from node to node.

Relations are defined as references to nodes with a keyword, which can be ‘finished’ or ‘started’. Depending on which keyword is entered, one of the two boolean variables (isSS or isFS) is set to true. These booleans are used in the transformation to decide whether a FS-relation or a SS-relation needs to be created.

4.3.3 File header

The contents of the concrete syntax have been specified in the previous two sections, but to implement it in Xtext, a root level element needs to be defined. This is done in the header of the file, which contains a ‘Model’ class that contains the high level components of the concrete syntax. Additional classes can be specified in the header section, such as the code to specify which elements need to be imported from different files. This functionality is desired, as it is often the case that models grow very large and they need to be split over multiple files. If a language is being used in real life, multiple developers work on the same model, so it is not wise to let them work on the same file. Thus, it should be possible to tell the system which files need to be imported so that references can be made to elements within these files.

Xtext has an importURI mechanism to do this, which looks for attributes with the name ‘importURI=String’. If such an attribute is encountered, the editor knows it should also allow references to objects defined in the file which are contained in the string. Another mechanism can be used to be able to use *namespaces*. This means it is possible to refer to an object by its *fully qualified name*, for example ‘*model1.examplesubplan.task1*’. Unfortunately, in Xtext 1.0 it is not possible to use these mechanisms together, thus the choice has been made to use the namespace mechanism.

The Xtext grammar part for the header of files uses the grammar parts which have been defined in the previous two sections through the component construct.

```
Model:
    'Model' name=ID
    components+=Component*
;
Component:
    Capability | Behavior | Subplan
;
```

4.4 Example model in concrete syntax

Once the grammar is implemented in Xtext, it is possible to generate a simple editor which can be used to write models for this language. The basic functionality of the editor is present without any additional specifications from the language designer. For example, syntax highlighting and content assist is implemented out of the box. Example 4.1 contains the example model as it is implemented in the Xtext generated editor.

Example 4.1

Example model in concrete syntax

```

1 Model coffee
2
3 Capability cup_dispenser has 2 instances
4 Capability water_supply has 1 instances
5 Capability sugar_provider has 1 instances
6
7 Behavior b_receive_choice
8 Behavior b_place_cup claims cup_dispenser
9 Behavior b_make_tea claims water_supply
10 Behavior b_add_sugar claims sugar_provider
11
12 Subplan coffee_machine_plan {
13     Task t_receive_choice of_type b_receive_choice
14     Task t_place_cup of_type b_place_cup requires t_receive_choice started
15     Decision d_coffee_decision
16         requires t_receive_choice finished and t_place_cup finished
17     branches
18     "TRUE" {
19         SubplanReference make_coffee_reference of_type make_coffee_subplan
20     }
21     "FALSE" {
22         Task t_make_tea of_type b_make_tea
23     }
24     Task t_add_sugar1 of_type b_add_sugar requires d_coffee_decision finished
25 }
26
27 Behavior b_make_coffee claims water_supply
28
29 Subplan make_coffee_subplan {
30     Task t_make_coffee of_type b_make_coffee
31 }

```

□

4.5 Summary

In this chapter, a number of concrete syntax alternatives were discussed and based on this, a verbose logical programming style syntax has been implemented in Xtext. A basic editor can be generated for it and the example model can be implemented in it. However, a number of other features need to be added to the editor for language users to be able to effectively model in it. Furthermore, transformations need to be added to be able to use the models written in the language.

Chapter 5

MDEE design

In the previous chapters, the design of the concrete syntax, the DAS and the FAS were discussed. This chapter elaborates on the model driven engineering environment in which RaPTR has been implemented. This encompasses the workings of the editor environment and the various transformations between concrete syntax, Ecore meta-models and target source code.

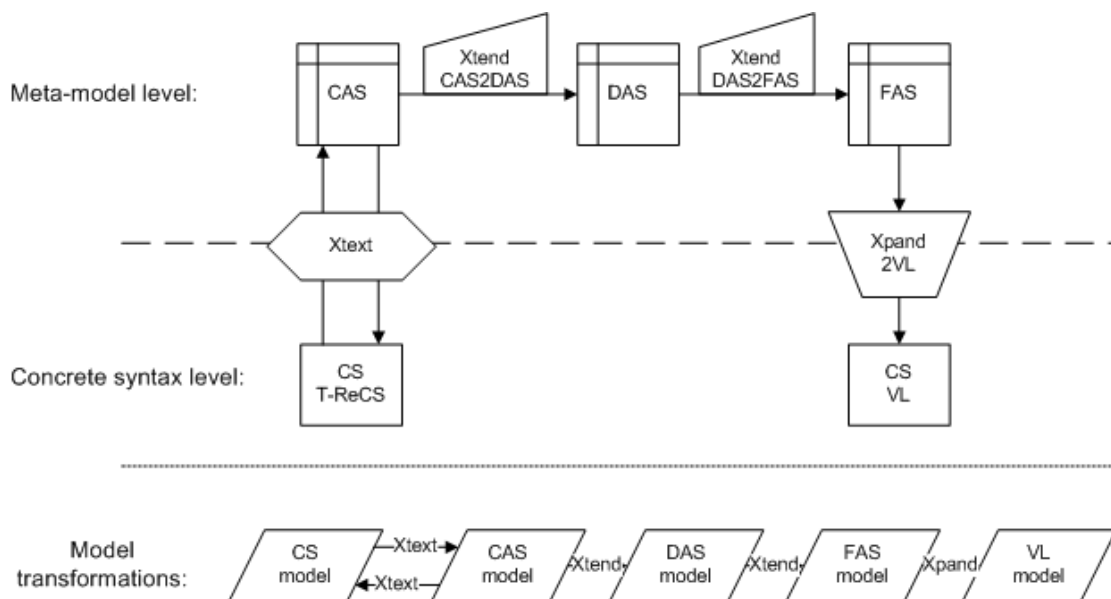


Figure 5.1: Model transformation flow of the MDEE

5.1 MDEE flow

The model driven engineering environment (MDEE) is the complete implementation of the concrete syntax through Xtext, the DAS and the FAS, including the various transformations between them (and to the verification language) written in Xtend and Xpand. Figure 5.1 gives a graphical overview of the components within the MDEE and the transformation flow between them. First, an overview of the transformations will be given, after which each of them is discussed in more detail.

5.1.1 CS2DAS (Xtext)

Xtext takes care of the automatic representation of concrete syntax models in the CAS representation. This is done through the editor environment, which is generated by Xtext. This editor environment can be enhanced by adding functionality such as scoping and validation rules.

5.1.2 CAS2DAS (Xtend)

The CAS2DAS transformation takes as input a model written in the CS, and transforms it through Xtend functions into a DAS representation.

5.1.3 DAS2FAS (Xtend)

The DAS2FAS transformation takes as input a model written in the DAS, and transforms it through Xtend functions into a FAS representation.

5.1.4 FAS2VL (Xpand)

The FAS2VL transformation takes as input a model written in the FAS, and transforms it through Xpand template into a textual representation conforming to the Verification Language (VL) concrete syntax.

5.2 Editor aspects in Xtext

Xtext is designed to generate an editor from just a grammar specification. However, a grammar cannot provide information for all necessary features such as scoping and static semantic checks. Xtext uses default implementations for these aspects. For basic languages, these default implementations are usually enough, but for real world industrial size languages such as T-ReCS, these are barely sufficient. Therefore, Xtext provides the functionality to override most of the functionality of the editor.

Table 5.1 contains an overview of the editor parts which can be manually overridden and for which a custom implementation can be defined.

Editor aspect	Description
Formatting	Provides overriding of automatic whitespace insertions. For example, automatic indentation after some element has been introduced, or deciding whether the editor uses automatic word-wrap at the end of the line or not.
Scoping	Provides overriding of some default scoping rules, which determine which elements are referable from which location. Importing from other files is part of the scoping definitions.
Validation	Provides the means to add static semantic checks to the editor, to check for constructs which are allowed by the grammar, but are not allowed by the semantics of the language.
Content assist	Provides a way to change the behavior of the content assist, which is invoked when Ctrl+space is pressed. It provides suggestions for what to type next.
Labeling	Provides a way to modify pictures (thumbnails) of some parts of the editor, such as the outline and the content assist window.
Outline	Provides the means to overwrite the outline of a model, for example to show some rules while omitting others.
Quickfix	Provides a way to create default fixes for some common errors. The quickfix functionality is an extension to the validation.

Table 5.1: Editor aspects in Xtext

Each of these aspects can be overridden by plain Java code. For RaPTR, it is necessary to introduce some validations rules and scoping rules, to prevent the language user from defining illegal models. The static semantic checks which are defined for the abstract syntax of Chapter 3 are partially covered by the concrete syntax, but for some of them validation or scoping is needed.

5.2.1 Validations

In Section 3.1.1, additional constraints have been specified for the DAS. These constraints could be checked by the transformation from concrete syntax to abstract syntax, but it would be better to warn the users for errors as soon as possible. Unless the constraints are covered by the grammar, validations need to be added. It turns out that validations need to be written for constraints 1, 4, 5 and 6. Rules 2 and 3 could also be covered by validations, but these are typical scoping issues.

Example 5.1

Example validation function for constraint 1

Xtext uses Google Guice [16] to allow overriding of functions. When the editor is generated, Xtext looks for functions which are tagged with the label '@Check' in the provided Java validation class. In this example, the tag is located on line 14. The parameter of the function needs to be a class-name in the Xtext grammar, in this case 'Rule'. Using this implementation, the function is executed for all Rule instances in a model. The function is not allowed to return anything, but it should contain at least one warning- or errorfunction (see line 20). The function takes the name of the current rule and checks whether it occurs in the precedence relation list of the rule. The rules which are referred to are also checked recursively, to see if they contain the name of the original rule. When it finds the name of the original rule, a boolean value false is returned and the recursion stops.

```

11 public class RaptrJavaValidator extends AbstractRaptrJavaValidator {
12
13 //The following check checks design constraint 1.
14 @Check
15 public void checkForLoops(Rule rule){
16     Boolean bool = true;
17     String testname = (rule.getName());
18     bool=recursiveCheck(testname, rule);
19     if (!bool) {
20         error("The relation path contains a loop!",
21             RaptrPackage.RELATION);
22     }
23 }
24 // Supporting function for checkForLoops
25 public Boolean recursiveCheck(String str, Rule rule){
26     Boolean bool = true;
27     for (int i=0; i<rule.getRelations().size(); i++){
28         if (rule.getRelations().get(i).getSourceNode().getName().equals(str)) {
29             bool = false;
30         } else {
31             bool = bool && recursiveCheck(str, rule.getRelations().get(i).getSourceNode());
32         }
33     }
34     return bool;
35 }

```

□

There are two approaches in Xtext to define validations. Either by defining them as rules in the Check language or as defining them as Java validations. The advantage of the Check language is that it is easier to understand and has a more concise notation than the Java validations. However, the Java validations allow for more advanced constructs, provide better performance and are necessary for linking with quickfix functionality. Therefore, the validations have been implemented in Java.

5.2.2 Scoping

Scoping in the language is necessary to prevent the linking between elements which are not allowed to be linked. Xtext has a number of standard implementations for this, which have a default behavior, which can be overridden for specific rule and reference combinations. For RaPTR, Example 5.2 contains the function which provides the necessary scoping rules to cover for constraints 2 and 3.

Example 5.2

Example scoping function for constraints 2 and 3

Xtext allows the language engineer to choose from a number of default scoping implementations, and for specific cases, overrides can be provided. Overriding is done by providing a so called IScope function. The name of the function is important, it should have the form: 'scope_<Classname>_<Referencename>'. The function returns an IScope, which is a collection of the elements a certain relation may refer to. In this case, a set of Nodes should be returned. Which nodes need to be returned can be determined by retrieving the container of the relation.

```

23 public class RaptrScopeProvider extends AbstractDeclarativeScopeProvider {
24
25     IScope scope_Relation_sourceNode(Relation rel, EReference node) {
26         Vector<Rule> outputlist = new Vector<Rule>();
27         EObject container = rel.eContainer().eContainer();
28         // Container is a Subplan
29         if (container.eClass().getName().equals("Subplan")) {
30             outputlist.addAll(((Subplan) container).getSubplanRules());
31         }
32         // Container is a Branch
33         if (container.eClass().getName().equals("Branch")) {
34             outputlist.addAll(((Branch) container).getBranchRules());
35         }
36         return Scopes.scopeFor(outputlist);
37     }
38 }

```

□

5.3 Model to model transformations

As can be seen in Figure 5.1, two model to model transformations have been implemented. These are the transformation from the concrete syntax Ecore model to the domain abstract syntax (CAS2DAS), and one from the domain abstract syntax to the formal abstract syntax (DAS2FAS).

For these transformations the Xtend language is used.

Xtend is a functional transformation language with side-effects. It uses a common expression language of oAW, which is also used by Xtext and Xpand. Xtend consists of *extensions*, which are in essence functions that take some input and produce some output. When model to model transformations are concerned, the main extension typically receives a model written in the input meta-model representation and transforms it into a model written in the output meta-model representation. To do this, it calls a number of other extensions, which transform smaller parts of the model.

The syntax for defining an extension is the following:

```
ReturnType extensionName(ParameterType1 parameterName1, ...) :
    expression, using parameters
;
```

Besides this basic extension, the *create extension* is also often used in model to model transformations. A create extension creates an object of the Returntype *before* the expression in the extension is executed. This provides an easy way to create models. Furthermore, using these create extensions prevents the duplication of objects, which is necessary when introducing references. The syntax for a create extension is the following:

```
'create' ReturnType extensionName(ParameterType1 parameterName1, ...) :
    expression, using parameters
;
```

Examples of these Xtend transformations can be found in the next section, where they will be used to illustrate the CAS2DAS and DAS2FAS transformations. For more information on Xtend, the oAW manual can be consulted [11].

5.3.1 Concrete Abstract Syntax to Domain Abstract Syntax

The concrete syntax and the domain abstract syntax are not extremely different, but a transformation still needs to be defined. The transformation in Xtend takes as an input a concrete syntax model file and transforms this into one abstract syntax model file, which is stored in XML. Alternatively, Xtend can also be configured to import a

complete directory of files or import a main concrete syntax model which transitively imports a number of other files.

In essence, both the concrete syntax model and the abstract syntax model are trees. Thus, the transformation is based on a tree-walk in either of the trees. This can be done in two directions: create a function which walks through the concrete syntax model tree and create the related elements in the abstract syntax (generation based approach), or walk through the abstract model tree that is being created and extract the needed elements from the concrete syntax tree (extraction based approach). Generally, the best approach is to walk through the smallest tree, because in that case it is not necessary to create redundant transformation rules.

In this case, both trees are about the same size, so the DAS is chosen as the basis for the transformation. The main reason for this is that the abstract syntax is the central point of the language and it should contain all elements in the language. When using this approach, we can also check whether for each rule in the abstract syntax a concrete syntax representation is present.

Example 5.3

Main Xtend function, which transforms CS models into a DAS model

As an example Xtend transformation, consider the main CAS2DAS Xtend transformation. It is a create extension, which creates a model in the DAS representation of RaPTR. The main thing to notice is the ‘->’ function, which mimics iterative behavior. The function on the left side of the arrow is executed first, followed by the function on the right. This way, all references of the Model class in the DAS can be ‘filled’ with information from the CAS. The functions *extractBehavior*, *extractCapability* and *extractSubplan* are other extensions, which transform the behaviors, capabilities and subplans to their corresponding representations in the DAS.

```

create RaptrAS::Model transform(List[raptr::Model] mdl) :
  this.setName(mdl.first().name)
  ->
  this.behaviors.addAll(mdl.components.extractBehavior())
  ->
  this.capabilities.addAll(mdl.components.extractCapability())
  ->
  this.subplans.addAll(mdl.components.extractSubplan())
;

```

□

Figure 5.2 gives the sample reflective Ecore editor view of the example model in the CAS and DAS representation. On the left side, the CAS tree is displayed. The right tree is the example model after it is transformed to the DAS. As can be seen, it is very similar to the concrete syntax tree representation. It seems like the only thing different is the ordering of nodes and some of the labels. However, as can be seen at the properties view at the bottom of the figure, the Capability has got two resources added to it, with names derived from the capabilityname.

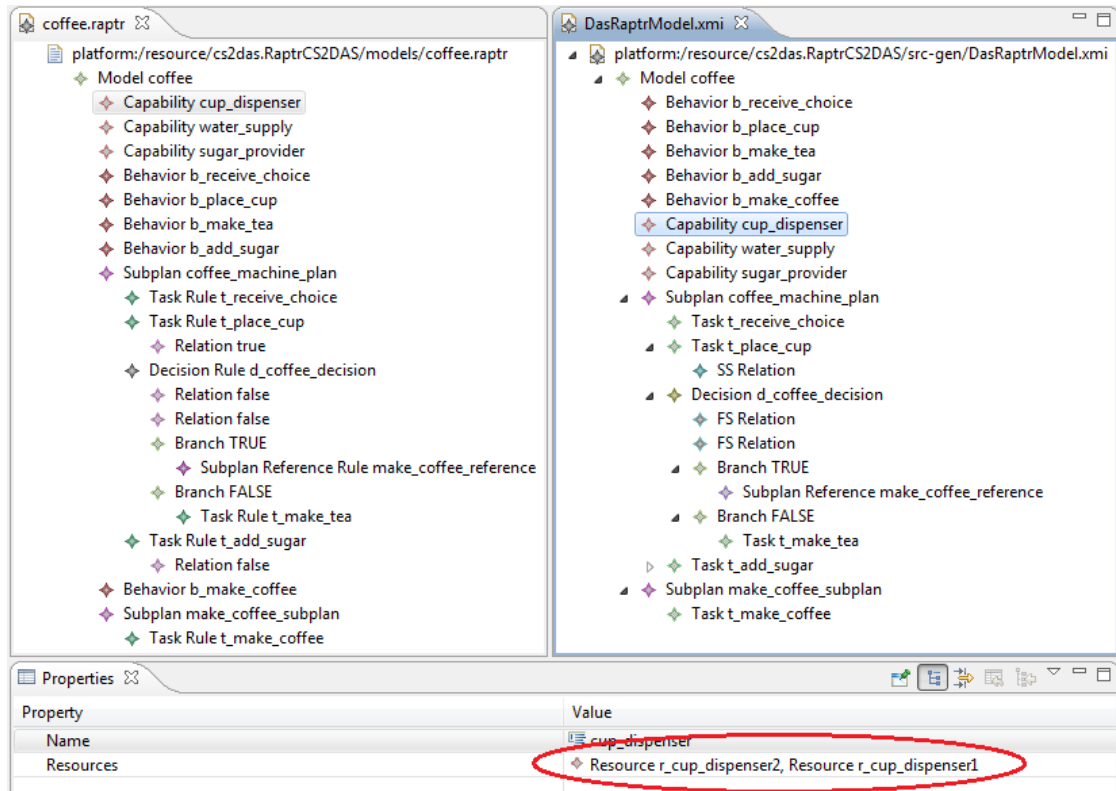


Figure 5.2: Ecore editor view of concrete syntax tree and an abstract syntax tree

5.3.2 Domain Abstract Syntax to Formal Abstract Syntax

The second Xtend transformation is the DAS2FAS transformation. It works in the same way as the CAS2DAS transformation, except that the transformation is partial: only the classes which are implemented in the FAS are transformed.

For some elements the mapping is straightforward, for others it is not so simple. Table 5.2 gives an overview of the classes which correspond to each other.

Two classes in the FAS are not in this list: The *SynchronizedParallelComposition* and

DAS class	Corresponding FAS class
Behavior	Task
Capability	Capability
Resource	Resource
Subplan	ProcessDefinition
Task	StartTask
Decision	ConditionalChoice
Branch	Branch
SubplanReference	ActivityDiagramReference
FS-Relation	SequentialComposition
SS-Relation	PreemptiveSequentialComposition

Table 5.2: Classes in the DAS and their corresponding classes in the FAS

the *FinishTask*. This is expected for the *FinishTask*. The *FinishTask* is an internal class, which is part of the execution semantics of the system. Once a *StartTask* is started, it is transformed in a *FinishTask*. In the DAS, this notion does not occur, since how tasks are processed is part of the engine definition. If the language designer *does* want to add the possibility to define *FinishTasks*, the CS and the DAS need to be modified.

The *SynchronizedParallelComposition* class specifies that two process terms are allowed to run in parallel. This is a notion which does occur in the DAS, but is not explicitly modeled. In the DAS, nodes within a subplan or branch which do not have a FS-relation or a SS-relation with each other, are allowed to run in parallel. This can be modeled in the FAS through a *SynchronizedParallelComposition*.

The transformation of subplans is the most interesting part of the DAS2FAS transformation. In the DAS, a subplan is a list of node declarations with constraints, while in the FAS this is modeled as a process term, which can have (sub) process terms (by the use of *PreemptiveSequentialCompositions*, *SynchronizedParallelCompositions* and *SequentialCompositions*).

The recursive algorithm to transform the DAS into the FAS is described in the following pseudocode. The *Skip* process term is used, which is used in the FAS definition as an empty process term. The input for the main algorithm (*processNodeList*) is the node list of the main subplan. The output is a process term, which can be anything from a basic skip to a large tree, starting with a *SynchronizedParallelComposition*.

```

PROCESSNODELIST(nodeList)
1  if nodeList.size == 0
2    then return Skip
3    else if nodeList.size == 1
4      return Process first node with processSingleNode
5    else Create SynchronizedParallelComposition,
6      with leftProcessTerm: Process first node with processSingleNode
7      and rightProcessTerm: Process tail of nodeList with processNodeList

```

The algorithm to process a single node is the following:

```

PROCESSSINGLENODE(node)
1  if node.relations.size == 0
2    then return Create StartTask from node name
3    else if node.relations.size == 1
4      then if node.relations.first == FSRelation
5        then Create SequentialComposition
6          with base node and node from Relation
7        else Create PreemptiveSequentialComposition
8          with base node and node from Relation
9      else Create SynchronizedParallelComposition,
10     with leftProcessTerm:
11     if node.relations.first == FSRelation
12       then Create SequentialComposition
13         with base node and node from Relation
14       else Create PreemptiveSequentialComposition
15         with base node and node from Relation
16     and rightProcessTerm:
17     Process node with tail of constraintList with processSingleNode

```

A schematic transformation of the first part of the example model is displayed in Figure 5.3. It shows the list structure of the DAS on the left, and the resulting tree-structure of the FAS on the right. One thing to notice is the large amount of duplication in the FAS. For example, ‘t_receive_choice’ is displayed three times. Although the transformation is correct and it allows for easy transformation to the process algebraic syntax, it might not be the most storage-friendly transformation, especially in the case the nodes which are being multiplied are composite nodes, as these will result in larger subtrees.

If the storage of models in the FAS becomes a problem, an internal model to model transformation could be written, reducing the tree in size, by combining similar subtrees. However, some duplication cannot be avoided: No matter how much rewriting is applied to Figure 5.3, ‘t_receive_choice’ and ‘t_place_cup’ will always be declared at least twice.

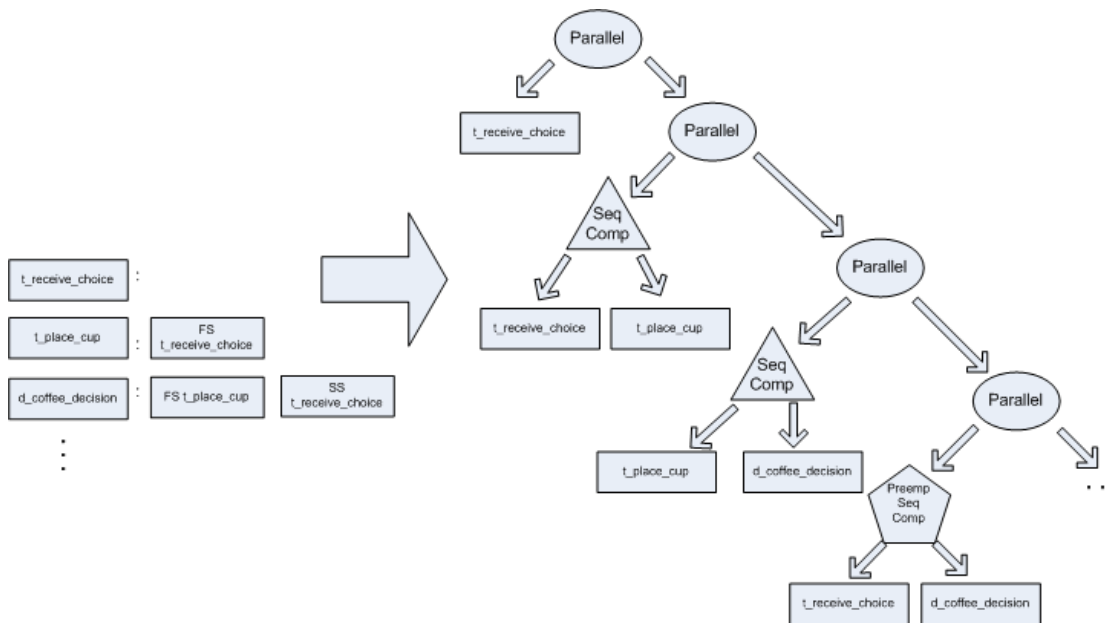


Figure 5.3: Visual representation of DAS to FAS transformation algorithm

5.4 Model to text transformation

Xpand is a template language that is part of oAW and can be used to generate text from Ecore models. The most important part of an Xpand template are the DEFINE blocks, which define the concrete syntax that needs to be generated once an object of a certain type is processed. The main DEFINE block, which is called from the workflow, expands a Model from the Ecore meta-model, most likely calling other DEFINE blocks through EXPAND statements.

The syntax for an Xpand template DEFINE block is the following.

```
«DEFINE expand-name FOR Class»
  static text, not enclosed in so-called guillemots.
  «EXPAND expand-name FOR Class.reference»
  «EXPAND expand-name FOREACH Class.referenceList»
«ENDDFIN»
```

The first thing to notice are the weird brackets, which are called *guillemots*. These are used because they are rare, and do most likely not occur in the target concrete syntax. Things outside the guillemots is static text, which will be written in the target file without further modification. The EXPAND statement calls other DEFINE blocks, for attributes or references of the class of the current DEFINE block. In case the EXPAND statement is called for a single element, FOR is used; if it is used for a list of elements, FOREACH is used.

Within the guillemots sections, the common expression language of oAW can be used, as well as Xtend functions. These Xtend functions should be defined in a Xtend file which is imported by the Xpand template. Other constructs, such as the FILE block (which defines the outputfile) and the IF block, are demonstrated in the example of the next section. For more information on Xpand, the oAW manual can be consulted [11].

5.4.1 Formal Abstract Syntax to Verification Language

The transformation from FAS2VL is very straightforward, as the structure of the FAS is directly represented in the Verification Language. The complexity of the VL representation is in the specification that *processes* the generated input, which is being developed in the related research project. This specification takes up the largest part of the generated VL file, but from the viewpoint of this transformation, it is static text.

Example 5.4

Main DEFINE block of FAS2VL Xpand template

```

1 «IMPORT fasfinal»
2 «EXTENSION xpandFAS2VL::v1Genhelperfunctions»
3
4 «DEFINE main FOR Model-»
5 «FILE name + ".mcr12"-»
6 sort Task_Label = struct «IF tasklists(processDefinitions).size==0»dummy1«ENDIF-»
7 «EXPAND taskexpand FOREACH tasklists(processDefinitions) SEPARATOR " | "»;
8 sort Process_Label = struct «IF processDefinitions.size==0»dummy2«ENDIF-»
9 «EXPAND activitydiagramexpand FOREACH processDefinitions SEPARATOR " | "»;
10 sort Capability = struct «IF capabilities.size==0»dummy3«ENDIF-»
11 «EXPAND capabilityexpand FOREACH capabilities SEPARATOR " | "»;
12 sort Resource = struct «IF resources.size==0»dummy4«ENDIF-»
13 «EXPAND resourceexpand FOREACH resources SEPARATOR " | "»;
14
15 map R_A: Capability -> List(Resource);
16 eqn «EXPAND capabilityResourceMapping FOREACH capabilities»
17
18 map R_Q: Task_Label -> List( Capability );
19 eqn «EXPAND taskCapabilityMapping FOREACH tasklists(processDefinitions)»
20
21 map processes: Process;
22 eqn «EXPAND processDefinitionexpand FOREACH processDefinitions»
23
24 init X( processes(«processDefinitions.first().name»), s([]) );
25
26 % STATIC STUFF:
27 «EXPAND staticExpand FOR this»
28 «ENDFILE»«ENDDEFINE»

```

□

The structure of the template is easy to understand. Recall a model in FAS contains lists of *tasks*, *processdefinitions*, *capabilities* and *resources*. Each of these are defined in the first part of the template. A ‘sort’ is defined, which contains the names of all the objects in the model. In the case of the task labels, the tasks are not used. Instead, it uses a Xtend function (‘tasklists()’) which extracts all task names from the processdefinitions. This way, unused tasks are not written in the model. Furthermore, this approach is needed because two StartTasks can have the same Task-name, which results in two different task labels in the VL.

The IF block contains a conditional. Everything within the block is written to text if the conditional is false. In this case, a check has been introduced to allow the transformation of models which have empty lists.

The EXPAND blocks make calls to other DEFINE blocks, which are not shown in Example 5.4. Most of them are quite straightforward, although the most interesting one is the *processDefinitionExpand* function. It contains an Xpand template which is defined for each type of processTerm. In the case of StartTasks, this results in a very simple definition, as only the name is written down. For the more advanced constructs, such as SynchronizedParallelComposition, it recursively calls its child processTerms. The DEFINE blocks for each type of processExpand look as follows.

```
«DEFINE processexpand FOR Skip»skip«ENDDFIN»
«DEFINE processexpand FOR StartTask»task_start(«this.name»)«ENDDFIN»
«DEFINE processexpand FOR FinishTask»task_finished(«this.name»)«ENDDFIN»

«DEFINE processexpand FOR SequentialComposition-»
seq( «EXPAND processexpand FOR leftProcess», «EXPAND processexpand FOR rightProcess» )«ENDDFIN»

«DEFINE processexpand FOR SynchronizedParallelComposition-»
upmodels( «EXPAND processexpand FOR leftProcess», «EXPAND processexpand FOR rightProcess» )«ENDDFIN»

«DEFINE processexpand FOR PreemptiveSequentialComposition-»
preempt( «EXPAND processexpand FOR leftProcess», «EXPAND processexpand FOR rightProcess» )«ENDDFIN»

«DEFINE processexpand FOR ConditionalChoice-»
choice( [«EXPAND processexpand FOREACH branches SEPARATOR ", "»] )«ENDDFIN»

«DEFINE processexpand FOR ProcessDefinition»«EXPAND processexpand FOR this.processTerm»«ENDDFIN»
«DEFINE processexpand FOR ProcessTerm»«ENDDFIN»
```

5.5 Summary

In this chapter the model driven engineering environment for RaPTR has been discussed. It consists of an Xtext project, two EMF based Ecore models, two Xtend transformations and an Xpand transformation. These meta-models and transformations provide a base prototype which can be used by language users and language developers. It is useful to extend this MDEE at some points, as will be demonstrated in the next chapter.

Chapter 6

MDEE Validation

There are two types of users of the MDEE: the *language user* and the *language developer*. The language user is mostly concerned with writing models in the concrete syntax and invoking transformations, while the language developer is mostly concerned with maintaining the MDEE, the abstract and concrete syntax and adding transformations. In this chapter, the MDEE is validated for both types of users.

The validation is done by performing three use cases (from a language users perspective) and three change cases (from a language developers perspective). First, some extensions to the MDEE are discussed, which are used to support the case studies.

6.1 MDEE extensions

During the validation of the MDEE, it proved that some cases were difficult to perform. From the language users perspective, it proved hard to get a quick overview of the models in the concrete syntax, as they could be arbitrarily structured. This will prove to be especially hard for users who are familiar with the legacy graphical implementation. From the language developers perspective, it was not desirable to modify the concrete syntax, as all models written in the old concrete syntax needed to be manually updated. The impact of the change was high, because models would always be stored in the concrete syntax representation.

To cope with these issues, two additional transformations have been added to the MDEE. These are Xpand template transformations, based on the DAS.

The DAS2DOT transformation has been added to support the language user. It transforms the subplans and state machine sections of the language into a dot file format [17],

which is used to generate graphical representations. This creates an overview of the models, which can help in the understanding of the logic behind models.

Another important additional transformation is the DAS2CS transformation, which transforms models stored in a DAS representation back to the verbose concrete syntax representation. This is a transformation that can be used by the language user to transform models into a standardized representation (increases readability of models), although it has mainly been added for the language developer. The DAS2CS transformation allows for automatic updating of models when the abstract syntax or the concrete syntax changes. How this is done is discussed in Section 6.3.

Figure 6.1 displays the model transformation flow of the MDEE including the added transformations.

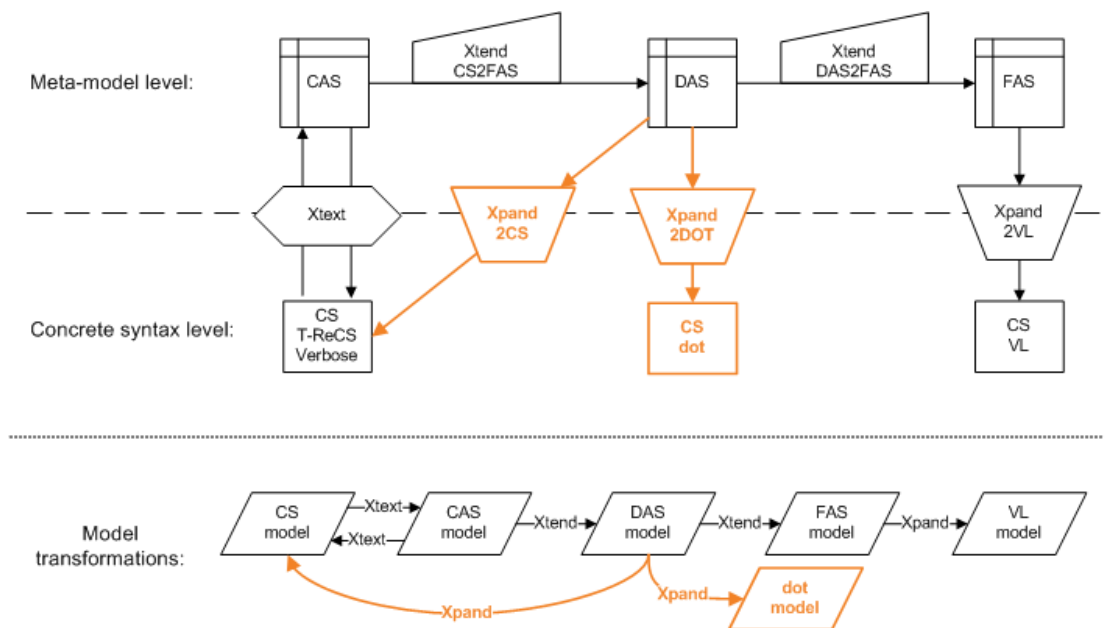


Figure 6.1: Final model transformation flow

6.1.1 Visualization of models

As has been described in Section 4.2.3, a drawback of the verbose logic style is that it is sometimes hard to get a quick overview of models. It works well for writing new models in it, but to get an overview of an already defined model is hard. The generation of a graphical representation through the dot language helps in this.

The structure of the dot language is very simple. It consists of vertexes and edges. Vertexes can be represented by rectangles, circles and other shapes, while edges can be

represented by arrows between these vertexes. Furthermore, subgraphs can be defined to cluster certain sets of nodes. From the DAS, it is quite simple to define a generation, since nodes and relations map directly to vertexes and edges.

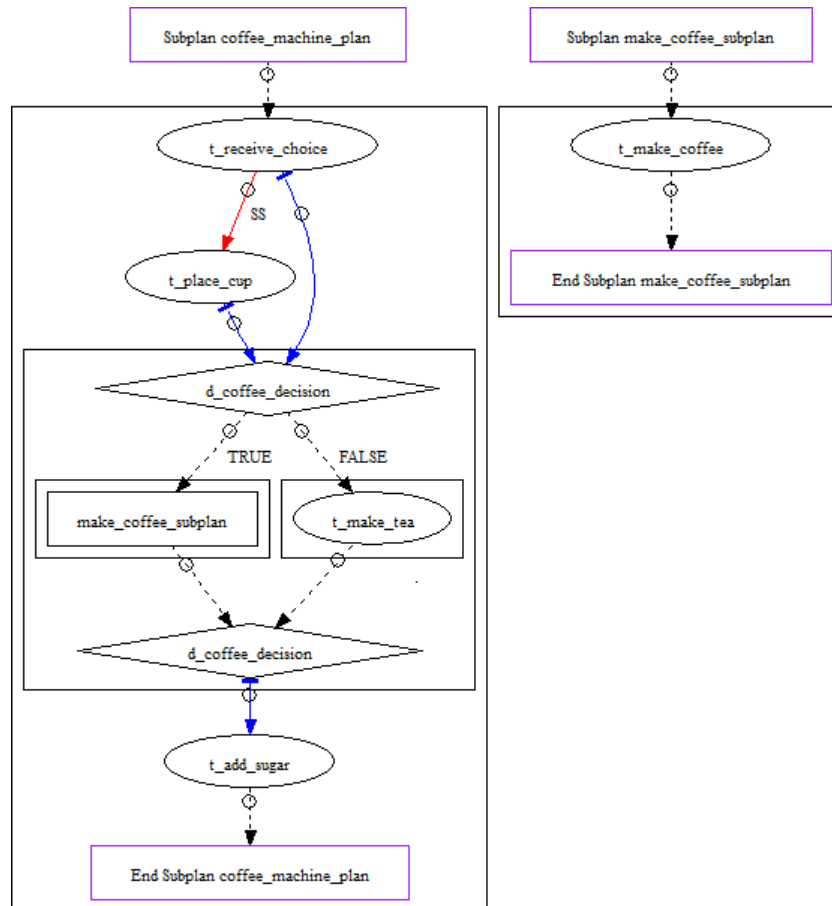


Figure 6.2: Graphical representation of model in dot format

Figure 6.2 shows the example model as it is graphically depicted by dotty [17]. Boxes are subplan references and ellipses are tasks. Decision sections are depicted as diamonds, with a number of associated branches. The end of a decision is marked with another diamond. Blue arrows are FS-relations, which are also marked with a bar at the source of the arrow. Red arrows are SS-relations and the dotted arrows signify a containment relation. In the example, two branches are contained within the decision section and a task and a subplan reference are contained in the branches.

6.1.2 Generation of CS models

The transformation from the DAS back to the concrete syntax representation has been designed for the language developer. Consider the situation where the DAS2CS transformation is not present. In this case, the language users write models in the concrete syntax representation and store them as such. The DAS is only used as an intermediate representation to the other representations, but the models are stored in the concrete syntax representation. When the language developer wants to change the concrete syntax, this would mean that all models written in the concrete syntax should be manually modified. The primary storage format of models will always be the concrete syntax representation, since once the models are transformed into the DAS representation, there is no way to modify them anymore.

```

1 Model coffee
2
3 /* This file is generated by Thomas' concreteRaPTRGenerator.
4 * It gives a standardized concrete syntax format for RaPTR,
5 * which can be regenerated to the DAS representation.
6 */
7 // BEHAVIORS
8 Behavior b_receive_choice
9 Behavior b_place_cup claims cup_dispenser
10 Behavior b_make_tea claims water_supply
11 Behavior b_add_sugar claims sugar_provider
12 Behavior b_make_coffee claims water_supply
13
14 // CAPABILITIES
15 Capability cup_dispenser has 2 instances
16 Capability water_supply has 1 instances
17 Capability sugar_provider has 1 instances
18
19 /*
20 *****SUBPLANS*****
21 */
22 Subplan coffee_machine_plan {
23     Task t_receive_choice of_type b_receive_choice
24     Task t_place_cup of_type b_place_cup
25     requires t_receive_choice started
26     Decision d_coffee_decision
27     requires t_place_cup finished and t_receive_choice finished
28     branches
29     "TRUE" {
30         SubplanReference make_coffee_reference of_type make_coffee_subplan
31     }
32     "FALSE" {
33         Task t_make_tea of_type b_make_tea
34     }
35     Task t_add_sugar of_type b_add_sugar
36     requires d_coffee_decision finished
37 }
38 Subplan make_coffee_subplan {
39     Task t_make_coffee of_type b_make_coffee
40 }

```

Figure 6.3: Standardized representation of concrete syntax, generated from DAS

With the addition of an Xpand DAS2CS transformation, maintenance of the language becomes much easier. The reason for this is that the primary storage format can be the concrete syntax representation but *also* the DAS representation. This makes it possible to transform models from concrete syntax to DAS and back, without losing information. When the language developer wants to change the concrete syntax, the models can be stored in the DAS representation and once the change has been implemented, the models can be transformed back into the new representation.

The transformation writes models back to concrete syntax, in a standardized layout. This standardized layout groups components such as behaviors and capabilities together, as can be seen in Figure 6.3. Furthermore, it uses an Xtend helperfunction to calculate the required indentation for tasks that occur in nested containers, such as branches.

The benefit of this additional transformation becomes clear when a change to the concrete syntax needs to be made. In this case, the models can be loaded into the DAS representation, the DAS to CS transformation could be adapted to cover for the new representation and the concrete syntax models could be generated easily. This even works if the DAS needs to be changed although a number of extra steps are needed. This will be demonstrated in Section 6.3.1.

The DAS to CS transformation does the exact opposite of the Xtend CAS2DAS transformation, as is to be expected. For each component in the concrete syntax, it takes the data from the DAS and produces a standardized concrete syntax representation. Many of these transformations simplify things: For example, when transforming resources, it can remove the resource IDs and just insert the length of the list into the concrete syntax representation. However, other issues are present which do not occur in the CAS2DAS transformation such as the layouting of the model file in a standardized way.

6.2 Language user perspective: Use cases

The use cases consist of typical usage scenarios of the language user. The language user writes and modifies models in the concrete syntax, after which these models are transformed into other representations. In this section, the most common use cases are executed, to illustrate some of the quality aspects of the concrete syntax and the editor. The use cases performed are the following:

- UC1:** Add a capability and a behavior
- UC2:** Add a task and a relation
- UC3:** Modify the subplan structure

The first use case shows the compositional structure of the concrete syntax, as two high level components are added to the model. The second use case demonstrates the most

common task a language user does: adding a task to a subplan and relating it to other tasks in this subplan. It should follow the locality of change principle: if a task and a relation are added, the existing model should not need to change. The third use case is introduced to be able to perform an extensive modification of the model, which will demonstrate that it is possible to modify the entire structure of a model with simple copy paste actions.

```

1 Model coffee
2
3 Capability cup_dispenser has 2 instances
4 Capability water_supply has 1 instances
5 Capability sugar_provider has 1 instances
6
7 Behavior b_receive_choice
8 Behavior b_place_cup claims cup_dispenser
9 Behavior b_make_tea claims water_supply
10 Behavior b_add_sugar claims sugar_provider
11
12 Subplan coffee_machine_plan {
13     Task t_receive_choice of_type b_receive_choice
14     Task t_place_cup of_type b_place_cup requires t_receive_choice started
15     Decision d_coffee_decision
16         requires t_receive_choice finished and t_place_cup finished
17         branches
18         "TRUE" {
19             SubplanReference
20         }
21         "FALSE" {
22             Task t_make_tea
23         }
24     Task t_add_sugar1 of_type
25 }
26
27 Behavior b_make_coffee claim
28
29 Subplan make_coffee_subplan
30     Task t_make_coffee of_ty
31 }
32
33 Behavior b_add_cream claims cream dispenser
34 Capability cream_dispenser

```

Figure 6.4: Example model with added capability and behavior, showing auto-completion

6.2.1 Use case 1: Add a capability and a behavior

In the example coffee model, three capabilities exist, *cup_dispenser*, *water_supply* and *sugar_provider*. For this use case, the capability *cream_dispenser* will be added, as well as the behavior *b_add_cream*, which claims this capability.

The flexibility of the language allows these components to be added anywhere in the model. In this case, they are added at the bottom of the file, after the subplan definitions. As can be seen in Figure 6.4, no resources are defined for the capability (see red circle). As was explained in the CS definition of Section 4.3.1, this means that the

Xtend transformation to DAS will add one resource to it, in its DAS representation.

The behavior is added to the model after the capability has been added, and after the keyword ‘claim’ has been written, the auto-completion gives the available capabilities in the model.

Results

With this case study, it has been demonstrated that the language allows full flexibility with respect to where components are defined. The behavior can claim any capability, whether it is defined below or above the behavior definition, which is demonstrated by the auto-completion feature of the editor.

6.2.2 Use case 2: Add a task

The capability and behavior added in the previous use case are not used yet in any subplan. In this case study, a task will be added which is an instance of *b_add_cream* and requires the decision *d_coffee_decision* to be finished. It can be defined anywhere in the subplan, but as a convention it should be declared after the nodes it claims. For this case, this means the task will be added at the end of the subplan definition. Figure 6.5 shows the part of the subplan where the task has been added.

```

12 Subplan coffee_machine_plan {
13     Task t_receive_choice of_type b_receive_choice
14     Task t_place_cup of_type b_place_cup requires t_receive_choice started
15     Decision d_coffee_decision
16         requires t_place_cup finished and t_receive_choice finished
17         branches
18         "TRUE" {
19             SubplanReference make_coffee_reference of_type coffee_machine_plan
20         }
21         "FALSE" {
22             Task t_make_tea of_type b_make_tea
23         }
24         Task t_add_sugar of_type b_add_sugar requires d_coffee_decision finished
25         Task t_add_cream of_type b_add_cream requires d_coffee_decision finished
26     }
27
28 Behavior b_make_coffee claims water_supply
29
30 Subplan make_coffee_subplan {
31     Task t_make_coffee of_type b_make_coffee
32 }

```

Figure 6.5: Subplan with added task

Results

With this case study, it has been shown that adding a task, which is a common use case, does not require any changes in other parts of the model. This demonstrates the compositionality of the definition of subplans.

6.2.3 Use case 3: Modify the subplan structure

As a final use case for the language user, the entire subplan in the example model will be changed. This will be done by placing the nodes which are outside the decision section, *inside* the decision section. To ensure the same behavior of the system, some tasks will be duplicated in the branches of the decision.

To perform this change, the following steps are performed on the model from Figure 6.5.

1. Copy lines containing tasks *t_receive_choice* and *t_place_cup* into the TRUE branch, add the number 1 to the task identifier
2. Move lines containing tasks *t_receive_choice* and *t_place_cup* into the FALSE branch, add the number 2 to the task identifier
3. Copy ‘requires’ section from Decision branch to *make_coffee_reference*, add number 1 to the referenced tasks
4. Move ‘requires’ section from Decision branch to *t_make_tea*, add number 2 to the referenced tasks
5. Copy lines containing tasks *t_add_sugar* and *t_add_cream* into the TRUE branch, add the number 1 to the task identifier
6. Change the ‘requires’ relation of *t_add_sugar* and *t_add_cream* into *make_coffee_reference finished*
7. Move lines containing tasks *t_add_sugar* and *t_add_cream* into the FALSE branch, add the number 2 to the task identifier
8. Change the ‘requires’ relation of *t_add_sugar* and *t_add_cream* into *t_make_tea finished*

This ensures that the Decision becomes the only (top level) node in the subplan, containing the two alternative sections within it.

Figure 6.6 shows the subplan section of the concrete syntax, combined with the graphical representation in dot, generated from this model. The original dot representation can be found in Figure 6.2.

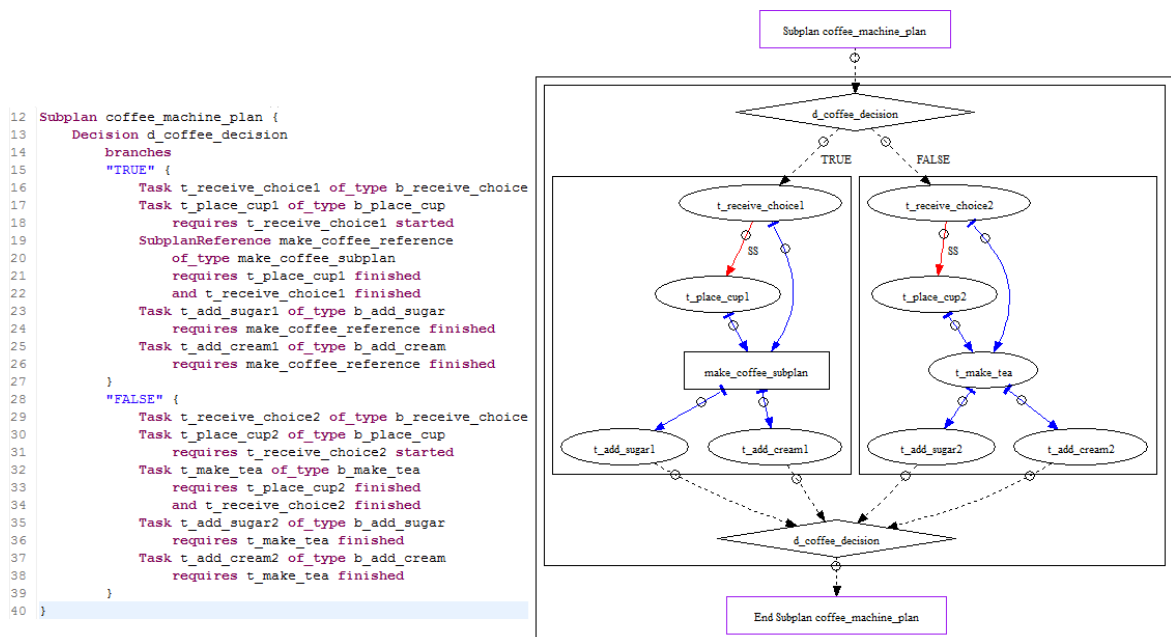


Figure 6.6: Modified subplan in concrete syntax and dot-representation

Results

With this case study, an extensive modification of the example subplan has been demonstrated. With the compositional structure of tasks and their relations, it is relatively easy to modify a model with using only copy paste operations. Furthermore, the change is clearly visualized by transforming the model to the dot format.

6.3 Language developers perspective: Change cases

The change cases are possible maintenance scenario's the language developer may encounter. Three change cases are performed on the MDEE, demonstrating the steps necessary to modify or add functionality to it.

- CC1:** Modify concrete syntax
- CC2:** Modify abstract syntax
- CC3:** Add a concrete syntax

The first change case is the most likely change case for the MDEE. Although the verbose concrete syntax is sufficient for creating RaPTR models, there may be opportunities to improve the concrete syntax. When the concrete syntax is validated with users, it will

almost certainly provide suggestions for improvement. The first change case will show that it is easy to modify the concrete syntax and update the models written in this concrete syntax with automatic transformations.

The second change case demonstrates a fundamental change case for the MDEE, in which the DAS is changed. Since the DAS is the linchpin of the MDEE, changes will affect most meta-models and transformations in the MDEE. This change case introduces a method to change the DAS without the need to manually modify any models stored in the old representation. This provides the possibility to migrate models stored in an old DAS representation to a new DAS representation.

The third change case illustrates the extendability of the MDEE. It is demonstrated that it is relatively easy to add a complete concrete syntax to the language and to transform models from one concrete syntax to the other, through the DAS representation. This paves the way for the implementation of multiple concrete syntaxes for the language.

6.3.1 Change case 1: Modify concrete syntax

The concrete syntax of the language should be very easy to modify, since it is the main language interface for language users. The verbose concrete syntax is sufficient to create RaPTR models but the syntax may be modified when language users start using the tool and feedback is gathered.

Depending on the complexity of the change, different parts of the MDEE need to be adapted. The models written in the concrete syntax need to be adapted in most cases, but they can be regenerated once the DAS2CS Xpand template has been adapted. Table 6.1 contains a number of concrete syntax changes and the MDEE aspects that need to be adapted in order to accommodate the change. It is assumed that the changes do not influence the DAS meta-model, as these changes have a bigger impact.

For this case study, we will consider the most difficult change case, in which an attribute is moved from one rule to another (in this case new rule), while the DAS remains the same.

As a change, consider the concept of explicit resource definition. In the current implementation, resources are modeled as a number which is part of the capability rule. The Xtext grammar for this is:

Capability:

```
'Capability' name=ID ('has' resourcenumber=INT 'instances')?  
;
```

Say, it is desired to be able to define resources separately from capabilities, to allow the

Change	Influenced MDEE components	Complexity
Add an optional keyword	Grammar	Low: Models do not need to change
Change a keyword or add a obligatory keyword	Grammar and DAS2CS Xpand template	Low: Models can be reloaded from the DAS if Xpand template is modified.
Change an attribute or rule	Grammar, CS2DAS Xtend files and DAS2CS Xpand template	Medium: The Xtend transformation needs to change. To change the models, the Xpand template needs to be modified to accommodate the changed attribute representation.

Table 6.1: Complexity of changes of the concrete syntax

language user to specify an explicit resource name. To accommodate for this, resources could be added as a top-level component, which could be implemented by the following grammar:

```

Capability:
    'Capability' name=ID
;
Resource:
    'Resource' name=ID 'of_type' capability=[Capability]
;

```

The procedure for implementing this change and modifying the models written in the verbose concrete syntax is done according to the following steps:

1. Load all models into the DAS representation, by executing the CS2DAS Xtend transformation.
2. Modify the Xtext grammar to allow for the new concrete syntax.
3. Modify the DAS2CS Xpand template, to generate the correct concrete syntax representation.
4. Transform the DAS models back into the new concrete syntax representation to check whether the transformation is implemented correctly.
5. Modify the CS2DAS Xtend transformation, to make sure the models written in the new concrete syntax transform to the correct DAS representation.
6. (Optional) Modify the validation functions of the Xtext project, to make sure they are still correct for the new concrete syntax.
7. (Optional) Rerun the Xtext workflow, export the Xtext project as a plugin and restart Eclipse. This ensures the editor works correctly.

With this approach, the models will be modified by the transformation. An interesting alternative to the approach described above is to not modify the grammar, Xtend and Xpand templates, but copy them and create new versions of the grammar. In effect, this would result in multiple concrete syntax versions for the same abstract syntax, which can be transformed into each other through the DAS.

Results

This case study has successfully been performed, requiring only slight modifications in the grammar and the Xpand template, although it required a little more work in the Xtend transformation. The reason for this is that in the DAS, resources are a part of Capabilities, while in the new concrete syntax, resources are a top level class with a reference to a capability. Therefore, an Xtend function needs to be written to extract the requirements with a reference to the current capability (which would otherwise be unnecessary). For this implementation, the old Xpand template and Xtend functions have been preserved, to allow easy switching between both representations of the resources. Figure 6.7 shows the top of the example model as it would look with the explicit resource definition.

A thing to keep in mind when modifying the concrete syntax, is that the editor and the validations implemented for this editor are dependent on the concrete syntax. Therefore, it might be the case that some validations do not work anymore after modification of the concrete syntax, or produce unexpected results. These need to be tested after the language is modified.

```
1 Model coffee
2
3 Capability cup_dispenser
4 Capability water_supply
5 Capability sugar_provider
6
7 Resource cup_dispenser1 of_type cup_dispenser
8 Resource cup_duspenser2 of_type cup_dispenser
9 Resource water_supply1 of_type water_supply
10 Resource sugar_provider1 of_type sugar_provider
11
12 Behavior b_receive_choice
13 Behavior b_place_cup claims cup_dispenser
14 Behavior b_make_tea claims water_supply
15 Behavior b_add_sugar claims sugar_provider
16
17 Subplan coffee_machine_plan {
18     Task t_receive_choice of_type b_receive_choice
```

Figure 6.7: Explicit resource definition for example model

6.3.2 Change case 2: Modify abstract syntax

Since the DAS is the central representation format of the language, great care is needed when modifying it. Modifications of the DAS may influence every transformation in the MDEE. Some modifications, such as *changes* in the abstract syntax are much harder to perform than *additions* to the abstract syntax.

Change	Influenced MDEE components	Complexity
Add a class, attribute or reference	Ecore model DAS	Low: The other artifacts do not <i>need</i> to change, although in most cases it is desired.
Remove a class, attribute or reference	Possibly all MDEE components	Medium: All transformations that use the removed element need to be changed, to remain correct.
Change a class, attribute or reference	Possibly all MDEE components	High: Depending on the size of the change, all transformations that are based on the changed element need to be adapted.

Table 6.2: Complexity of changes to the DAS

The third change case of Table 6.2 is the most complex change, since it potentially impacts everything. Therefore, it might be wise to avoid the change as much as possible, and try to model it as an addition to the DAS. Also, if the change is structural but does not effect the data required of the models, it might be smart to create a new version of the DAS (say DAS') and make a transformation from DAS to DAS' and back. This way, the old DAS can be used as basis for the existing transformations and the DAS' can be used for future transformations.

For this change case, a structural change is chosen which does not add information. This means that the concrete syntax does not need to change.

In the current implementation (for RaPTR), Relation is an abstract class, which can be an FS-Relation or a SS-Relation. Instead of modeling it as an abstract class with two concrete subclasses, the choice could be made to model Relation as a concrete class with a boolean called 'isFS', which is true in the case the Relation is a FS-relation and false if it is a SS-relation.

To make this change in the MDEE and to make sure all models are automatically transformed, the language developer should perform the following steps:

1. Transform all models into the concrete syntax representation, by executing the DAS2CS Xpand transformation.

2. Modify the Ecore model to allow for the new abstract syntax representation.
3. Modify the DAS2FAS Xtend template, to generate a correct FAS-representation of the model.
4. Modify the Xpand templates, to accommodate for the new abstract syntax representation.
5. Modify the CS2DAS Xtend transformation, to make sure the models written in the concrete syntax correctly transform to the new DAS representation.

When these changes are made, the models stored in the concrete syntax representation can be transformed into the DAS representation.

Results

This case study has been performed by introducing a new DAS', in order to leave the old DAS intact. Furthermore, since it is much easier to *extend* the DAS with the new information instead of *changing* it, the old representation for relations is not removed. This would mean that both representations are possible in the Ecore model: An FS-Relation is stored both as a subclass of Relation, and as a concrete Relation, with the boolean 'isFS' set to true. This approach has both advantages and disadvantages.

The main disadvantage is that the Ecore model representation becomes larger and stores redundant information, which needs to be generated by the Xtend transformation from the concrete syntax. However, once one of the representations is derived from the concrete syntax, it is very easy to derive the other representation from it. This could even be done by an internal model to model transformation on the DAS model, to prevent cluttering of the CS2DAS transformation.

The main advantage of this alternative approach is that the existing transformation *from* the DAS do not need to be modified, since the classes, attributes and relations needed for these transformations are still present. Furthermore, future transformations can be based on either or both Relation representations in the DAS, depending on which is more convenient for that particular transformation.

6.3.3 Change case 3: Add a concrete syntax

With the addition of the Xpand DAS2CS transformation, the main storage format for RaPTR models has become the DAS representation. This allows the language developer to define multiple concrete syntax representations for this DAS. To connect a concrete syntax to the DAS, an Xtend transformation needs to be added from the new concrete syntax to the DAS, as well as an Xpand transformation from the DAS to the concrete syntax.

The new concrete syntax could be partial or complete. If a concrete syntax is complete, this means that all classes, attributes and relations are expressible in the concrete syntax. A partial concrete syntax covers only part of a model, which means that if a model is transformed from the DAS to the partial concrete syntax and back again, information is lost, unless models are merged.

For this case study, we will create a complete concrete syntax, based on the verbose concrete syntax. This means that the Xtend transformation and the Xpand transformation from and to the verbose concrete syntax can be reused for the new complete concrete syntax, and no merging of models needs to be implemented. Furthermore, since it is a complete concrete syntax, models can be transformed from the verbose concrete syntax into the new concrete syntax (through the DAS), which effectively demonstrates multiple views on the domain abstract syntax.

To implement a new concrete syntax, the following steps need to be done.

1. Create a new Xtext project, define the grammar for the syntax.
2. (Optionally) Run the workflow to generate an editor for the concrete syntax, export the project as a plugin and restart Eclipse.
3. Create (or copy and modify) the DAS2CS Xtend transformation, according to the new CS.
4. Create (or copy and modify) the CS2DAS Xpand transformation, according to the new CS.

The structure for the new concrete syntax will be very similar to the verbose concrete syntax. Only keywords and the ordering of attributes and references in rules will be changed, which will make sure that the original Xtend CS2DAS transformation can be reused. The Xpand DAS2CS template can also be based on the initial version. It needs to change, but it will be a relatively simple modification, as only keywords are replaced and the ordering of some functions is changed.

As a style for the new concrete syntax, an ad-hoc representation is made, in which a very concise style is chosen for creating models, in contrary to the verbose style. As little as possible keywords are used, and replaced by punctuation marks. This concrete syntax will be called the minimalistic concrete syntax. The following Xtext grammar gives the part of the concrete syntax that deals with subplans.

```
Subplan:
    'Subplan' name=ID
    '{' subplanComponents+=SubplanRule+ '}'
;
SubplanRule:
    Task | SubplanReference | Decision
;
```

```

Task:
  'Tsk' name=ID ':' behavior=[Behavior]
  ('<-' relations+=Relation (',' relations+=Relation)*)?
  ','
;
SubplanReference:
  'SRef' name=ID ':' subplanname=[Subplan]
  ('<-' relations+=Relation (',' relations+=Relation)*)?
  ','
;
Decision:
  'Dec' name=ID
  ('<-' relations+=Relation (',' relations+=Relation)*)?
  branches+=Branch branches+=Branch (branches+=Branch)*
;
Branch:
  label=ID
  '{' containercomponents+=SubplanRule* '}'
;
Relation:
  node=[SubplanRule]
  (isSS?='SS' | isFS?='FS')
;

```

Results

The minimalistic concrete syntax has been implemented for the final version of the project. The grammar and the transformations can be found in the code of the final project. No validations have been implemented for it, and the same scoping configuration as the verbose concrete syntax has been used. As an example, the generated minimalistic code for the subplan of the (original) example model is displayed in Figure 6.8.

6.4 Summary

In this chapter, the validation of the MDEE has been described. Three use cases and three change cases have been successfully performed, as well as an implementation of an industrial case study. To support the language user and the language developer in their tasks, two additional Xpand transformations have been added to the MDEE. Based on these validations, recommendations for the use of the MDEE can be formulated, which will be the topic of the final chapter of this thesis.

```
59 /*
60 *****SUBPLANS*****
61 */
62 Subplan coffee_machine_plan {
63     Tsk t_receive_choice : b_receive_choice.
64     Tsk t_place_cup : b_place_cup
65     <- t_receive_choice SS .
66     Dec d_coffee_decision
67     <- t_place_cup FS , t_receive_choice FS
68     TRUE {
69         SRef make_coffee_reference : make_coffee_subplan.
70     }
71     FALSE {
72         Tsk t_make_tea : b_make_tea.
73     }
74     Tsk t_add_sugar : b_add_sugar
75     <- d_coffee_decision FS .
76 }
77 Subplan make_coffee_subplan{
78     Tsk t_make_coffee : b_make_coffee.
79 }
```

Figure 6.8: The example model in the (generated) minimalistic concrete syntax

Chapter 7

Conclusions and recommendations

7.1 Conclusions

The research question for this project as stated in the introduction of this thesis was the following:

Using model-driven engineering techniques, is it possible to create and maintain a complete model driven engineering environment for the development and use of an industrial sized language, including transformation support?

In this thesis an attempt has been made to provide an answer to this question by implementing a prototype model driven engineering environment for the RaPTR and the T-ReCS language (not discussed in this thesis). The answer to the research question will be provided by reflecting on each chapter of this thesis.

7.1.1 Language engineering approach

A language engineering approach has been defined for this project, consisting of the development of a number of meta-models and transformations. This approach uses EMF and the former openArchitectureWare tooling to define meta-models and transformations for it. It has been shown that it is required to separate the abstract syntax generated from the concrete syntax of Xtext (CAS) from the abstract syntax designed from the domain analysis (DAS). This in turn required a Xtend transformation from the CAS to the DAS, which was the basis for the MDEE design.

7.1.2 Abstract and Concrete syntax design

Based on a domain analysis an abstract syntax has been designed, which covers all aspects of RaPTR. It is used as the central storage point of models in RaPTR and it has proven to be a solid basis for transformations.

Three concrete syntax alternatives have been designed to help decide a suitable concrete syntax implementation. These were the imperative, functional and logical styles. Based on these styles, the logical style is chosen as a basis for implementing the concrete syntax in Xtext. The notation of the concrete syntax is verbose, which allows for writing models in a business rule like manner. The concrete syntax has been designed according to basic design guidelines and requirements from ASML. The verbose concrete syntax is able to express all elements present in the abstract syntax, which makes it possible to transform complete models from concrete syntax to abstract syntax. A basic editor environment has been generated for this concrete syntax that supports language users with writing models in the concrete syntax.

7.1.3 Model driven engineering environment design

The MDEE as it was defined in the language engineering approach of this thesis has been implemented in EMF and the former oAW tooling. It consists of an Xtext project (CS and CAS), two EMF based Ecore models (DAS and FAS), two Xtend transformations (CAS2DAS and DAS2FAS) and an Xpand transformation (FAS2VL).

These meta-models and transformations provide a MDEE prototype that is able to transform concrete syntax models into the verification language representation through several meta-models. It has been shown that the language engineering approach can be implemented for an industrial sized language with the provided technologies.

7.1.4 Validation

Two additional transformations have been added to the MDEE, to provide additional support for both the language user and the language developer. First, a DAS2DOT transformation has been added, which can generate graphical models from the concrete syntax models. Second, a DAS2CS transformation has been added, which transforms models stored in DAS representation back into a standardized concrete syntax representation. This allows models to be stored in either DAS or CS representation, which is helpful for maintenance of the MDEE.

It has been shown that it is possible to implement an existing model in the new concrete syntax, and transform it to the verification language as well as a graphical representation.

It has also been shown that it is easy to modify the concrete syntax and regenerate models written in the concrete syntax by means of the DAS2CS Xpand template. Modifications of the abstract syntax are harder, although there are ways to make these changes have less impact on the MDEE. Furthermore, it has been shown that it is possible to add different concrete syntaxes to the MDEE and transforming models written in one representation into the other representation.

7.1.5 Final conclusion

Based on the results from the previous sections, it can be concluded that it is possible to create and maintain a complete model driven engineering environment for the development and use of an industrial sized language, including transformation support. This has been demonstrated by the design and validation of a prototype MDEE in EMF and oAW. Furthermore, a language engineering approach has been devised which supports the evolution of a language and its implementation in a MDEE. Figure 7.1 shows the final model transformation flow of the different components which are part of the MDEE prototype.

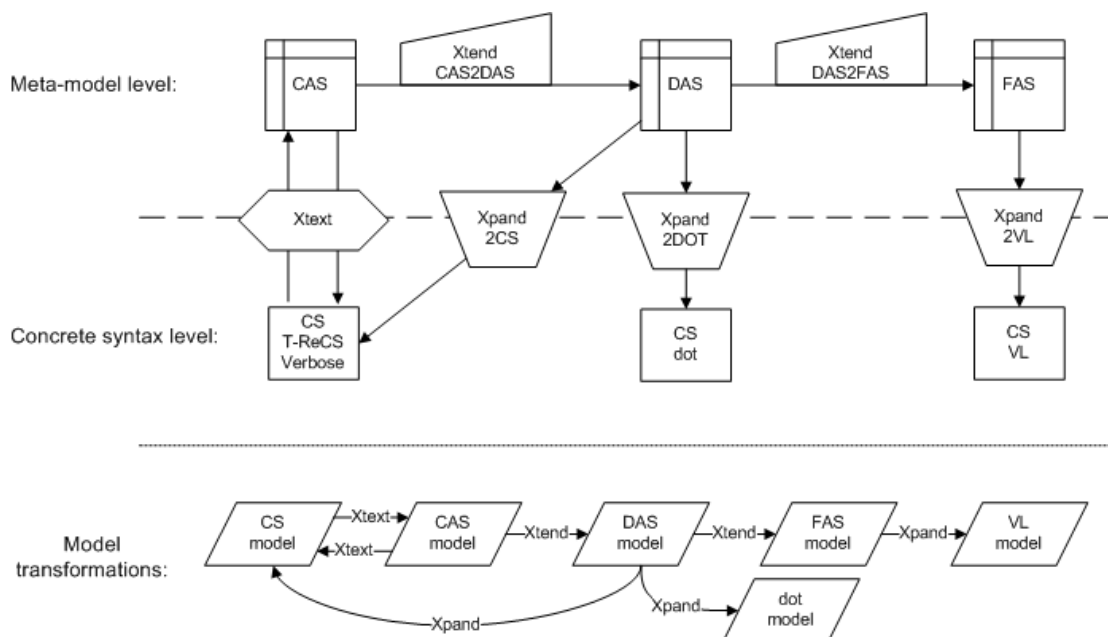


Figure 7.1: Final model transformation flow

7.2 Recommendations

Based on the conclusions, a number of recommendations can be formulated to improve the MDEE and the language engineering approach. The recommendations can be grouped by the different chapters of the thesis.

7.2.1 Abstract and concrete syntax design

Testing the concrete syntax with users

The verbose concrete syntax has been designed with care, based on input from domain experts and general design principles. However, it has not been validated extensively with users. As Gould et al. describe it in their paper on usability [18], one of the key principles of design is to focus on the users and tasks of the system. In this case, only the domain experts have been involved in the design of the concrete syntax and not the language users.

Therefore, it is recommended to evaluate the MDEE with language users, to gather suggestions for improving the concrete syntax. Based on the evaluation, the concrete syntax should be adapted or a new concrete syntax should be made. As part of this project, an interview plan was created which could be used as a starting point for such an evaluation.

Combining namespace scoping with multiple files

The current editor implementation uses a scoping mechanism that allows to reference objects by their fully qualified name, for example ‘coffee.make_coffee_subplan.make_coffee’. Another scoping mechanism allows for importing elements from other files. For the current Xtext distribution, it is technically very hard to implement *both* mechanisms. In fact, for these technical reasons, we were not able to do this.

However, such a scoping implementation could be very beneficial for using the language. In industrial projects, typically many developers work on the same configuration. To avoid merger conflicts, it may be very desirable to be able to split the model over various files. It is desired that this is done by import statements at the beginning of a file. Besides this, it is also required that elements can be imported by their fully qualified name, to avoid importing unnecessary elements. The concrete syntax is designed in such a way that this is possible (it is very compositional).

A recommendation for imports would be to do some research on the possibilities of Xtext

with regard to scoping and importing, to see whether the required functionality is at all possible. A way to combine the import-URI mechanism with namespace scoping is described by Meinte Boersma [19], which suggests a way to write your own *ScopingFragment* to combine both approaches. Figure 7.2 shows the Java code to do this. However, in Xtext 1.1, we were not able to get this code to work, so it may be better to wait for the 2.0 release of Xtext, which should have better scoping support.

```
13 public class ThomasScopingFragment extends AbstractScopingFragment {
14
15     @Override
16     protected Class<? extends IScopeProvider> getLocalScopeProvider() {
17         return isIgnoreCase()
18             ? IgnoreCaseImportedNamespaceAwareScopeProvider.class
19             : ImportedNamespaceAwareLocalScopeProvider.class;
20     }
21     @Override
22     protected Class<? extends IGlobalScopeProvider> getGlobalScopeProvider() {
23         return isIgnoreCase()
24             ? IgnoreCaseImportUriGlobalScopeProvider.class
25             : ImportUriGlobalScopeProvider.class;
26     }
27
28 }
```

Figure 7.2: Java scoping fragment to combine two scoping mechanisms

7.2.2 Model driven engineering environment design

With respect to the MDEE, some recommendations can be made with respect to the editor generated by Xtext, the model to model transformations in Xtend and the model to text transformations in Xpand.

Editor aspects

For the prototype of RaPTR, the basic editor features have been implemented that are needed to get a usable editor. However, a large amount of customization could be done to improve it further. Additional validation rules could be provided, to support the language user even more during development. Furthermore, some quick fixes could be provided, to provide easy solutions for many common errors.

A downside of these customizations is that they might influence the performance of the editor. Furthermore, it needs to be tested what the result on performance is once the editor needs to process larger models than the example models.

Model to model transformations

During the model to model transformations, information from the source model can be extracted, but new information can also be generated when there is no input. A useful feature for this is to allow less restrictions in the concrete syntax than in the abstract syntax. An example of this is the optional resource number of capabilities: the Xtend transformation adds this information if it is not provided.

With these ‘default’ transformations, models can be written much faster since less information needs to be explicitly modeled. It will make models much more concise. However, when using this approach, an apparent risk is that language users might not be aware of these implicit default values. If they simply forget to enter a value when it *is* necessary, the model to model transformation does not warn the language user of this fact. Therefore, these implicit generation assumptions need to be well documented.

Model to text transformations

Model to text transformations are relatively easy to define, so a number of them could be written for various purposes. The most important one is the transformation from DAS to C-code, as this will allow the language to be used in the engineering process of ASML. Other useful transformations could be documentation generation or the generation of code metrics.

Maintenance of the current Xpand templates is not very hard. The largest model to text transformation is the DAS to concrete syntax transformation, which is also the most complex one. It is the only transformation which takes layouting into account, which is necessary because the concrete syntax should be human readable (which is not an issue for the formal verification language and the dot representation).

7.2.3 Validation

Multiple concrete syntaxes

The concrete syntax added in the validation chapter was a *complete* concrete syntax, which can describe every aspect of the language. A simpler approach is to create a *partial* concrete syntax, which focus on a particular part of the abstract syntax. These partial concrete syntaxes would all transform into part of a DAS model. The only thing that needs to be added to work with partial concrete syntaxes is an Xtend transformation which merges the models at the DAS level, to make sure no information is lost.

Using multiple partial concrete syntaxes has several advantages:

1. Reuse: Common parts of the language may occur in other domains, such as state machines. If within ASML there is a common concrete syntax for state machines, it can be connected to multiple abstract syntaxes.
2. Combining graphical and textual concrete syntaxes: The Graphical Modeling Framework (GMF), which is also part of EMF, allows for creating a graphical syntax for Ecore meta-models.
3. Creating multiple ‘views’: You can generate multiple concrete syntax representations for the same part of the language, depending on the preferred viewpoint.

The Xtend merger transformation can be very simple, but also very complex. In the most simple case, the models in the DAS are simply overwritten with the data provided by the partial models, while keeping the data which is not present in the partial model. In the difficult case, an extensive version-control mechanism can be written as part of the Xtend transformation. In essence, this would allow to merge partial models that have changed the same part of the model.

7.3 Summary

In this chapter, an answer to the research question has been provided, by discussing the various chapters of the thesis. Recommendations for extensions of the MDEE have been made, including suggestions for future use. The recommended model flow with the use of multiple concrete syntaxes is depicted in Figure 7.3.

Bibliography

- [1] J. M. Nieuwelaar, *Supervisory machine control by predictive-reactive scheduling*. 2004.
- [2] “www.asml.com.” ASML website, 2010.
- [3] J. Rooda, “Supervisory machine control.” Slides of course 4K420, Eindhoven University of Technology, 2010.
- [4] K. Slonneger and B. Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1995.
- [5] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM Comput. Surv.*, vol. 37, pp. 316–344, December 2005.
- [6] M. Fowler, *Domain Specific Languages*. Addison-Wesley Professional, 1st ed., 2010.
- [7] A. Kleppe, *Software Language Engineering: Creating Domain-specific Languages Using Metamodels*. Upper Saddle River, NJ: Addison-Wesley, 2009.
- [8] S. Kent, “Model driven engineering,” in *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, (London, UK), pp. 286–298, Springer-Verlag, 2002.
- [9] M. Fowler, “martinfowler.com/articles/languageworkbench.html.” website, June 2005.
- [10] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Boston, MA: Addison-Wesley, 2. ed., 2009.
- [11] “http://openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html.” Xtend and Xpand manual, 2008.

- [12] “www.eclipse.org/xtext/documentation/latest/xtext.html.” Xtext manual, 2010.
- [13] S. Mauw, W. Wiersma, and T. Willemse, “Language driven system design,” *Hawaii International Conference on System Sciences*, vol. 9, p. 280b, 2002.
- [14] M. F. Amstel, M. G. Brand, Z. Protić, and T. Verhoeff, “Transforming process algebra models into uml state machines: Bridging a semantic gap?,” in *Proceedings of the 1st international conference on Theory and Practice of Model Transformations, ICMT '08*, (Berlin, Heidelberg), pp. 61–75, Springer-Verlag, 2008.
- [15] H. Groenniger, H. Krahn, B. Rumpe, M. Schindler, and S. Volkel, “Text-based modeling,” *ATEM*, 2007.
- [16] “<http://code.google.com/p/google-guice/>.” Google guice.
- [17] “<http://www.graphviz.org/doc/info/lang.html>.” dot language manual.
- [18] J. D. Gould and C. Lewis, “Designing for usability, key principles and what designers think,” in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems, CHI '83*, (New York, NY, USA), pp. 50–53, ACM, 1983.
- [19] M. Boersma, “Blog: Configuring local and global scope providers at the same time.” website: <http://dslmeinte.wordpress.com/2010/09/02/configuring-local-and-global-scope-providers-at-the-same-time/>.