UNIVERSITY OF SOUTHAMPTON

Extension in Domain Specific Code Generation with Meta-Model Based Aspect Weaving

by

Meng Tian

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Physical Sciences and Engineering
Electronics and Computer Science

May 2016

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Meng Tian

Domain specific code generation improves software productivity and reliability. However, these advantages are lost if the generated code needs to be manually modified or adapted before deployment. Thus, the systematic extensibility of domain specific code generation becomes increasingly important to ensure that these advantages are maintained. However, the traditional extension approaches, like round-trip engineering, have their limitations in supporting certain code customization scenarios. In this thesis, we address this problem with aspect-oriented techniques. We first show that the meta-model and the code generator can be used to derive a domain specific aspect language whose join points are based on domain specific elements. We then show that a corresponding aspect weaver can be derived as well, provided a proper model tracing facility can be made available for the code generator. We demonstrate the viability of our approach on several concrete domain specific code generation case studies, respectively with the AUTOFILTER code generator, the ANTLR parser generator, and the CUP parser generator. We successfully construct a few Java program analysis tools as a result of these case studies.

Contents

List of Figures

Listings

Academic Thesis: Declaration Of Authorship

I, Meng Tian , declare that this thesis entitled, “Extension in Domain Specific Code Generation with Meta-Model Based Aspect Weaving”, and the work presented in it are my own.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University.
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
3. Where I have consulted the published work of others, this is always clearly attributed.
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
5. I have acknowledged all main sources of help.
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

Acknowledgements

I owe my deepest gratitude to Dr Bernd Fischer and Dr Julian Rathke for their enlightening ideas and patient supervision. Thanks also go to my learned colleagues Teresa Binks, Tristan Aubrey-Jones, Neville Grech, Jeremy Morse, Owen Stephens, and Seyed Hossein Ghotbi for their patient help on my naive questions.

Nomenclature

ANTLR	ANother Tool for Language Recognition
AOM	Aspect-Oriented Modelling
AOP	Aspect-Oriented Programming
AST	Abstract Syntax Tree
BYACC	Berkeley YACC
CFG	Context Free Grammar
CFL	Context Free Language
CUP	Constructor of Useful Parsers
DFA	Deterministic Finite Automata
DFS	Depth First Search
DFSM	Deterministic Finite State Machine
DMM	Direct Manual Modification
DSAL	Domain Specific Aspect Language
DSCG	Domain Specific Code Generation
DSES	Domain Specific Extension Statement
DSJPM	Domain Specific Join Point Model
DSL	Domain Specific Language
DSML	Domain Specific Modelling Language
DSM	Domain Specific Modelling
DSP	Domain Specific Procedure
EBNF	Extended Backus-Naur Form
GP	Generalized Procedure
GPAL	General Purpose Aspect Language
GPPL	General Purpose Programming Language
JPM	Join Point Model
KFAL	Kalman Filter Aspect Language
LALR	Look Ahead LR
LHS	Left Hand Side
LL	Left to right, Leftmost derivation (parsing)
LOC	Lines Of Code
LR	Left to right, Rightmost derivation (parsing)
LTL	Linear Temporal Logic

MBSD	Model Based Software Development
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MOF	Meta Object Facility
MRTE	Model Round Trip Engineering
ODM	Ontology Definition Meta-model
OMG	Object Management Group
PCD	PointCut Descriptor
RHS	Right Hand Side
SDF	Syntax Definition Formalism
SLOC	Source Lines Of Code
SLR	Simple LR (parsing)
SP	Specialized Procedure
XSD	XML Schema Definition
XSLT	eXtensible Stylesheet Language Transformations
YACC	Yet Another Compiler Compiler

Chapter 1

Introduction

Model based software engineering approaches, such as Model Driven Engineering (MDE) [1, 2], have raised the abstraction level at which engineers develop software from code to models in specific domains, and have brought about many advantages in productivity and reliability over traditional software development [3]. These advantages mainly come from the use of automatic code generation techniques, which enable the systematic and reliable transformation of models into code and provide a "correct-by-construction" guarantee [4]. This allows domain experts to ignore implementation details and to focus on the more abstract models by automatically bridging the gap between different abstraction levels. Different abstraction gaps lead to different applications of code generation. For instance, a code generator that maps between a code template and its instantiated code blocks can be used as a background supporting tool for the corresponding wizard of an IDE. Although code generation techniques can be utilized in a broad range of applications, our discussion about code generation in this thesis will focus on its application in model based software engineering approaches, in particular, the code generation from domain specific models.

Domain Specific Code Generation (DSCG), as a specialized model based code generation approach, provides an automatic mapping between domain specific models and their corresponding implementation. Compared with models that are built in a general purpose modelling language, domain specific models are normally more effective and less complicated [5]. This is because they are built with the elements that are specifically used in the target domain, which would otherwise need to be built from scratch with more general and basic elements. With the help of DSCG, domain experts can build models directly with domain specific concepts that they are familiar with. Therefore, DSCG has become increasingly popular over the last decade. In DSCG, code generators encapsulate the details of the mapping between the domain specific models and the generated code. These details are normally based on the best practice in similar code generation scenarios. As a result, DSCG not only increases the productivity of software developers significantly, but also further raises the quality of the generated code [3].

On the other hand, DSCG also has its limitations. In order to maintain the "correct-by-construction" guarantee, the code that is automatically generated from DSCG is, in general, expected to remain unchanged once it is generated. However, there might be a multitude of different reasons to introduce modifications to the generated code to make it work well. For instance, the exposed interface of a component needs to adjust due to a recent change of a dependent library, or the database communication layer needs to be modified since a database migration has just been launched on a user's demand. These modifications may vary greatly in the scope or the complexity of the affected code, for example, from renaming a single variable to a complete refactoring of the framework of the code, or even to migrating to a new implementation platform. Figure 1.1 shows a general overview of the scenario.

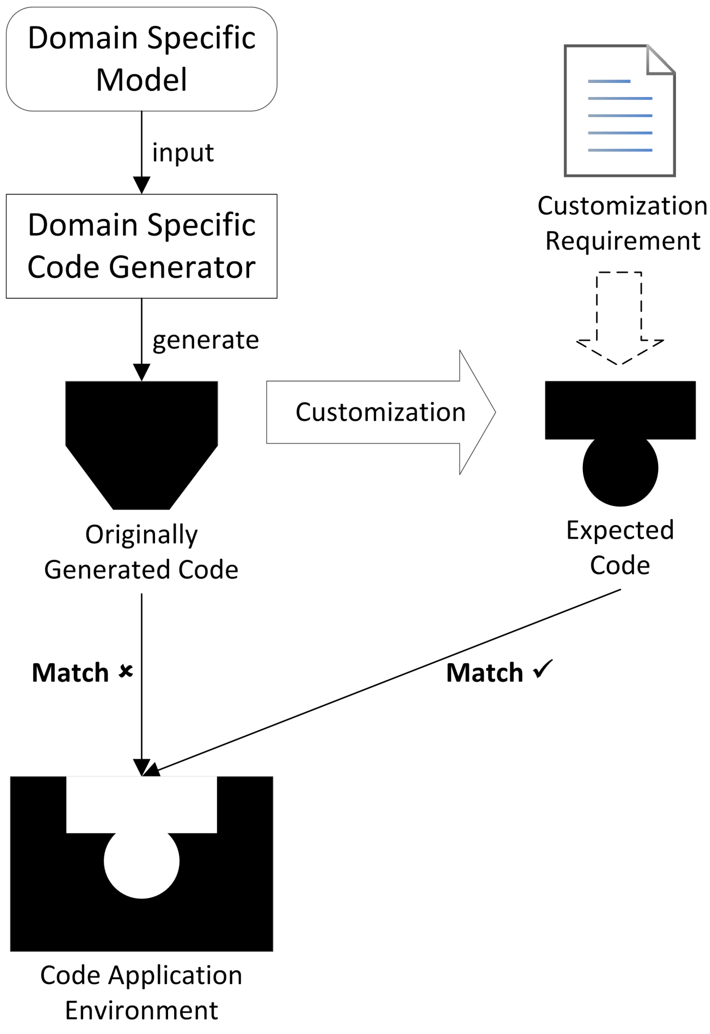

Domain Specific Model
input
Domain Specific Code Generator
generate
Originally Generated Code
Customization
Customization Requirement
Expected Code
Match ✗
Match ✓
Code Application Environment

FIGURE 1.1: Generated code often needs customization before deployment

However, if this kind of arbitrary modification gets directly introduced to the generated code, no matter how trivial the modification may be, it will break the synchronization previously maintained between the domain specific model and its generated code. The "correct-by-construction" guarantee will disappear due to the syntactic or semantic alteration of the generated code. The quality of the modified code will thus be uncertain, unless the modification can be somehow verified or validated. In addition, DSCG is often used in complex system development, which could make the modification time-consuming and error-prone. For instance, if a widely used interface of an internal library in a complicated system gets changed, the modification to the generated code would entail updates in many scattered code segments. Unfortunately, these modifications are sometimes necessary and inevitable. We cannot simply prohibit all modifications to the generated code.

In order to support the full software development life-cycle and to accommodate unforeseen feature requests or changes to the existing system, the code generation process needs to be able to adapt systematically and efficiently. Researchers have addressed different extension approaches [3, 6], which can be categorized into two groups: the pure model based philosophy and model round-trip engineering approach. Unfortunately, both of them have their limitations in supporting different DSCG extension scenarios.

In a pure model based philosophy, models are regarded as first class citizens, and code is considered as a subsidiary artifact that can be easily generated at any point. Therefore, the system evolution process follows a "model change ⇒ code re-generation" style as shown in Figure 1.2. In principle, this style is elegant and well supported. In practice, unfortunately, it does not always work well, because it requires the models to be detailed enough to reflect the changes, while many changes (e.g. adding logging functionality or code refactorings) can be very difficult to express in the model level. A natural remedy for this is to enrich the domain modelling language, so that it can be more expressive and able to describe supplementary details in models. However, this can lead to the pollution of the model with concepts in a lower abstraction level and, eventually, to a breakdown of the abstraction hierarchy. Moreover, a pure model based approach requires meta-model updates, which are supported much less well than model updates, and are sometimes even impossible. For example, updating a meta-model is not appropriate if one cannot update the code generator to account for the meta-model updates. Besides, such updates can lead to incompatibilities between different meta-model versions.

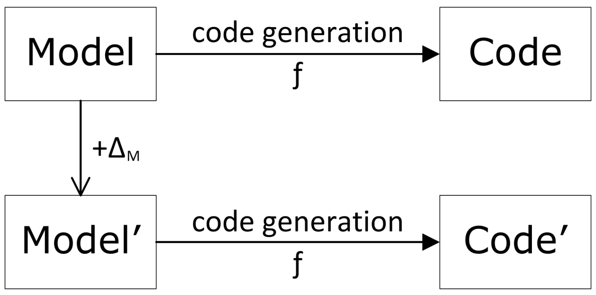

$$\begin{tikzcd}
\text{Model} \arrow[r, "\text{code generation}", "f"'] \arrow[d, "+\Delta_M"] & \text{Code} \\
\text{Model}' \arrow[r, "\text{code generation}", "f"'] & \text{Code}'
\end{tikzcd}$$

FIGURE 1.2: System Evolution in A Pure Model Based Philosophy.

The other approach is to directly modify the generated code via some cost effective solutions, and then rebuild the synchronization between the modified code and the model after the code modification. This approach is called the *Model Round Trip Engineering* (MRTE) [7, 8] approach. The basic idea is to rebuild the synchronization by enabling a reverse engineering process that derives an evolved domain specific model from the modified code as shown in Figure 1.3. This approach implicitly requires that it is always possible to find or derive a reverse process of the target code generation, so that a bidirectional (ideally bijective) mapping between domain specific models and code can be established. Unfortunately, in practice the domain specific code generation is a general, often non-injective, transformation of domain specific models into code.

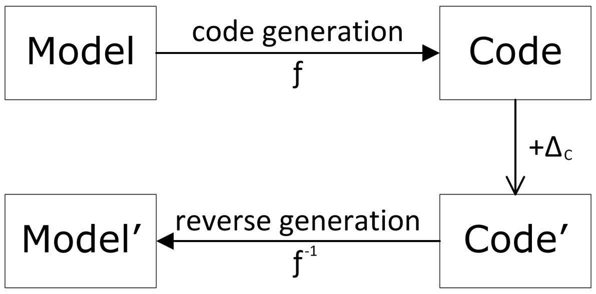


FIGURE 1.3: System Evolution by MRTE.

On the other hand, MRTE was originally introduced to address model update problems in model-to-model transformations, where artifacts on each side are equally important core products. However, our scenario is model to code. As such, using MRTE places an important assumption on the code, i.e. considering the code as a model as well. However, code needs to deal with implementation details, which are exactly what we try to conceal from models. Figure 1.4 illustrates the transfer of information in DSCG process. We can see that the information finally conveyed to the generated code comes from two sources: the model and the code generator. When we change the code, if the information involved comes from the model, we want to propagate the change to the model level. Otherwise, if the information comes from the code generator, we do not pollute the model with it. The MRTE DSCG extension approach enforces that the code change is always propagated to the model side.

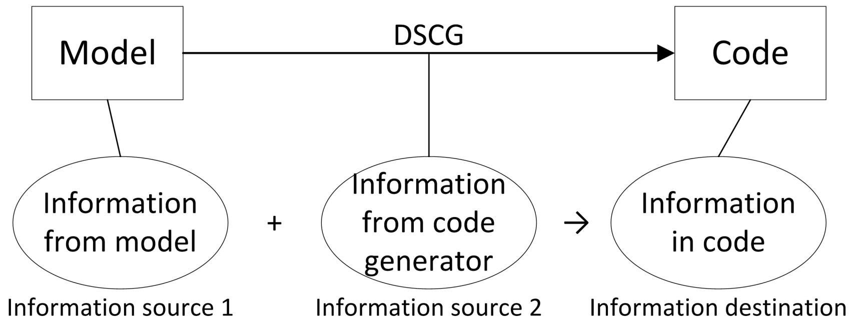


FIGURE 1.4: Transfer of Information in DSCG.

1.1 Problem Statement

In this section, we list the limitations and drawbacks that we observed in the existing DSCG extension approaches and identify the problems that we are tackling.

Observation 1: It can be very difficult to describe the desired modifications in terms of the domain specific models, if at all possible.

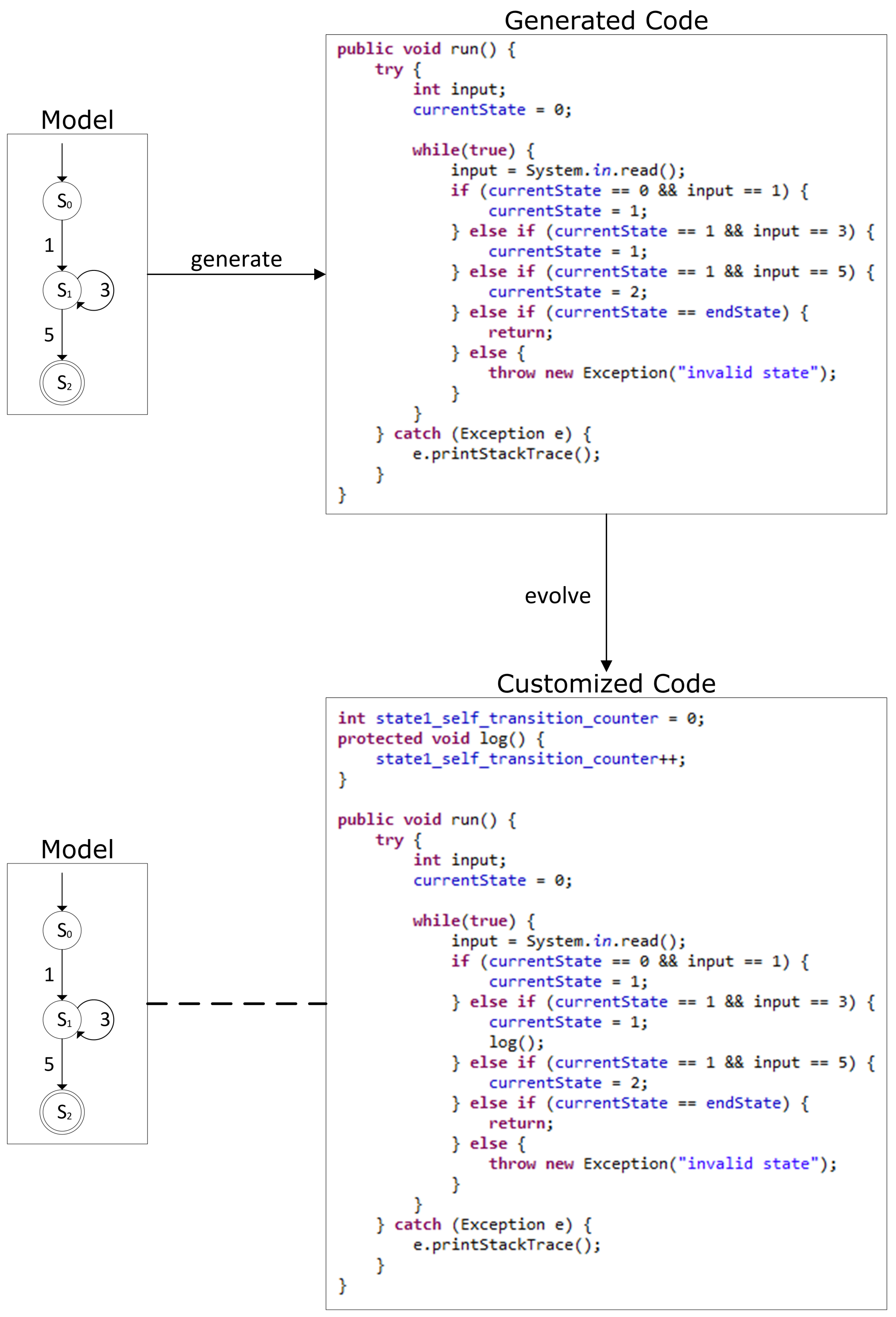
As mentioned before, the modification requirements are introduced in arbitrary and unforeseen ways. It is very likely that sometimes the modification descriptions are beyond the expressiveness of the original modelling language. This will lead to the failure to construct a new domain specific model to reflect the modification.

Figure 1.5 illustrates a simple example where the expected modification cannot be reflected in the domain specific model. The domain specific model is a deterministic finite automaton with three states: S_0, S_1 and S_2. S_0 is the start state. S_2 is the end (or accept) state. There are three transitions among them. Each transition fires when a specific character is read from the input stream. Apparently, this automaton accepts digit string in the format of "`1(3)*5`".

Let us assume that the current modification requirement is to supplement a logging method and invoke it after the self-transition of S_1 fires to count the number of digit "3" in the input string. The modelling domain here is the deterministic automaton construction and execution, where logging is generally not regarded as a domain element. In other words, the domain specific modelling language is "too abstract" to reflect the expected modification. In addition, this scenario also makes a solution based on MRTE [9] quite difficult to implement, as the change to the code cannot be translated into a corresponding exact change to the domain specific model. This kind of change is regarded as "invalid" by Thomas et al. [8]. Even if we managed to establish a round-trip transformation, i.e. the reverse code generation process, the new model that is rebuilt from the modified code is very likely to be identical to the original one.

Observation 2: Crosscutting concerns in the expected changes make the modification process inefficient and error-prone.

In the expected changes to a software system, some concerns are closely coupled with "primary functionality of the system", or "core concerns" [10], and thus cannot be effectively expressed in a modular way using traditional development approaches. For example, logging or asserting functions, often have to crosscut the whole software system. Returning to the above state machine example, let us assume the change we need is to log every transition. If the logging function can be added directly into the model, it entails



Generated Code

```java
public void run() {
    try {
        int input;
        currentState = 0;

        while(true) {
            input = System.in.read();
            if (currentState == 0 && input == 1) {
                currentState = 1;
            } else if (currentState == 1 && input == 3) {
                currentState = 1;
            } else if (currentState == 1 && input == 5) {
                currentState = 2;
            } else if (currentState == endState) {
                return;
            } else {
                throw new Exception("invalid state");
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Customized Code

```java
int state1_self_transition_counter = 0;
protected void log() {
    state1_self_transition_counter++;
}

public void run() {
    try {
        int input;
        currentState = 0;

        while(true) {
            input = System.in.read();
            if (currentState == 0 && input == 1) {
                currentState = 1;
            } else if (currentState == 1 && input == 3) {
                currentState = 1;
                log();
            } else if (currentState == 1 && input == 5) {
                currentState = 2;
            } else if (currentState == endState) {
                return;
            } else {
                throw new Exception("invalid state");
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

FIGURE 1.5: A manual DSCG extension where the change cannot be reflected in the domain specific model.

manual update of every transition in it. If not, it requires modifying every conditional branch inside the "while" loop (except the final exception raising branch). In both cases, the required change leads to manual changes that are scattered in the system. It is obvious that when the expected change involves such crosscutting concerns, no matter whether they can be described in the model level or not, they have to be introduced separately in different places of the system, which often contain very similar or even the same modification. This makes the modification process rather inefficient, and such duplication may easily lead to errors.

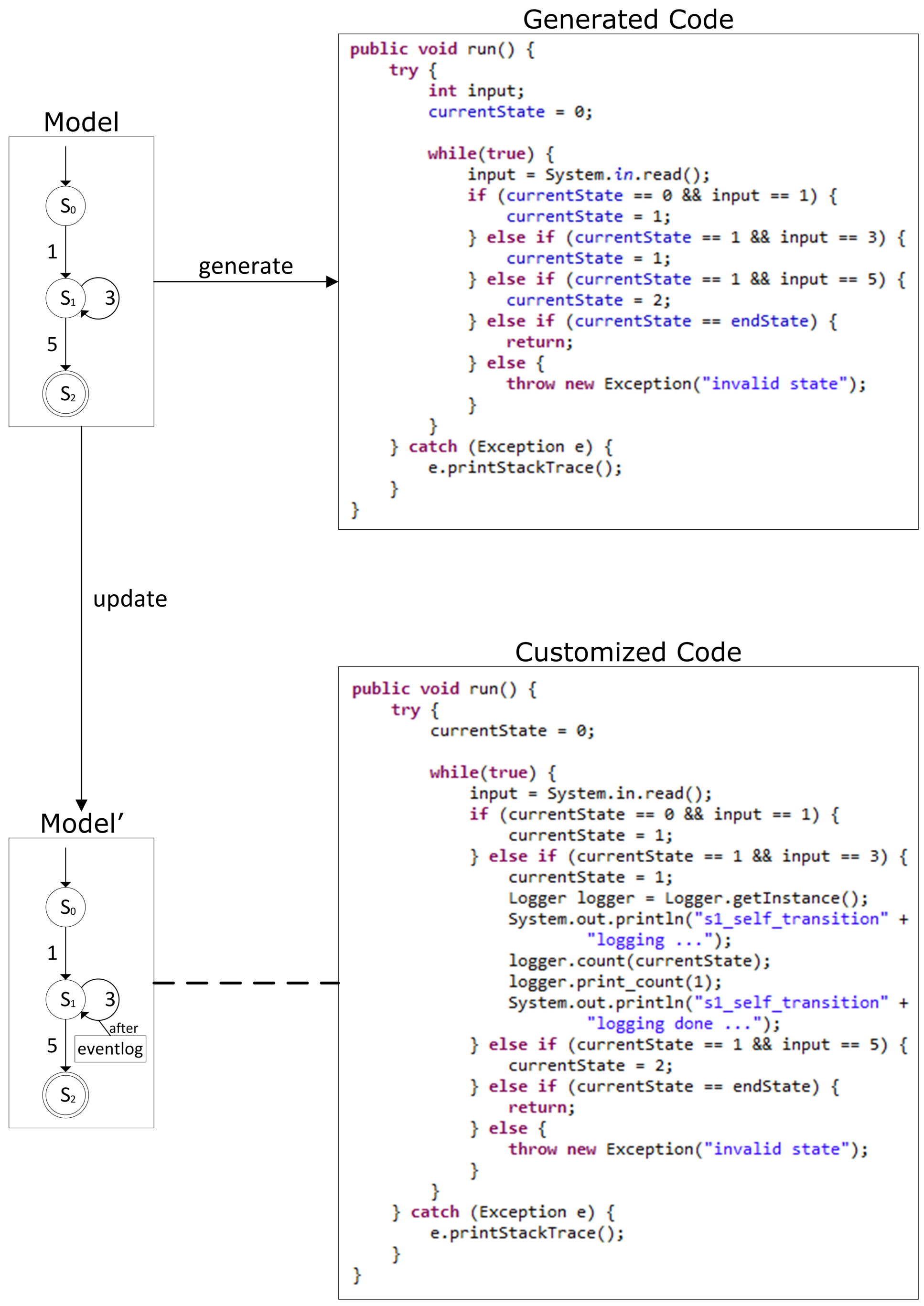
Observation 3: The existing DSCG extension approaches always seek to maintain a direct mapping between domain specific models and code.

An obvious common point of the above solutions via existing approaches is that they are both based on a direct mapping between model and code. Whenever a valid model is created, there exists a version of code that could be generated from it. As for the pure model based approach, it introduces the restriction that code can only be produced through model based code generation. Therefore, whenever there is a version of code generated, there is a model used as the DSCG input in the first place. In the MRTE approach, if the code is generated through code generation, there is a corresponding model involved to produce it. If not, there will be a corresponding model computed by the reverse engineering. In brief, the existing approaches bind model and code with a direct mapping, so that they would update in lock-step. No matter whether the model model' is directly modified with Δ_{M} in the pure model based approach, or model' is computed from Δ_{C} in the MRTE approach, the extension is always achieved by a concomitant update of both model and code. In other words, a change introduced from either model or code side is always propagated to the other side. As a result, an extension via existing approaches always leads to an update of the domain specific model.

Taking into account the observations above, we list the following problems in the existing DSCG extension approaches.

Problem 1: The existing approaches may lead to a breakdown of the abstraction hierarchy.

In Observation 1, we show an example where the expected change cannot be reflected in the domain specific model. But it is not a scenario where the existing approaches cannot accomplish the extension requirement. In fact, the pure model based approach can provide a working solution. However, there is a serious problem in this solution. As shown in Figure 1.6, a solution following the pure model based approach first extends the domain specific modelling language with a new domain concept, i.e. eventlog, which is bound either before or after a certain transition. It then updates the model by adding an instance of eventlog after the self-transition of S_1. Finally, the updated model is used to regenerate the code. The problem here is that such a solution pollutes the model with an eventlog concept, which is commonly seen in the less abstract level (code level), rather than in the deterministic finite automaton model level.



```
public void run() {
    try {
        int input;
        currentState = 0;

        while(true) {
            input = System.in.read();
            if (currentState == 0 && input == 1) {
                currentState = 1;
            } else if (currentState == 1 && input == 3) {
                currentState = 1;
            } else if (currentState == 1 && input == 5) {
                currentState = 2;
            } else if (currentState == endState) {
                return;
            } else {
                throw new Exception("invalid state");
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
public void run() {
    try {
        currentState = 0;

        while(true) {
            input = System.in.read();
            if (currentState == 0 && input == 1) {
                currentState = 1;
            } else if (currentState == 1 && input == 3) {
                currentState = 1;
                Logger logger = Logger.getInstance();
                System.out.println("s1_self_transition" +
                        "logging ...");
                logger.count(currentState);
                logger.print_count(1);
                System.out.println("s1_self_transition" +
                        "logging done ...");
            } else if (currentState == 1 && input == 5) {
                currentState = 2;
            } else if (currentState == endState) {
                return;
            } else {
                throw new Exception("invalid state");
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

FIGURE 1.6: An extended solution following the pure model based approach in the deterministic finite automaton example.

Problem 2: The existing approaches implicitly demand that the relationship between model and code is a surjective function. This deviates from the original intention of model based engineering.

Apart from the risk of breaking down the abstraction hierarchy, there is even a deeper problem in the solutions via the existing approaches. As mentioned in Observation 2, the existing approaches bind model and code with a direct mapping throughout the software maintenance life-cycle. More formally, they define a function from domain specific models to code. In other words, for each valid domain specific model, there exist one and only one version of code, as the same model always generates the same version of code once the code generator is specific. In particular, this function is surjective, as for each version of code, there exists at least one domain specific model. In the pure model based approach, this restriction is intrinsic as code can only be generated through DSCG process. In the MRTE approach, it is also required. Because the round-trip would be cut if there exists a certain version of code, for which a corresponding domain specific model cannot be found. In fact, the restriction in the MRTE approach is even more strict. If the function is non-injective, different domain specific models can generate the same version of code, which would make it impossible to establish a reverse function from code to model to complete the round-trip. Therefore, the function from model to code in the MRTE approach needs to be not only surjective, but also injective, i.e. bijective.

All above restrictions on the relationship between model and code may seem inevitable. However, they are not required by the nature of model based engineering. The defining characteristic of model based engineering is that the primary focus and product in traditional software development, programs, are superseded by models [11]. The models, as core products, should not be imperatively bound with some non-core products, like code. The relationship between model and code does not necessarily need to be a function at all. As an example, consider a refactoring of some generated code C to C'. A loose analogy of the relationship between model and code in model based engineering is the relationship between code and machine code in traditional software engineering. In model based engineering, a model can be related to multiple versions of code, just like a version of high-level programming language code can be compiled into different machine code on different platforms in traditional software engineering, where the code is regarded as core product. In short, the restrictions to keep model and code in stringent synchronization deviates from the original intention of model based engineering.

Problem 3: The existing approaches may lose track of the software when the introduced change cannot be reflected in the model.

Version control, as Yuehua et al. [12] argued, is a vital technique towards improving quality and maintainability in the latter stages of the model based software life-cycle. The tracking of model based software is naturally implemented by tracking the versions of the model. Unfortunately, the existing approaches cannot support it well in certain scenarios. From the above discussion, we may find that both the pure model based approach and the MRTE approach can be inadequate when the expected changes cannot be reflected in the models. In fact, even if we somehow managed to implement a solution to introduce such a change to the code, it might hinder the proper tracking of the software. In the pure model based approach, the changes have to be introduced by extending the modelling language first, and then translating the model from the original modelling language to the extended modelling language, and finally updating the translated model according to the expected change. Obviously, such kind of model updates with a prerequisite of modelling language change cannot be tracked the same way as the common model updates. A common version reversing request may fail due to the model incompatibility problem.

1.2 Our Solution

Our primary research objective is to sustain the benefits of DSCG that may get lost during the introduction of changes that cannot be expressed by the original modelling language. To achieve this, we develop a systematic and partially automatic approach to extend the given DSCG. This research objective can be expounded as to develop a new DSCG extension approach, which does not affect the use of the other tools that conform to the modelling language of the given DSCG, to accommodate changes that cannot be described by the same modelling language.

Inspired by the aspect-oriented software development methods [13, 14], which naturally supports the direct manipulation of the code and the encapsulation of customization concerns, we propose our DSCG extension approach via domain specific aspect weaving. We first dynamically generate a domain specific aspect language according to the meta-model of the target domain, and then encapsulate the changes into domain specific aspects. Finally we weave these aspects to introduce the expected changes. Our approach allows domain experts, who have little knowledge of the generated code, to customize the code generation process conveniently by organizing their modification concerns in terms of aspects that written in the generated domain specific aspect language, which coordinates both domain concepts and code elements to describe the expected modifications in a flexible and concise way.

Our approach properly solves the problems we listed above respectively. First, when the models cannot reflect the expected changes, we can describe them in aspect, where elements of both the problem domain and the solution domain can be used, e.g. code level depiction is allowed. Thus, unnecessary modelling language updates can be effectively avoided, so that the risk of a breakdown of the abstract hierarchy can be reduced greatly. Second, the aspects can be regarded as an intermediate abstract level between model and code. A model can thus relate to multiple versions of code with the help of different aspects. Last, our approach clearly distinguishes between the software updates due to model change and the updates due to aspect weaving. This bidimensional version mechanism can support version control system well. Figure 1.7 illustrates the comparison between the existing approaches and our approach over the version control mechanism.

From Figure 1.7, we can observe that our DSCG extension approach is not designed as a replacement to the existing approaches. Instead, we attempt to introduce it as a powerful supplement to them. When the expected changes can be reflected in the models, domain experts can still introduce the change by directly modifying the models and launching the corresponding code regeneration. However, when the changes cannot be expressed by the current modelling languages, our domain specific aspect based DSCG extension approach becomes necessary. Besides, sometimes our approach can be more efficient to introduce the changes than the existing approaches provided all of them can do the job. The remainder of this thesis is organized as follows:

In Chapter 2, we discuss the background context related to this thesis. It mainly includes three areas: the model based software development, the domain specific code generation, and the aspect-oriented software development. In Chapter 3, we elaborate on our DSCG extension approach, and use a typical scenario to demonstrate how it can solve the target problem. In the next three chapters, we demonstrate the viability of our approach by experimenting on several concrete DSCG case studies, where we respectively implement an extension mechanism that allow us to introduce some common customizations in the different domains. In Chapter 4, we start with *Kalman Filter* domain as our first target domain, and extend the code generation by a specific generator called *AUTOFILTER* [15] that generates the C implementations of the Kalman Filter algorithm from high-level specifications. In Chapter 5, we move on to a more well-established parser generation domain, and work with a very popular parser generator, *ANTLR* [16], which accepts an LL(*) grammar specification of a certain language, and generates a parser of the language implemented in Java. In Chapter 6, we work with another widely used parser generator, *CUP* [17], which accepts an LALR(1) grammar specification of a certain language, and generates a parser of the language implemented in Java. In Chapter 7, we evaluate our approach by comparing it with the two existing approaches. Finally, we provide concluding remarks and potential future work in Chapter 8.

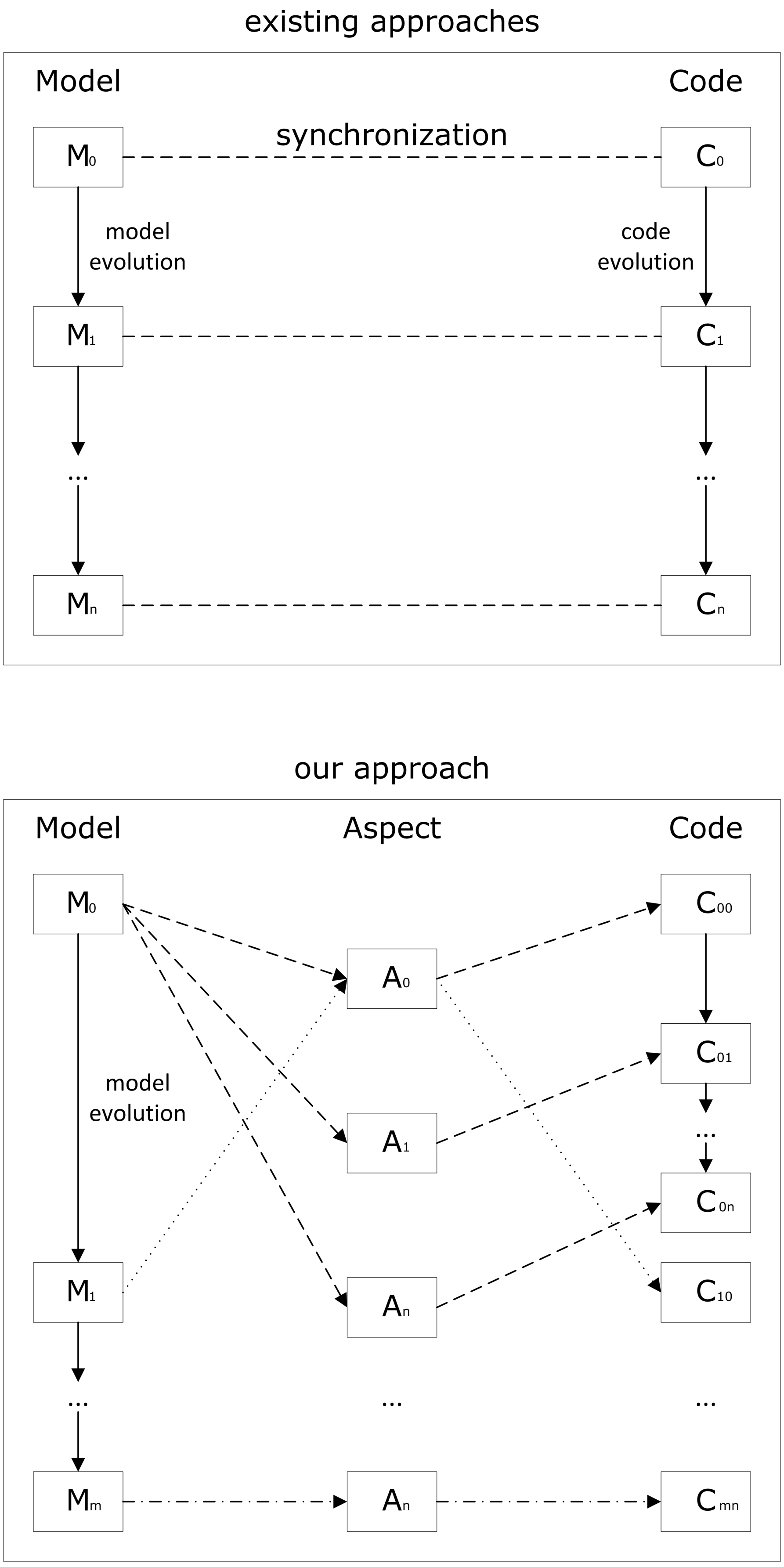


FIGURE 1.7: A comparison between the existing approaches and our approach over the relationship between the model and code, as well as the versioning mechanism.

Chapter 2

Background

In this chapter, we discuss the background of extending domain specific code generation and the related literature. We start with the background knowledge involved in domain specific code generation, which mainly about model based software development, such as domain specific modelling, etc. We then discuss the code generation techniques and model traceability in domain specific code generation. Finally, we introduce the aspect-oriented software development methods that directly inspires our work.

2.1 Model Based Software Development

The term *Model Based Software Development* (MBSD) subsumes a number of related software development approaches such as *Model Driven Engineering* (MDE) [2, 18], *Model Driven Development* (MDD) [19, 20], or *Model Driven Architecture* (MDA) [21]. These approaches are essentially much alike, but there are subtle differences between them. For example, MDD and MDE refer to the same software development approach, while MDA refers to a specific, architecture based implementation [22] of the approach, as defined by the *Object Management Group* (OMG).

2.1.1 MDD, MDE, and MDA

There are three concepts that are often referred to in different literature related to MBSD. Here we first provide a more detailed description of each concept, and then clarify our terminology in this thesis.

Model Driven Development *Model Driven Development* (MDD) [2, 19, 20, 23] is a common name for a family of software development approaches that raise the abstraction level of software development from programming language implementation to problem domain modelling. According to Atkinson and Kühne [19], MDD is an approach modelling the necessary functionality and system architecture, instead of spelling out detailed implementation in a programming language, with the aim of automating many of the complex (but routine) programming tasks. Later on, Hailpern and Tarr [20] provided a more detailed description of MDD. They described MDD as "a software engineering approach consisting of the application of models and model technologies to raise the level of abstraction at which developers create and evolve software, with the goal of both simplifying (making easier) and formalizing (standardizing, so that automation is possible) the various activities and tasks that comprise the software life-cycle".

Model Driven Engineering *Model Driven Engineering* (MDE) [1, 18] is a software engineering methodology that concentrates on model creation and exploitation instead of the traditional implementation in programming languages. According to Schmidt [18], MDE is "a promising approach to address platform complexity and the inability of third-generation languages to alleviate this complexity and express domain concepts effectively", in which there are a set of technologies that combine domain specific modelling languages, model transformation engines and code generators. Compared to this long definition, Kent [1] provided a much more brief explanation of MDE as "a methodology combining process and analysis with the MDA", where MDA is explained below.

From the above definitions and explanations we can see that MDD and MDE are much alike in definition, and the terms are used interchangeably in most situations. As we understand, the only difference between MDD and MDE is that sometimes the former is preferred in discussing software development paradigms and the latter is preferred in discussing software engineering methods.

Model Driven Architecture According to Mellor et al. [21], *Model Driven Architecture* (MDA) is a software development framework using models as descriptions of the system to be built. These descriptions "can be expressed at various levels of abstraction, with each level emphasizing certain aspects or viewpoints of the system." Later, Beydeda et al. [23] defined MDA with more detailed technical concerns. According to their definition, MDA is a software development architecture in which "we can define rules for automating many of the steps needed to convert one model representation to another, for tracing between model elements, and for analyzing important characteristics of the models".

On the other hand, the term MDA also refers to the specific standardized architecture formulated by the Object Management Group (OMG). In 2001, MDA was adopted by the OMG as a standard framework for model driven software development. According to OMG's MDA specification [22], MDA can be considered as a development process in which models are first class artifacts and are integrated through the chain of transformation from model to code. The latest official MDA specification version number is 1.0.1 [24].

From the above introduction, we can find that the differences between these model based approaches are very subtle in terms of the scope of our discussion. It is reasonable for us to ignore these differences and focus instead on their much larger commonalities. Rather than emphasizing any one of these approaches, in this thesis, we will use the neutral term *Model Based Development* (MBD) to refer to such commonalities.

The main goal of MBD is to raise the abstraction level of software development from programming language level implementation to problem domain modelling. Software development thus starts with the creation of an abstract model of the target system, which is then repeatedly refined and systematically transformed into a concrete implementation. Model based approaches combine a number of technologies. The models are formulated in one or possibly multiple modelling languages that conform to an (explicit) meta-model. The concretization is implemented via a series of model-to-model transformations, followed by a final model-to-code transformation step. Obviously, comparing to the traditional software development approaches, the most essential change in MBD is that models replace code as the core product to be designed, refined, and reused in the software development processes.

2.1.2 Models and Model Transformation

According to Beydeda et al. [23], models and modelling make up the basis of model based development. This argument conveys two fundamental ideas in MBD. First, models are the core products. Second, models need to evolve or develop to be gradually refined to the final software product. In this section, we will present a brief introduction of a few related concepts, including model, modelling dimensions, and model transformation.

Models No matter what technique or architecture is adopted in practice, the principal element of MBD is models. The term *model* here refers to a conceptual or abstract representation of something or some system in a specific domain. As a basis for subsequent development, models are a vital part of any MBD method. As Selic indicated [11], the defining characteristic of MDD is that programs, the primary focus and product in traditional software development, are superseded by models. This substantial change makes the design focus less bound to the underlying implementation and closer to the problem domain, which turns out to be such a significant raise in abstraction level that Selic believed that “MDD holds promise of being the first true generational leap in software development since the introduction of the compiler”.

The major difference between model and code is that they belong to different abstract levels. Models represent the design concerns with the elements or relationships in a the problem modelling domain, while code conveys detailed information about the implementation. According to Kleppe et al. [25], "a *model* is a description of (part of) a system written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer." The explanation of "well-defined language" here can be regarded as a concise definition of "modelling language."

Modelling Dimensions The modelling space can be structured in various dimensions of concerns, for example, security or reliability. A common dimension of concern is the abstraction level. In an MBD project, there should be one or more problem domain models derived from the requirements analysis in the beginning, which are constructed using only problem domain concepts. These models are transformed later into solution domain models, which are built with solution domain concepts. The abstraction level of these models thus forms a dimension of the modelling space. OMG's MDA standard defines two terms *Platform Independent Models* (PIMs) and *Platform Specific Models* (PSMs). As their names suggest, PIMs model platform independent concerns of the target system, while PSMs refine PIMs by including platform specific concerns in them. Kent [1] proposed that the MDA framework group models into PIMs and PSMs by categorizing perspectives. Laforcade et al. [26] refined this categorization by indicating a third model class in the MDA, *Computation Independent Models* (CIM). Kent addressed that several other dimensions of concerns have begun to be identified in the area of *Aspect Oriented Software Development* (AOSD) [27]. Kent also pointed out some seldom considered dimensions: abstraction/concretization, in regard with managerial and societal aspects, like authorship, version.

Model Transformation Model transformation plays a very important role in refining the models in MBD processes. According to Sendall [9], model transformation refers to the processes that "take one or more source models as input and produce one or more target models as output, following a set of transformation rules." Based on a taxonomy of the design features of model transformation approaches, Czarnecki and Helsen [28] categorize the transformation approaches into six categories: direct-manipulation, relational, graph-transformation-based, structure-driven, hybrid, and other. According to the definition of "well-defined language" and "model" we discussed in Section 2.1.2, we can infer that a programming language can be considered as a "well-defined language", and an implementation in a certain programming language can then be regarded a model. Hence, the model-to-code transformation, i.e. the code generation, is also a specialized kind of model transformation. Without special explanation, "model transformantion" in this thesisalways refers to "model-to-model" transformations. We will separately discuss code generation in Section 2.2.1.

2.1.3 Domain Specific Modelling

In order to describe a software system more precisely, developers often need to concentrate only on a specific area and its related entities, attributes, and relationships. In software engineering, these areas are often referred to as *domains*, and the related entities, attributes, and relationships are often referred to as *domain elements*. The increasing complexity and variety of different domains prevent general purpose modelling languages from building models precisely and concisely. It would be more convenient and cost effective if we could build models directly with the basic concepts in the specific domains. This leads to the notion of domain specific modelling. In simple terms, *domain specific modelling* (DSM) is a specialized modelling method based on domain specific knowledge. From the model dimensional view, all domain specific models are problem domain models. We did not find a standardized or generally accepted definition of DSM, as different people define it from different angles. According to Kelly and Tolvanen [3], DSM is an approach requiring three parts to form a development environment: a domain specific modelling language, a domain specific code generator, and a domain framework, while Choi et al. [29] define it as design representation with terms of parameters associated with a specific domain. As we understand, DSM is a software engineering methodology that systematically utilizes a domain specific modelling language to describe the domain ontology of the specific domain. Next, we explain several related concepts.

Domain According to Simos et al. [30], the term “domain” has two different meanings. First, a domain is the “real world” encapsulating the knowledge about a problem area, which excludes any detail about its solution or the corresponding software automating. Second, a domain is a set of systems. As Czarnecki and Eisenecker [31] argued, “in the software reuse community (and particularly in the field of Domain Engineering), the term ‘domain’ encompasses not only the ‘real world’ knowledge in a given problem area, but also the knowledge about how to build softwares in that area”.

Domain Ontology and Meta-Model The term *ontology* originally denotes the philosophical probing of existence. To our knowledge, it was first introduced into Computer Science by Gruber [32], referring to “a specification of a conceptualization.” An ontology is “a description of the concepts and relationships that can exist.” It is now widely used in Computer Science, especially in Artificial Intelligence and Domain Modelling, to describe the knowledge within a domain. According to Musen [33], models consist of the foundational domain concepts, and possibly the problem-solving procedures that might be applied to those concepts. These enumerations of the domain concepts and their relationships are referred to as *domain ontologies*. In MBD, the design models are transformed into various artifacts, all of which conceptually map to the abstract design models that are described in terms of domain ontology. To this extent, the domain ontology can be considered as the reference frame through the whole MBD procedure.

In Sergeevitch's paper [34], the domain ontology is the mechanism of creating continuous information field within the domain. It consists of two perception subjects, domain objects and the relation between these objects. According to Grüninger and Lee [35], an ontology is generally defined as a"formal explicit specification of shared conceptualization" or the "classifications of the existing concepts." A domain ontology can thus be considered as a formal representation of domain knowledge. In this thesis, we define domain ontology as a formal representation of the existing domain knowledge of a certain application domain, which mainly includes two parts: the specific elements in the domain, and the relations among these elements.

Simply stated, *meta-model* is a model of models, which conform to a specific domain ontology. They can be regarded as concrete descriptions of how models conforming to a certain domain ontology can be built, especially in terms of constructs and rules. From the perspective of graph theory, a meta model can be considered as a graph, in which nodes are classes and properties. In practice, domain specific modelling Languages often serve as the meta-models describing the corresponding domain ontologies. Interestingly, meta modellingitself can also be considered as a particular domain, which can be modeled within MDA. Djuric et al. [36] presented the *Ontology Definition Metamodel* (ODM) that is defined using the *Meta Object Facility* (MOF) [37] to promote the MDA standards in the ontological engineering.

Domain Specific Modelling Language A *Domain Specific Modelling Language* (DSML) is a formalized modelling language that is capable of specifying all parts or certain desirable facets of the domain ontology in a particular domain. Different from the general purpose modelling languages, like UML, DSMLs focus on a specific problem domains, offer higher abstraction levels that enable building models directly using the elements in the problem domains, and hence reduce design space.

Domain Specific Language *Domain Specific Languages* (DSLs) are sometimes identified with DSMLs. However, according to Deursen et al. [5], DSL is actually a different concept that is defined as "a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." In brief, the difference between DSML and DSL is that models built in DSML do not need to be executable while models in DSL need to be executable. In this thesis, we will not distinguish them, and will use them interchangeably.

To some extent, DSM can be considered as an extension of the general purpose modelling approach. It pushes the design context to the problem domain, which brings about the highest abstraction level that can be achieved from modelling. According to Booch et al. [38], the full value of MBD can only be achieved when the modelling concepts map directly to domain concepts rather than computer technology concepts. To this extent, the foundation of MBD is automation. Without automation, models would merely end up as documentation. Therefore, appropriate automation techniques are required in MBD to fulfill its promise.

2.2 Code Generation and Domain Specific Modelling

In this section, we first give a general introduction to code generation and its relevant techniques. Then we discuss how code generation works together with DSM technology to deliver the working software product in MBD.

2.2.1 Code Generation

Code generation [3, 39–41] has become a very popular topic in computer science research. However, there has been little agreement on a precise definition of the term "code generation." A major reason is that the meaning behind it has gradually changed since the notion was first proposed. Initially, the notion of code generation was used to refer to an internal program construction process in compilers for high-level programming languages [42]. Hence, the "code" here refers to the intermediate code or machine code. With the maturity of the compiler related code generation techniques and the boom of model driven development, code generation now refers to the source code generation process in MBD in increasingly more papers [3, 41]. In terms of the way that different code generation techniques assure the correctness of their generated code, code generation techniques can be grouped into two main categories: *deduction based code generation* and *transformation based code generation.*

Deduction Based Code Generation According to Stickel et al. [43], deduction based code generation (or "deductive program synthesis" [4]) is a code generation approach in which the code is "developed from the logical form of the specification by a deductive approach." In this approach, the code is generated as the byproduct of the proving process, which gives the basis of ensuring and demonstrating the correctness of the generated code. The structure of the generated code thus "reflects the proof from which it was extracted." Since the foundation of this code generation approach is the underlying theorem proving process, the code generated in this approach is guaranteed to be "correct-by-construction".

Transformation Based Code Generation Different from the "correct-by-construction" guarantee provided in the deduction based code generation approach, the correctness of the code that is generated as a direct product of certain transformation processes depends on the tools that actually conduct the transformation, i.e. the *code generator* [4]. A code generator takes in a model as input, puts it through certain transformation processes, and finally produces a corresponding program as output.Code generators may vary greatly in terms of the nature and strategy of the transformations they implement.

In the famous book "generative programming" by Czarnecki et al. [31], model-to-code transformation is categorized into *horizontal transformation*, *vertical transformation* and *oblique transformation.* A horizontal transformation operates on one level of abstraction and modifies the modular structure of a model. In contrast, a vertical transformation, or *forward refinement*, performs strictly hierarchical decomposition, preserves the modular structure of the higher level representation and implements it with one or more modules at the lower level. An oblique transformation does not only perform hierarchical decomposition, but also changes the modular structure of the higher-level representation. Figure 2.1 illustrates these three kinds of transformations.

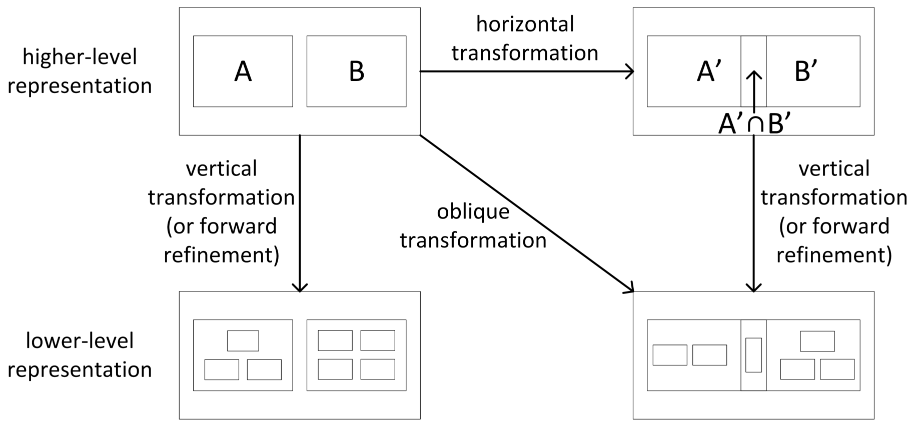

higher-level representation
A
B
horizontal transformation
A'
B'
A'∩B'
vertical transformation (or forward refinement)
oblique transformation
vertical transformation (or forward refinement)
lower-level representation

Figure 2.1: Transformations in code generation discussed in the book "Generative Programming: Methods, Techniques, and Applications Tutorial Abstract" by Czarnecki et al. [31]

Accordingly, Czarnecki et al. group code generators into two categories: *compositional generators* and *transformational generators*. Compositional generators only implement forward refinement, while transformational generators perform horizontal or oblique transformation. It is worth noting the "transformation" here does not refer to the general "model transformation". Instead, it specifically refers to the transformation in which the modular structure of the model at higher level of abstraction is breached. Most generators found in practice by then have a "predominantly compositional character" [31].

On the other hand, in terms of their implementation strategies, code generation can be categorized into two main kind: *template based code generation*, and *rewrite based code generation*.

- **Template Based Code Generation** As its name suggests, *template based code generation* is a code generation approach based on template expansion techniques. As Czarnecki and Helsen [28] argued, a template “usually consists of the target text containing splices of metacode to access information from the source and to perform code selection and iterative expansion.” A template resembles closely the code to be generated in terms of structure, and is independent of the target language. These features make this approach rather easy to implement. Nonetheless, a template is essentially a container of textual patterns, which are untyped. It inherently allows syntactic and/or semantical errors. Thus the correctness of this code generation approach relies on the correctness of the base templates and their expansion process. Further discussion about template based code generation can be found in the book by Cleaveland et al. [44].

- **Rewriting Based Code Generation** Pfenning and Elliott [45] indicated that *Abstract Syntax Trees* (AST) have been proved very useful in contexts like efficient and correct program transformation or inference rule application. AST is the core artifact in rewriting based code generation, which generally consists of three consecutive steps. First, the textual representation of the input model is transformed into an AST that conforms to a regular tree grammar. Second, a sequence of rewriting rules are applied to the AST. A rewriting rule can be described in the form $R : T1 \rightarrow T2 \; where \; C$, which means on the match of term *T1* when condition C is satisfied, a rule named R rewrites *T1* with new term *T2*. Finally, the modified AST, which represents the transformed model, is translated to the output programming language to complete the code generation process. According to the conclusion of a case study by Hemel et al. [46] on a typical rewriting based framework Stratego/XT [47], term rewriting has several advantages over template expansion in code generation, which include the ability to ensure syntactical correctness of generated code. However, such improvement relies on a correct grammar to construct the right AST in the first step. From this perspective, the correctness of this approach depends on the correctness of the AST construction and rewriting rule application.

2.2.2 Benefits of Code Generation

The benefits of code generation have been presented and discussed in many papers and books [3, 11, 41, 48]. Here we focus on two major benefits of code generation: productivity improvement and quality improvement.

Productivity Improvement The productivity improvements include the following aspects:

- **Lower Learning Cost** Since models hide many code level details that are often complicated and error-prone, software developers only need to master the domain knowledge at the model level, instead of having to understand model level domain knowledge as well as code level details and their mappings with domain elements.
- **Higher Development Efficiency** With the automatic code generation process, the cost of making design changes even in late stage of software engineering life-cycle is greatly reduced.
- **Better Maintainability** With code generation, developers no longer have to directly modify the code in order to introduce a design change. Instead, they only need to change the corresponding model (though sometimes a certain amount of coding is still needed), which is relatively simpler and less error-prone.

Quality Improvement The quality improvements include the following aspects:

- **Increased Reliability** Code generators often adopt existing best practice and specialized design patterns as the basis of the generated code. The adoption of these "built-in code templates" effectively avoids the tedious and error-prone manual coding process and increases the reliability of the generated code in return.
- **Improved Performance** Besides increased reliability, raised performance of the generated code is another benefit that is often gained from these "built-in code templates." It is because the templates often introduce optimized algorithms and specialized programming tricks to a certain problem, which greatly enhances the performance of the generated code.

However, the above benefits may get lost if direct manual code modifications are introduced after the code generation. That is exactly the main motivation for our work.

2.2.3 Domain Specific Code Generation

Now that we have presented a general introduction about domain specific modelling and code generation. In this section, we introduce the problem domain of our research, *domain specific code generation*. *Domain specific code generation* (DSCG) is a specialized code generation technique applied in domain specific modelling. It translates the domain specific models that are built by domain experts to the corresponding implementation of source code. In other words, the DSCG transforms the DSL description of a system in the problem domain into its corresponding representation in the solution domain, e.g. a Java implementation. Before further discussion, we need to clarify the meanings of several relevant terms that will be used in our discussion.

The Problem Domain Despite its vagueness and volatility [5], "problem domain" often serves as a defining term in DSM related definitions. According to Czarnecki and Eisenecker [49], the *problem domain* (or *problem space* in Czarnecki's terminology) "consists of the application-oriented concepts and features that application programmers would like to use to express their needs". As for the coordination of the problem domain and the solution domain, they explicitly define a concept called *configuration knowledge*, which maps abstract requirements onto appropriate configurations of solution components. According to their definition, the configuration knowledge consists of five parts: illegal feature combinations, default settings, default dependencies, construction rules, and optimizations. They believe such separation between the problem space and the solution space and the configuration knowledge helps in requesting concrete systems or components.

However, for the discussion under the context of DSCG, this separation does not often help. In DSCG, the input problem model includes the goals that the problem owner wishes to achieve, the context within which the problem exists, and all rules that define essential functions or other aspects of any solution product. In other words, the problem domain here represents the environment in which a solution will have to operate, as well as the problem itself. In this thesis, we define "problem domain" as all information that defines the problem and the constraints on the solution. In terms of DSCG, the problem domain is the context where domain specific models are constructed. It can be regarded as a meta-model of the input problems.

The Solution Domain While the "problem domain" defines the environment in which the solution work, the "solution domain" defines the environment where the solution is developed. As Czarnecki and Eisenecker [49] put it, the *solution domain* (or *solution space*) "consists of the implementation components with all their possible combinations". Accordingly, in this thesis, we define the "solution domain" as all elements that are used to form a solution. It is worth noting that the elements in the solution domain may reside at more than one levels of abstraction. The higher level solution domain defines the abstract context in which a solution is constructed, while the lower level solution domain includes the corresponding implementation, e.g. in the form of source code. As there is not a consensus about the definition of the "problem domain" and the "solution domain", we will not introduce the ontologies of them respectively in the case studies of our extension approach. Instead, our discussion will be around the *target domains*, which refer to a general combination of the corresponding problem domain and solution domain.

Output Languages, Target Languages and Host Languages The terms *target languages* and *output languages* are introduced from the perspective of DSCG code generators. They normally refer to the same concept, namely the programming languages in which the code is generated. There is difference between them only when the model tracing information is directly tangled with the generated code. In that case, we use the term *target languages* to refer to the original programming languages without the definition of the grammatical constructs for model tracing information, e.g. the comments for tracing an instance of a certain domain class. We use the term *output languages* to refer to extended programming languages which include the definition of such model tracing constructs. In practice, *target languages* are often standard versions of some mainstream programming languages, such as *C11 (ISO/IEC 9899:2011)* [50]. *Output languages*, on the other hand, are the extended versions of the *target languages*, which include the model tracing constructs. We can see some examples in later discussion. It is worth noting that such extensions only enrich the semantics of the target languages conforming to the existing syntactic specifications of them, so that they do not require extended compilers. It is because that interpreting the extra semantics for model tracing is not the job of the target language compilers. Another relevant term is *host languages*. As Hutchins [51] indicated, “DSLs are often implemented by code generation, in which domain specific constructs are translated to a general purpose ‘host’ language.” From this perspective, the DSCG process is essentially the DSL compilation process. Therefore, the *host languages* of DSLs are in fact the *output languages* in the underlying DSCGs, provided that the DSLs are implemented by the DSCGs.

In the following discussion, the term *code generation* only refers to the code generation process in MBD, i.e. the “model-to-code” transformation. With the help of DSCG, software designers can be relieved from worrying about tedious details in the solution domain, such as system portability, or adopting best practice for certain solution patterns. Thus they can indeed work at raised abstraction level. Moreover, the higher abstraction level also leads to significantly reduced design space, which helps in decreasing design errors. On the other hand, the greater abstraction level gap between the original models and the generated code makes the mapping between them more tenuous. Therefore, model traceability becomes more important in maintaining the link between the models and their corresponding code.

2.2.4 Model Traceability

The notion of *traceability* in software development was originally defined as the “degree to which a relationship can be established between multiple products in the software development process or the degree to which each element in a single product establishes its reason for existing” [52]. Aizenbud et al. [53] argued that this definition was “strongly influenced by the originators of traceability–the requirements management community”, and broadened it to include “any relationship between artifacts involved in the software engineering life-cycle.” Traceability is often mandated in industrial projects, especially in safety-critical systems [54] and is often achieved by generating human-readable documents about how requirements can be related to various artifacts involved in the whole process [55].

In MBD, traceability is naturally gained by its transformation chain, along which each transformation relates its source and target models to each other. The application of repeated transformations from the original models to the finally delivered code make it very difficult to recognize the system elements that are specified in the original models. In details, the traceability is often achieved by maintaining extra information that conforms to a traceability meta-model, which may be used along the transformation tool chain. This extra information from the applied sequence of transformations can be chained together, and so provide an argument for how the generated programs conform to the original models.

In this thesis, with DSCG as our research problem domain, we are interested in the traceability of the domain specific models along the DSCG process. We use the term *model traceability* specifically to refer to the ability of tracing the domain specific model in the generated code. We use the term *traceability links*, which is used by Czarnecki and Helsen [28], to refer to any artifact that provides or maintains the traceability from the generated code back to the domain specific models. As they indicated, traceability links may be constructed in different ways, e.g. by encoding in transformation rules, or by storing unique GUID in each model element. In practice, there are different types of traceability links in terms of the way we keep them. For example, they can be kept inside the model and/or the generated code, or be kept separately, e.g. as a standalone tracing log file. In particular, we often call the traceability links intertwined in code at the entry and exit of the generated code blocks corresponding to a specific domain element, the *sentinels* (of the domain elements). In respect of the DSL and the output languages, the traceability links may exist in different forms, such as annotations in models, comments or function invocations in code, etc. Some examples of these traceability link implementations are shown in Appendix A.1. It is worth noting that traceability links are very important to model traceability, especially when the code block corresponding to the domain specific elements are not original linguistic constructs in the output language, such as variable modifications or function invocations.

Model traceability is often an expected property of code generators in MBD. Most commercial code generators support it by directly adding tracing information to the generated code in the form of comments or embeded links [56], while academic research has worked on maintaining and recovering it [57, 58].

2.3 Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) is, as Mehmood et al. summarized, an approach "which allows explicit identification, separation, and encapsulation of concerns that cut across the primary modularization of a system." [10] AOSD provides an effective way to separate and express these cross-cutting concerns. Early research works on AOSD are often from the perspective of programming paradigm based on such idea, which is well known as *Aspect-Oriented Programming* (AOP) [13, 14]. In AOP, cross-cutting concerns are encapsulated into a separate category of modules called *aspects*, which are invasively composed with (or *woven into*) a base system. It allows an "aspectual" decomposition of the system instead of (or in addition to) the traditional modular decomposition, and also enables an easy modification of the existing systems. In the following discussion, we follow the terminology used by Kiczales et al. [13]. We call the program of the target system the *base program*, and the language in which the base program is written the *component language*. We call the program of the aspect the *aspect program*, and the language in which the aspect program is written the *aspect language*. We call the tool that takes in aspect programs and then composes them with their corresponding base programs the *aspect weaver*. Finally, we call the output program of the aspect weaver the *woven program*.

2.3.1 Component Language

The above concepts by Kiczales et al. are based on the assumption that the target software system is implemented in a *Generalized Procedure* (GP) language, whose key abstraction and composition mechanism is always rooted in a certain form of generalized procedure [13]. The components, which are the system's functional units, are thus encapsulated in these GPs in a "well-localized, easily accessed and composed as necessary" manner. Therefore, the languages that these components are implemented in are called *component languages*.

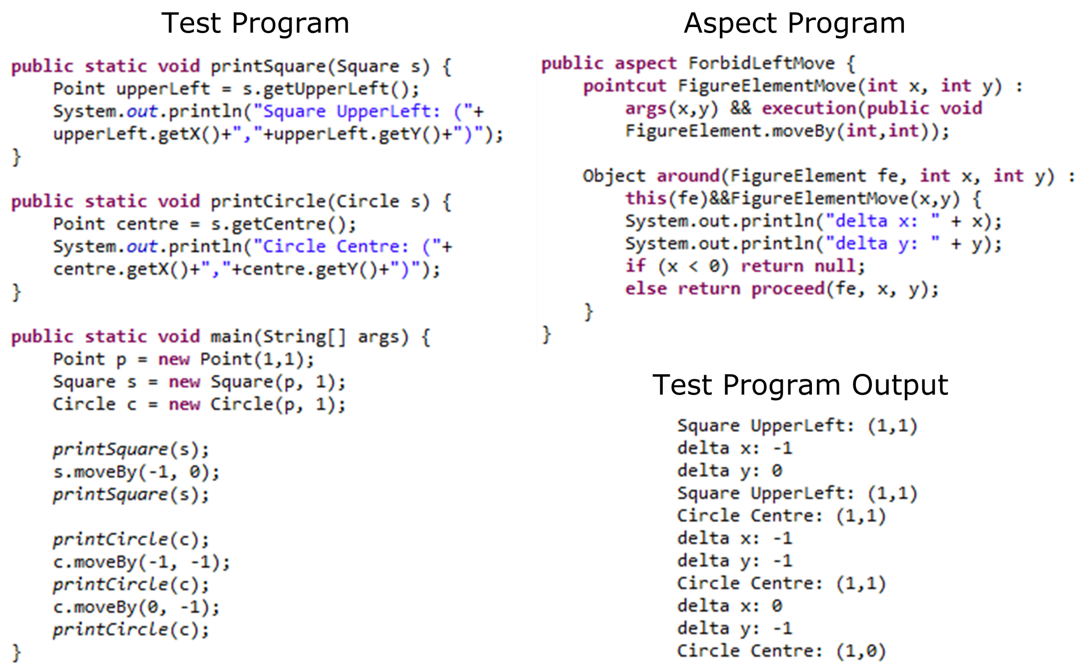
2.3.2 Aspect Languages

To give a precise definition of *aspect*, we first need to clarify the notion of *crosscut*. In general software system design, whenever two properties have to be built up following different composition rules and yet be coordinated, we say that they *crosscut* each other [13]. From this perspective, *components* can be regarded as the system properties that are built up via the major composition mechanism provided by GP languages. *Aspects* can thus be considered as the system properties that crosscut the components. Or as Kiczales et al. defined, *aspects* are the system properties that "affect the performance or semantics of its components in systematic way". The languages in which aspects are written are called *Aspect Languages* (ALs). According to the different crosscutting concerns that they focus on, ALs can be grouped into two categories: *general purpose aspect languages* and *domain specific aspect languages.*

General Purpose Aspect Languages An aspect language is often referred to as a *General Purpose Aspect Language* (GPAL), if it "is not coupled to any specific crosscutting concern and provides general language constructs that permit modularisation of a broad range of concerns" [59]. It is worth noting that a GPAL is not defined as an aspect language working with a *General Purpose Programming Language* (GPPL) as its component language. The distinguishing characteristic of an aspect language is the crosscutting concerns it focuses on, instead of the component language it works with. GPALs are very useful in software development, as there are many crosscutting concerns in software systems at source code level, for example, system logging, or error handling.

AspectJ [60] is a very widely used GPAL, whose component language is Java. As a simple and practical aspect-oriented extension to Java [61], it allows its users to encapsulate their crosscutting concerns for modification in separate aspects that are declared in a similar form to Java classes. An aspect declaration in AspectJ may contain three kinds of declarations: pointcut declarations, advice declarations, and all other kinds of declarations allowed in class declarations. To show a general idea of AspectJ, here we present an example of how it helps to modify an existing picture drawing tool. Figure 2.3 shows the current implementation of the tool.

Assume that we want to forbid any attempt to move a FigureElement to its left. We do not need to modify the code of every class that inherits FigureElement interface. Instead, we only need to write an aspect containing all the custom code we need to introduce, as shown in the AspectProgram in Figure 2.2.



Test Program

```
public static void printSquare(Square s) {
    Point upperLeft = s.getUpperLeft();
    System.out.println("Square UpperLeft: ("+
    upperLeft.getX()+","+upperLeft.getY()+")");
}

public static void printCircle(Circle s) {
    Point centre = s.getCentre();
    System.out.println("Circle Centre: ("+
    centre.getX()+","+centre.getY()+")");
}

public static void main(String[] args) {
    Point p = new Point(1,1);
    Square s = new Square(p, 1);
    Circle c = new Circle(p, 1);

    printSquare(s);
    s.moveBy(-1, 0);
    printSquare(s);

    printCircle(c);
    c.moveBy(-1, -1);
    printCircle(c);
    c.moveBy(0, -1);
    printCircle(c);
}
```

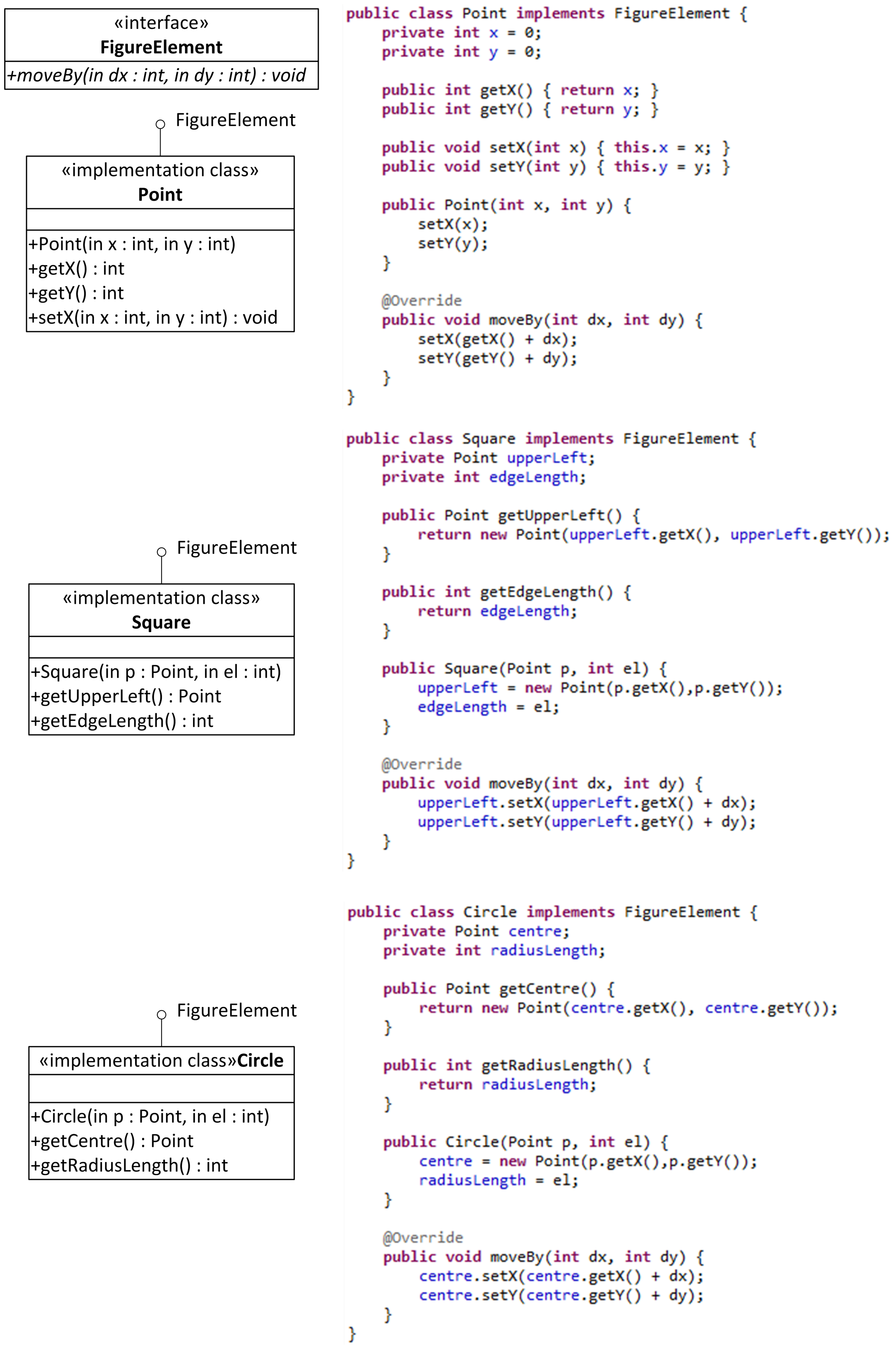
Aspect Program

```
public aspect ForbidLeftMove {
    pointcut FigureElementMove(int x, int y) :
        args(x,y) && execution(public void
        FigureElement.moveBy(int,int));

    Object around(FigureElement fe, int x, int y) :
        this(fe)&&FigureElementMove(x,y) {
        System.out.println("delta x: " + x);
        System.out.println("delta y: " + y);
        if (x < 0) return null;
        else return proceed(fe, x, y);
    }
}
```

Test Program Output

```
Square UpperLeft: (1,1)
delta x: -1
delta y: 0
Square UpperLeft: (1,1)
Circle Centre: (1,1)
delta x: -1
delta y: -1
Circle Centre: (1,1)
delta x: 0
delta y: -1
Circle Centre: (1,0)
```

FIGURE 2.2: An example of how AspectJ modifies the picture drawing tool.



```
public class Point implements FigureElement {
    private int x = 0;
    private int y = 0;

    public int getX() { return x; }
    public int getY() { return y; }

    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }

    public Point(int x, int y) {
        setX(x);
        setY(y);
    }

    @Override
    public void moveBy(int dx, int dy) {
        setX(getX() + dx);
        setY(getY() + dy);
    }
}

public class Square implements FigureElement {
    private Point upperLeft;
    private int edgeLength;

    public Point getUpperLeft() {
        return new Point(upperLeft.getX(), upperLeft.getY());
    }

    public int getEdgeLength() {
        return edgeLength;
    }

    public Square(Point p, int el) {
        upperLeft = new Point(p.getX(),p.getY());
        edgeLength = el;
    }

    @Override
    public void moveBy(int dx, int dy) {
        upperLeft.setX(upperLeft.getX() + dx);
        upperLeft.setY(upperLeft.getY() + dy);
    }
}

public class Circle implements FigureElement {
    private Point centre;
    private int radiusLength;

    public Point getCentre() {
        return new Point(centre.getX(), centre.getY());
    }

    public int getRadiusLength() {
        return radiusLength;
    }

    public Circle(Point p, int el) {
        centre = new Point(p.getX(),p.getY());
        radiusLength = el;
    }

    @Override
    public void moveBy(int dx, int dy) {
        centre.setX(centre.getX() + dx);
        centre.setY(centre.getY() + dy);
    }
}
```

Figure 2.3: Part of the sample picture drawing tool.

Domain Specific Aspect Languages Apart from the general concerns that relate to the component languages, there can be some particular concerns related to specific domains, i.e. domain specific concerns. For example, Harbulot and Gurd [62] developed the *loop join point model* that focuses on the specific semantic structures in the component language, in particular, the loop structure in Java. Ali and Rashid proposed the *state based join point model* [63] aiming for the safety critical system domain, in which state and state transitions of the system are the particular domain specific concerns.

An aspect langauge is often referred to as a *Domain Specific Aspect Language* (DSAL), if it "allows special forms of crosscutting concerns to be decomposed into modularized constructs" [64]. As we emphasized above, whether an aspect language is a DSAL does not depend on its component language, but on the forms of crosscutting concerns it focuses on. The component language of a DSAL does not necessarily need to be a DSL. An AL, whose component language is a GPPL, can still be a DSAL, as long as it focuses on specific concerns. Interestingly, the majority of DSALs have been developed to work with GPPLs [59].

2.3.3 Join Point Models

The general definition of the term *join points* is given by Kiczales et al. [13] as the "elements of the component language semantics that the aspect programs coordinate with." From the perspective of program execution, join points are the points (or actions) in base programs where aspect programs interact with base programs. The models that formalize the join points and the allowed interactions at them are referred to as the *Join Point Models* (JPMs). In particular, JPMs specify at which level the join points may crosscut the components. According to Kiczales et al. [31], the join point may crosscut the component at either the class level, i.e. crosscutting all instances of a class, or the instance level, i.e. crosscutting specific instance of the class. As Masuhara et al. [65] indicated, JPM is both a syntactic concept and a semantic concept. From semantic side, it defines which specific points are join points in base programs and which specific interactions or operations are allowed at these join points. From syntactic side, it defines the pointcut mechanism, i.e. how to identify join points, and the advice mechanism, i.e. how to specify the interactions allowed at join points. The core of an aspect language is its JPM, as it determines how it crosscuts its component language by defining its pointcut and advice mechanisms. We will elaborate upon the pointcut mechanism in Section 2.3.4 and the advice mechanism in Section 2.3.7. The JPMs of GPALs are often built in accordance with their component languages. In detail, their join points are the linguistic actions, like function invocations or variable value modifications. The basic elements we manipulate with in the interactions at these join points are also the linguistic entities, like variables. There are several general JPMs defined based on the different concerns in join point definition or interactions at join points. For example, based on the concern about the timing of join point interactions, there are *static/dynamic join point models*, in which aspect programs interact with base programs at compile-time/runtime. AspectJ, for example, includes the dynamic join point model, in which join points are specified as "principled points in the execution of the program" [61]. Returning to our AspectJ example shown in Figure 2.2, the ForbidLeftMove aspect captures the points in the TestProgram wherever an object whose class type implements interface FigureElement invokes its moveBy method.

2.3.4 Pointcut Mechanism

The notion of *pointcut* is another basic concept in aspect languages. A pointcut is a collection of join points that satisfy the conditions it specifies. To describe a *pointcut* in an aspect, a specific syntactic construct is introduced into the corresponding aspect language, which is called *pointcut designator* [61] or *pointcut descriptor*. Both of them are often abbreviated as *PCD*. It is worth noting that people may refer to the languages in which PCDs are defined as *crosscut languages* [66] or *pointcut languages* [67]. A crosscut language can be considered as an embedded language or mechanism in its "host" aspect language to describe pointcuts. Different aspect languages may have their own terms in referring to their crosscut languages. For example, in AspectJ, it is called "pointcut language" [68]. In DJ, it is called "traversal strategies" [69]. Crosscut languages do not only formalize the syntax to describe the pointcuts, but also reflect the underlying join point models adopted by the corresponding aspect languages. Returning to our AspectJ example shown above, we define the ForbidLeftMove aspect containing a PCD called FigureElementMove(intx, inty). We can see clearly that the PCD consists of three parts: the initial keyword pointcut, the declarative name of the pointcut before ":", and the join point description part after ":." From the final part args(x, y)&&execution(public void FigureElement.moveBy(int, int)), we identify all join points where an object of a type implementing FigureElement invokes its moveBy method with two int type parameters x and y. Moreover, it also reflects that AspectJ includes a dynamic join point model that allows user to capture the arguments of the execution of a method invocation as a join point.

2.3.5 Domain Specific Join Point Models

As its name suggests, *Domain Specific Join Point Models* (DSJPMs) are referred to the JPMs of DSALs, according to which domain specific elements can be used in both join point descriptions and the operations at join points. The basis of such a special crosscutting mechanism requires a unique perspective upon component languages; as we indicated in Section 2.3.1, Kiczales et al. introduce the general AOP architecture based on an assumption about the component languages of GPALs that they are GP languages, i.e. their key abstraction and composition mechanism is always rooted in a certain form of generalized procedure. Similarly, when we talk about the component languages of DSALs, we implicitly view the programs in the component languages as organized in certain forms of specialized procedures defined in specific domains. With respect to the consistency of the terminology used by Kiczales et al. , we call such component languages *Specialized Procedure* (SP) languages. In this thesis, any discussion about DSALs is based on the assumption that the component languages are always SP languages.

A DSJPM defines the specialized procedures in its domain, or *Domain Specific Procedures* (DSPs), as their join points, which normally correspond to some elements in the domain meta-model. There are often other domain elements involved in the DSP at each join point, which can be classes or properties in the meta-model, e.g. from_state and to_state in state transitions in our automata example. From the perspective of DSALs, a DSP is represented by a pointcut. which can be defined by domain specific elements, instead of the basic linguistic constructs in the GPPLs. Domain specific elements can also be used in the advice code, which modifies the behavior of the captured join point.

2.3.6 Problems Related to Pointcut Designators

There are a few problems related to PCDs that we need to discuss, as they have influenced our design of the pointcut designator in our aspect languages.

Coupling of Pointcut Designators to Base Programs The pointcut designators in many aspect languages are based on pattern matching of the base program's structures and behaviours. For instance, our previous examples in AspectJ have shown how pointcuts are defined by certain patterns of the method declarations. This tight coupling between pointcut designators and base programs may lead to problems in further evolutions. For example, Koppen et al. address the *fragile pointcut problem* [70], in which the autonomous evolution of base programs may incidentally lead to the failure of aspect weaving. Gybels et al. indicate the *arranged pattern problem* [66], where the evolution of base programs may render the aspects inoperative if the pointcuts are based on certain conventions and patterns in programs. All these problems hinder the evolution of aspect oriented programs, as they imply restrictions on the autonomous evolution of the base programs.

Attempts to Decouple Pointcut Designators from Base Programs There are several attempts made to decouple pointcut definition mechanisms from programs. AspectJ designers introduced *abstract aspects*, which contain *abstract pointcuts* that can be defined by inheriting concrete aspects. However, this only decouples the pointcuts from the aspects, instead of the base programs. The *fragile pointcut problem* still remains among the concrete aspects that implement the abstract aspects. Stoerzer et al. present the *pointcut delta analysis* [71] approach to address the fragile pointcut problem. They introduce a program analysis to both base programs and modified programs, in which they could (partly) detect the changes in pointcut matching (which they call *pointcut delta*) and finally reveal unintended code modifications that cause pointcut deltas. This approach might be useful to partially solve the fragile pointcut problem. However, it relies on “a good approximation of dynamic pointcut designators” [71] and requires dedicated supporting tools, which come with considerable cost. Moreover, it is merely treating the symptoms. It is not attacking the coupling which is the root cause of these problems.

Ostermann et al. present *expressive pointcuts* [72], which improves the expressiveness and robustness of the pointcuts by exploiting various information from the program, such as the execution trace or the syntax tree. However, the extra information introduced to increase the pointcut precision is based on static analysis of the program. Moreover, the abstraction capability of expressive pointcuts, such as functional composition and higher-order pointcuts, allows the composition of primitive pointcuts to build complicated pointcuts. The increased complexity of the aspects increases the possibility of aspect interaction problems. For example, the precedence of multiple aspects is more likely to lead to different program behaviours [73]. Different from the expressive pointcuts approach, which requires specific program analysis, the *program annotations* [71] approach achieves more precision via annotations to the programs. This approach maintains a conceptual view of the programs by adding meta-information in the form of annotation or comment. Though it demands the extra decoration of the program, the decoration itself does not affect the basic compilation of the program. All annotation related functions are encapsulated by the aspect weaver, which makes this decoupling approach the most light weight one with regard to the cost.

2.3.7 Advice Mechanism

Whenever a join point is successfully captured, the aspect needs to apply the expected modification to the corresponding base program. A semantic rule conveying such expected modification is often referred to as *advice*. Advice normally consists of two parts: *header* and *body*. A header declares the target pointcut that the current advice associates with, or to be applied to. This can be done in two ways, either by directly describing the target pointcut, or by referring to an existing pointcut that is defined elsewhere. A body contains the code snippet for adapting the captured join points which are specified by the pointcut declared in its header. In this thesis, we refer to such code snippet defined in the body of an advice as *custom code*. We refer to the language in which custom code is written as *custom language*.

There are mainly three concerns related to advice in an aspect language design. The first concern is how an advice communicates with associated pointcut, in particular, how information is exchanged between the captured join points and the custom code scope, i.e. context exchange [59]. For example, sometimes an advice needs to retrieve specific information from its captured join points, such as the runtime values of the parameters passed into a function, so that its custom code can work properly. On the other hand, if the custom code is an independent code block to append to the captured join points, there is no need for much context exchange. The second concern is how the custom code is applied to the captured join points, specifically, the pattern in which the custom code is integrated with the corresponding piece of code in the base program. Such patterns are often referred to as *advice patterns*. Different aspect languages may have their own advice patterns. Take AspectJ for instance, advice is designed to be "a method-like mechanism used to declare that certain code should execute at each of the join points in a pointcut" [61]. Thus the related advice patterns supported by AspectJ include: *before*, *after*, and *around*. As an example, we declare a piece of *around* advice in the ForbidLeftMove aspect, as shown in Figure 2.2. The third concern is in which order multiple advice should be applied when they are associated with the same pointcut. This may seem not important at the first glance. However, it does influence the modification result if the custom code of these advice have logical overlap. For example, assume we have two aspects A and B. Both are applied to pointcut P in *after* pattern. There is a variable x visible in the scope of the captured join points. A tries to update the value of x to 3. B tries to update the value of x to 5. If A is applied before B, then the modified code would finally set the value of x to 5. Otherwise, that would be 3.

From above discussion, we can observe that advice design relates closely to both pointcut language design and aspect weaver design. In the first concern, if we want to have more information available in the context exchange in an advice, we have to make the corresponding pointcut designator finer grained, so that it is able to capture more information in the first place. In the second concern, the advice patterns exposed to the users of an aspect language depends on how its aspect weaver can actually weave the custom code into the base programs. In the third concern, the advice application order known to the aspect language users must reflect in the implementation of its aspect weaver.

2.3.8 Aspect Weaver

The semantics of aspect languages, which are mainly about join points capture and advice application, are supported by their auxiliary tools, which accepts base programs and aspect programs as input and emits modified (or woven) programs as output. Such auxiliary tools are often referred to as *aspect weavers*, and the transformation process is called *aspect weaving*. Kiczales et al. [13] argued that aspect weaving "must process the component and aspect languages, composing them properly to produce the desired total system operation.” They define three distinct phases in the aspect weaving process. The first phase is to generate a data flow graph from the base program. The second phase is to run the aspect program to adjust the generated graph accordingly. The last phase is to walk through the adjusted graph and generate the modified program. In this process, they emphasized that the weaver should not be a “smart” compiler. Instead, its job should be “integration, rather than inspiration.” That means the weaver should work as a rewriting engine which only performs predefined primitive rewriting rules. All the “actual smarts” should be provided in the aspect by the programmer. To this extent, aspect weaving is essentially the compilation process of both aspects and their base programs. Aspect weavers can thus be regarded as compilers of aspect languages. Although the aspect weaver is often not considered as part of the aspect language, the aspect language relies on its underlying aspect weaver to work. A weaver of a GPAL can be referred to as a *general purpose aspect weaver*. Similarly, a weaver of a DSAL can be referred to as a *domain specific aspect weaver*.

2.4 Aspect-Oriented Approaches for Domain Specific Code Generation

Many aspect-oriented approaches have been proposed to work in the MDE context. Research works in this area have covered diverse topics ranging from techniques focused on specific modelling languages, like [74], to general software modelling paradigm like *Aspect-Oriented Modelling* (AOM) [75, 76], which uses aspect technology for the modularization and composition of crosscutting concerns during the design stage of software development. The general idea of them is to separate concerns in several models (including aspects) and then compose them together to derive the software system. In this section, we briefly introduce some of these works related to our work.

2.4.1 Symmetry of Aspect-Oriented Approaches

Symmetry is an important property of aspect oriented approaches, or more precisely, their composition paradigms. Interestingly, it was not proposed alongside the aspect-oriented approaches. The concept is not denoted until the base (or *component*) and aspects are clearly delineated as different elements involved in composition. Asymmetric aspect-oriented approaches explicitly distinguish the base and aspects that affect the base. Their composition is based on the join point model of the aspects, which is specified by the corresponding aspect language. In this style of aspect-oriented approaches, the base is the primary model with stand-alone descriptions. The aspects, on the other hand, can be considered as decorators of the base model, which can only be described in relation to the base. Their composition is essentially a one-way influence of aspect over the base through join points, e.g. the aspect application (or weaving) following the "pointcut-advice" pattern in AspectJ. The opposite influence is impossible, i.e. there is no way the base can affect the aspects. In contrast, symmetric aspect-oriented approaches do not have the above distinction. There is no "base" or "aspects" at all. Instead, there are only "partial views", which contain separated concerns of the system. Each view may affect every other view being composed. Their composition follows rules that are often independent of the views themselves.

To analyze the ramifications of using symmetric and asymmetric paradigms, Harrison et al. [77] define three levels of symmetry: "element symmetry", "join point symmetry" and "relationship symmetry", which are corresponding to the three kinds of entities involved in a composition paradigm, "composable element", "join points" and "composition relationship". As its name suggests, "composable element" denotes the element that can be composed, e.g. the base, aspect, and partial views. Each "join point", as the point inside the composable elements where composition can occur, can be intrinsically asymmetric or symmetric. Asymmetric join points contain explicit references to specific concerns, while symmetric join points allow composition of any concern. The details of how composition happens at join points are specified by "composition relationship". Asymmetric composition relationship has to be binary, which refers explicitly to the base and implicitly to the aspects that composed to the base. On the other hand, symmetric composition relationship has a scope ranging across all concerns being composed.

In practice, most AOSD approaches follow the asymmetric paradigm, including PARC AOP [78] and AspectJ. Contemporary software development has, to some extent, incorporated asymmetric aspect orientation. However, the industry adoption still does not fulfill the high hopes put into aspect orientation at its beginning, that it can improve software modularity. With regards to this, symmetric aspect orientation becomes promising, particularly to academia, as it shows the possibility of keeping concerns modularized from specification to implementation/code, e.g. the "Theme" approach by Clarke and Baniassad [79].

2.4.2 Model Composition Paradigms

The crux of aspect-oriented model-driven code generation is how to compose the base model and the related aspects, which contains separate concerns, into the complete software system. This process is often referred to as *model composition*. According to a systematic mapping study in this area by Mehmood et al. [10], model composition approaches can be categorized into at least two main groups.

The "Weave-Then-Generate" Approach The approaches in the first group, such as [75] and [80], directly compose the base model and the aspect model into an enhanced model, and then transform the obtained model into code of a target programming language through the DSCG process. For this reason, these approaches are also referred to as "Weave-Then-Generate" [81] approaches. The "Weave-Then-Generate" approach seems ideal from the MDE perspective. The outcome of composition is a composed model, which can be considered as a modification of the original model. The composed model can then be used to analyze or to generate new implementation/code. Thus the composed model remains as the primary artifact in the software development, and the typical "model change ⇒ code re-generation" evolution pattern in MDE orthodoxy can be maintained. The weakness of this approach, on the other hand, is a potentially large semantic gap between the composed model and code. First, it can be quite difficult to develop the composition tool. For example, the aspects being composed may require modification of the meta-model of the base. A trade-off has to be made between the expressiveness of aspects and the consistency of the meta-model of the base. Second, the traceability of code errors may be hindered. When an error is detected in code, it could be difficult to identify if it comes from the base or a certain aspect.

The "Generate-Then-Weave" Approach The second group of model composition approaches first generate the base program from the base model, and then explore the direct transformation of aspect model into code of a target aspect language, which is often a GPAL, e.g. AspectJ. Finally, the obtained aspect program is applied to the base program using the weaver of the target aspect language. These approaches are also referred to as "Generate-Then-Weave" [81] approaches. The "Generate-Then-Weave" paradigm is adopted by many code generation tools/frameworks, such as the *Formal Design Analysis Framework* (FDAF) by Bennet et al. [82] and the "GenERTiCA" generation tool by Wehrmeister et al. [83]. Compared with the "Weave-Then-Generate" approach, the "Generate-Then-Weave" approach produces potentially smaller semantic gap between model and code, and it simplifies the generation of the composition tool. It only requires translation tools from the DSALs into some target GPALs. The translated aspects can then be applied to the base program using the the weavers of the underlying GPALs. Moreover, a smaller semantic gap makes it easier to relate the errors detected in code back to the base models or certain domain specific aspects.

2.4.3 Transformation-Oriented Views of Model Composition

If we consider the code generated in DSCG as the lower-level (i.e. code level) models that correspond to the input domain specific models, the DSCG process can thus be considered as a special case of model transformation. Thus the composition of the code level models and the aspect models can be examined from the transformation-oriented views. According to an exploration work on the relationship between model composition and model transformation by Fleurey et al. [84], there is a spectrum of transformation-oriented composition approaches from dedicated composition to generic composition.

In a dedicated composition, all content that a primary model needs to incorporate is specified by a context specific aspect model, while no knowledge about the primary model, aspect model, composition directives, bindings and the signatures is included in the transformation actions in a generic composition. As they argued, such generic transformation would be "extremely difficult (if not impossible) to implement". To make our approach as generic as possible, we develop the approach following a pattern of template aspect model and model-transformation-specialized framework, on which we will elaborate in the next chapter.

2.4.4 Generic Model Composition

Although aspect-oriented approaches show a propitious prospect in dealing with system complexity and maintainability, they rely on the ability of automatic model composition in various domains. This attracts research effort in a generic model composition framework that can be adapted to different domains, i.e. modelling languages. Bézivin et al. [85] examined three different model composition frameworks, from which they extracted a core set of common definitions in model composition, such as "match operation", "merge operation". Fleurey et al. [86] proposed a generic model composition framework, in which model composition is decomposed into two major steps: a domain-specific "Matching" step and a domain-independent "Merging" step. The framework defines "Signature" to identify meta-model elements, and "Mergeable" to represent any domain element that can be merged in a generic mechanism. As the first step to adapt the framework to a specific modelling language L, a "composition strategy" $\mathsf{M_C}$ needs to be defined to specify the three things: "Mergeable" elements in the meta-model of the given modelling language $\mathsf{M_L}$, their "signature", and a specific matching operator based on these "signatures". $\mathsf{M_L}$ can then be composed with $\mathsf{M_C}$ to obtain $\mathsf{M'_L}$ that has composition capabilities. Any model that conforms to $\mathsf{M_L}$ should also conform to $\mathsf{M'_L}$. When the adapted framework takes two models conforming to $\mathsf{M'_L}$, all "Mergeable" elements in the models are checked using the "signature" matching operator. All matching elements are finally merged using generic algorithm to finish the model composition.

We gleaned two pieces of inspiration from this work. First, meta-model can be used as a clear interface to specialize a generic model manipulation framework for a given domain specific modelling language. In particular, the meta-model of a modelling language can be extended through a composition with a functional model, e.g. "composition strategy" in this work, to obtain the corresponding functionalities, i.e. the composition capability. Second, not every element in the given meta-model is used by the framework, only a subset of the given classes and properties is involved in its functional extension. In real-life DSCG cases, the meta-model of a given modelling language can be large and complex. An effective way to derive such a simplified meta model may benefit the extension in terms of its complexity and efficiency.

2.4.5 Meta-model Pruning

Meta-model pruning is an algorithm proposed by Sen et al. [87] to prune meta-models by removing unnecessary elements. In detail, the algorithm takes as input a meta-model M and a set of its classes and properties marked as required, and trims off any other classes and properties that are determined as unnecessary based on a set of rules and options. As a result, a potentially smaller meta-model M' is derived, which contains only the elements in the input set and their mandatory dependents. The derived meta-model M' is often referred to as an *effective meta-model* (of the original meta model M). From the graph-theoretical perspective, this pruning process is to simplify a given meta-model by cutting off all unnecessary nodes (classes or properties) from its graph. Moreover, it preserves the names of the meta-classes and meta-properties retained from the original meta-model in the effective meta model. From a type-theoretical perspective, the effective meta-model can be taken as a "super-type" of the original meta-model. As Sen et al. argued, "all models of the effective meta-model are exchangeable across tools that use the large input meta-model as a standard." As a conclusion, the meta-model pruning algorithm provides us a generic method of meta-model simplification, which can help us tailor the meta-models before any further customization.

Chapter 3

Extension in DSCG with Domain Specific Aspects

In Section 1.2, we mentioned that the primary objective of our work is to accommodate changes that can not be expressed by the original domain specific modelling languages, whilst the benefits of DSCG can be maintained. The benefits here mainly refer to the reliability, i.e. the correctness of the modified code, and the productivity, i.e. the effectiveness of the modification. To achieve this, we develop an approach that takes as input the meta-model of the target DSCG, and extend it with an aspect oriented system, so that we can express such changes as domain specific aspects and compose them systematically with the current system by weaving them into the base code. In this chapter, we first explain some concepts and terms involved in our work, and then raise some assumptions of our "DSCG extension approach". We then elaborate on this approach and demonstrate the modification process using a simple extension scenario of the automata example in Section 1.1.

3.1 Terms and Assumptions

Before we start the detailed introduction of our approach, we first explain a few concepts and terms that are involved in the following discussion, and raise several assumptions to narrow the scope of the discussion.

3.1.1 Context Free Grammar

In our approach, DSAL is generated according to the target DSCG. The generated DSAL is specified as a *Context Free Grammar* (CFG) [88]. As its name suggests, production rules in its grammar can be applied regardless of the context. There is a formal definition of CFG given by DeRemer et al. [89]. The definition denotes a CFG by a quadruple $(\mathsf{V_T}, \mathsf{V_N}, \mathsf{S}, \mathsf{P})$, where

- $\mathsf{V_T}$ is a finite set of symbols that often called *terminals*
- $\mathsf{V_N}$ is a finite set of symbols distinct from $\mathsf{V_T}$ that often called *nonterminals*
- S is a unique symbol called *start symbol*, where $\mathsf{S} \in \mathsf{V_N}$
- P is a finite set of rules that often called *production rules*
- $\forall\ p \in \mathsf{P}$, p is a pair (α, β), where $\alpha \in \mathsf{V_N}$ and $\beta \in (\mathsf{V_T} \bigcup \mathsf{V_N})^*$

As α always appears on the left hand side of production rules, it is often referred to as the *Left Hand Side* (LHS). Accordingly, β is often referred to as the *Right Hand Side* (RHS). When one α appears as LHS in more than one production rule, it is common to list all the βs as the RHS of a "merged" production rule, where each of the original βs is called an *alternative* and separated by symbol "$|$".

3.1.2 Abstract Syntax Tree and Parse Tree

To parse a program according to a given grammar, a parser normally first interprets the program into a token string through lexical analysis, and then generates a "tree-like intermediate representation that depicts the grammatical structure of the token stream" [40]. There are two typical types of "tree-like" representations that can be created in the parsing process. One is *abstract syntax tree* and the other is *parse tree*.

Abstract Syntax Tree An *Abstract Syntax Tree* (AST), or sometimes shortened as *syntax tree*, is a tree representation of the syntactic structure of an input string in a given language. It is "abstract" because its nodes represent the abstract syntax of the language, instead of the concrete tokens defined by some formal grammar that specifies the language. In other words, an AST is a conceptual representation built with language constructs. For example, the AST of the expression "$\mathsf{3 + 4}$" in a simple arithmetic operation language is shown in Figure 3.1.

Parse Tree Similar to an AST, a *parse tree* is also a tree representation of the syntactic structure of an input string in a given language. The difference is that a parse tree is always related to a formal grammar specifying the given language, instead of the language itself. The nodes of a parse tree represent concrete syntax. Or more precisely, the internal nodes of a parse tree represent the nonterminals defined in the related grammar, whereas its leave nodes represent the terminals. To emphasize this

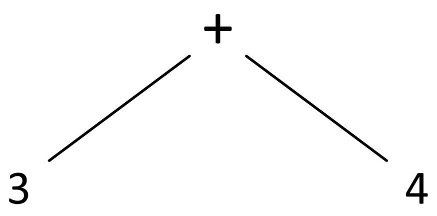


FIGURE 3.1: The AST of input “3 + 4”

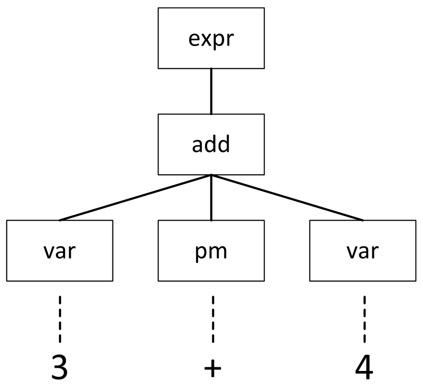


FIGURE 3.2: The parse tree of input “3 + 4”

contrast, a parse tree is sometimes called a *Concrete Syntax Tree* (CST). Assume that the above arithmetic operation language is specified with a simple grammar, whose EBNF representation is shown in Listing 3.1, the parse tree of the expression “3 + 4” is shown in Figure 3.2.

```
expr := add | mul ;
add := var pm var ;
pm := '+' | '-' ;
add := var md var ;
md := '*' | '/' ;
var := '0' | '1-9' {'0-9'}*
```

LISTING 3.1: A simple grammar specifying the arithmetic operation language

In summary, a parse tree is a representation of a program, which contains the concrete syntax constructs defined in the given grammar. It is a record of grammar rule applications. An AST, on the other hand, is an abstract representation of a program, which trims off the concrete syntax constructs and retains the “real content” of the program.

3.1.3 Languages in DSCG

In the context of DSCG and DSAL, concepts and terms can sometimes be quite confusing, as the artifacts they refer to may have multiple appellations from different perspectives or in different terminologies. For example, the programming language, in which the code is generated, can be called the *output language* of the DSCG code generator in the context of DSCG . The same language can, from the DSL perspective, be referred to as the *host language* of the DSL. In this section, we clarify the definitions of some terms commonly used in our discussion.

The terms *target languages* and *output languages* are both defined from the perspective of DSCG. In general, they are used to refer to the same artifact, i.e. the programming languages in which the generated code is written. *Target languages* emphasize the model-to-code transformation process, in which the modelling languages are the *source* and the programming languages are the *target*. *Output languages* emphasize the code generators, which use the programming languages to describe the *output*. In our approach, as the code generator would be updated to include the traceability link in the code generation process, we use them to distinguish the programming language before the code generator update and after that. In detail, we use the term *target languages* to refer to the original programming languages used in the target DSCG. We use the term *output languages* to refer to the programming languages extended with the extra syntax and semantics for the intertwined traceability links, e.g. the traceability links based on comments.

As an example, Listing A.1 shows a code block wrapped by a pair of comment based traceability links in a programming language. From the perspective of the target language, the two traceability links are merely two lines of textual comments with no semantics. Removing the comments will not affect the generated code in terms of its functional correctness and performance. On the other hand, from the perspective of the output language, these comments can be parsed with finer grained syntactical rules, to get some extra model tracing semantics. Although their removal still does not affect the functional correctness and performance of the generated code, it will impede the proper crosscutting behaviour of our DSAL weaver. In practice, *target languages* are often standard versions of some mainstream programming languages, e.g. *C11 (ISO/IEC 9899:2011)* [50].

3.1.4 Assumptions

In practice, there are many factors involved in the DSCG contexts that may influence the feasibility of our approach, e.g. the implementation of the DSCG code generators. As such, we raise several assumptions to focus on the concrete problem to be addressed by our approach.

Assumption 1: We focus on the DSCG for only textual DSLs.

As Grönniger et al. [90] mentioned, there types of DSLs used nowadays: textual DSLs, graphical DSLs and hybrid DSLs. The experience from practical work indicates that there are indeed some advantages of the textual DSLs over the graphical DSLs, in terms of the interpretation and manipulation of the DSLs. In this work, to accommodate the modification requirements that cannot be expressed by the original DSLs, our approach aims at extending the DSLs by generating the corresponding *Domain Specific Aspect Languages* (DSALs) from them. The very first step in this process is to extract the domain meta-models from the DSLs. Although this is still possible with graphical DSLs, it would be much easier to work with textual DSLs. Besides, Grönniger et al. indicated some other important advantages of textual DSLs, such as less dependence on the platforms and tools.

Assumption 2: We only focus on the transformation based DSCG.

The basis of the DSALs generated in our approach is the model traceability in the generated code. If the target DSCG is transformation based, the generated code would be a direct product of certain transformation process. No matter whether the transformations are rule based rewritings or template expansions, it is still possible to maintain the model traceability in the final transformation result, i.e. the generated code. However, if the target DSCG is deduction based, the code would be generated only as the byproduct of the proving process, which "reflects the proof from which it was extracted" instead of the target domain elements. As a result, there is no guarantee that model traceability can be established in the generated code. Therefore, we only focus on only the transformation based DSCG.

Assumption 3: The DSCG code generators must be compositional generators.

In Section 2.2.4, we mentioned that it can be very difficult to trace domain specific elements in the code generated through multiple transformations conducted by the code generator. If the code generator allows horizontal transformations that breach the modular boundary in their higher level representations, the potential overlapped modules in the lower level representations would make it very difficult to trace certain domain specific elements in the generated code. Particularly in our approach, we use sentinel pairs at the boundary of the code block corresponding to certain domain element instance. Such impediment in model traceability would severely hinder the generation of the DSALs. Figure 3.3 shows an example of the problem when the higher level modular structure is not preserved.

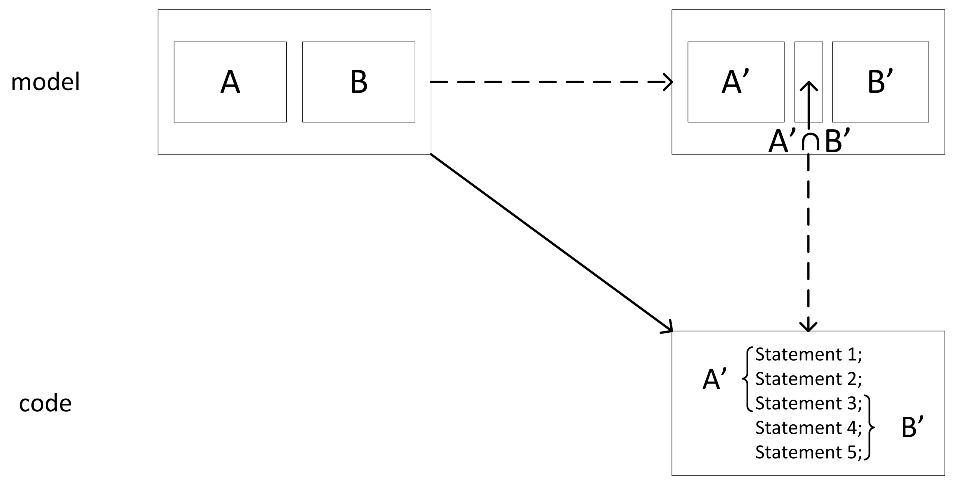


FIGURE 3.3: An example in which higher level modular structure is broken.

In this example, Action A and B in the domain specific model does not have any overlap at the beginning. However, as the result of an internal optimization (a horizontal transformation), Action A and B are restructured into Action A′ and B′ which share an overlap part A′ ∩ B′. In the final output program, Statement 1-3 are generated from Action A′ while Statement 3-5 are generated from Action B′. Obviously, Statement 3, as an overlap, would cause problems when we try to trace A′ or B′. For example, if we changed statement3 in an attempt to modify Action A′, we actually also changed Action B′ in the meantime. Therefore, some tricks need to be developed to address such tracing problems. As such tricks are beyond our research interest, we just forbid any horizontal transformations in the target code generators to avoid such problems in our discussion. Compositional code generators, as we introduced in Section 2.2.1, only allow vertical transformations. Thus we simplify the environment by assuming that the code generators in our discussion are always compositional.

3.2 The DSCG Extension Approach

The basic idea of our approach is to develop a generic aspect oriented framework, which can take as input a target DSCG, the involved DSL in particular, and dynamically generate as output a DSAL and a corresponding aspect weaver from it. The generated DSAL allows domain experts to describe the expected changes, which cannot be expressed in the original modelling language, and automatically weave them into the base code using the generated DSAL weaver. From the perspective of MDE, the generated DSAL can be considered as an auxiliary of the target domain modelling language, and its weaver as an appendage of the modelling tool chain, at the end of the code generator. They as a whole can be regarded as an extension of the target DSCG to accommodate certain modification requirement in a systematic way. For this reason, we call it the "DSCG extension approach".

3.2.1 Overview

Figure 3.4 illustrates the general process to accommodate changes using our DSCG extension approach, which can be divided into three main parts: *Target DSCG*, *DSCG extension* and *DSAL application*.

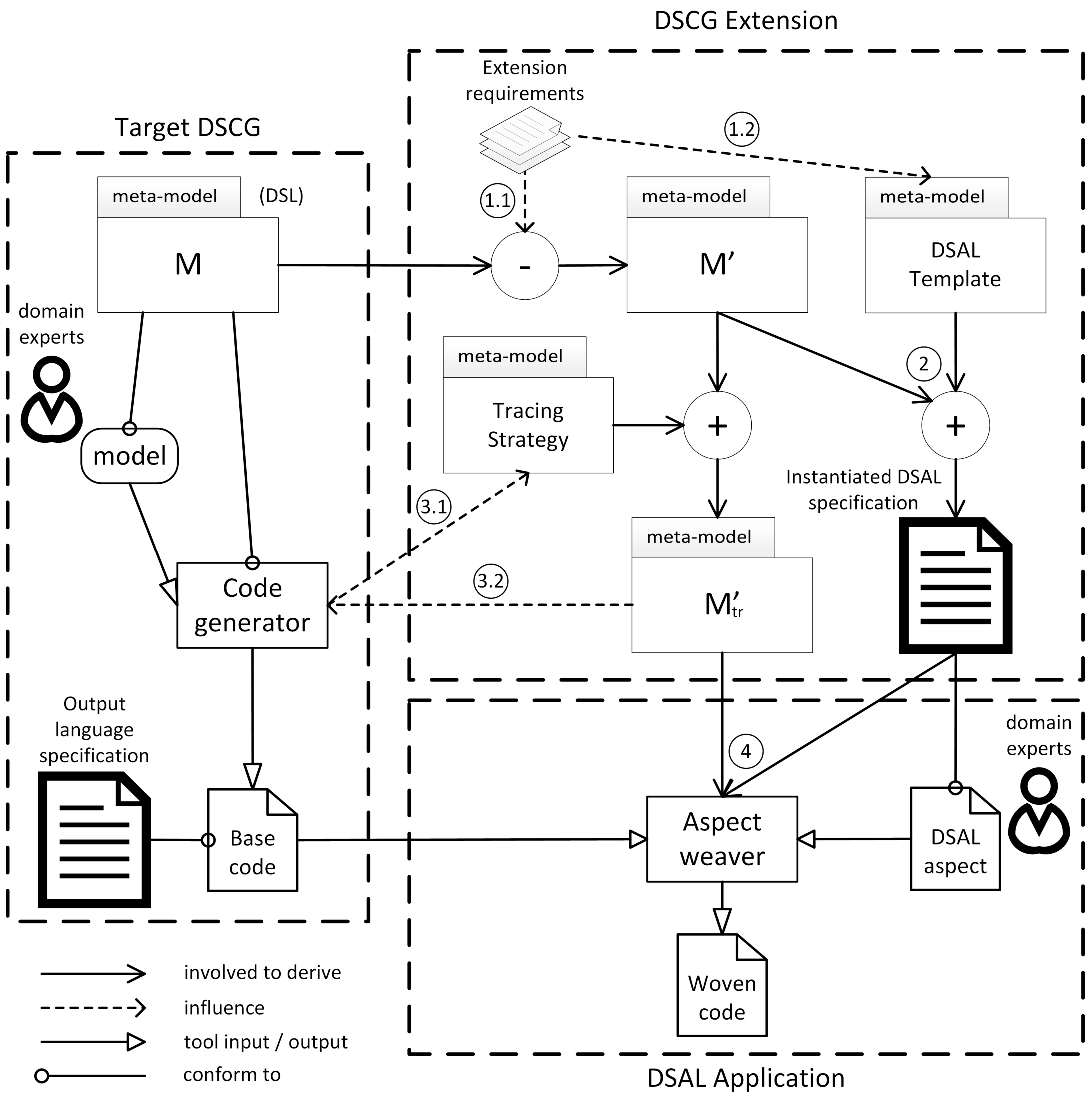


FIGURE 3.4: An overview of our DSCG extension approach.

The target DSCG process is shown in the left rectangle with the dashed outline. Domain experts write models conforming to the domain meta-model, i.e. the DSL, and then send them into the DSL code generator to generate programs, or "base code", that conform to the output language specification. The extension of the target DSCG is shown in the upper right rectangle with the dashed outline. This is the core step in our approach, to generate the modification systems. Simply stated, the extension requirements are first interpreted to specify how domain meta-models need to be involved in the expected change and what DSAL patterns can make it easy to describe these changes. With regards to the interpreted specification, a traceable meta-model of the target domain and a specification of the DSAL are then generated respectively. Finally, a corresponding aspect weaver is created in accordance with them. We will elaborate on these generation process in Section 3.3. The next step in our approach is to apply the generated DSAL, in which domain experts can describe the expected change in DSAL aspects, and weave them into the base code using the generated DSAL weaver, to finish the modification process.

It is not difficult to find that meta-models are used throughout the DSCG extension process. These meta-models are the basis in constructing the modification systems in our approach. Thus we also refer to this approach as the *meta-model based DSCG extension approach*. In practice, these meta-models can be specified using different techniques, e.g. the *Unified Modelling Language* (UML) [91], the *XML Schema Definition* (XSD) language [92], or the *Kernel MetaMetaModel* (KM3) [122]. The technique selection may depend on the preference of the DSCG extension implementor, e.g. UML for graphical representation and XSD for textual representation, as well as the availability of the tool set, e.g. *Visual Paradigm* [93] for UML and *Visual Studio* [94] for XSD.

Within the entire modification process, we can see that there are four steps annotated with circled numbers. These steps may require human communication and decisions, and thus affect the efficiency of introducing the expected changes. There may be trade-offs between productivity and some property that we prefer in these steps. We will have further discussion on this later.

3.2.2 An Example of Extension Scenario

To demonstrate how our approach works in detail, we use a hypothetical scenario about our automata example shown in Section 1.1, in which the expected changes cannot be described with the original DSL.

Meta-Model of Target Domain The target domain is the simulation of *Deterministic Finite Automata* (DFA), or *Deterministic Finite State Machine* (DFSM) [95]. The corresponding DSL is very simple. As shown in Figure 3.5, there are only four domain classes defined: State, Input, Transition, Automata. Each State has a name property and may have zero or more inbound or outbound Transition objects. Each Transition has three properties: a from_state, a to_state, and an input object. Each Automata may have one or more State objects, zero or more transitions, and two other properties, start_state and accept_state.

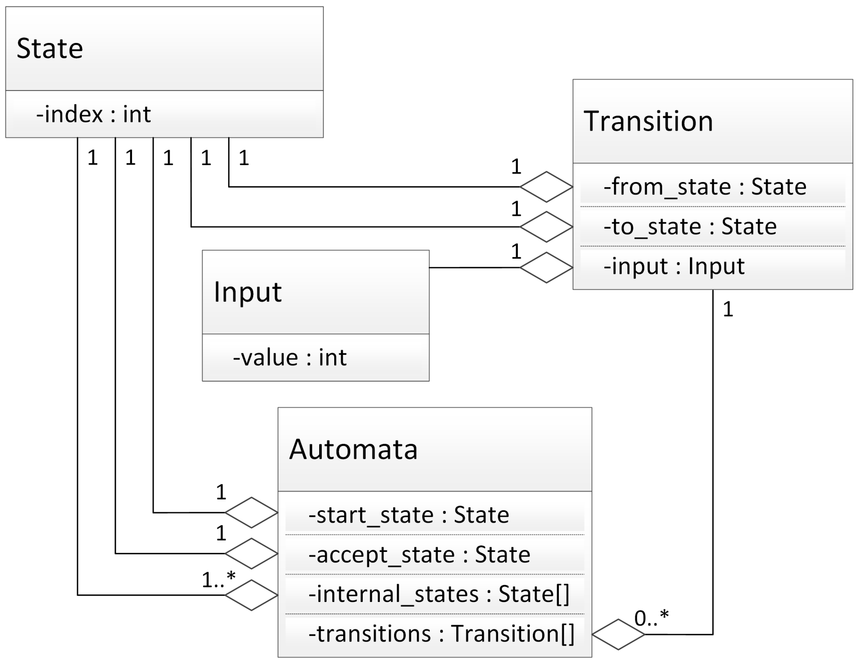


FIGURE 3.5: UML diagram of DFA domain modelling language.

The DSL used in this example is called "DFA". It is specified by a language specification framework called *ANTLR* [16, 96, 97], upon which we will elaborate in Section 5.2. The complete ANTLR specification of the DFA language is attached in Appendix A.5. The corresponding description in *Extended Backus-Naur Form* (EBNF) [98, 99] is shown in Listing 3.2.

```
INTLITERAL := '0' | '1'..'9' {'0'..'9'}* ;
INTLIST := '{' INTLITERAL (',' INTLITERAL)* '}' ;
alphabet_declaration := 'Alphabet' INTLIST ';' ;
states_decl := 'States' INTLIST ';' ;
start_state_decl := 'StartState' '(' INTLITERAL ')' ';' ;
accept_states_decl := 'AcceptStates' INTLIST ';' ;
state_declaration := states_decl start_state_decl
   accept_states_decl ;
transition_decl := 'Transition' '(' INTLITERAL ','
   INTLITERAL ',' INTLITERAL ')' ';' ;
transition_declaration := (transition_decl)* ;
program := 'DFA' '{' alphabet_declaration state_declaration
   transition_declaration '}' ;
```

LISTING 3.2: The EBNF representation of the "DFA" language

Extension Requirement Now assume that the domain experts want to trace certain transitions in simulation, e.g. to count and log any transition from state x to state y. Obviously, the current DSL cannot express this requirement, since there is no domain class capable of counting numbers or logging transitions. According to the pure model based approaches, the DSL needs to be extended with supplementary domain classes, e.g. a "logger" class, and the code generator needs to be updated accordingly, so that the base model can be rewritten in the extended DSL, and then be used to regenerate the code with the customized code generator. However, the domain experts do not agree to extend the DSL with any counting or logging facilities. They do not want to get into any compatibility trouble with the existing models or the MDE tool chain. Besides, there is an existing function "log()" in the legacy library code, which implements exactly the expected counting and logging function. The ideal way to modify the code is to insert a line to invoke this function at the entry of each code block corresponding to a target transition. However, such manual modification will sacrifice the reliability of the code and the productivity based on the modelling tool chain. This is a typical scenario where our approach can be a helpful alternative. Now consider a simple automata model that is used to generate the code shown in Figure 1.5, its description in the "DFA" language is shown in Listing 3.3.

```
DFA {
    States {0,1,2};
    StartState (0);
    AcceptStates {2};

    Transition (0,1,1);
    Transition (1,3,1);
    Transition (1,5,2);
}
```

Listing 3.3: The DFA model involved in our automata example

In the next section, we will elaborate on how our approach can help domain experts to introduce this change step by step.

3.3 Meta-Model Based DSCG Extension

In our DSCG extension approach, the generation of the DSAL is the first and fundamental step. It consists of four main processes. First, the extension requirements are interpreted. Accordingly, a compact meta-model of the target domain M' containing all domain elements involved in the expected changes is generated, and a generic DSAL specification is created. Second, a trace strategy model is created with insight of the implementation of the code generator, and composed with the derived meta-model M'.

Thus all domain elements in the composed meta-model $\mathsf{M}'_{\mathsf{tr}}$ obtain the tracing functionalities. The code generator is then updated according to $\mathsf{M}'_{\mathsf{tr}}$ to become capable of building the traceability links in its code generation. Third, the created DSAL specification is composed with M' to derive a specification of the DSAL. Finally, a corresponding aspect weaver is created with regards to both $\mathsf{M}'_{\mathsf{tr}}$ and the DSAL specification.

3.3.1 Extension Requirement Analysis

In our approach, the analysis of the extension requirements aims to extract formal specifications of the expected changes from the description by domain experts in natural language, in particular, the answers to the following two questions. The first question is *How domain classes and properties may be involved in the expected changes?*. The second question is *Which DSAL patterns would be the best to accommodate these changes?* The first answer, annotated with $\textcircled{1.1}$, helps to derive a subset of the target domain meta-model. The second answer, annotated with $\textcircled{1.2}$, helps to create a template DSAL specification.

Effective Meta-Model The first answer from domain experts specifies which classes and properties in the original domain meta-model M would be involved in the expected changes. In particular, the classes they want to use to crosscut the base model, i.e. the classes to be used as join points, should be indicated explicitly. There can be one or more class in M selected as join point classes. Each join point class normally corresponds to one specific PCD pattern and potentially an advice pattern. Each join point class should be specified that at which level they are supposed to crosscut the base model, e.g. the class level or the instance level, as explained in Section2.3.3. For instance, in our UML-based meta-model in our automata example, we can use join point interfaces like "IJoinInstance" as the indicators. Besides, the domain experts also need to confirm if each property in the selected join point class is involved in the expected changes. Each join point class will be composed later into our DSAL template, which means a corresponding type of traceability link will need to be established, and a corresponding DSAL grammar rule will need to be created. If we could trim off some unrelated domain elements, we can reduce the complexity of the involved domain meta-models and potentially reduce the complexity of the composition with the DSAL templates. For this reason, the classes and properties selected by the domain experts can be sent as input of a pruning method. The method will calculate a closed subset of M, according to certain interests or concerns. The calculated meta-model, i.e. M' is often referred to as *effective meta-model*. In our following experiments, we always use the algorithm proposed by Sen et al. [87], which calculates the effective meta-model through the dependencies of the given subset of domain meta-model.

Returning to our DFA example, here we use UML to specify the domain meta-models.

As the domain experts only want to trace certain transitions determined by their properties, i.e. from_state, input or to_state. They do not care which specific automata instance has included these transitions. Therefore, we do not need to include class "Automata" in the effective meta-model. To mark the "join point" class, we enforce them to implement the "IJoinInstance" interface, which represents join points that crosscut at instance level. The corresponding effective meta-model in UML is shown in Figure 3.6.

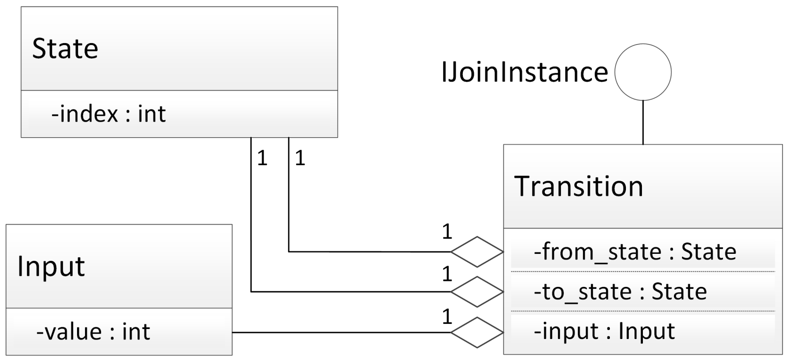


FIGURE 3.6: Effective Meta-Model of the DFA domain.

DSAL Template Selection The second answer from domain experts specifies which aspect oriented modification patterns can simplify and facilitate the description of the expected changes. As domain experts may not be familiar with AOP or DSAL, we create a number of formal language templates, or DSAL grammar templates to be precise, which are based on different aspect oriented modification patterns, e.g. PCD patterns and advice patterns, so that they can select the most convenient way to accommodate the changes they expected. If no existing template can satisfy their need, we can of course create new templates. The selected template will then be expanded according to the derived effective meta-model M', to generate a valid DSAL that domain experts can use to describe the changes. In these DSAL templates, placeholders are used to represent the domain specific content, such as the filtering condition in the PCDs. When a selected template is expanded in accordance with an effective meta-model M', the placeholders will be expanded with the specific join point classes and their related properties in M'. The templates we use in our experiments are defined as *Context-Free Grammars* (CFGs), as there are a number of parser generation frameworks supporting CFGs, e.g. the *ANTLR framework* [16], with which the generation of the DSAL weaver might be partially automated.

Our DSAL templates follow the asymmetric aspect orientation paradigm, i.e. to take the generated code as the primary base model, and describe the changes as aspects. There are two reasons behind it. First, our purpose of this aspect oriented extension of the target DSCG is to provide an auxiliary way to modify the model-based generated code, without breaking the existing tool chain conforming to the given DSL. We still want to keep the existing models as the primary artifact, and to distinguish them from any artifact we introduce for modification purpose, e.g. the aspects. Second, symmetric aspect orientation paradigm does not distinguish aspects from models. This may introduce undesirable influence in the detailed behavior of model composition, especially when there are multiple relevant aspects involved. For example, when there are two aspects to be applied to the same model, the two aspects may have implicit and unwanted impact over each other, before they are applied to the model.

The DSAL templates can be specified with any language specification formalism, such as *Syntax Definition Formalism* (SDF) or the *ANTLR grammar specification language* [96], as long as the templates can be easily composed with the derived effective meta-model M'. From the perspective of AOP, our templates follow a "pointcut-advice" pattern. In this pattern, each pointcut describes a collection of join points that satisfy a specific condition. The placeholders are often used in such conditional filters, which can be a simple value check on one of the domain properties, or a compound condition involving multiple domain elements. Domain experts need to determine which specific type of modification they want to make at the join points. For example, if the expected changes at a join point instance would only happen at the entry and exit of the join points, the "before" and "after" advice pattern would suffice. On the other hand, if the join points are supposed to be modified by replacement or rewriting, the "around" advice pattern would become necessary.

The placeholders may also be used in advice, such as the custom code, i.e. the code block to be woven into the join points. According to the expected changes, domain experts sometimes only want to add a snippet of proven code at the join points, and do not need to worry about its correctness. In this case, we can select an advice pattern that takes the custom code block as plain text, and apply it directly. On the other hand, if the custom code has not been proved, e.g. temporary code from the aspect writers, we can select an advice pattern capable of checking the syntactical correctness of the custom code, according to the grammar of the output language in the target DSCG. In particular, if domain experts would like to use certain domain element directly in the custom code block, we would need to select an advice pattern that uses placeholders in the custom code block to represent the domain specific elements.

Another concern in selecting the DSAL templates is the support of generic pointcuts. Generic pointcuts provide aspect writers an effective way to express the crosscutting concerns. Many aspect languages, such as AspectJ [60] and AspectC++ [100], support generic wildcards in their PCDs. Although they use different syntax, like '+' and '$*$' in AspectJ, and '%' in AspectC++, these generic wildcards are used to define generic pointcuts. For example, AspectJ allows users to define the pointcut shown in Listing 3.4.

```
    pointcut generic_pc() : call(public final * Foo.*(int));
```

Listing 3.4: Using wildcards in the pointcuts of AspectJ

This pointcut captures the invocations of any public final method of Class Foo, which takes in one parameter of int type. The first wildcard symbol '∗' stands for any return type, and the second '∗' stands for any method name. In our templates, we support generic pointcut by either allowing the absence of filters in PCD conditions or explicitly using symbol '∗' or '?' in the basic value type definitions.

Now that we have explained about the basic structure of our DSAL templates, and the general concerns involved in their selection. Back to our automata example, the domain experts want to insert an invocation of function "logging", to each join point instance corresponding to a target transition. In other words, they need to add proven code at the beginning of captured join point. They can select a template with placeholders for domain specific PCD to capture the target join points, i.e. the "Transition" class instances, and the "before&after" advice pattern for the insertion of proven code. The placeholder production rules for the conditional filter in PCD are shown in List 3.5. The complete ANTLR specification of the selected template is shown in List A.6.

```
/* Placeholder: property_filter
 * Use: a conditional expression of
 * the properties of join point class
 * Expansion sample:
 * "filter_property1 operator value_literal"
 * Note:
 * rule "operator" and "value_literal"
 * will be selected according to
 * the type of "filter_property1"
 */
property_filter
    :
    ;

/* Placeholder: filter_property
 * Use: the properties of join point class
 * Expansion sample:
 * "jp_class1_name COLON property1.1_name"
 * "jp_class1_name COLON property1.2_name"
 *                ...
 * "jp_class2_name COLON property1.1_name"
 *                ...
 */
filter_property
```

```
    :
    ;
```

Listing 3.5: The PCD placeholders in the DSAL template selected in the automata example

3.3.2 Generation of DSAL

As explained in Section 3.3.1, once a DSAL template is selected, it can be expanded according to the derived effective meta-model. Due to the diverse modification requirements in different domains, this expansion is normally a manual process. In detail, the template expansion process often includes two steps. First, the placeholders, e.g. for PCD or its filter, are expanded according to the domain specific join point models, which mainly include the join point classes and properties in the effective meta-model M'. Second, there may be some functions or macros in the legacy code, which needs to be used directly in the custom code, e.g. a logging function. Returning to our automata example, the only join point class is the Transition class, with three properties from_state, input and to_state. Accordingly, the placeholders shown in List A.6 are expanded as shown below. In the following discussion, we call this generated DSAL "AspectDFA". The complete ANTLR specification of *AspectDFA* is attached in Listing A.7.

```
property_filter
    :   filter_property1 numerical_comparer INTLITERAL
    |   filter_property2 numerical_comparer INTLITERAL
    |   filter_property3 numerical_comparer INTLITERAL
    ;
filter_property1
    :    'Transition' COLON 'from_state'
    ;

filter_property2
    :    'Transition' COLON 'to_state'
    ;

filter_property1
    :    'Transition' COLON 'input'
    ;
```

Listing 3.6: Production rules created in template expansion

3.3.3 Generation of Traceable Domain Meta-Model

The DSALs to generate in our approach rely on the model traceability in the generated code with regards to the domain specific join points. After generating the DSALs from the templates, we need to guarantee the correct generation of the traceability links in the DSCG process. To do this, we acquire deep insight of the target code generator, and select an appropriate model tracing strategy, which is then composed with the effective meta-model M', to derive a traceable meta-model $\mathsf{M}'_{\mathsf{tr}}$. The derived traceable meta-model is used to guide the update of the code generator, so that the updated code generator will be capable of generating traceability links in the code generation process.

Tracing Strategy Selection To select a proper tracing strategy is the first step to generate a traceable meta-model. It is annotated with $\textcircled{3.1}$ in Figure 3.4. In Section 2.2.4, we introduced a few different traceability link implementations, such as standalone links based on separate XML file and sentinels based on programming language constructs, like comments and annotations.

By the term "tracing strategy", we refer to the strategy to implement the traceability links. From the perspective of DSAL, these links can be regarded as markers of the domain elements in the generated code. The detailed implementation of these links determines how our DSALs can crosscut the code generated from the domain specific models, since the PCDs in the DSALs directly work with these traceability links. As discussed in Section 2.3.6, there are many problems related to the PCDs, such as the *fragile pointcut problem* [71]. In our experiments, we define the traceability links as the profiles of the join points, which contain the static information of the corresponding join points. Such information can be a name mapping between the domain class instances and their corresponding variables in the generated code.

A tracing strategy is basically a definition of a set of class attributes, which often exist as class attribute functions, containing the definition of certain traceability links. A tracing strategy model can be composed with the join point classes in the effective meta-model. Each tracing strategy corresponds to a specific implementation of sentinels, and contains a set of functions defined with string templates to be used in the code generator update. In our experiments, we use String Template [101], a textual-based template expansion technique, to describe the templates, and if possible guide their expansion in the corresponding update of the code generator. It is worth noting that a tracing strategy is language-specific, depending on the output language of the target DSCG.

To decide what specific tracing strategy to use, we need to obtain deep insight of the code generator with regard to the potential effect of the sentinels over the generated code. In detail, there are four major concerns. The first concern is the correctness of the generated code with sentinels inserted. Apart from the basic syntactical and semantical correctness requirement, there are often extra agreements upon the generated code, such as safety certifications [4] and the conformance to predefined library interfaces. The insertion of our sentinels must not break such agreements. The second concern is to minimize the cost of adding the traceability links, which includes the performance impact of generating the traceability links during the code generation process, and the complexity of modifying the implementation of the code generator. The third concern is to minimize the effect of the traceability links to the generated code, in terms of its functional correctness and performance. Sometimes, the choices based on these two concerns may conflict with each other. For example, the comment-based sentinels may generally require lower cost in code generator customization, while the other forms of sentinels may potentially affect the functional correctness of the generated code, as well as its performance. In practice, we may sometimes insert function invocations as sentinels into the generated code. Even though they are idle functions that do nothing inside, their invocation would still change the call stack of the code, and thus result in little performance penalty. The last concern is the limitation from the output language of the target DSCG. For instance, if the output language does not support annotation at all, we obviously cannot choose annotation based sentinels. Sometimes, the consideration over these concerns can lead to a specific choice. For example, if the given DSL host language is Java, and there is no strict time constraints on the generated code, and the performance penalty caused by the invocations of the sentinel functions is not problematic, and a reliable AspectJ weaver is available, it would be easy to reach a consensus on adopting sentinel functions as traceability links. Unfortunately, these factors may be incompatible in other cases and no choice can satisfy all of them. Trade-offs have to be made on ad hoc basis.

Returning to our DFA example, as the code generator is template based, it is not difficult to add sentinel templates directly in its existing code template. As the output language is Java, we can select sentinel template based on annotation, comment, or idle function invocation. Although comment based sentinels may have no impact of the generated code in terms of its correctness and performance, it is not an appealing option to us, since there is, to the best of our knowledge, no existing aspect language that supports Java comments based join point. We finally select a function invocation based sentinel template, as the code generated here is not time sensitive software, like embedded system, the performance hindrance due to the sentinel function invocations will not be a big concern for us. Moreover, using function based sentinels could facilitate the reuse of AspectJ in generating our DSAL. Besides, we can see that the members of the join point class, i.e. Transition, are all of simple type, i.e. String. The code generator is generating separate code blocks for join points with different transition member values. We can thus even include value mapping in our sentinels, so that we do not need to separately check on the member values when try to capture the target join points. The selected tracing methods and their underlying sentinel templates are shown in List 3.7.

```
// public interface
InstanceLevel_EntrySentinel() ::= <<
```

```
String getInstanceEntrySentinel() {
    return "<InstanceLevel_BeginSentinel>";
}
>>
InstanceLevel_ExitSentinel() ::= <<
String getInstanceExitSentinel() {
    return "<InstanceLevel_EndSentinel>";
}
>>

// instance level sentinel templates
InstanceLevel_BeginSentinel(className, memList) ::= <<
Begin___<className>___<memList:{p | memberMapping(<p>)};
   separator="__">
>>
InstanceLevel_EndSentinel(className, memList) ::= <<
End___<className>___<memList:{p | memberMapping(<p>)};
   separator="__">
>>

// for each join point class member, inform the code
   generator to
// trace their domain element name and value
memberMapping(name) ::= <<
<name>_\<prpt_value\>
>>
```

Listing 3.7: The tracing sentinel template selected in the automata example

Composition of Domain Meta-Model with Tracing Strategy Once the tracing strategy is selected, it can be automatically composed with the effective meta-model M'. Their composition is essentially a two step process. The first step is a partial expansion of the selected sentinel templates according to the class specific information of the join point classes in M'. The expansion is not complete here, because some information is only available at generation-time, e.g. the variable name that correspond to a specific domain element instance. The second step is to append the tracing attributes, which are defined with the partially expanded templates, to the join point classes in M'. With these tracing attributes, the target code generator can be updated to generate code that can be traced back to the corresponding domain specific model. Therefore, we call the composed meta-model *traceable meta-model* $\mathsf{M}'_{\mathsf{tr}}$. Returning to our automata example, the implementation of the tracing methods according to the selected tracing strategy is shown in Listing 3.8.

```
// in the "Transition" class
string getInstanceEntrySentinel() {
    return "Begin___Transition___from_state__<prpt_value>
   __to_state__<prpt_value>__input__<prpt_value>";
}

string getInstanceExitSentinel() {
    return "End___Transition___from_state__<prpt_value>
   __to_state__<prpt_value>__input__<prpt_value>";
}
```

LISTING 3.8: The tracing sentinel template selected in the automata example

It is worth noting that $\mathsf{M}' \rightarrow \mathsf{M}'_{\mathsf{tr}}$ is a restricted extension. $\mathsf{M}'_{\mathsf{tr}}$ can be regarded as a transient modelling language that enhances the original modelling language M in a sense that models conforming to M can now be used to generate traceable code. Although $\mathsf{M}'_{\mathsf{tr}}$ only exists within the scope of our DSAL generation framework, it will influence the domain specific code generator in its update to support model traceability, and further affect the code generated with it. To guarantee the correctness of the generated code, we need to ensure the sentinels do not have write access to the rest of the code. Any sentinel implementation in the tracing strategies needs to satisfy this standard.

Code Generator Customization With the traceable meta-model $\mathsf{M}'_{\mathsf{tr}}$, we can customize the target code generator accordingly to enable the model traceability in the code generation process. This step is annotated with (3.2) in Figure 3.4. Unfortunately, as different domain specific code generators may vary greatly in terms of their system architecture designs, implementation technologies, etc, it is very difficult to distill a generalized process or technique to customize domain specific code generators. Yet we can define two general principles in the customization of the code generators. First, the sentinel insertion should be configurable. For example, our customization of the ANTLR parser generator (in Section 5.4.3) only generates the sentinels when the "-trace" option is selected. Ideally, when the sentinel generation option is off, the customized code generator should output exactly the same code as the code previously generated by the original code generator from the same model. Second, provided the sentinels themselves are executable lines, e.g. function invocations, the sentinel code block *MUST* not change the value of any variable defined outside the sentinel code blocks. This can guarantee that the code generated by the customized code generators is functionally identical with the code generated by the original code generator from the same model.

To make the tracing strategy, especially their underlying sentinel templates, more reusable in code generation, we can define a unified model of syntax and semantics for the special placeholders in our sentinel templates, e.g. <*inst_name*>, <*prpt_value*>, etc, which requires expansion in the code generation stage. For example, "*className___<instance_name>*" means "the name of the current instance of domain class '*className*'". Returning to our automata example, the placeholder *<prpt_value>* used in the template of the entry sentinel means "the value of property '*propertyName*' in the current instance". It is worth noting that some generator-specific factors may still affect the generation of the sentinels in the code, even with the traceable domain meta-models as formal guide. An example is the addition of "if" conditional check for the value of backtracking in the ANTLR experiment, on which will be elaborated in Section 5.4.3.

3.3.4 Generation of DSAL Weaver

The last step in our DSCG extension process is to generate a tool that can automatically apply the expected changes described in the DSAL aspects to the base code, which is generated from the base model using the updated code generator. In detail, the tool, i.e. the DSAL weaver, takes as input a DSAL aspect and the target base code, and generates as output the code customized according to the input DSAL aspect. This process can be decomposed into two steps. The first step is the parsing of the DSAL aspect, which includes lexical analysis, syntactical analysis. The second step is the semantic analysis of the DSAL aspect and its application into the base code. To generate a DSAL weaver, is basically to implement the above two functions. The first function is relatively easy to achieve. As mentioned in Section 3.3.1, there are some frameworks capable of generating lexers and parsers automatically from the given language specifications, like the DSAL specifications derived in our approach. To implement the second function, the traceable domain meta-model M′_tr is required for the detailed implementation of the sentinels for each join point class, as the sentinels are often the "real" underlying join points the DSAL weavers capture. There are generally two approaches to do the semantic analysis of the DSAL aspects and weave them into the corresponding base code.

DSAL Weaver Based on Existing Aspect Weaver The first approach is to leverage an existing aspect weaving system, e.g. AspectJ. In this approach, we can choose another aspect language, and define translation rules from our DSAL to the target aspect language. For any aspect written in our DSAL, it can thus be translated into the target aspect language first, and then woven into the base code using an aspect weaver of the target aspect language. In practice, the *Syntax-Directed Translation* (SDT) [102] technique, which binds extra semantics to the correpsonding production rules in the language specification, can be used to translate the PCDs based on the traceability links and their related advices, from the DSAL syntax to the corresponding syntax of the underlying aspect language. Once the DSAL aspect has been successfully translated, we can weave the translated aspect into the base code using its existing aspect weavers. The major advantage of this approach is flexibility. By reusing the existing aspect weaving system, we only need to build translators from our generated DSALs into the underlying aspect languages, which means the implementation of our DSAL weaver does not need to be restricted to a specific technique or framework. As long as there is a valid translation from our DSAL into another aspect language, we can build a translator and reuse an existing weaver. Besides, the translators can be automatically generated through techniques like SDT. In our automata example, we use ANTLR framework to build a translator from our AspectDFA to AspectJ, and then use AspectJ weaver to perform the underlying weaving work. In detail, we modify the ANTLR specification of AspectDFA to attach extra semantics to the relevant production rules for the translation into AspectJ. For example, production rule aspect_statement is expanded as shown in Listing 3.9.

```
aspect_statement
    :   lm=loc_modifier pd=pcd_decl ad=adv_decl
        -> aspectStatement(pcdloc={$lm.pcd}, advloc={$lm.adv
   }, pcd={$pd.st}, body={$ad.st})
    ;

aspectStatement(pcdloc, advloc, pcd, body) ::= <<
<advloc>(): <pcdloc>__<pcd>() <body>
>>

loc_modifier
    :   'before' {$pcd = "Begin"; $adv = "after"}
    |   'after' {$pcd = "End"; $adv = "before"}
    ;
```

Listing 3.9: Production rule aspect_statement with semantic action for translation

With the above syntax-directed semantics, aspect translators from our DSAL to AspectJ can be automatically generated. However, it is possible that there is no valid translation from the generated DSAL into any other aspect language, or there is no existing aspect weaving system that can work with the current output language. In that case, we have to turn to the second approach, to build our own AST rewriter.

DSAL Weaver Built as AST Rewriter The second approach is basically to build our DSAL aspect weavers as a rewriter working at the AST level. There are mainly two components in such a weaver. The first component takes in the parsing results of a DSAL aspect and interprets the corresponding semantics into an AST rewriting strategy, i.e. a collection of rewriting rules over the input base code. With the interpreted rewriting strategy, the second component can walk through the AST parsed from the base code and rewrite it accordingly. We will elaborate on this approach in our case study with the AUTOFILTER code generator discussed in Chapter 4.

Sentinel Location Maintenance in Custom Code Weaving At last, we need to emphasize the importance of maintaining the sentinel locations. Sentinels are the primary model tracing infrastructure in our approach. Their locations directly convey the model tracing information by indicating the boundaries of code block corresponding to certain domain elements. Therefore, it is important that we maintain the proper locations of the sentinels in each custom code weaving operation, so that we can guarantee the validity of the sentinels for the forthcoming aspect weaving request. Let us see a piece of generated code corresponding to a certain join point, as shown in Listing 3.10.

```
...

// Location A
Begin_Sentinel();
// Location B

<code block corresponding to target join point>

// Location C
End_Sentinel();
// Location D

...
```

LISTING 3.10: Problem in maintaining the sentinel locations

If there is an advice trying to insert some custom code into the location "before this join point", the custom code should be inserted into "Location B" instead of "Location A". Otherwise, Begin_Sentinel() would be left in a wrong location after the custom code is woven, as it no longer marks the beginning of the join point related code block. Similarly, the custom code in an after advice should go to "Location C" instead of "Location D". The maintenance of sentinel location is important as it ensures the correctness of applying multiple DSAL aspects to the same base code. In particular, if we build the DSAL weavers on top of existing aspect language weavers, we need to guarantee the translation semantics generate the correct location modifiers.

3.4 Code Modification Through DSAL Aspect Weaving

So far we have generated a DSAL that can crosscut the base model at the domain specific join points selected by the domain experts, and a corresponding weaver that can apply the aspects written in the DSAL, the domain experts can now describe the changes they want in DSAL aspects. It is not difficult for them to learn the syntax and semantics of the generated DSAL, as they already have a general understanding about it during our communication about selecting the DSAL template. Returning to our automata example, the domain experts can now extend the current DFA model to log transitions that satisfy the given conditions. For example, to log any transition from state 1 to state 3, the domain experts can write a DSAL aspect as shown below.

```
aspect sample {
  after (Transition:from_state=1&&Transition:to_state=3) {
    log();
  }
}
```

LISTING 3.11: AspectDFA aspect for logging self transition over state 1

The AspectDFA weaver then translates this aspect into AspectJ. If there is any syntactic issue, the weavers would stop the compilation and emit proper error messages. For example, if we deliberately give a different property name other than the three property names expanded during the DSAL generation step, e.g. replacing "from_state" with "previous_state", the translation process will stop with the following error message.

```
line 2:22 mismatched input 'previous_state' expecting 'from_state'
```

If there is no syntax error, the aspect should be translated into an AspectJ aspect as shown in Listing 3.12.

```
package antlr.DFA;
public aspect sample {
    before(): call(
   End___Transition___from_state__1__to_state__1*()) {
      log();
    }
}
```

LISTING 3.12: Translated AspectJ aspect

It is worth noting that the translated advice location modifier is "before", although it was "after" in the DSAL aspect. This is because the modifier "after" in the DSAL aspect refers to the ending of the domain specific join point, i.e. "transition", which is labeled by our "end sentinel". Therefore, the correct location to insert the custom code is "before" the "end sentinel" instead of "after" it. This is achieved by the corresponding SDT semantics to the AspectDFA grammar, which is shown in List 3.13.

```
aspect_statement
    :   lm=loc_modifier pd=pcd_decl ad=adv_decl
        -> aspectStatement(pcdloc={$lm.pcd}, advloc={$lm.adv
   }, pcd={$pd.st}, body={$ad.st})
    ;
```

```
aspectStatement(pcdloc, advloc, pcd, body) ::= <<
<advloc>(): <pcdloc>__<pcd>() <body>
>>

loc_modifier
    :   'before' {$pcd = "Begin"; $adv = "after"}
    |   'after' {$pcd = "End"; $adv = "before"}
    ;
```

Listing 3.13: SDT semantics in AspectDFA for correct location of custom code insertion

The translated AspectJ aspect will then be passed to a reliable AspectJ weaver, such as the default "ajc" weaver, to finish the weaving process. As AspectJ actually weaves custom code at Java bytecode level, which is not very easy to understand, we present the equivalent customized Java code in Listing 3.14, to give an intuition of the effect of this aspect weaving into the base code.

```
        if (currentState == 0 && input == 1) {

   Begin___Transition___from_state__0__to_state__1__input__1
   (); // begin sentinel
            currentState = 1;

   End___Transition___from_state__0__to_state__1__input__1()
   ; // end sentinel
        } else if (currentState == 1 && input == 3) {

   Begin___Transition___from_state__1__to_state__1__input__3
   (); // begin sentinel
            currentState = 1;
            log();

   End___Transition___from_state__1__to_state__1__input__3()
   ; // end sentinel
        } else if (currentState == 1 && input == 5) {

   Begin___Transition___from_state__1__to_state__2__input__5
   (); // begin sentinel
            currentState = 2;

   End___Transition___from_state__1__to_state__2__input__5()
```

```
; // end sentinel
      } else if (currentState == endState) {
        return;
      } else {
        throw new Exception("invalid state");
      }
    }
```

Listing 3.14: Customized Java code in our automata example

Compared with the manual modification shown in Figure 1.5, our custom code is introduced systematically by the DSAL aspect weavering process. The generated DSAL and the aspect weaver can be reused for similar modification requirements. Besides, our approach does not break the existing MDE tool chain. The simulator program regenerated by the customized code generator is still synchronized with the input model.

In this chapter, we have given an introduction about our meta-model based DSCG extension approach for accommodating changes to the DSCG generated code that cannot be described at model level. Although this approach entails considerable human effort to implement, it provides the capability to formalize the changes and to guarantee the correctness of the corresponding code modification. In the next three chapters, we respectively test our approach in three different real-life DSCG extention scenarios as concrete case studies. In Chapter 4, we extend the generation of state estimators based on Kalman filter algorithms with the code generator AUTOFILTER. In Chapter 5, we extend the generation of LL(*) grammar parsers with the ANTLR parser generator,. In Chapter 6, we extend the generation of LALR(1) grammar parsers with the CUP parser generator.

Chapter 4

DSCG Extension for AUTOFILTER

The first concrete case study for our meta-model based DSCG extension approach is to modify the C code generated by a domain specific code generator called *AUTOFILTER* [15]. AUTOFILTER is a code generator that takes the descriptions of state estimation problems as input domain specific models, and generates the implementation of a process that computes statistically optimal estimates using the *Kalman filter* algorithm [103, 104]. In this chapter, we first introduce the Kalman filter algorithm and the AUTOFILTER code generator. We then extract the domain meta-model and describe the extension scenario. Finally, we illustrate the construction of the DSCG extension system, and demonstrate how to accommodate the expected changes with it.

4.1 The Kalman Filter and The AUTOFILTER Code Generator

4.1.1 The Kalman Filter Algorithm

The *Kalman filter* algorithm was first introduced by Kalman [103], as an algorithm that provides a recursive computational solution to the discrete data linear filtering problems. It addresses the general estimation problem in a discrete time controlled stochastic process. The process to be estimated is influenced respectively by the process and measurement noise. They are assumed to be independent of each other, and to be Gaussian white noise, which means that they are serially uncorrelated and with normal distributions. Formally, the process is governed thus:

$$x_{k+1} = A_k x_k + B u_k + w_k, \tag{4.1}$$

and the measurement vector is

$$z_k = H_k x_k + v_k \tag{4.2}$$

Here the subscript means “at step k”. x denotes the state vector. The length of x is often denoted by n. A, sometimes also noted as F, denotes the n $\times$ n state transition matrix. u denotes the control vector. The length of u is often denoted by l. B denotes the n $\times$ l control matrix, which maps the control to the state variables. w denotes the process noise vector, whose length is also n. z denotes the measurement vector. The length of z is often denoted by m. H denotes an n $\times$ m matrix, which extracts the measurement from the state variables. v denotes the measurement noise vector, whose length is also m.

$$p(w) \sim N(0, Q) \tag{4.3}$$

$$p(v) \sim N(0, R) \tag{4.4}$$

Q denotes an n $\times$ n state variance matrix, which represents the estimation error. R denotes an m $\times$ m measurement variance matrix, which represents the measurement error.

The term “filter” comes from the signal process domain, where it is used to refer to the process that removes some undesirable component or feature from a signal. The Kalman filter estimates the above process as a recursive filter, which predicts the process state at some time and then procures feedback on measurements. In each iteration of the recursion, there are two stages: predict and update. The “predict” stage is also referred to as the “time update” stage. In this stage, it estimates the process state according to the process model, and then calculates an *a priori* error covariance estimate, which is represented as an n $\times$ n matrix P, according to the process model and the process noise model. Here we use the denotation that used by Welch and Bishop [105]. The superscript “−” is used to mark the intermediate results, meaning “in between step k and k+1”.

$$x^-_{k+1} = A_k x_k + B u_k \tag{4.5}$$

$$P^-_{k+1} = A_k P_k A^T_k + Q_k \tag{4.6}$$

The “update” stage is also referred to as the “measurement update” stage. In this stage, the Kalman filter first computes an n $\times$ m matrix K, which is introduced to minimize the estimation error covariance. K denotes the *Kalman gain*, or *blending factor*. It is computed with the state error covariance matrix P, the measurement matrix H, and the measurement error covariance matrix R, according to the following equation.

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \tag{4.7}$$

It then computes the *measurement innovation*, or the *residual*, to evaluate the discrepancy of the predicted measurement, i.e. $H_k x_k$, and the actual measurement z. This innovation is used to compute the *a posteriori* state estimate.

$$x_k = x_k^- + K(z_k - H_k x_k^-) \tag{4.8}$$

Finally, it computes the *a posteriori* error covariance estimate according to the *priori* error covariance obtained in the "predict" stage.

$$P_k = (I - K_k H_k) P_k^- \tag{4.9}$$

To make the whole computation process more clear, we illustrates the corresponding pseudo code in Figure 4.1.

```
⟨Initialization⟩
while(⟨more input available⟩) do
    input $\mathbf{u}, \mathbf{z}$             // $\boldsymbol{u}$ control vector, $\boldsymbol{z}$ measurement vector
    // Prediction
   $\mathbf{x} = \mathbf{Fx} + \mathbf{Bu}$        // $\boldsymbol{x}$ (a priori) state estimate, $\boldsymbol{F}$ process model, $\boldsymbol{B}$ control model
   $\mathbf{P} = \mathbf{FPF}^{\mathrm{T}} + \mathbf{Q}$      // $\boldsymbol{P}$ (a priori) estimate covariance, $\boldsymbol{Q}$ process noise covariance
    // Update
   $\mathbf{y} = \mathbf{z} - \mathbf{Hx}$         // $\boldsymbol{y}$ innovation, $\boldsymbol{H}$ observation model
   $\mathbf{S} = \mathbf{HPH}^{\mathrm{T}} + \mathbf{R}$      // $\boldsymbol{S}$ innovation covariance, $\boldsymbol{R}$ measurement noise covariance
   $\mathbf{K} = \mathbf{PH}^{\mathrm{T}}\mathbf{S}^{-1}$       // $\boldsymbol{K}$ Kalman gain
   $\mathbf{x} = \mathbf{x} + \mathbf{Ky}$         // (a posteriori) state estimate
   $\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{P}$     // (a posteriori) estimate covariance, $\mathbf{I}$ identity matrix
    output $\mathbf{x}$
end
```

FIGURE 4.1: Abstract form of the discrete Kalman filter algorithm.

The Kalman filter algorithm is the basis of mathematical models formulated to solve the estimation problems. It is widely used in solutions estimating the internal states of dynamic systems with a series of observation measurements. To cope with different process models, there are a number of variants of the standard Kalman filter developed, such as the *Extended Kalman filter* [106] and the *Unscented Kalman filter* [107] for the nonlinear systems. Different estimation problems require different mathematical models that involve different variants of the Kalman filter. However, for each formulated mathematical model, it still takes a lot of time and effort to build an associated estimator for testing. This is the motivation behind the AUTOFILTER code generator.

4.1.2 The AUTOFILTER Code Generator

The *AUTOFILTER* code generator is a "knowledge-based" tool developed by Whittle and Schumann [15], which takes as input the estimation problem description, and generates as output the implementation of its estimator based on a properly selected Kalman filter. The input description defines the physical characteristics of the problem in terms of the process model and measurement model. The output implementation can be configured into a number of languages, including C, C++, and Modula 2 [108]. AUTOFILTER significantly reduces the coding effort in generating the implementations of various Kalman filter variants for different problems.It requires no low-level programming skill, such as to "glue" toolkit function invocations. AUTOFILTER also facilitates the prototyping of the estimators for the input problems. The information given in the input specification is used to generate an corresponding simulation or testing. Thus the iterations on the given models can be made quickly and easily.

From the perspective of DSCG, AUTOFILTER can be regarded as a domain specific code generator. It takes as input a mathematical description of a given estimation problem, then selects a specific Kalman filter algorithm schema as its solution, and finally generates the corresponding implementation. Strictly speaking, the solution selection process in AUTOFILTER is a set-valued function. For each input problem specification, there is a specific set of solutions. However, the output can be any solution in the set. In other words, an input description may lead to more than one output solutions. For the sake of simplicity, we make two assumptions. The first assumption is that the C language is only output language AUTOFILTER supports. The second one is that the same solution is always selected for the same problem each time. Based on this assumption, the target domain can be considered as a composition of the state estimation problem domain and their Kalman filter based solution domain. Accordingly, we can regard the input estimation problem and its specific Kalman filter algorithm together as the input model, and the underlying algorithm implementation as the DSCG process.

The estimation problems for AUTOFILTER can be specified in terms of continuous or discrete, linear or nonlinear process and measurement dynamics. The problem domain here encompasses the state vector x, the process model, the measurement model, and some basic controlling properties over the Kalman filter based solutions. AUTOFILTER uses its proprietary problem specification language, i.e. the *Domain Specific Language* (DSL), to describe these elements. As Whittle and Schumann [15] put it, the language "allows the concise specification of the process model, the measurement model, and other important design information."

The input language of AUTOFILTER is not only designed to provide syntax close to the notations that are familiar to the domain experts (i.e. vectors, matrices, and differential equations) to describe the basics of the estimation problems, e.g. the process models and measurement models. It also allows model designers to specify certain details about the desired output estimator, e.g. its name and interface, or how it is initialized with the declared elements, etc. The specification of the input language consists of two parts: the basics of the estimation problems and the information of the desired filter(s). The first part specifies the declarations of the basics (i.e. scalars, vectors, or matrices) of the estimation problems, and the dynamics of the process and measurement. The second part specifies certain information of the estimator code, such as the interfaces and synthesis goal.

Declarations In the declaration part, a domain element can be declared as a scalar, vector, or matrix, with a modifier indicating its usage. const means "used as constant"; data means "used as input data"; absent modifier means "used as variable". The basic data types include nat (natural number), int (integer), double (double-precision floating-point number). For vectors and matrices, the dimensions are specified with a lower and upper bound, which can be arbitrary expressions. The corresponding EBNF specification is shown in Listing 4.1.

```
DECL := [ const | data ] TYPE VAR [ := EXPR ] [ as COMMENT ]
      | IVAR ~ gauss(EXPR, EXPR) ;
TYPE := nat | int | double ;
VAR := NAME | NAME(EXPR .. EXPR) | NAME(EXPR .. EXPR, EXPR
   .. EXPR) ;
IVAR := NAME | NAME(IND) | NAME(IND, IND) ;
IND := '0' | '1'..'9' {'0'..'9'}* ;
COMMENT := STRING+ ;
```

Listing 4.1: Input language specification for the basic declarations of the estimation problems

EXPR represents scalar arithmetic expressions. Vector and matrix are declared in a FORTRAN-style: NAME(EXPR), and NAME(EXPR, EXPR), respectively. IND are all-quantified specification variables to denote generic indices. As some variables are statistical variables that have a distribution, in particular the Gaussian distribution, they are declared as $\mathsf{IVAR} \sim \mathsf{gauss}(\mathsf{EXPR}, \mathsf{EXPR})$. For example, $\mathsf{x}(\mathsf{I}) \sim \mathsf{gauss}(0, \mathsf{sigma}(\mathsf{I}))$ means that each element of the vector x has a zero-mean Gaussian distribution with a standard deviation corresponding to the vector element sigma(I).

Process Models and Measurement Models As both the process model and the measurement models relate to the state vector x, they are specified as a set of equations over the state vectors. For continuous models, differential equations ($\hat{x} = F(x)$, represented as CONT_EQU) are used; a discrete model is given as a set of difference equations ($x_{k+1} = F(x_k)$, represented as DISC_EQU). The corresponding input language specification is shown in Listing 4.2.

```
MODEL_DEF := equation_set NAME has '[' DISC_EQU+ | CONT_EQU+
    ']' '.' ;
DISC_EQU := disc NAME(EXPR) := EXPR ;
CONT_EQU := d/dt NAME(EXPR) := EXPR ;
```

Listing 4.2: Input language specification for the dynamics of the estimation problems

Output Controls and Constraints This part specifies the controlling and constraining information over the desired output, e.g. state vectors or filters, the interface, and additional information, e.g. on the filter initialization. The name of the output Kalman filter is given after the estimator keyword. Then various properties of the filter are specified. The corresponding EBNF specification is shown in Listing 4.3.

```
CONTROL := ESTIMATOR_DECL SET_ATTR+ SYNTH_GOAL ;
ESTIMATOR_DECL := estimator NAME '.' ;
SET_ATTR := 'attribute(' NAME ',' 'steps' ')' '::=' EXPR
    % number of filter execution steps
    | 'attribute(' NAME ',' 'process_eqs' ')' '::=' NAME
    % name of process equation set
    | 'attribute(' NAME ',' 'measurement_eqs' ')' '::=' NAME
    % name of measurement equation set
    | 'attribute(' NAME ',' 'initials' ')' '::=' NAME(IND)
    % initial values state vector
    | 'attribute(' NAME ',' 'initial_covariance' ')' '::='
   EXPR
    % initial covariance matrix
    ;
SYNTH_GOAL := DECL* output_filter FNAME ;
FNAME := NAME | NAME 'par' FNAME ;
```

Listing 4.3: Input language specification for the output code control

This part of the specification mainly defines how to link the problem domain, i.e. the estimation problem domain, and the solution domain, i.e. the Kalman filter domain. The expected output may be a single filter or a number of Kalman filters running in parallel (defined by par). Each output filter is given a name NAME. We can specify the output parameters as part of the synthesis goal. For example, the following input description specifies that we want to generate a Kalman filter named test, along with an output matrix named xhat of type double.

```
double xhat(0..n_statevars-1, 0..n_steps-1).
output_filter test.
```

Listing 4.4: Example of the output code control

For the convenience of the model builders, it also defines how to control the basics of the output code. In fact, the language also supports the definition of operating modes in which different filters run in each mode. However, that is beyond our interest. So we will not cover here.

4.1.3 The Target Domain Meta-Model

The target domain includes the estimation problem domain, as well as the solution domain, i.e. the Kalman filter domain. Accordingly, we define the target domain meta-model as two parts respectively from the given DSL specification and the AUTOFILTER code generator. From the above DSL specification, we can derive the first part of the target domain meta-model in the form of XML schema. Accordingly, the estimation problem domain meta-model consists of the basic types such as "scalar" and "vector", components classes, e.g. the process model and measurement model, and the controls and constraints on a certain Kalman filter. For example, the problem domain class "scalar" in the meta-model is shown in Listing 4.5. The complete meta-model is attached in Appendix B.1.

```
<xs:simpleType name="expr_string_type">
  <xs:restriction base="xs:normalizedString"/>
</xs:simpleType>

  <xs:simpleType name="name_string_type">
  <xs:restriction base="xs:string">
    <xs:pattern value="([A-Za-z_])+"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="basic_data_ref_type">
  <xs:restriction base="xs:string">
    <xs:enumeration value="nat"/>
    <xs:enumeration value="int"/>
    <xs:enumeration value="double"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="role_type">
  <xs:restriction base="xs:string">
    <xs:enumeration value="const"/>
```

```
      <xs:enumeration value="variable"/>
      <xs:enumeration value="data"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="scalar_type">
    <xs:sequence>
      <xs:element name="role" type="role_type"/>
      <xs:element name="name" type="name_string_type"/>
      <xs:element name="element_type" type="
 basic_data_ref_type"/>
      <xs:element name="init_value" type="expr_string_type"
 minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
```

Listing 4.5: The "scalar" class in the estimation problem domain meta-model

Unlike the problem domain meta-model that is explicitly defined in the DSL specification, the rest of the target domain meta-model, i.e. the solution domain meta-model, is formed implicitly in the AUTOFILTER code generator. It includes all participants involved in the Kalman filter construction and the estimations based on them. More specifically, it consists of the calculations of the state estimate vector x and its error covariance P respectively in the "predict" and "update" stage. In detail, the calculations are comprised by several finer grained sub calculations, such as the calculation of the Kalman gain K. There are two points worth noting here. First, as mentioned in Section 4.1.1, the Kalman filters use a two-stage model, i.e. *a priori* state and *a posteriori* state, in the process state estimation. This is reflected in the representation of both the state vector and its estimation error covariance. For instance, the solution domain respectively defines the priori state estimate $\hat{x}^-$, whose value is calculated in the "predict" stage, and the posteriori state estimate $\hat{x}$, whose value is calculated in the "update" stage. Similarly, it defines the prior process covariance P^-, and the posterior process covariance P^+. Second, the solution domain here spans two levels of abstraction. The higher level solution domain defines the abstract elements in the Kalman filter domain, such as $\hat{x}^-$ and P^-, whereas the lower level solution domain specifies the corresponding implementation in the form of C code. The solution domain meta-model in XSD schema is attached in Appendix B.2.

4.2 Extension Scenario

In this section, we first introduce a specific estimation problem, and the corresponding Kalman filter algorithm selected by AUTOFILTER. We then describe the expected changes to the selected Kalman filter algorithm, which cannot be expressed by the target domain modelling language. This scenario will be used as a driving example to demonstrate how we can accommodate such changes using our DSCG extension approach.

4.2.1 The Cruiser Estimation Problem

The input description in our driving example specifies a problem about a craft cruising at constant attitude in a 2D coordinates with random perturbations. We call it “the cruiser problem”. The state vector in the cruiser problem is declared as “x”, where x[0] represent the “angle to X axis of trajectory”, x[1] represents the “X coordinate of craft”, x[2] represents the “Y coordinate of the craft”. The state vector is the major concern in the estimation problem, and the key variable in the corresponding Kalman filter algorithm. Apart from the state vector, the problem description also specify a number of related scalars and vectors as shown in Listing 4.6.

```
const nat n_statevars := 3 as 'Number of state variables'.

double x(0..n_statevars-1) as 'state variables vector'.

double w(0..n_statevars-1) as 'process noise vector'.

const double sigma(0..n_statevars-1) := [0.1,0.1,0.1] as '
   standard deviation of process noise'.
         where 0 =< sigma(_).

w(I) ~ gauss(0,sigma(I)).

double xinit(0..n_statevars-1) as 'initial values of state
   variables'.

data double x_init_noise(0..n_statevars-1) as 'initial state
    noise'.

data double xinit_mean(0..n_statevars-1) as 'initial value
   means'.

xinit(I) ~ gauss(xinit_mean(I), x_init_noise(I)).

const double s:=10.0 as 'Constant speed'.
const double ts:= 1.0 as 'Time step in seconds'.
```

Listing 4.6: Declaration of scalars and vectors in the cruiser problem description

The process model and the measurement model of the cruiser problem is shown in Listing 4.7.

```
/*
  Process model.
*/
equation_set cruising_model has
  [
    disc x(0) := x(0) + w(0),
    disc x(1) := x(1) + ts * s * cos(x(0)) + w(1),
    disc x(2) := x(2) + ts * s * sin(x(0)) + w(2)
  ].
equation_set measurement_model has
 [
  z(0,tvar) := x(1) + v(0),
  z(1,tvar) := x(2) + v(1)
 ].
```

Listing 4.7: The process model and the measurement model in the cruiser problem description

The description finally specifies several controlling properties and constraints over the solution, i.e. the Kalman filter. For example, the "constant speed" s, the "time step in seconds" ts, and "sampling interval" t, etc. Besides, the solution controlling information indicates details such as the name and the initialization of the estimator, etc.

```
estimator test.
attribute(test, update_interval) ::= t.
attribute(test, steps) ::= 100. % note has to be a known
   constant integer, cannot be a symbolic constant. This is
   a deficiency.
attribute(test, process_eqs) ::= cruising_model.
attribute(test, measurement_eqs) ::= measurement_model.
attribute(test, initials) ::= xinit(I).
attribute(test, timevar) ::= tvar.
attribute(test, initial_covariance) ::=
mx(idx(pvar(990), 0, 2), idx(pvar(991), 0, 2),
[[x_init_noise(0),0,0],[0,x_init_noise(1),0],[0,0,
   x_init_noise(2)]]).

double xhat(0..n_statevars-1, 0..n_steps-1).
output_filter test.
```

Listing 4.8: The controlling and constraining information in the cruiser problem description

4.2.2 Extension Requirements

From the abstract form of the Kalman filter shown in Figure 4.1, we can see the code generated by AUTOFILTER is based on matrix computations. However, due to the variety of the Kalman filter algorithm family, the exact number of the involved matrices, as well as the form and order of the matrix computations may vary greatly from one model to another. Worse still, there are some anti-patterns in the code generated by AUTOFILTER. For example, it keeps a bad naming convention of array variables, which constructs array names by adding a positive number to a common prefix "pv". Besides, *magic numbers* [109] are used as matrix sizes and indexes throughout the generated code without much explanation about their meanings. All these factors make it difficult to understand the code generated by AUTOFILTER, despite it being well documented. Simply stated, manual changes to the generated code can be rather difficult and error-prone.

The expected extension is to allow domain experts to monitor the updates of the variables involved in the Kalman filter algorithm, in particular, the state vector x, the estimate error covariance P, and the Kalman gain K, respectively in the "init", "predict" and "update" stages. For monitoring purpose, they require the ability to modify these join points with simple functions in terms of output language statements, e.g. printing and logging. Sometimes, they may also want to check some rare runtime status by tampering with certain join points, e.g. skipping their normal execution. These changes cannot be expressed in terms of the elements in either the estimation problem domain, or the Kalman filter domain. This is a good scenario to test our meta-model based DSCG extension approach. First, the modelling language, i.e. the problem description cannot describe certain details in the solution domain, such as the "predict" and "update" stages. Besides, neither the estimation problem domain elements or the Kalman filter domain elements are capable of supporting the simple functions needed by the domain experts. Second, due to the difficulty in understanding the generated code, manual modification can be quite difficult or even impractical. Third, we cannot directly access the source code of AUTOFILTER. We have to use some formal descriptions, i.e. the meta-models in our approach, to guide the code generator maintainers to modify it for us. This helps to verify that our meta-model based approach is truly independent from the specific domains.

4.3 The Extension of AUTOFILTER DSCG

Following our approach, we first need to analyze the extension requirement to derive the effective meta-model for crosscutting the Kalman filter model and to select a DSAL template for the expected modification.

4.3.1 The Kalman Filter Domain Effective Meta-Model

In this example, the join points involved in the extension requirements are the variables involved in the Kalman filter algorithm, including scalars, vectors and matrices, and the three algorithm stages, i.e. “init”, “predict” and “update”. Accordingly, the join point classes are the “AlgorithmicParticipant” class and the “AlgorithmicStage” class. To specify the corresponding meta-model, we use the *XML Schema Definition* (XSD) language, which can be easily reused later in the generation of the DSAL. In XSD, the meta-model classes are defined as “<xs:simpleType>” or “<xs:complexType>”, and the properties are defined as “<xs:element>”. In particular, the join point classes are annotated with the “IJoinInstance” annotation. The complete specification of the effective meta-model is shown in Appendix B.3.

4.3.2 DSAL Template in SDF

As domain experts are concerned about the correctness of the C code snippet inserted into the base programs through the advice in the generated DSAL. We want to select a DSAL template, in which the custom code in advice can be syntactically checked at the aspect weaving time. In this experiment, we use *syntax definition formalism* to specify the DSAL template, as there is a reusable specification module of the C programming language.

Syntax Definition Formalism *Syntax Definition Formalism* (SDF) is a formalism for the definition of both lexical and context-free syntax [110]. It is a declarative language that allows a concise and natural expression of the syntax of a context-free language. It is modular, and richer than the traditional formalisms, such as BNF and Yacc. It is less restrictive than Yacc, which can generate a parser only if the input grammar falls within the LALR subclass [111]. Besides, it supports disambiguation by applying special purpose facilities in SDF, such as priorities and reject productions.

In this example, the simple functions we need to support in custom code, can be implemented directly as C statements. For example, the printing function can be implemented by calling C function “printf()”. By including the C grammar module, we can have finer-grained definition for the custom code in the advice of our DSAL specification. Another benefit of the SDF modularity is that we can encapsulate all syntax placeholders corresponding to the domain elements in a dedicated, “domain specific” module. We can thus define the rest part of the DSAL template as an independent module, which will include the expanded “domain specific” module later to complete the generation of the DSAL.

Pointcut and Advice Patterns As mentioned in Section 4.2.2, sometimes an expected change may entail thorough modification of a join point, e.g. removing or replacing its corresponding code block in the generated code. Therefore, we need to select a template that supports the "around" advice pattern. This pattern allows the custom code to overwrite the join point code block, or to insert extra code around it. In the latter case, the original code block is normally represented by some predefined syntax in the custom code, e.g. the "proceed();" in AspectJ. As shown in Listing 4.9, the corresponding syntax in our DSAL template is "Base;", which will be parsed as a normal C statement. The complete DSAL template we select is shown in Appendix B.4.

```
 "Base;"                              -> Statement {cons("
BasecodePlaceholderStatement")}
```

LISTING 4.9: The SDF module of our DSAL template

4.3.3 Traceable Domain Meta-Model

Now that we produced the effective meta-model of the target domain, we can now select a tracing strategy and compose it into the effective meta-model to derive the traceable domain meta-model.

Tracing Strategy As mentioned in Section 4.2.2, we have no direct access the source code of AUTOFILTER. But we know that the generated code is time-critical C program. We are thus inclined to use the sentinel pairs based on C style comments. A sentinel pair includes a beginning sentinel that resides at the beginning of the join point code block and an ending sentinel that resides at the end of it. Each sentinel comment consists of three parts, *prefix*, *body*, and *suffix*. Only prefix part differs between the beginning sentinel and the ending sentinel in the same pair. All sentinels share the same suffix. More details of the sentinel template are shown in Listing 4.10.

```
 "/*<JoinPoint-Begin"                 ->
JoinPointSentinelBeginTagPrefix {cons("
JoinPointSentinelBeginTagPrefix")}
 "/*<JoinPoint-End"                   ->
JoinPointSentinelEndTagPrefix {cons("
JoinPointSentinelEndTagPrefix")}
 "/*<Declaration"                     ->
DeclarationSentinelPrefix {cons("
DeclarationSentinelPrefix")}
 "/>*/"                               ->
SentinelCommonSuffix {cons("SentinelCommonSuffix")}
```

LISTING 4.10: Sentinel string in tracing strategy

The body of each sentinel records the stage in which the value of an AlgorithmicParticipant instance is updated and a name mapping of all involved AlgorithmicParticipant instances and their corresponding variables in the generated code. Sometimes, an instance is updated with its old value. For example, $x = Fx + Bu$ in the predict stage. As the AUTOFILTER generate two different variables to represent the previous and current value of x, we keep different entries in the name mapping for them respectively. The entry name for the variable storing the latest value is KeyRole, and the variable storing the old value is ForeRole.

```
 AlgorithmicParticipant ":" Identifier       ->
Participant {cons("Participant")}

 "Stage=\"" AlgorithmicStage "\""    -> StageDescription
{cons("StageDescription")}
 "KeyRole=\"" Participant "\""       ->
KeyRoleDescription {cons("KeyRoleDescription")}
 "ForeRole=\"" Participant "\""      ->
ForeRoleDescription {cons("ForeRoleDescription")}
 StageDescription "," KeyRoleDescription
           -> JoinPointDescription {cons("
CommonJoinPointDescription")}
 StageDescription "," KeyRoleDescription ","
ForeRoleDescription     -> JoinPointDescription {cons("
UpdateJoinPointDescription")}

 JoinPointSentinelBeginTagPrefix JoinPointDescription
SentinelCommonSuffix   -> JoinPointSentinelBeginTag {cons
("JoinPointSentinelBeginTag")}
 JoinPointSentinelEndTagPrefix JoinPointDescription
SentinelCommonSuffix     -> JoinPointSentinelEndTag {cons
("JoinPointSentinelEndTag")}
```

Listing 4.11: SDF definition of the body of sentinels

4.3.4 Generation of Kalman Filter Aspect Language

We call the DSAL to generate here the *Kalman Filter Aspect Language* (KFAL). The generation process is basically an expansion of the produced DSAL template in accordance with the above domain effective meta-model. In detail, we transform the above effective meta-model into an SDF module called `KalmanFilterAlgorithm`, in which the root elements are AlgorithmicParticipant and AlgorithmicStage that respectively correspond to the two join point classes in the effective meta-model with the same names.

The complete SDF specification of `KalmanFilterAlgorithm` module is shown in Appendix B.5.

The general structure of a KFAL aspect is similar to that of an AspectJ aspect. The aspect declaration is followed by a list of PCD and advice, which are wrapped by a pair of braces. The keyword for aspect declaration is "customization" instead of "aspect" in AspectJ. KFAL supports both specific pointcuts and generic pointcuts. A specific pointcut captures the update of a specific AlgorithmicParticipant in a specific stage. In contrast, a generic pointcut captures the update of any AlgorithmicParticipant in a specific stage. Technically, we can easily support an even "more generic" pointcut, which captures the update of any AlgorithmicParticipant in any stage. As domain experts do not think it necessary, we do not support this kind of PCD in KFAL. The SDF definition of KFAL PCD is shown in Listing 4.12.

```
 "$" AlgorithmicParticipant          -> APIdentifier {
cons("APIdentifier")}
 APIdentifier                        -> Identifier {cons(
"APApplication")}

 Identifier                          ->
PointcutDescriptorName {cons("PointcutDescriptorName")}
 AlgorithmicStage                    ->
PointcutDescriptorName {reject}

 AlgorithmicStage                                   ->
GenericPointcutDescriptor {cons("
GenericPointcutDescriptor")}
 AlgorithmicStage "$" AlgorithmicParticipant        ->
SpecificPointcutDescriptor {cons("
SpecificPointcutDescriptor")}

 "pointcut" PointcutDescriptorName ":"
GenericPointcutDescriptor             ->
PointcutDescriptorDeclaration {cons("
GenericPointcutDescriptorDeclaration")}
 "pointcut" PointcutDescriptorName ":"
SpecificPointcutDescriptor            ->
PointcutDescriptorDeclaration {cons("
SpecificPointcutDescriptorDeclaration")}
```

Listing 4.12: SDF definition of pointcut in KFAL

Here, GenericPointcutDescriptor and SpecificPointcutDescriptor specify the PCD for generic pointcuts and specific pointcuts respectively. AlgorithmicParticipant and AlgorithmicStage are imported from module `KalmanFilterAlgorithm`.

```
pointcut spc_update_x : update $x
pointcut gpc_predict_any : predict
```

Listing 4.13: Pointcut examples of KFAL

Listing 4.13 shows two PCD examples in KFAL. The first pointcut, whose name is prefixed by "spc" (Specific PointCut), is a specific pointcut that captures the join points updating the state estimate vector x in the update stage. Note that we use a special character $ before the name of any AlgorithmicParticipant in KFAL, to indicate it as a domain element instead of a common variable in the generated C code. This helps to avoid ambiguity caused by name conflicts. The second pointcut, whose name is prefixed by "gpc" (Generic PointCut), is a general pointcut that captures the join points updating any traced domain element, i.e. the state estimate vector x and the estimate covariance matrix P, in the predict stage.

Similar to AspectJ, KFAL supports three advice patterns, *before*, *after* and *around*, which allows aspect writers to insert custom code either before a specific join point or after it, or simply to replace the whole join point with the given custom code. Specifically, *around* advice can also insert custom code to both the start and the end of a specific join point, without removing the original code block. In that case, the original code block is denoted by keyword Base in advice code. The corresponding SDF definition is shown in Listing 4.14.

```
 "before"                            ->
CrosscutPatternModifier {cons("before")}
 "after"                             ->
CrosscutPatternModifier {cons("after")}
 "around"                            ->
CrosscutPatternModifier {cons("around")}
 CrosscutPatternModifier             -> Keyword {cons("
CrosscutPatternModifier")}

 Keyword                             -> Identifier {
reject}

 "Base"                              -> Keyword {cons("
BasecodePlaceholder")}
 CrosscutPatternModifier PointcutDescriptorName "()" "{"
Statement "};"       -> AdviceBlock {cons("
PointcutNameBasedAdvice")}
 CrosscutPatternModifier GenericPointcutDescriptor "()" "
{" Statement "};"    -> AdviceBlock {cons("GenericAdvice")
}
 CrosscutPatternModifier SpecificPointcutDescriptor "()"
"{" Statement "};"  -> AdviceBlock {cons("SpecificAdvice"
)}
 "Base;"                               -> Statement {cons("
BasecodePlaceholderStatement")}
```

Listing 4.14: SDF definition of advice in KFAL

From the above SDF definition, we can see that Base; is recognized as a Statement, which is imported from module `Default-C`. The term represents a common C statement. On this matching, a node called "BasecodePlaceholderStatement" will be created in the parsed syntax tree, whereas the nodes created by the other "normal" C statements all have different names. This is defined by syntax cons("BasecodePlaceholderStatement"). Based on this difference, our rewriting rule can distinguish this placeholder from normal C statements, so that we can replace it with the original code block. For the sake of the simplicity, KFAL does not support logic combination of multiple crosscutting conditions. As for advice declaration, KFAL supports referencing a defined pointcut by its name, as well as directly embedding a pointcut definition. For example, both of the two code blocks shown in Listing 4.15 insert some custom code before initializing the process model F.

```
before pcd1 () {
    ...
};
    ...
};
```

Listing 4.15: Two ways to link advice with pointcut in KFAL

Two pieces of advice examples are shown in Listing 4.16.

```
after predict $x () {
    log_current_var($x);
    log_current_var($P);
};
around init $H () {
    before_init_H();
    Base;
    after_init_H();
};
```

Listing 4.16: Advice examples of KFAL

The first piece of advice logs both the state estimate vector x and the covariance matrix P after each prediction of x. The second piece of advice reuses two functions before_init_H()

and after_init_H() from the legacy code around the join points initializing the observation model H.

4.3.5 KFAL Weaver Generation with Stratego/XT

To the best of our knowledge, there is no existing aspect weaver that supports crosscutting C programs by C style comments. Therefore, we build KFAL aspect weaver as an AST rewriter which accepts the base code and the KFAL aspects as input, and generates the customized code according to the rewriting strategies specified in the KFAL aspects. More specifically, the KFAL weaver first parses the input base code into corresponding ASTs, then traverses them and applies the detailed rewriting strategies interpreted from the advice in the input KFAL aspects, it finally restores the rewritten ASTs back into customized code.

The Stratego/XT framework provides a complete tool chain to generate parsers according to given SDF specifications, and generate AST rewriters with Stratego rewriting rules. Simply stated, parse tables are generated for built-in generic parsers, so that any program written in the languages defined by the given SDF specifications can be parsed into corresponding tree structures, i.e. AST or CST. The parsed tree structures are then rewritten according to the given Stratego rewriting strategies, and finally pretty-printted back to programs to finish the modification, i.e. aspect weaving process.

SDF to Parse Table First, we use tool `sdf2table` to automatically generate parse table `KFOL.tbl` from the SDF definition of the updated output language, i.e. `KFOL.sdf` and parse table `KFAL.tbl` from `KFAL.sdf`. The generated parse tables can then be used as input to a generalized LR parser `sglri`, which checks the syntactic correctness of the input base code and KFAL aspects by parsing them into structured ASTs. Further discussion about LR parsing can be found in Section 6.1.2. Strictly speaking, parsing via `sglri` is a two-step process. It first invokes the base parser `sglr` to parse the input programs into *concrete syntax trees* or *parse trees*. It then implodes the parse trees into the corresponding ASTs represented in *ATerm*, or more precisely, AsFix, by tool `implode-asfix`. *Annotated Term* (ATerm) format [112, 113] is a format for exchanging structured data in Stratego/XT. The ATerm format is a generic internal and external representation of data by means of simple prefix terms. AsFix [114], namely "ASF+SDF fixed format", is a format for representing parse trees in the ATerm format. In detail, there are two kinds of AsFix: *AsFix2ME*, a compact version with lists, layout and literals flattened, and *AsFix2*, a very verbose version with a structured representation of its content. Specifically, the ASTs generated by `sglri` is by default in the format of AsFix2.

Stratego Rewriting Rules Construction The seamless support of SDF in the Stratego/XT framework is reflected in the convenient transformation from SDF definition to Stratego signature. As the basis of the rewriting transformation, the rewriting rules derived from KFAL aspects need to recognize the comment sentinel structures, as well as the traced Kalman filter domain elements. The Stratego/XT framework provides the tool `sdf2rtg` to generate abstract *Regular Tree Grammar* (RTG) from SDF definitions, and the tool `rtg2sig` to generate the corresponding Stratego signatures from the generated RTGs. As for KFAL, we generate the Stratego signature module (`KFAL.str`) from the SDF module (`KFAL.sdf`), we then import it as the basis of the AST rewriting module. The second step of KFAL weaver generation is to construct Stratego rewriting rules to interpret the input KFAL aspects into a series of rewriting rules that can be applied during a traversal of the ASTs generated from the input base code. In general, Stratego rewriting strategies follow a "first match, and then apply" pattern. Therefore, we define a number of Stratego rewriting rules to translate different kinds of KFAL pointcuts and advice into the corresponding Stratego rewriting rules. In detail, we define a series of fundamental rewriting strategies for each advice pattern we support. Back to our cruise filter example, as our advice model supports *before*, *after* and *around* patterns, we define the corresponding Stratego rewriting strategies as shown in Appendix B.6.

The SpecificAdviceApplication strategy shown above is a combination of Stratego built-in rewriting strategies, such as oncebu, which applies its parameter strategy at one position in an AST, and our customized strategies, such as SpecificAdviceApplicationBefore, which merges the custom code block represented by parameter customcode and the code block wrapped by comment sentinels matching both stage information represented by algostage and involved domain elements information represented by algoparticipant and participantlist into a new AST branch, and replace the branch of the original code block with it. There are three points worth noting here. First, during this merging process, any domain element name will be replaced with the name of its corresponding variable in base code. For example, the observation model H will be replaced with variable name h_text. Second, only the domain elements originally involved in a certain join point are visible in extension. Any other domain elements are regarded as "undefined", and their names cannot be properly replaced with the corresponding variable names. For example, only the observation model H is visible in the scope of the join point initializing it. If we try to modify the innovation vector y, it will not be replaced with the proper variable name, so that a compilation error will be thrown when we compile the customized code. This design is to prevent potential tampering with irrelevant domain elements in certain join points. Third, the advice application strategies apply to the advice defining the PCDs directly instead of referencing by their names. The relevant rewriting strategies are shown in Appendix B.7.

In short, the SpecificAdviceApplicationBefore rewriting strategy modifies the custom code by proper name replacements of the domain elements involved, and then inserts it into the AST branch of the target join points at its beginning. Similarly, we define the strategies for *around* and *after* patterns, as well as the strategies for *general advice*. On top of these strategies for advice application, we define strategies traversing the ASTs of the input KFAL aspects and translating the "pointcut-advice" pairs into Stratego rewriting strategies against the ASTs of the input base code, which match the join point conditions specified in the pointcuts and then apply the corresponding advice application strategies. The Stratego rewriting rules that translate the KFAL pointcut-advice pairs are shown in Listing 4.17.

```
PointcutReplacer :
([],list) -> list

PointcutReplacer :
([pt | pts],list1) -> list3
where list2 := <PointcutReplacer> (pt,list1);
list3 := <PointcutReplacer> (pts,list2)

PointcutReplacer :
(SpecificPointcutDescriptorDeclaration(
 PointcutDescriptorName(pdn), spd), advlist1) -> advlist2
where advlist2 := <map(try(SpecificPointcutReplacer(|pdn,
 spd)))> advlist1

PointcutReplacer :
(GeneralPointcutDescriptorDeclaration(
 PointcutDescriptorName(pdn), gpd), advlist1) -> advlist2
where advlist2 := <map(try(GeneralPointcutReplacer(|pdn,
 gpd)))> advlist1

SpecificPointcutReplacer(|pdn,spd) :
PointcutNameBasedAdvice(wp,PointcutDescriptorName(name),cc
 ) -> SpecificAdvice(wp,spd,cc) where equal(|name,pdn)


GeneralPointcutReplacer(|pdn,gpd) :
PointcutNameBasedAdvice(wp,PointcutDescriptorName(name),cc
 ) -> GeneralAdvice(wp,gpd,cc) where equal(|name,pdn)
```

Listing 4.17: Stratego rewriting rules translating KFAL pointcut-advice pairs

The above strategies traverse the ASTs of the input KFAL aspects for the advice list, and replace every pointcut name involved in advice with its definition, so that the advice application strategies like SpecificAdviceApplication can be applied. The strategies shown in Listing 4.18 distribute the proper advice application strategies to each advice in the list.

```
CustomizationApplier :
([], pl, code) -> code

CustomizationApplier :
([task | tasks], pl, code1) -> code3
where code2 := <CustomizationApplier>(task, pl, code1);
code3 := <CustomizationApplier>(tasks, pl, code2)

CustomizationApplier :
(SpecificAdvice(wp,SpecificPointcutDescriptor(as,ap),cc),
 pl, code1) -> code2  where code2 := <
 SpecificAdviceApplication(|wp,as,ap,cc,pl)> code1

CustomizationApplier :
(GeneralAdvice(wp,GeneralPointcutDescriptor(as),cc), pl,
 code1) -> code2  where code2 := <GeneralAdviceApplication
 (|wp,as,cc,pl)> code1
```

Listing 4.18: Stratego rewriting rules applying proper advice application strategies

All above Stratego rewriting rules implement two major functions. The first one is to traverse the ASTs of the input KFAL aspects to load the appropriate advice application strategies. The second one is to traverse the ASTs of the input base code to apply each loaded advice application strategy. It is worth noting that the advice in KFAL aspects will be parsed and applied to base code ASTs in the order they appear. This is important as it avoids the ambiguity in applying multiple advice to the same join point. We will show a detailed example later in Section 4.4. Now that we have finished the construction of all needed rewriting rules. With tool `strc`, we compile them into a single executable, which we call *KFRewriter*. It is worth noting that the KFRewriter can recognize and manipulate ASTs derived from both base code in KFOL and aspects in KFAL, as the imported Stratego signature module `KFAL.str` contains all the language construct definition of KFOL in it.

Pretty Printing in XT In Stratego/XT, the process unparsing ASTs back to code text is often referred to as *Pretty Printing*. The XT tool set contains a *GPP* package, which is a tool suite for generic pretty printing. GPP supports pretty printing of ASTs in AsFix format into text in a number of output formats, including plain text, LaTeX and HTML. Similar to the two-step code-to-AST parsing process, the pretty printing process also consists of two step process. In the first step, the ASTs in AsFix format are unparsed by tool `ast2abox` into intermediate representation in an internal format called the Box language [115]. The unparsing step requires pretty print tables to specify how the target language constructs need to be pretty printed. The Stratego/XT framework shows its hospitality again to SDF language definition. It provides the tool `ppgen` to generates pretty print tables directly from SDF definitions. We use it to generate the unparse table of KFOL, `KFOL.pp`, from `KFOL.sdf`. In the second step, the unparsed box representations are further translated into code text by tool `abox2text`.

KFAL Aspect Weaving So far we have explained the generation of all the components required in the KFAL aspect weaving process. The final step of the KFAL aspect weaver generation is to get these components to work together. We write a bash script we call `KFALWeaver.sh` to integrate the whole weaver system. The KFAL aspect weaving process consists of three steps. First, the input base code is parsed into AST in AsFix format with the help of the corresponding parse table `KFOL.tbl`. The XT command to parse the base code into ASTs is shown in Listing 4.19.

```
sglri -i Input/Aspect.akf -p Utility/KFAL.tbl -o tmp/Aspect.
   trm
```

LISTING 4.19: XT command parsing base code into ASTs

Similarly, the input KFAL aspects are also parsed into ASTs. A little trick here is to concatenate the ASTs of both base code and KFAL aspects into the intermediate file storing the base code ASTs, so that our core rewriter `KFRewriter` can handle them from a single file. Second, the underlying rewriter takes the merged ASTs and launches the traversal for a two-step rewriting, i.e. loading required advice application strategies from KFAL aspect ASTs and then applying them to base code ASTs. The XT command to rewrite the base code ASTs is shown in Listing 4.20.

```
Utility/KFRewriter -i tmp/BaseCode.trm -o tmp/CustomizedCode
   .trm
```

LISTING 4.20: XT command rewriting base code ASTs

Finally, the rewritten ASTs go through a two-step pretty printing process to be unparsed into code text with the woven custom code. The XT command to pretty print the rewritten ASTs in Listing 4.21.

```
ast2abox -p Utility/KFOL.pp -i tmp/CustomizedCode.trm -o tmp
   /CustomizedCode.abox
abox2text -i tmp/CustomizedCode.abox -o Output/
   CustomizedCode.c
```

LISTING 4.21: XT command pretty printing rewritten ASTs

4.4 KFAL Aspect Application in Cruise Filter

So far we have generated the DSAL for Kalman filter domain, i.e. KFAL and its AST rewriting based aspect weaver. To complete our case study, we write a testing aspect in KFAL and apply it to the Kalman filter implementation generated by AUTOFILTER in the cruise filter example described in Section 4.1.3, and then check the modified program for the weaving result. We basically include five testing purposes in this aspect. The first purpose is to verify that both ways of referencing PCDs in advice work. The second purpose is to ensure both specific pointcut and generic pointcut work. The third purpose is to check whether multiple advice are applied to the same join point in the same order as they are defined in the example KFAL aspects. The fourth purpose is to verify that the model element names are properly replaced with the corresponding variable names in the custom code. The fifth purpose is to verify that the *around* advice can completely replace the captured join points with custom code. The testing KFAL aspect is shown in Listing 4.22.

```
customization sample {
    pointcut spt_predict_P : predict $P
    pointcut spt_predict_x : predict $x
    pointcut spt_update_P : update $P
    pointcut gpt_predict : predict

    before spt_predict_P () {
        SET_ZERO($P); // A macro to set all elements of the
   input parameter as "0".
    };

    // A generic advice, which is to append a time logging
   after any "predict" process.
    before predict () {
        log_current_var($H);
    };
    // A generic advice, which is to append a time logging
   after any "predict" process.
    after gpt_predict () {
        log_current_var($x);
        log_current_var($P);
    };

    around init $H () {
        before_init_H();
        Base;
        after_init_H();
    };

    after spt_update_P () {
        printf("estimation error: %f", norm($P));
    };

    around init $Q () {
        SET_ZERO($Q);
    };
}
```

LISTING 4.22: KFAL aspect to extend the cruise filter example

As we can see, the first piece of advice references a specific pointcut by its name spt_predict_P, whereas the second piece of advice directly declares a general pointcut. Besides, the first two pieces of advice are applied to the same join point, i.e. predicting the estimate covariance matrix P. In the third advice, we deliberately use the irrelevant domain element x in the join points predicting P. For this, we define the last and third-to-last advice to test these two different applications respectively. Besides, the SET_ZERO macro from legacy code is to set all elements of its parameter to 0. The norm method is also from legacy code, which returns the estimation error according to the given covariance matrix. Our aspect example diversifies as much as possible in the extension requirements that KFAL supports. It covers both specific and general pointcuts, and all three patterns of advice.

4.4.1 Aspect Weaving Result

The test environment of our KFAL weaver is `Ubuntu 12.04.3 LTS`, with three core packages installed for Stratego/XT: `aterm-2.5`, `sdf2-bundle-2.4` and `strategoxt-0.17`. We apply the example aspect shown above to the base code generated by AUTOFILTER in the cruise filter example, and then verify the weaving result by manually checking and compiling the customized code generated by the KFAL weaver script. The customized code gives positive answers to all our test concerns. Take the code block predicting P for example, as all the first three pieces of advice is applied to it. The corresponding base code is shown in Listing 4.23.

```
    /*<JoinPoint-Begin Stage="predict",KeyRole="P:
  pminus_test",ForeRole="P:pplus_test"/>*/
    // Calculation of matrix transpose
    pv92[0 ][ 0] = phi_test[0 ][ 0];

      ...
    pminus_test[2 ][ 2] = q_test[2 ][ 2] + pv90[2 ][ 2];
    /*<JoinPoint-End Stage="predict",KeyRole="P:pminus_test
  ",ForeRole="P:pplus_test"/>*/
```

Listing 4.23: Base code of join point predicting P

There are two points to explain. First, any comment in the base code, except for our comment sentinels, is discarded in the aspect weaving process according to our grammar `KFOL.sdf`. Second, there are many extra white spaces inserted in the customized code, which are introduced in the pretty printing process due to the default Box language translation rules. As they do not affect the compilation of the customized code, we simply leave it untouched. The customized code is shown in Listing 4.24.

```
    /*<JoinPoint-Begin Stage=" predict " , KeyRole=" P :
  pminus_test " , ForeRole=" P : pplus_test " />*/
    log_current_var ( h_test ) ;
    SET_ZERO ( phi_test ) ;
    pv92 [ 0 ] [ 0 ] = phi_test [ 0 ] [ 0 ] ;

    ...

    pminus_test [ 2 ] [ 2 ] = q_test [ 2 ] [ 2 ] + pv90 [
  2 ] [ 2 ] ;
    log_current_var ( $ x ) ;
    log_current_var ( pminus_test ) ;
    /*<JoinPoint-End Stage=" predict " , KeyRole=" P :
  pminus_test " , ForeRole=" P : pplus_test " />*/
```

Listing 4.24: Customized code of join point predicting P

The first piece of advice inserts the SET_ZERO macro at the beginning of the code block, and then the second piece of advice inserts a statement logging H also at the beginning of it. Therefore, it comes before the inserted macro. This result proves that multiple pieces of advice are actually applied in the order they are defined in KFAL aspects. The third advice applied here defines custom code involves both x and P. As this join point predicts only estimate covariance P, its name is replaced with the corresponding variable name pminus_test, whereas x remains the same in the woven code, which would lead to a compilation error indicating x is undefined. In summary, the KFAL aspect weaving system successfully extends the DSCG of AUTOFILTER for a model, i.e. the cruise filter, as we expected.

In this chapter, we illustrate that our DSCG extension approach can successfully accommodate modifications that previously cannot be described by the target DSL to AUTOFILTER generated code. With the help of the Stratego/XT framework, we implement a complete system to establish proper tracing mechanism on certain elements in the Kalman filter domain, and to generate a DSAL called *KFAL* and its aspect weaver on top of the meta-model tracing system. We then test the whole KFAL based extension system with the cruise filter example by writing a test KFAL aspect with diversified pointcuts and advice and then verifying the customized code generated by the aspect weaving process. The test result shows the feasibility of our meta-model based DSCG extension approach, in a circumstance where we aim at a very specific target domain and build our aspect weaver as an AST rewriter.

Chapter 5

DSCG Extension for ANTLR

As a second case study, we test our approach within a well-established domain, the parsing of *Context Free Grammar* (CFG) languages.There are a number of different parsing algorithms involved in this domain. According to the order of grammar rule application, they can be mainly categorized into *Left-to-right Leftmost* (LL) parsing algorithms and *Left-to-right Rightmost* parsing algorithms, on which we will elaborate in Section 5.2. There are many parser generators developed with different techniques for each kind of parsing algorithm. In this case study, we focus on one of the most widely used parser generators, ANTLR [16, 96, 97].

5.1 Context Free Language Parsing

A language, as conceived by Chomsky [116], is "a set (finite or infinite) of sentences, each finite in length and constructed out of a finite set of elements", which have "a finite number of phonemes (or letters in its alphabet) and each sentence is representable as a finite sequence of these phonemes (or letters)". Instead of the loose concept of general languages, the term "language" in our discussion refers to the formal languages, in particular, those involved in the computer science domain, which may include programming languages, modelling languages, representation languages, etc. According to Earley [88], a formal *language* is "a set of strings over a finite set of symbols". A grammar of a language can be considered as a formal device that specifies the symbol sets that are allowed in the language. According to Aho et al. [117], a grammar defines a language by imposing a structure on each sentence in the language. In computer science, formal grammars can be categorized into two types according to whether their rules can be applied regardless of context. In Section 3.1.1, we have introduced *Context Free Grammar* (CFG), which is widely used to specify formal languages. A formal languages specified by CFG is often referred to as *Context Free Language* (CFL). In this case study, the target DSCG is exactly the generation of the parsers for context free grammars. On the other hand, a production rule of a formal grammar can be constrained to a specific context, e.g. after matching another production rule, such a grammar is a *Context Sensitive Grammar* (CSG). In practice, CSG is seldom used as there is no efficient context sensitive recognition algorithm, whereas CFGs describe formal languages in a way that their sentences are built recursively from smaller terms. This allows the construction of efficient parsing algorithms, and CFG parsing becomes popular and arouses much research effort in computer science. The reason we choose CFG parsing as the target domain is because this is a well-established domain, and many parsing algorithms have been invented and improved. Moreover, a number of mature parser generation tools are available to us as potential code generators in our experiments.

5.1.1 CFG Parsing Algorithms

According to Aho et al. [40], the compilation process comprises seven phases. The first phase is called *lexical analysis*, which transforms the given input text into a string of tokens, i.e. terminals, defined in the formal grammar. The second phase is called *syntactic analysis* or *parsing*, which analyze the transformed token string. Note that all formal grammars involved in our discussion are CFGs unless specifically indicated.

A common problem encountered in CFG parsing is *ambiguity*. Given the fact that a sentence of a language specified by CFG can be constructed by applying a sequence of production rules to the start symbol, a CFG is regarded to be ambiguous if we can find more than one sequence for a specific sentence. In other words, in an unambiguous grammar, a syntactically correct sentence can be derived from the start symbol by applying only one specific sequence of production rules. A rule application sequence is called a *derivation*. In each step of a derivation, the start symbol is rewritten into an updated string, which may include both terminals and nonterminals. Such a string is often called a *sentential form*. Obviously, a sentence is a sentential form containing only terminals.

From this perspective, the purpose of parsing is exactly to identify the specific derivation of an input sentence, e.g. a program, according to the given grammar, and to return the reverse of the identified derivation, which is often called a *parse*. Obviously, the basis of a successful parsing is that the grammar must be unambiguous. Note that all grammars involved in our discussion are unambiguous unless specifically indicated.

The goal of a parsing algorithm is thus to construct a parsing system according to a given grammar, and find out which production rules are applied in building the input string and the application order. There are many parsing algorithms developed for CFGs, among which two groups of parsing algorithms have become entrenched, i.e. *LL* parsing and *LR* parsing. In this chapter, we focus on *LL* parsing algorithms. We will elaborate on *LR* parsing algorithms in Section 6.1.2.

LL Parsing Each LL parsing algorithm is a top-down, recursive descent parsing algorithm. The first *L* in its name means that an LL parser scans the input string from "**L**eft to right". The second *L* indicates an LL parser's preference for "**L**eftmost derivation". "Top-down" is a conceptual description of the parsing process, in which parse trees are constructed in a "top-down" pattern, i.e. from roots (start symbols) to leaves (terminals). "Recursive descent" describes two features of the parsing process. "Recursive" stands for the recursive applications of CFG production rules in the parsing process. "Descent" indicates that the parsing is actually based on *Depth First Search* (DFS) on derivation trees. Obviously, the parsing will be in infinite loop if a production rule of the given grammar is "left recursive". Therefore, LL parsers normally do not support left recursive grammars by default. In the parsing process of an LL parser, it scans one token from the input string at each time, and then applies an appropriate production rule according to the scanned token based on some selection strategy.

The original LL parsing algorithm relies on sheer luck, and there is no production rule selection strategy at all. The parser simply performs a DFS and backtracks whenever hitting an unexpected sentential form due to some wrong application of production rules. To reduce the performance penalty caused by backtracking, the concept of *predictive parsing* is developed to avoid backtracking. The trick is to build a parse table according to the given CFG, assuming the parser can always look ahead to a fixed number of tokens that follow the current token that it scans, to predict the current state, which is essentially decided by the remaining input. By convention, predictive LL parsing algorithms are collectively called *LL(k)* (k>0), where *k* stands for the number of lookahead tokens. The original LL parsing algorithm is thus denoted as *LL(0)*. Here we use a very simple CFG and the *LL(1)* parsing algorithm to give an intuition about how a predictive parsing algorithm works. The EBNF representation of the example CFG is shown in Listing 5.1.

```
Expr := 'int' ;                    (* Rule 1 *)
Expr := '(' Expr Op Expr ')' ;        (* Rule 2 *)
Op := '+' ;                        (* Rule 3 *)
Op := '*' ;                        (* Rule 4 *)
```

Listing 5.1: The example grammar

The parse table of this *LL(1)* grammar can be constructed as shown below.

	int	(	)	+	*
Expr	Rule 1	Rule 2	–	–	–
Op	–	–	–	Rule 3	Rule 4

Each row of the parse table represents a specific LHS nonterminal in one of the production rules. Each column stands for the first token of a certain RHS in one of the production rules. The table values are calculated according to the *FIRST* set and the *FOLLOW* set of each LHS nonterminal, which are computed with a *transitive closure algorithm* (or a *fixed point iteration*). More details of the computation of these sets can be found in the famous book about compilers by Aho et al. [40]. These details are outside the scope of this work and are not further discussed here. We only need to know that we are able to generate this kind of parse tables from an unambiguous LL(1) grammar. An LL(1) parser maintains a token stack S to record its internal states and an index p pointing to an input string to mark the parsing progress. Before each parsing process, the parser initializes S with the start symbol and set p pointing to the left of the first token of the input string. The parser then performs a recursion containing the following steps.

Step 1 If p is pointing to the end of the input string, parsing is complete.

Step 2 Look ahead to the first input token lat to the right of p.

Step 3 Check the top token t in S. If t is a terminal, go to **Step 4**. Otherwise, go to **Step 5**.

Step 4 Match t against las. If successful, pop t from S, move p to the right of las, and then skip the current iteration. Otherwise, report error and end the parsing process. This step is often called a *match* step.

Step 5 Look for a cell in the parse table, whose row matches t and whose column matches las. If the search is successful, the cell value r is the production rule to apply according to the prediction. Otherwise, report error and end the parsing process. This step is often called a *predict* step.

Step 6 Pop t from S and move t to the right of las. This step is in fact a simulation of the application of production rule r.

Apparently, the more tokens a parser can look ahead, the more accurate its prediction would be. In the extreme case, if all the remaining tokens were kept as lookahead tokens, "prediction" would actually become "lookup" in the whole state space with 100% accuracy. However, a rising number of lookahead tokens does not necessarily result in more effective parsing. The increase of the number of lookahead tokens will lead to exponential increase of the cell number of parse tables and incredibly complicate the parser, not to mention the fact that real life parsing requires arbitrary lookahead as the length of an input string is normally unpredictable. Besides, with the increase of k, stronger restrictions are implicitly enforced to CFGs that can be parsed by *LL(k)* algorithm, which in return makes *LL(k)* less expressive.

In summary, *LL(0)* and *LL(k)* (k>0) are two types of conventional LL parsing algorithms. By allowing backtracking, *LL(0)* accepts all but left recursive CFGs, while suffers from performance penalty in parsing. By enforcing looking ahead to a fixed number i.e. k, of following tokens and disabling backtracking, *LL(k)* avoids performance penalty, at the cost of accepting fewer CFGs and increased complexity of parser.

5.1.2 LL(*) Parsing

So far we have explained the advantages and disadvantages of the two types of traditional LL parsing algorithms. *LL(0)* is simple and expressive, but produces slow parsers; whereas *LL(k)* (k>0) produces fast parsers, but is more complex and less expressive. As an improvement, another LL parsing strategy called *LL(*)* [118] is proposed. It matches the lookahead tokens with regular expressions instead of using linear search of token sequences in prediction of production rules. When predicting an appropriate production rule for an LHS nonterminal, *LL(*)* parsing relies on a *Deterministic Finite Automaton* (DFA), which can be cyclic to deal with arbitrary lookahead tokens. The design of DFA based lookahead in *LL(*)* does not only make specifying lookahead depth unnecessary as in *LL(k)*, but also improves the performance of an *LL(*)* parser. Compared to backtracking with full parser, which entails method invocations for production rule applications and unrolling their effects, backtracking of lookahead DFAs in *LL(*)* is more lightweight, in terms of simpler implementation and better performance.

These lookahead DFAs in *LL(*)* are constructed by the grammar analyzer in ANTLR in a heuristic way. A valid LL(*) DFA has the following properties.

- All states are reachable.
- All states can reach an accept state.
- At least one accept state can be reached for each alternative.

The DFA construction would be terminated when there is recursion in at least two alternatives for a certain LHS nonterminal. On such an unsuccessful attempt, which is rare according to empirical results [118], *LL(*)* would fail over to *LL(1)* by enabling backtracking.

5.2 The ANTLR Parser Generation Framework

In this case study, the target DSCG is the ANTLR parser generation process, which takes an ANTLR specification of an *LL(*)* grammar as input, and generates a lexer and a parser as output. In this section, we introduce the key artifacts involved in the ANTLR parser generation framework.

5.2.1 ANTLR

The domain specific code generator here is called ***AN****other* ***T****ool for* ***L****anguage* ***R****ecognition*, *ANTLR* [16, 96, 97, 118]. As the successor to the *Purdue Compiler Construction Tool Set* (PCCTS) [119], it was first developed by Professor Terence J. Parr in 1989 for *LL(k)* parser generation [16]. It is maintained by Parr since then and is under active development. With the development of *LL(*)* parsing strategy, ANTLR becomes a dedicated *LL(*)* parser generator.

Strictly speaking, ANTLR is a powerful framework, capable of constructing various tools involved in language development, such as recognizers, interpreters, translators, etc. In our experiment, we work with one of its recent stable release, *ANTLR v3.5* [120]. It takes as input formal grammars, i.e. *ANTLR grammars*, and generates as output the source code of the corresponding parsers. In each code generation process, there are two main output artifacts. One is a lexical analyzer called *lexer* or *recognizer*, which transforms an input string into a token stream. The other one is a parser, which would parse the transformed token stream into the corresponding parse tree and walk through it to execute required actions defined in the input ANTLR grammar. By default, the output language of ANTLR is Java. As other options, a number of mainstream languages are available, such as C, C#. As the lexer class can be considered as a helper class of the parser and will not be involved by the the extension requirements in our experiments, we focus on only the parser class program generated by ANTLR in our experiments. When we mention "base program" in the following discussion, we always refer to the source code of the parser class.

5.2.2 ANTLR Grammar

In the ANTLR parser generation process, the input grammars are specified using *YACC*-like syntax with EBNF operators and token literals in single quotes. This specification language is called the *ANTLR grammar description language*. The grammars defined in it are called *ANTLR grammars*. There are four kinds of ANTLR grammars: lexer, parser, tree, and combined lexer and parser. Lexer grammars define the tokens, or lexemes, of input grammars, and the lexical rules. Parser grammars define the production rules, also known as the *parsing rules*. Tree grammars define the tree matching rules. Combined grammars integrate the related lexer and parser grammars together. All of the four kinds of grammars have the same basic structure as shown in Listing 5.2.

```
<grammarType> grammar grammarName;
<optionsSpec>
<tokensSpec>
<attributeScopes>
<actions>
```

```
rule1 : ... | ... | ... ;
rule2 : ... | ... | ... ;
...
```

Listing 5.2: General structure of ANTLR grammars

In the first line, grammarType indicates which specific type of grammar is defined. Its value can be one of the four values: "lexer", "parser", "tree", "lexer parser". The following grammar is the keyword reserved to define ANTLR grammars, which is followed by the name of the ANTLR grammar. The optionsSpec block is used to define the general configurations of the generated parsers. For example, the option block shown in Listing 5.3 sets the output language as C# instead of the default Java and sets the parsing output in form of template.

```
options {
  language = C#;
  output=template;
}
```

Listing 5.3: An example of ANTLR grammar options specification

As its name suggests, tokensSpec block is used to define the tokens, or lexemes, of input grammars. By convention, tokens are named with uppercase letters, and can be declared with or without initialization values. Take the grammar of the arithmetic operation language we used in Section 3.1.2 for example, its lexer grammar is shown in Listing 5.4.

```
tokens {
  PM;
  MD;
  VAR;
}
```

Listing 5.4: An example of ANTLR grammar token specification

Generally speaking, the only way supported by ANTLR to pass information between two production rules is to use function parameters and return values. This can be quite inconvenient in situations like communicating with deeply nested rules. ANTLR allows users to define their attribute scopes. All attributes defined in a certain scope are visible to any rule declared to be in the scope. The scope shown in Listing 5.5 defines an attribute called x of type int.

```
scope ScopeOne {
  int x;
}
```

Listing 5.5: An example of ANTLR grammar attribute scopes

ANTLR generates a method for each rules in the given ANTLR grammar. These methods are encapsulated respectively in a lexer class and a parser class in the ouput language. ANTLR allows users to directly modify such classes in ANTLR grammars through actions called GrammarActions, which can be used to modify the members or the headers of either class.

```
@lexer::header {
  import some.dependent.class;
}

@member {
  int y;
}
```

Listing 5.6: Two examples of ANTLR grammar actions

The first grammar action imports an additional class some.dependent.class for the lexer class. The second action adds a new member variable y of type int to the parser class. As for the rule part, different ANTLR grammars define different kinds of rules. Lexer grammars define lexical rules. For example, the lexical rules related to the tokens of the arithmetic operation grammar are shown in Listing 5.7.

```
PM : '+' | '-' ;
MD : '*' | '/' ;
VAR : '0' | '1-9' {'0-9'}*
```

Listing 5.7: An example of lexical rules in ANTLR grammar

Tree grammars define tree matching rules with tree pattern, which is in the form of ^(root child1 child2 ... childN). For example, the tree grammar rule shown in Listing 5.8 matches an AST of an "add" operation in the arithmetic operation grammar.

```
add : ^( '+' VAR VAR )
    | ^( '-' VAR VAR )
    ;
```

Listing 5.8: An example of tree matching rules

In ANTLR, there are two ways to rewrite input streams. One is to directly manipulate the existing parse trees to accommodate expected changes, which can be implemented with tree rewriting patterns. As such is not related to our extension requirements, we will not go further about tree grammars and tree rewriting. Instead, we focus on the other way, i.e. template rewriting, which is based on extra attributes and actions that are defined at the production rule level in the input grammars.

Production Rule in ANTLR Grammars Each production rule in an ANTLR grammar consists of an LHS nonterminal, which is also used as the name of the rule, and one or more alternatives. Each alternative can reference other rules by their names. The general structure of an ANTLR production rule is illustrated in Listing 5.9.

```
ruleName <arguments> returns <returnValues>

<throwSpec>
<optionspec>
<ruleAttributeScopes>
<ruleActions>

  : alternative1 -> rewriteRule1
  | alternative2 -> rewriteRule2
  ...
  ;

<exceptionsSpec>
```

Listing 5.9: General structure of an ANTLR production rule

In ANTLR parser generation, ANTLR generates a method in output language for each production rule of the given grammars, which will be invoked when the corresponding rule is applied according to the current matching. The method name is the same as the rule name. By default, such methods have no parameters or returns values. However, when one rule reference another rule, ANTLR allows users to declare input parameters and return values directly to pass information. Besides, ANTLR allows multiple variables to be defined within a production rule, which may be used to store information or handle input or output parameters. These variables are called *rule attributes*. As Parr [97] argued, "this is analogous to the normal programming language functionality whereby methods communicate directly through parameters and return values".

Apart from the basic syntax recognition of the input program, parsers often need to provide extra functionalities during the parsing process, such as semantic computation. ANTLR allows users to define *embedded actions* within production rules. In detail, actions can be inserted before a production rule within @init block, or after it within @after block, or simply before or after any specific alternative of the rule. Embedded actions are written in output languages. The example shown in Listing 5.10 illustrates a production rule named add with all these optional components.

```
add returns [int result]
@init { System.out.println("begin to apply add rule"); }
@after { System.out.println("add rule applied"); }
  : a=VAR '+' b=VAR
```

```
    {$result = Integer.parseInt($a.text)+Integer.parseInt($b
   .text);
      System.out.println($result);}
  |
    {System.out.println("Empty alternative!");}
  ;
```

Listing 5.10: An example of ANTLR production rule

5.2.3 The ANTLR Grammar Domain Meta-Model

So far we have introduced the basics of ANTLR grammars. From the perspective of DSCG, the code generator is ANTLR, and the DSL used is the ANTLR grammar specification language. Accordingly, the target domain is the ANTLR grammar domain. To define the meta-model, we take two documents as guide. One the official ANTLR grammar of the ANTLR grammar specification language itself, which is published on the ANTLR home page [96] (shown in Appendix B.8). The other is the meta-model written by Jouault [121] in KM3 [122]. We use XSD to specify our meta-model of the ANTLR grammar domain, in a module way. We first define all the domain classes and properties involved in the general CFG parsing process, e.g. “terminal”, “nonterminal”, “production_rule”, etc, in one XSD module `General_Parsing_Domain`, which is shown in Appendix B.9. Thus the `General_Parsing_Domain` module can be reused in our next case study. We can thus include the `General_Parsing_Domain` module and define the *LL(*)* domain specific elements, such as token peeking, production rule parameters, etc, in another XSD module `LL_STAR_Domain`, which is shown in Appendix B.10.

In the rest of this chapter, we will elaborate on two of our experiments based on this domain meta-model. The experiments have different domain specific models, i.e. ANTLR grammars, and different extension requirements, to test the feasibility of our DSCG extension based approach. The first experiment tries to enhance the parser with some statistic functions in parsing an ANTLR grammar called *RERS*. The second experiment adds similar functions for more concerns in the parsing process of the Java programs.

5.3 Extension Scenario

ANTLR allows domain experts to insert additional functions when applying a production rule, e.g. by defining extra rule attribute, embedded actions within the corresponding production rule. This modification mechanism is rather flexible, as the additional artifacts can be defined in statements of any programming language supported by ANTLR, e.g. Java. However, it can still be very difficult to introduce functions that involve multiple production rules. In ANTLR, passing information among different production rules is achieved by either explicitly declaring specific parameters and return values, or using dedicated scopes. If we consider the declaration of the parameters, return values, and scope information of a certain production rule as its *signature*, both ways make the signatures of ANTLR production rules vulnerable to systematic changes. Any concern crosscutting multiple production rules may lead to a cascade of signature updates of all relevant production rules.

5.3.1 RERS ANTLR Grammar

In our first experiment, the input model is a simple grammar specifying a tiny formal language called *RERS*, which is used to define *Linear Temporal Logic (LTL)* [123] formulae. The token part of the RERS grammar includes the basic input/output elements and keywords for different relations between propositions in the LTL formulae, such as “NEXT”, “EVENTUALLY”, and “GLOBALLY”. More details are as shown in Listing 5.11.

```
tokens {
  NOT = '!';
  AND = '&';
  OR = '|';

  NEXT = 'X';
  EVENTUALLY = 'F';
  GLOBALLY = 'G';
  UNTIL = 'U';
  WEAKUNTIL = 'WU';
  RELEASE = 'R';

  LPAR = '(';
  RPAR = ')';
}
```

Listing 5.11: Token definition in the RERS grammar

The production rule section of RERS grammar starts with the prog rule. The complete section is shown in Appendix B.11.

5.3.2 Extension Requirements

The modification requirement is to add finer grained and more flexible monitoring functions in the parsing process. For example, to monitor domain events like token matchings and rule attribute settings. Basically, domain experts want to monitor the events according to their production rule application context. For example, they want to trace all the matching of a specific token T in the application of any production rules. Obviously, this demand may crosscut multiple production rules, in which T appears. If we use ANTLR built-in support we mentioned above to implement this, the solution would entail the signature updates of all relevant production rules, though it might still be practicable. The real problem is that each specific tracing requirement would result in such a cascade of signature updates, in other words, an update of the input model. Worse still, ANTLR built-in solution would become deficient to cope with more complicated tracing requirements. By "tracing", we refer to monitoring the *dynamic* behavior, not a static code insertion. For example, domain experts only want to trace the matching of T in a certain context, e.g. during the application of a specific production rule p. This requirement leaves ANTLR users to identify which production rules that involve the matching of T may be applied during the application of p.

5.4 Extension of ANTLR Parser Generation

From the requirement analysis, we can identify three modification concerns, i.e. token matching, rule attribute value changing, and production rule application. Accordingly, we can tailor our domain meta-model into an effective meta-model, in which class "terminalType", "attributeType", and "alternativeType" are defined as join point classes. For example, the "terminalType" class is defined in the effective meta-model as shown in listing 5.12.

```
<xs:complexType name="terminalType">
  <xs:sequence>
    <xs:element name="name" type="terminalNameConvention"/
 >
    <xs:element name="value" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Listing 5.12: The "terminalType" join point class in the meta-model

5.4.1 Generation of AspectRERS

We call the ANTLR domain specific aspect language "AspectRERS", as it is generated on demand of modifying the "RERS" model. In Section 5.4.2, we mentioned our intension of reusing AspectJ in our DSAL generation. By "reusing" AspectJ, we refer to reusing both its join point model and its weaver. Therefore, we select a DSAL template close to AspectJ syntactically and semantically, so that it is easier for us to build the

AspectRERS weaver as a translator from AspectRERS to the underlying AspectJ. Like AspectJ, this template allows definition of local variables inside an aspect. Each PCD is defined with a name, which can be referenced to be bound with a piece of advice. For each PCD, there can be one or more filter conditions, which can be combined by logical "and" operator and logical "or" operator. Placeholders are used for the production rule of PCD. Besides, we also reuse the existing ANTLR grammar blocks for the Java grammar, so that we can syntactically check the correctness of the custom code blocks in advice.

There is a significant difference in advice pattern between this template and AspectJ. It does not support the *around* advice. This is because the modifications requiring the *around* advice are essentially trying to achieve two goals. One is to change the internal behavior of the code generator, e.g. the behavior in matching a token or setting an attribute. This kind of modifications should be accommodated by a customization of the code generator. The other is to alter the production rules or alternatives in the input grammar. This kind of change should be achieved by direct modification to the input grammar. We believe neither of them should be introduced in a "patching" way by using aspects in the DSAL generated in our approach. Therefore, our join point model only supports "insertion" changes, which only insert extra code in the generated code. A major benefit of this is we can leave all statements in the base programs immutable, as the sentinels are inserted in the "slots" between two adjacent statements in the base program. The sentinels also decouple the PCDs from the base program, which prevents the problems caused by the coupling between them, such as *fragile pointcut* [70] and *arranged pattern*. More details of the template are shown in Appendix B.12.

Now we can expand the selected DSAL template with the effective meta-model. The expansion is basically to replace placeholder *pointcut_filter* with several ANTLR grammar domain specific crosscutting patterns. In AspectRERS, we support two groups of pointcuts. The first group of pointcuts locates some specific points in the parser execution, like token matching, etc. We call such pointcuts *pinpoint pointcuts*. In AspectRERS, we support four types of pinpoint pointcuts, *attribute setting pointcuts*, *token matching pointcuts*, *sub rule pointcuts*, and *branch pointcuts*.

Attribute Setting Pointcuts relate to the setting of a rule attribute in one or more production rule or alternative. For example, pointcut after(@temporal : 2 : $esbmc) captures all "slots" just after the points in the application of the second alternative of rule temporal where the value of attribute esbmc is changed. Note that the syntax of AspectRERS demands @ before rule names and $ before attribute names. Besides, detailed context restriction defined in the first two columns is optional. For instance, (@temporal : $esbmc) refers to the value setting of esbmc in any alternative of rule temporal. Similarly, (: 2 : $esbmc) refers to the value setting of esbmc in the second alternative of any rule, and ($esbmc) simply refers to the value setting of esbmc in any context. To make pointcut definitions more flexible, we also support an unary operator ! to get the complement, which means “any ... other than”. For example, (@temporal : 2 :!$esbmc) refers to the setting of any attribute other than esbmc in the second alternative of rule temporal. This operator can also be applied to the first two context columns.

Token Matching Pointcuts relate to the token matchings. They work in a very similar way to attribute setting pointcuts. The only difference is that its syntax expects # instead of $ before token names. For example, the before(@temporal : 2 : #EVENTUALLY) pointcut captures the slots just before the matching of token EVENTUALLY in the application of the second alternative of rule temporal, while (@temporal :!#EVENTUALLY) refers to the matching of any token other than EVENTUALLY in the application of rule temporal.

Sub Rule Pointcuts relate to the application of a production under the control flow of a production rule or one of its alternatives. They also work in a similar way to the above two kinds of pointcuts, although there are a few differences. First, the syntax expects @ in the third column before the names of the nonterminal involved as sub rules. Second, the third column accepts empty sub rule name, i.e. a single @. For example, pointcut before(@temporal : 2 : @) captures the spots just before the application of any nonterminals applied as a sub rule of rule temporal.

Branch Pointcuts relate to the application of one or more alternatives of one or more production rules. Unlike the above pointcuts, branch pointcuts only have the first two columns. The branch pointcuts help to pinpoint the branching spots in the control flow of the parsing process. For example, pointcut before(@temporal : 2) captures the spots just before the application of the second alternative of rule temporal. In branch pointcuts, both column support the ! operator. For instance, (@temporal :!2) refers to the application of any alternative of rule temporal except the second one, whereas (!@temporal : 2) refers to the application of the second alternative of any rule except temporal.

Strictly speaking, the second kind of pointcuts are not pointcuts, as they cannot capture any join points by themselves. Instead, they help to define a certain range of the static code or its dynamic execution. So that pinpoint pointcuts can combine them to introduce more detailed range constraints. For the convenience of AspectRERS users to reuse such constraints, we allow them to be defined independently, just like normal pointcuts. We call such “pointcuts” *range pointcuts.* In detail, we support two types of range pointcuts *within pointcuts* and *cflow pointcuts.*

Within Pointcuts restrict a certain range of the base program, which can be either one or all alternatives of a production rule. As the rule application process is regarded as function invocations instead of class definition, they actually map to the "withincode" pointcuts in AspectJ. It is worth noting that the basic ranges are the alternatives of rules, instead of the rules themselves. For example, pointcut within(@temporal : 2) restricts the range as the second alternative of rule temporal, whereas pointcut within(@temporal) restricts the range as any alternative of rule temporal. Similar to pinpoint pointcuts, range pointcuts also support the ! operator in both columns. For instance, pointcut within(@temporal :!2) restricts the range as any but the second alternative of rule temporal, while pointcut within(!@temporal : 2) restricts the range as the second alternative of any rule other than temporal.

Cflow Pointcuts restrict a certain range of the dynamic execution of the base program, instead of the static code. They help to capture dynamic join points in rule application control flow in a similar way as the "cflow" pointcuts in AspectJ. The major difference here is that the control flows here are the applications of the production rules, or more precisely, the alternatives of them. For example, the production rule application control flow in parsing the RERS expression F oZ would look as follows.

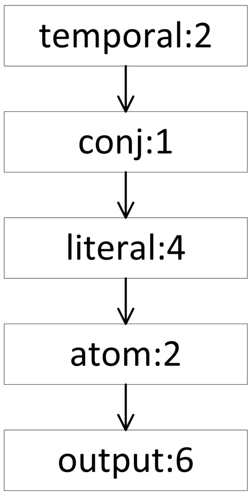


Figure 5.1: Control flow sample of ANTLR rule applications in parsing the RERS expression "F oZ".

Besides, cflow pointcuts should also support the ! operator in both columns. For example, pointcut cflow(!@temporal : 2) defines the range as the application of the second alternative of any rule other than temporal, whereas pointcut cflow(@temporal :!2) defines the range as the application of any but the second alternative of rule temporal. However, although the inserted sentinels can store the static information about the code block, they cannot store the dynamic information at runtime about the "parsing stack" in terms of alternative branches. To achieve this, we use a little trick by keeping a local stack (*Stack<String>* in Java) for branch tracing, as shown in Listing 5.13.

```
@members {
    public static Stack<String> branchTrace = new Stack<
   String>();
}
```

Listing 5.13: The little trick for *range_pointcut* in expansion

This trick also entails a customization of the sentinel templates in the tracing strategy, i.e. to add a Java statement to push the current production rule and alternative information, into the local "branchTrace", which we will explain later. The complete expanded part is shown in Appendix B.13.

5.4.2 Traceable Domain Meta-Model

From the perspective of DSCG, the DSL in ANTLR parser generation is the ANTLR grammar description language. By default, the output language is Java. As the generated parser programs have no strict restrictions in performance, the only restriction for our tracing strategy selection is that the inserted sentinels cannot break the compilation of the generated parser programs in Java. With regard to the possibility of reusing AspectJ, we choose the sentinels based on idle function invocations. By "idle function", we refer to function with empty body. In particular, we specify the tracing strategy in *eXtensible Stylesheet Language Transformations* (XSLT) [124], which can automatically transform the XSD based meta-models. In the strategy, we use three specific sentinel pairs in accordance with the join point classes in the effective meta-model. Take the "terminalType" class for instance, we define the following tracing strategy rule.

```
  <xsl:template match="terminalType">
    <xsl:copy>
      <xsl:apply-templates select="@* | node()" />
      <xsl:element name="xs:element">
        <xsl:attribute name="name">before_match</xsl:
   attribute>
        <xsl:attribute name="default">Begin_Match_Token_&lt;
   instance_name&gt;();</xsl:attribute>
      </xsl:element>
      <xsl:element name="xs:element">
        <xsl:attribute name="name">after_match</xsl:
   attribute>
        <xsl:attribute name="default">End_Match_Token_&lt;
   instance_name&gt;();</xsl:attribute>
      </xsl:element>
    </xsl:copy>
```

```
</xsl:template>
```

LISTING 5.14: Tracing rule for “terminal” join point class

The composition of the tracing strategy with the effective meta-model is essentially a transformation of the XSD file representing the effective meta-model according to the XSLT file representing the tracing strategy. For example, the “terminalType” class after the composition is as shown in listing 5.16.

```
<xs:complexType name="terminalType">
  <xs:sequence>
    <xs:element name="name" type="terminalNameConvention"/
 >
    <xs:element name="value" type="xs:string"/>
    <xs:element name="before_match" default="
 Begin_Match_Token_&lt;instance_name&gt;();" type="
 xs:string"/>
    <xs:element name="after_match" default="
 End_Match_Token_&lt;instance_name&gt;();" type="xs:string
 "/>
  </xs:sequence>
</xs:complexType>
```

LISTING 5.15: The “terminalType” join point class in the traceable domain meta-model

The placeholder “instance_name” will be replaced at generation-time, with the specific terminal instance name, through our customization of the code generator. It is worth noting that the change for the trick to support the *cflow* pointcuts is reflected in the declaration of the sentinel functions. Prior to the trick, the function bodies are empty. Now a Java statement is respectively added to the function bodies.

```
  <xs:complexType name="sentinelDeclaration">
    <xs:sequence>
      <xs:element name="before_match" default="public static
  Begin_Match_Token_&lt;instance_name&gt;() {branchTrace.
 push(&quot;&lt;ruleName&gt;:&quot;+&quot;&lt;altNum&gt;&
 quot;);}" type="xs:string"/>
      <xs:element name="after_match" default="public static
 End_Match_Token_&lt;instance_name&gt;() {branchTrace.pop
 ();}" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

LISTING 5.16: The declaration of the sentinel functions for the “terminalType” class in the traceable domain meta-model

5.4.3 Code Generator Customization

Now that we have produced the traceable ANTLR domain meta-model, we can modify the ANTLR code generator with it to enable the model traceability in the generated code. As mentioned in Section 5.2.1, the output language of ANTLR is Java by default, while a number of other languages are available, such as C, C#. This is achieved by the use of StringTemplate [101] technique in ANTLR. For each target language, ANTLR defines a dedicated template to generate output code. Such templates are defined in String Template Group (.stg) files. This design makes our customization of the ANTLR code generator very easy. We first back up the original template file for Java code generation, i.e. `/org/antlr/codegen/templates/Java/Java.stg` in the ANTLR tool package `antlr-3.5-complete.jar`, and then introduce all sentinel related Java code change in the template. If any of our modification blocked the original DSCG, we can easily replace the updated template with the backup file. Neither the modification process nor the error recovery process requires a recompilation of the ANTLR tool package.

As mentioned in Section 5.1.2, sometimes *LL(*)* parsing may fail over to *LL(1)* enabling backtracking. In the modification requirements, the applications of production rules to monitor obviously refer to the correct applications of production rules, instead of the applications "rolled back" in backtracking. ANTLR generates a global variable `backtracking` in the parser code, to store the current depth of backtracking. To make this transparent to our DSAL users, we add an "if" conditional check for each sentinel function invocations, to ensure the invocations occurs only if `backtracking` equals to `0`.

Besides, ANTLR parser generation supports several different modes, including basic mode, `trace` mode, and verbose mode, which allows users to generate more extra information in the generated code. To ensure that our modification to the out template will not break the basic parser generation function, we encapsulate all our modification snippet with a conditional check <`if(trace)`>, so that users can still use any generation mode other than "trace" to ensure the generated code will not be affected by our change. Instead of showing full details of the modification, we show our modification for token matching sentinels only. This serves to illustrate how we add the declarations of the sentinel functions and how we add their invocation statements in the template.

```
<if(trace)>
/* Begin token hook function declaration block */
<rest(tokens):{tk | <declTokenHook(tk)>}>
/* End token hook function declaration block */<\n><endif>
declTokenHook(token) ::= <<
public static void Begin_Match_Token_<token.name>() {}
public static void Begin_Match_Token_<token.type>() {}
public static void End_Match_Token_<token.type>() {}
public static void End_Match_Token_<token.name>() {}<\n>
>>
```

Listing 5.17: Declarations of token matching sentinel functions

```
/** match a token optionally with a label in front */
tokenRef(token,label,elementIndex,terminalOptions={}) ::=
 <<
<if(trace)><if(!isSynpredRule)>if(state.backtracking==0)
 Begin_Match_Token_<token>();<endif><endif>
<if(label)><label>=(<labelType>)<endif>match(input,<token
 >,FOLLOW_<token>_in_<ruleName><elementIndex>); <
 checkRuleBacktrackFailure()>
<if(trace)><if(!isSynpredRule)>if(state.backtracking==0)
 End_Match_Token_<token>();<endif><endif>
>>
```

Listing 5.18: Invocations of token matching sentinel functions

5.4.4 Generation of AspectRERS Weaver

As mentioned in Section 5.4.1, to generate the AspectRERS weaver is to build a translator from AspectRERS into AspectJ. Now that we have an ANTLR grammar of *AspectRERS*, we can define the corresponding string templates in this grammar to rewrite the AspectRERS aspects into the corresponding AspectJ aspects. In detail, we append a proper string template to each production rule in `AspectRERS.g`, so that ANTLR knows which string template to use when applying the rules. All relevant string templates are encapsulated in `AspectRERS.stg`. Except for the pinpoint and range pointcuts which are domain specific, AspectRERS syntax can be almost directly mapped to AspectJ syntax with minimal change. For example, an AspectRERS aspect definition public aspect Sample(RERS) {...} is simply translated into an AspectJ aspect definition with exactly the same name public aspect Sample {...}. The relevant string template appending in `AspectRERS.g` is shown in Listing 5.19 and Listing 5.20.

```
aspect_declaration
    :   am=access_modifier ASP an=IDENTIFIER LPAREN grn=
   IDENTIFIER RPAREN LBRACE asl=aspect_statement_list RBRACE
        -> asptDecl(accessModifier={$am.code}, aspectName={
   $an.text}, grammarName={$grn.text}, aspectBody={$asl.st})
    ;
```

Listing 5.19: String template asptDecl to translate aspect definition

```
asptDecl(accessModifier, aspectName, grammarName, aspectBody
   ) ::= <<
```

```
<accessModifier> aspect <aspectName> {
    <aspectBody>
}
>>
```

LISTING 5.20: Definition of string template asptDecl in AspectRERS.stg

The translation of pointcuts in AspectRERS is essentially a concatenation over a series of sentinel function signatures to form valid AspectJ pointcut definitions. In detail, domain specific information, such as rule names and alternative indices, is first extracted from AspectRERS pointcuts. The information is then sent to different string templates to construct the corresponding sentinel function signature strings, which are then used to build different kinds of pointcuts. These constructed strings are finally linked with proper logic operations to form a valid AspectJ pointcut descriptor. In particular, the translation of pinpoint pointcuts is different from that of range pointcuts.

Pinpoint Pointcut Translation The pinpoint pointcuts capture the invocations of our sentinel functions. So their translation is mainly straightforward mapping to some "call" pointcuts capturing the proper sentinel function invocations. The only nontrivial part is how to deal with alternatives in the pointcuts. The "call" pointcuts cannot help us in distinguishing different alternatives in the same rule, as their corresponding code blocks are all defined within the same function, i.e. the function correspond to the production rule. Instead, we use our internal tracing stack, i.e. branchTrace, to translate the alternative constraints. Assume we want to confine the range of pointcut matching within the code block correspond to alternative N of rule X. At runtime, if the control flow enters this specific alternative, our sentinel function Begin_Parse_Rule_X_Alternative_N() must have been invoked. In the meantime, the tracing string "X : N" must have been pushed into branchTrace. Moreover, the string "X : N" must be the top element in the stack, as there is no further sentinel function invocation. This means that we can tell if the target code is statically "within" alternative N of rule X, by matching string "X : N" as the top element of branchTrace. As an example, Appendix B.14 and B.15 illustrate how we implement the translation of production rule attribute_filter with string templates.

As all our sentinel functions are declared with the same prefix string public static void, the string templates for translating the pinpoint PCDs, such as attrFilterN, always start with call(public static void. The first alternative of attribute_filter matches the pattern <slotLocation>(@<ruleName> : <alternativeIndex> : $<attributeName>). The slot location information is extracted when applying rule location_modifier. If it is before, we know that Begin should be the prefix of the sentinel function name. It is then sent to template attrFilter1 as its parameter locationModifier. Similarly, the name of the target production rule is extracted when applying rule rule_indicator, and then sent to attrFilter1 as parameter ruleNameValue. The name of the target attribute is extracted when applying rule attr_indicator, and then sent to attrFilter1 as parameter attributeName. With above information, string template attrFilter1 can thus construct the main body of the corresponding AspectJ as shown below.

```
call(public static void <grammarName>Parser.<locationModifier>
   _Set_Rule_<ruleNameValue>_Attribute_<attributeName>())
```

However, this is not a complete translation for the input attribute setting pointcut. What is missing here is the constraint on alternatives. As we mentioned in Section 5.4.2, we maintain an internal stack called branchTrace to trace the control flow of production rule applications. We match the strings stored in branchTrace against certain patterns to construct the predicate expression representing the missing constraints. In detail, this is achieved with string template altFilterForWithin or reverseAltFilterForWithin, depending on whether the target code range is a specific alternative or its complement. The constructed constraint string is then sent to attrFilter1 as parameter alt, and appended to the end of the above incomplete AspectJ pointcut.

String templates involved in other alternatives of attribute_filter work in quite similar ways. Take the second alternative for example, it deals with pointcuts in form of (@<ruleName> : <alternativeIndex> :!$<attributeName>), which aim for any attribute except attributeName. This is reflected as two AspectJ PCD combined with "and" by string template attrFilter2. The first PCD captures any attribute setting in the target rule by the following AspectJ PCD.

```
call(public static void <grammarName>Parser.<locationModifier>
   _Set_Rule_<ruleNameValue>_Attribute_*())
```

The second PCD filters out the matching of a specific attribute setting in the target rule by the AspectJ PCD as shown below.

```
!call(public static void <grammarName>Parser.<locationModifier>
   _Set_Rule_<ruleNameValue>_Attribute_<attributeName>())
```

This is because the second alternative is meant to match the complement of a specific attribute setting with the same rule and alternative constraints.

Range Pointcut Translation The translation of within pointcuts in AspectRERS is even simpler than that of the pinpoint pointcuts mentioned above. We translate a within pointcut in AspectRERS into a withincode pointcut in AspectJ confining the range as the parsing function generated for the target production rule, which is then appended with the corresponding alternative constraint string. For example, pointcut within(@<ruleName> : <alternativeIndex>) is translated into withincode(public final $*$ <grammarName>
It is then combined with the alternative constraint string as shown below.

```
if(<grammarName>Parser.branchTrace.peek().matches("<ruleName>:<
    alternativeIndex>"))
```

More details of the relevant string templates are shown in Listing 5.21, and their definitions in Listing 5.22.

```
within_filter
     :   WTN LPAREN rn=rule_indicator ai=
    alt_indicator_for_within RPAREN
          -> withinFilter1(rangeModifier={"withincode"},
    ruleNameValue={$rn.value}, ruleNamePattern={$rn.pattern},
     alt={$ai.st})
     |   WTN LPAREN NOT rn=rule_indicator ai=
    alt_indicator_for_within RPAREN
          -> withinFilter2(rangeModifier={"withincode"},
    ruleNameValue={$rn.value}, ruleNamePattern={"\\\\w*"},
    alt={$ai.st})
     ;
```

LISTING 5.21: String templates withinFilterN to translate within pointcuts

```
withinFilter1(rangeModifier, ruleNameValue, ruleNamePattern,
     alt) ::= <<
<rangeModifier>(public final * <grammarName>Parser.<
    ruleNameValue>(..))<alt>
>>

withinFilter2(rangeModifier, ruleNameValue, ruleNamePattern,
     alt) ::= <<
<rangeModifier>(public final * <grammarName>Parser.*(..))
    &&!<rangeModifier>(public final * <grammarName>Parser.<
    ruleNameValue>(..))<alt>
>>
```

LISTING 5.22: Definition of relevant string templates for within_filter in AspectRERS.stg

The cflow pointcuts are translated in a quite similar way, with only two differences here. First, the corresponding AspectJ pointcuts are "cflow", instead of "withincode". Second, the alternative constraint strings are in a different pattern. From the perspective of the parsing control flows, the within pointcuts focus on only the current "layer" of the target rule application, whereas the cflow pointcuts also concern about all the descending control flows under the target layer. In terms of our branchTrace string matching, the alternative constraint strings for within pointcuts match the target rule application string, i.e. "<ruleName> : <alternativeIndex>", against the top element in branchTrace. The constraint strings for cflow pointcuts, on the other hand, only care about whether the target string is contained by branchTrace. For example, the constraint string for pointcut cflow(@<ruleName> : <alternativeIndex>) is shown below.

```
if(java.util.Arrays.toString(<grammarName>Parser.branchTrace.
   toArray()).contains("<ruleName>:<alternativeIndex>"))
```

The relevant string templates are shown in Listing 5.23, and their definitions in Listing 5.24.

```
cflow_filter
     :   CFW LPAREN rn=rule_indicator ai=
    alt_indicator_for_cflow RPAREN
         -> controlflowFilter1(rangeModifier={$CFW.text},
    ruleNameValue={$rn.value}, ruleNamePattern={$rn.pattern},
     ruleNameCflowPattern={$rn.pattern}, alt={$ai.st})
     |   CFW LPAREN NOT rn=rule_indicator ai=
    alt_indicator_for_cflow RPAREN
         -> controlflowFilter2(rangeModifier={$CFW.text},
    ruleNameValue={$rn.value}, ruleNamePattern={"\\\\w*"},
    ruleNameCflowPattern={""}, alt={$ai.st})
     ;

alt_indicator_for_cflow
     :   COLON ai=INTLITERAL -> altFilterForCflow(altIndex={
    $ai.text})
     |   COLON NOT ai=INTLITERAL -> reverseAltFilterForCflow(
    altIndex={$ai.text})
     |   -> eptStr()
     ;
```

LISTING 5.23: String templates controlflowFilterN to translate cflow pointcuts

```
controlflowFilter1(rangeModifier, ruleNameValue,
    ruleNamePattern, alt) ::= <<
<rangeModifier>(call(public final * <grammarName>Parser.<
    ruleNameValue>(..)))<alt>
>>

controlflowFilter2(rangeModifier, ruleNameValue,
    ruleNamePattern, alt) ::= <<
<rangeModifier>(call(public final * <grammarName>Parser
    .*(..)))&&!<rangeModifier>(call(public final * <
    grammarName>Parser.<ruleNameValue>(..)))<alt>
```

```
>>
```

Listing 5.24: Definition of relevant string templates for cflow_filter in `AspectRERS.stg`

With the string templates like the above ones defined in `AspectRERS.g` and `AspectRERS.stg`, an AspectRERS-to-AspectJ translator can be generated by ANTLR. We then finish the AspectRERS weaver generation by wrapping the generated translator and the AspectJ weaver in one single script, so that it behaves as a proper weaver for AspectRERS.

5.5 RERS Parser Modification with AspectRERS Aspect Weaving

In this section, we test the above AspectRERS based modification system with a valid input program in RERS language and concrete extension requirements. The RERS program is shown in Listing 5.25.

```
(! oZ WU (oU & ! oZ))
(G (! iC | (F oZ)))
((G ! oW) | (F (oW & (F oU))))
(G (! iE | (F oY)))
(G (! (iB & ! oU) | (! oV WU oU)))
(! oV WU oX)
((G ! oW) | (F (oW & (F oU))))
(! oU WU (oX & ! oU))
```

Listing 5.25: Test program in RERS language

There are two requirements to accommodate in the parsing process. One is to count the number of GLOBALLY temporal logic expressions in the program. The other one is to count how many output expression other than oU are involved in those GLOBALLY expressions. According to our DSAL extension approach, we first regenerate the base program, i.e. `RERSParser.java`, using the modified ANTLR generator, so that model tracing information is inserted as the infrastructure for AspectRERS to work. We then describe the extension requirements within an AspectRERS aspect as shown in Listing 5.26.

```
package antlr.RERS;

public aspect test(RERS) {

    // counting "GLOBALLY" expression
    int GLOBALLY_expression_counter=0;
    pointcut GLOBALLY_expression=after(@temporal:3);
    after : GLOBALLY_expression() {
        GLOBALLY_expression_counter++;
    }

    // counting "oU" expression in "GLOBALLY" expression
    int oU_in_GLOBALLY_expression_counter=0;
    pointcut oU_in_GLOBALLY_expression=after(@output:1)&&
   cflow(@temporal:3);
    after : oU_in_GLOBALLY_expression() {
        oU_in_GLOBALLY_expression_counter++;
    }

    pointcut summary=%BeforeMainExit;
    after : summary() {
        System.out.println("There are " +
   GLOBALLY_expression_counter + " \"GLOBALLY\" expressions.
   ");
        System.out.println("There are " +
   oU_in_GLOBALLY_expression_counter + " \"oU\" expressions
   found in these \"GLOBALLY\" expressions.");
    }
}
```

Listing 5.26: The AspectRERS aspect to accommodate the extension requirements

This aspect is rather logically straightforward. In the first block, we define a counter and a pointcut to capture all applications of the third alternative of rule temporal, which represents the GLOBALLY expression. The corresponding advice increases the counter. Similarly, the pointcut in the second block for the second counting requirement is defined to capture the occurrence of non-“oU” expression in the application, or under the control flow, of the third alternative of rule temporal. Finally, before the parser program exits, it prints out the counting results. We then compile this aspect with our AspectRERS weaver, which first translates it into the AspectJ aspect shown in Listing 5.27, and then invokes the AspectJ weaver to complete the “actual” weaving process.

```
package antlr.RERS;
public aspect test {
     int GLOBALLY_expression_counter=0;
    private pointcut GLOBALLY_expression() : within(
   RERSParser)&&((call(public static void RERSParser.
   End_Parse_Rule_temporal_Alternative_3())));
    after(): GLOBALLY_expression()
    {
    GLOBALLY_expression_counter++;
    }
     int oU_in_GLOBALLY_expression_counter=0;
    private pointcut oU_in_GLOBALLY_expression() : within(
   RERSParser)&&((call(public static void RERSParser.
   End_Parse_Rule_output_Alternative_1()))&&(cflow(call(
   public final * RERSParser.temporal(..)))&&if(java.util.
   Arrays.toString(RERSParser.branchTrace.toArray()).
   contains("temporal:3"))));
    after(): oU_in_GLOBALLY_expression()
    {
    oU_in_GLOBALLY_expression_counter++;
    }
    private pointcut summary() : execution(public static
   void main(String[]));
    after(): summary()
    {
    System.out.println("There are "+
   GLOBALLY_expression_counter+" \"GLOBALLY\" expressions.")
   ;
    System.out.println("There are "+
   oU_in_GLOBALLY_expression_counter+" \"oU\" expressions
   found in these \"GLOBALLY\" expressions.");
    }
}
```

Listing 5.27: The translated AspectJ aspect to accommodate the extension requirements

We finally use the woven parser to parse the above RERS program. The output of the counting result is shown in Listing 5.28.

```
There are 5 "GLOBALLY" expressions.
There are 2 non-"oU" expressions involved in these "GLOBALLY
   " expressions.
```

Listing 5.28: The counting result from the parser

Without difficulty, the results can be verified manually. This experiment shows that the DSCG extension approach can be used to build practical modification system for generated code. Given the fact that ANTLR provides the grammars of many mainstream programming languages, our approach also shows its potential capability of generating generic program synthesis tools for these languages. In order to test this position, we performed a number of additional experiment to generate Java program analysis tools through DSAL aspect weaving.

5.6 Further Experiments to Build Java Program Analysis Tools

In this experiment, we still use our approach to customize the Java parser generated by ANTLR. Precisely, we aim to generate Java program analysis tools with different functionalities by weaving the corresponding DSAL aspects into the generated Java parser. The term "program analysis" mainly refers to syntax-directed searching tasks for code statistics and inspections, such as the detection of code smells [125].

The whole DSCG scenario remains the same except that the input model is now the ANTLR grammar for Java, i.e. `Java.g`, instead of the RERS grammar, and the base program is `JavaParser.java`. The ANTLR grammar of Java is downloaded from the ANTLR homepage [126]. As Java is a quite widely used programming language, we do not introduce its grammar here.

5.6.1 Extension Requirements Analysis

The key feature of the Java analysis tools generated in our approach is that they are, to some extent, "programmable". The programming language is exactly our DSAL. We can generate Java analysis tools with different functionalities on the fly, by weaving the proper DSAL aspects into the generated Java parser. More precisely, we use the DSAL pointcuts to describe the points in which we are interested with descriptions in terms of Java syntax, and we use the DSAL advice to describe the expected modification to these points. In this experiment, our DSAL aspects need to be "powerful" enough to supplement some really useful functionalities to the generated Java parser.

In detail, we generate two Java program analysis tools with two DSAL aspects. The first aspect enables the woven Java parser to count the "simple 'if' statements" [127], or the "if-only" statements, i.e. the "if" statements without defining their corresponding "else" parts. This function is useful, as the "if-only" statements leave the "else" branch uncovered during the program execution, which may lead to potential bugs. Besides, the aspect also helps to count and locate the expressions for "equality checks" in the predicates in the "if" statements. The involved comparison operators include "==" and "!=". The second aspect enables the woven Java parser to count the *Line Of Code* (LOC), which is a common code metric, of the parsed programs according to the counting rules that are defined in the given aspects.

These functionalities can be interpreted as one or more queries to our model tracing system. From the perspective of model traceability, the domain elements to be traced remain unchanged since our AspectRERS experiment, i.e. the matching of Java syntactical tokens under static / dynamic range constraints in terms of the applications of production rules and their alternatives, as well as the value settings of the involved attributes. As a result, we can reuse the traceable domain meta-model derived in our AspectRERS experiment, which means we can also reuse the ANTLR code generator we customized accordingly. This reuse shows the meta-model based reusability of our approach. On the other hand, such reuse does not necessarily guarantee the reuse of the DSALs generated in our approach, as the modification requirements may have different prospectives and concerns of the same effective meta-model, which can be reflected in the join point model of the DSALs.

5.6.2 Generation of AspectJava

Although we can reuse meta-models in the AspectRERS experiment, the new modification requirements entail more expressive DSAL to support more complex crosscutting patterns and finer grained access to the join points. We call the new DSAL *AspectJava*. Compared with the ANTLR grammar of RERS, the ANTLR grammar of Java has much more production rules, which leads to more complicated patterns in the control flows of the rule applications, e.g. larger loops in rule application stack. Therefore, to interpret the above Java program analysis concerns in terms of domain crosscutting queries demands a more precise and effective way to describe the expected join points, as well as finer grained access and manipulation of the captured join points. As a result, we introduce a new PCD and four macros that can be used in custom code block in advice.

CflowPattern Pointcuts This is a more precise PCD than the *cflow* PCD we use in AspectRERS. For example, assume we need to capture the matching of token T in the control flow of the second alternative of rule R1, which should be under the control flow of the third alternative of another rule R2. With only "cflow" pattern, the PCD will look like $\mathsf{cflow(R1:2)\&\&cflow(R2:3)}$. But this is actually not correct, as it will also capture the matching of token T under the control flow of "R2:3", which is under the control flow of "R1:2". Obviously, this is not expected. With our cflowPattern pattern, we can distinguish these two scenarios with different expressions: $\mathsf{cflowPattern(R1:2, R2:3)}$ and $\mathsf{cflowPattern(R2:3, R1:2)}$.

It is worth noting that the length of control flow patterns we need to express is not necessarily constrained to two. For example, the target control flow pattern may be "R1:ALT1" $\rightarrow$ "R2:ALT2" $\rightarrow$ "R3:ALT3" $\rightarrow \ldots$. To support this, we introduce a new crosscutting pattern cflowpattern. The above control flow pattern can be expressed simply as $\mathsf{cflowpattern(@R1:ALT1, R2:ALT2, R3:ALT3, \ldots)}$. The rule name column here

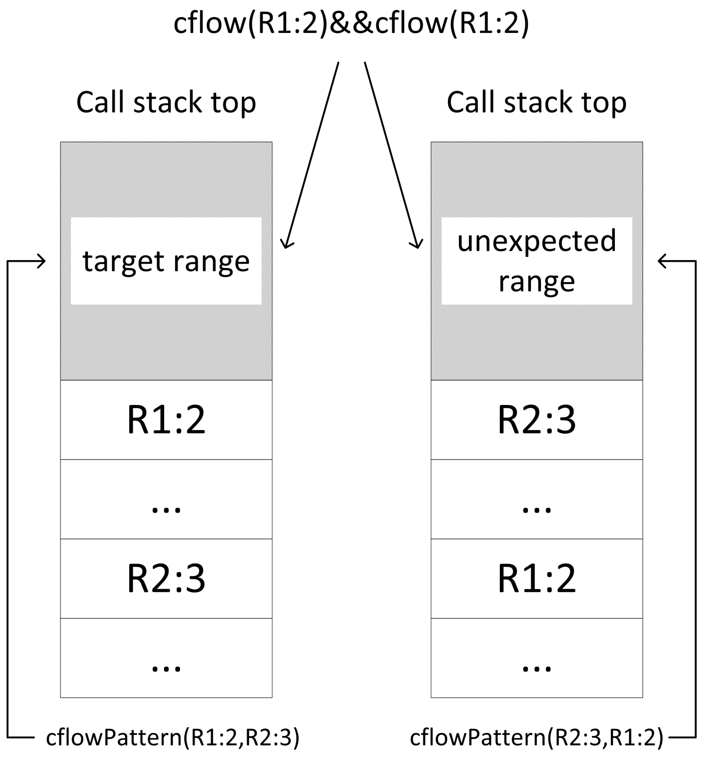


FIGURE 5.2: Different control flow patterns that "cflow" fails to distinguish.

can be left empty, which captures any rule. It is similar for the alternative index column. For example, "R1:*" → "*:ALT2" captures the control flow where the "ALT2" alternative of any rule is under any alternative of rule "R1". The corresponding specification snippet for the "cflowpattern" is shown in Listing 5.33.

```
cflowPattern_filter
    :   CFP LPAREN b=branch_element_list RPAREN
    ;

branch_element_list
    :   branch_element (',' branch_element)*
    ;

branch_element
    :   rule_indicator alt_indicator_for_cflowPattern
    ;

alt_indicator_for_cflowPattern
    :   COLON INTLITERAL
    |
    ;
```

Listing 5.29: Specification snippet for the “cflowpattern” pointcut

Four DSESs In the custom code block of DSAL advice, we introduce four finer grained operations, to facilitate the modifications at the captured join points. We call these operations the *Domain Specific Extension Statements* (DSESs). The syntax of these DSESs are much alike that of the normal Java function invocations. To distinguish these DSESs from the normal Java statements, each DSES starts with a % symbol. The first DSES is GetToken. It takes in an int parameter as the index of the lookahead token, and returns a String containing the corresponding lookahead token.The indices here are 1-based. For example, %GetToken(1) returns the first lookahead token. The second DSES is GetParsedText. It has no parameter and returns a String containing the text that has been successfully parsed so far. It has an overloaded version, which takes in an int parameter as the index of the ending token according to their order being parsed. The indices are 0-based, but count in a descending order. %GetParsedText(k) returns the parsed text ending at the last but k token. For example, %GetParsedText(-1) returns a String containing the parsed text that ends at the last but one token. From this perspective, the non-parameterized GetParsedText can be considered as a special form of the parameterized GetParsedText, i.e. %GetParsedText(0). The third DSES is GetParsingStackTrace. It has no parameter and returns a String representing the control flow information stored in our branchTrace from bottom to top. It is helpful to identify the specific point in the parsing process. The last DSES is GetBacktrackLevel has no parameter and returns an int value that indicates the current backtrack level.

5.6.3 Generation of AspectJava Weaver

Similar to the weaver generation for AspectRERS, we still use string templates to generate the translator from AspectJava to AspectJ. Here we only need to add the templates for translating the cflowPattern pointcut and the four DSESs.

We have briefly explained how cflowPattern pointcut works in Section 5.6.2. In detail, we concatenate a regular expression according to the given control flow pattern, and then use it to launch a matching against the string converted from the content of our branchTrace. As shown in Listing 5.30, three templates are involved, cflowPatternFilter, branchElementList, and branchElement, to generate the corresponding AspectJ code.

```
cflowPattern_filter
    :   CFP LPAREN b=branch_element_list RPAREN
        -> cflowPatternFilter(pc={current_cflowpattern}, fc
   ={$b.st})
    ;
```

```
branch_element_list
@init {
current_cflowpattern="\\\\[([\\\\w:,\\\\s]*)";
}
    :   b0=branch_element (',' b+=branch_element)*
        {current_cflowpattern+="\\\\]";}
        -> branchElementList(fb={$b0.st}, bl={$b})
    ;

branch_element
    :   r=rule_indicator a=alt_indicator_for_cflowPattern
        {current_cflowpattern=current_cflowpattern+$r.
   pattern+":"+$a.altIndex+"([\\\\w:,\\\\s]*)";}
        -> branchElement(rulePattern={$r.value})
    ;

alt_indicator_for_cflowPattern returns [String altIndex]
    :   COLON INTLITERAL
        {$altIndex=$INTLITERAL.text;}
    |   {$altIndex="\\\\d*";}
    ;
```

Listing 5.30: Specification snippet for the cflowpattern pointcut with the string templates

The definitions of these templates are shown in Listing 5.33.

```
cflowPatternFilter(pc, fc) ::= <<
<fc>&&if(java.util.Arrays.toString(<grammarName>Parser.
   branchTrace.toArray()).matches("<pc>"))
>>

branchElementList(fb, bl) ::= <<
<fb>&&<bl:{b | <b>}; separator="&&">
>>

branchElement(rulePattern) ::= <<
cflow(call(public final * <grammarName>Parser.<rulePattern
   >(..)))
>>
```

Listing 5.31: Definitions of the string templates for the cflowpattern pointcut

As for the DSESs, our implementation relies on a special reference variable thisJoinPoint in AspectJ. It helps to reference the generated parser class, i.e. class JavaParser. The class exposes a field called input of type TokenStream, which stores the information of the input token stream. From this object, we can access both the parsed tokens and lookahead tokens. In detail, we get the index of the last parsed token in the stream by calling intJavaParser.input.index(). We can thus access part of the parsed text by calling StringJavaParser.input.toString(startTokenIndex, endTokenIndex. We can access the lookahead tokens by calling GetToken(k), where k is the index of the expected lookahead token. As for the DSES GetBacktrackLevel, we can get the value of the current backtrack level by calling JavaParser.getBacktrackingLevel().

5.6.4 Build Java Program Analysis Tools By Aspect Weaving

As mentioned in Section 5.6.1, the extended functionalities we want for the parser are first interpreted in one or more model tracing and modification queries. We then express these queries as AspectJava aspects, and respectively weave them into the generated parser to equip it to be the expected Java program analysis tool. In this section, we briefly explain the two AspectJava aspects and test the generated analysis tools.

If Predicate Checker As its name suggests, the first aspect helps to analyze the predicates in the "if" statements in Java programs. In particular, it counts the "if-only" statements that do not have a corresponding "else" part and counts the equality checking expressions in the "if" predicates.

In the input model, i.e. the ANTLR grammar of Java, the definition of the basic Java statement is shown in Listing 5.32. We can see that the "if" statement is defined in the fourth alternative of the statement production rule.

```
statement
    :   block

    |   ('assert'
        )
        expression (':' expression)? ';'
    |   'assert'  expression (':' expression)? ';'
    |   'if' parExpression statement ('else' statement)?
    |   forstatement
    |   'while' parExpression statement
    |   'do' statement 'while' parExpression ';'
    |   trystatement
    |   'switch' parExpression '{'
   switchBlockStatementGroups '}'
```

```
|   'synchronized' parExpression block
|   'return' (expression )? ';'
|   'throw' expression ';'
|   'break'
        (IDENTIFIER
        )? ';'
|   'continue'
        (IDENTIFIER
        )? ';'
|   expression  ';'
|   IDENTIFIER ':' statement
|   ';'


;
```

Listing 5.32: Definition of basic "statement" in Java ANTLR grammar

For the first task, the straightforward solution is to count the application of the fourth alternative of rule statement, where the statement following the else token is resolved as empty string. However, this condition is not easy to describe in AspectJava. Instead, we can count all "if" statements, i.e. with and without the "else" parts, and save the result in c1. We then count the "if-else" statements, and save the result in c2. Their difference is exactly what we want. In terms of the DSL here, to parse an "if" statement is to apply the fourth alternative of the statement production rule. Given the fact that we can parse an "if-else" statement if and only if we can match an "else" token, we can simply count the instances of the "else" token. The number of "if-only" statements thus equals to c1 − c2. In AspectJava, these DSESs can be expressed as shown in Listing 5.33.

```
pointcut summary=%BeforeMainExit;

//===== begin: count if-only statement =====
int c1=0;
int c2=0;

pointcut ifStatement=after(@statement:4);
pointcut ifElseStatement=after(#ELSE);

after : ifStatement() {
    c1++;
}

after : ifElseStatement() {
        c2++;
    }

    after : summary() {
        System.out.println("there are " + (c1-c2) + " if-
   only if-statements.");
    }
    //===== end: count if-only statement =====
```

Listing 5.33: AspectJava code to count the "if-only" statement

For the second task, to locate the expressions in the "if" predicates entails that the application of parExpression is under the control flow of the application of the fourth alternative of statement. As for the application of parExpression, the involved production rules are shown in Appendix B.16.

In terms of the DSL, an "equality check" in expressions implies an application of instanceOfExpression under the control flow of the application of equalityExpression, where the first lookahead token is either == or parExpression. Besides, the whole context has to be under the control flow of "@statement : 4,@parExpression : $*$". We can use the cflowPattern pointcut to locate such an "equality check". Apart from counting these "equality checks", we also want to reflect the location of them. So we print the first lookahead token and the parsed text when we meet any "if-only" statement. In AspectJava, these DSESs can be expressed as shown in Listing 5.34.

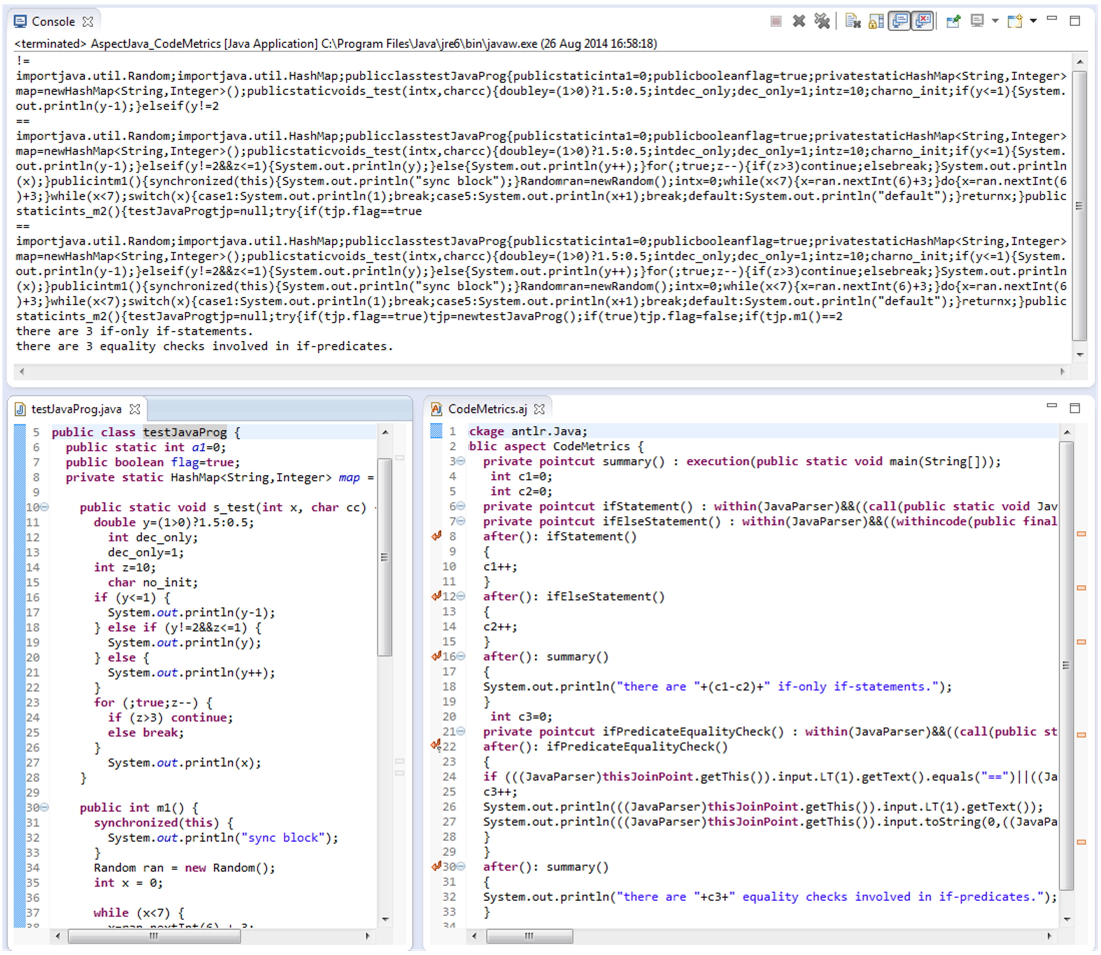
```
    //===== begin: count equality checks =====
    int c3=0;

    pointcut ifPredicateEqualityCheck=after(
   @instanceOfExpression:)&&cflowpattern(@statement:4,
   @parExpression,@equalityExpression);

    after : ifPredicateEqualityCheck() {
        if (%GetToken(1).equals("==") || %GetToken(1).equals
   ("!=")) {
            c3++;
            System.out.println(%GetToken(1));
            System.out.println(%GetParsedText(1));
        }
    }

    after : summary() {
        System.out.println("there are " + c3 + " equality
   checks involved in if-predicates.");
```

```
}
//===== end: count equality checks =====
```

LISTING 5.34: AspectJava code to count and locate the "equality checks" in "if" predicates

Now that we have written an AspectJava aspect with the expected "if" statement analysis functionalities, we weave it to the base code, i.e. the Java parser generated by ANTLR, to finish our first Java program analysis tool. Finally, we test the tool with a simple Java program, which is contrived to have as many different grammatical structures as we can think of. The test Java program is shown in Appendix B.17.

Our test environment, in which the translated AspectJ aspect gets woven and executed, is "`Eclipse Java EE IDE for Web Developers (version:Juno SR2)`" [128] + `AJDT 2.1.2` [129]. The parsing result shows that each "if-only" statement is captured, as well as each "equality check". A snapshot of the detailed output is shown in Figure 5.3.



```
<terminated> AspectJava_CodeMetrics [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (26 Aug 2014 16:58:18)
!=
importjava.util.Random;importjava.util.HashMap;publicclasstestJavaProg{publicstaticinta1=0;publicbooleanflag=true;privatestaticHashMap<String,Integer>
map=newHashMap<String,Integer>();publicstaticvoids_test(intx,charcc){doubley=(1>0)?1.5:0.5;intdec_only;dec_only=1;intz=10;charno_init;if(y<=1){System.
out.println(y-1);}elseif(y!=2
==
importjava.util.Random;importjava.util.HashMap;publicclasstestJavaProg{publicstaticinta1=0;publicbooleanflag=true;privatestaticHashMap<String,Integer>
map=newHashMap<String,Integer>();publicstaticvoids_test(intx,charcc){doubley=(1>0)?1.5:0.5;intdec_only;dec_only=1;intz=10;charno_init;if(y<=1){System.
out.println(y-1);}elseif(y!=2&&z<=1){System.out.println(y);}else{System.out.println(y++);}for(;true;z--){if(z>3)continue;elsebreak;}System.out.println
(x);}publicintm1(){synchronized(this){System.out.println("sync block");}Randomran=newRandom();intx=0;while(x<7){x=ran.nextInt(6)+3;}do{x=ran.nextInt(6
)+3;}while(x<7);switch(x){case1:System.out.println(1);break;case5:System.out.println(x+1);break;default:System.out.println("default");}returnx;}public
staticints_m2(){testJavaProgtjp=null;try{if(tjp.flag==true
==
importjava.util.Random;importjava.util.HashMap;publicclasstestJavaProg{publicstaticinta1=0;publicbooleanflag=true;privatestaticHashMap<String,Integer>
map=newHashMap<String,Integer>();publicstaticvoids_test(intx,charcc){doubley=(1>0)?1.5:0.5;intdec_only;dec_only=1;intz=10;charno_init;if(y<=1){System.
out.println(y-1);}elseif(y!=2&&z<=1){System.out.println(y);}else{System.out.println(y++);}for(;true;z--){if(z>3)continue;elsebreak;}System.out.println
(x);}publicintm1(){synchronized(this){System.out.println("sync block");}Randomran=newRandom();intx=0;while(x<7){x=ran.nextInt(6)+3;}do{x=ran.nextInt(6
)+3;}while(x<7);switch(x){case1:System.out.println(1);break;case5:System.out.println(x+1);break;default:System.out.println("default");}returnx;}public
staticints_m2(){testJavaProgtjp=null;try{if(tjp.flag==true)tjp=newtestJavaProg();if(true)tjp.flag=false;if(tjp.m1()==2
there are 3 if-only if-statements.
there are 3 equality checks involved in if-predicates.
```

```java
public class testJavaProg {
  public static int a1=0;
  public boolean flag=true;
  private static HashMap<String,Integer> map =

    public static void s_test(int x, char cc)
      double y=(1>0)?1.5:0.5;
        int dec_only;
        dec_only=1;
      int z=10;
        char no_init;
      if (y<=1) {
        System.out.println(y-1);
      } else if (y!=2&&z<=1) {
        System.out.println(y);
      } else {
        System.out.println(y++);
      }
      for (;true;z--) {
        if (z>3) continue;
        else break;
      }
        System.out.println(x);
    }

    public int m1() {
      synchronized(this) {
        System.out.println("sync block");
      }
      Random ran = new Random();
      int x = 0;

      while (x<7) {
```

```
ckage antlr.Java;
blic aspect CodeMetrics {
  private pointcut summary() : execution(public static void main(String[]));
   int c1=0;
   int c2=0;
  private pointcut ifStatement() : within(JavaParser)&&((call(public static void Jav
  private pointcut ifElseStatement() : within(JavaParser)&&((withincode(public final
  after(): ifStatement()
  {
  c1++;
  }
  after(): ifElseStatement()
  {
  c2++;
  }
  after(): summary()
  {
  System.out.println("there are "+(c1-c2)+" if-only if-statements.");
  }
   int c3=0;
  private pointcut ifPredicateEqualityCheck() : within(JavaParser)&&((call(public st
  after(): ifPredicateEqualityCheck()
  {
  if (((JavaParser)thisJoinPoint.getThis()).input.LT(1).getText().equals("==")||((Ja
  c3++;
  System.out.println(((JavaParser)thisJoinPoint.getThis()).input.LT(1).getText());
  System.out.println(((JavaParser)thisJoinPoint.getThis()).input.toString(0,((JavaPa
  }
  }
  after(): summary()
  {
  System.out.println("there are "+c3+" equality checks involved in if-predicates.");
  }
```

FIGURE 5.3: The output of weaving aspect `IfPredicateChecker` in Eclipse.

LOC Counter *Lines Of Code* (LOC), also known as *Source Lines Of Code* (SLOC), is an important software metric, which measures the size of a program by counting the number of lines in the text of its source code. It is commonly used in the prediction of development effort, the estimation of the software quality, etc. However, the LOC number of the same program may vary if we follow different counting principles. For example, we can simply count any nonempty line in the program. However, the LOC number we get may be misleading, as programmers may write big block of comments, or break a long single statement into multiple lines for better appearance of the text, etc. Therefore, we need finer grained counting principles to describe which kinds of grammatical structure should be counted. For instance, we can specify whether to count the Java "import" statement or not. In our aspect, we count the following Java grammatical structures as valid lines.

1. Primitive statements including "assert" statements, "return" statements, "throw" statements, and normal statements. Here normal statements refers to the statements represented by the fifteenth alternative of the `statement` rule.
2. The declarations of local variables
3. The declarations of class variables
4. The declarations of class methods.
5. The declarations of classes.
6. Sub statements including "if" predicates, "for" conditions, "while" conditions, "do-while" conditions, "catch" clauses, "switch" conditions, and "sync" resources.

To organize the above DSES in a modular way, we first specify all the primitive statements according to the `statement` rule. The corresponding AspectJava code snippet is shown in Appendix B.18. With the pointcuts that represent primitive statements, we can easily change the counting principles. We then specify the declarations to count as shown in Appendix B.19. We then specify which sub statements to count as shown in Appendix B.20.

So far we have specified all the grammatical structures that we want to count as "lines". We then describe the grammatical structures that we do not count. Strictly speaking, we do not this part in the aspect, as we do not have much to do with these structures under our current goal. We list them here just for the clarification of the whole aspect structure. The corresponding AspectJava code snippet is shown in Listing 5.35.

```
/* <begin: uncountable statement> */
/* <begin: import statement> */
pointcut import_stmt=after(@importDeclaration:);
after : import_stmt() {
        System.out.println("end of an import statement\n");
    }
    /* <end: import statement> */

    pointcut uncountable_stmt=import_stmt()||
   uncountable_primitive_stmt();
    /* <end: uncountable statement> */
```

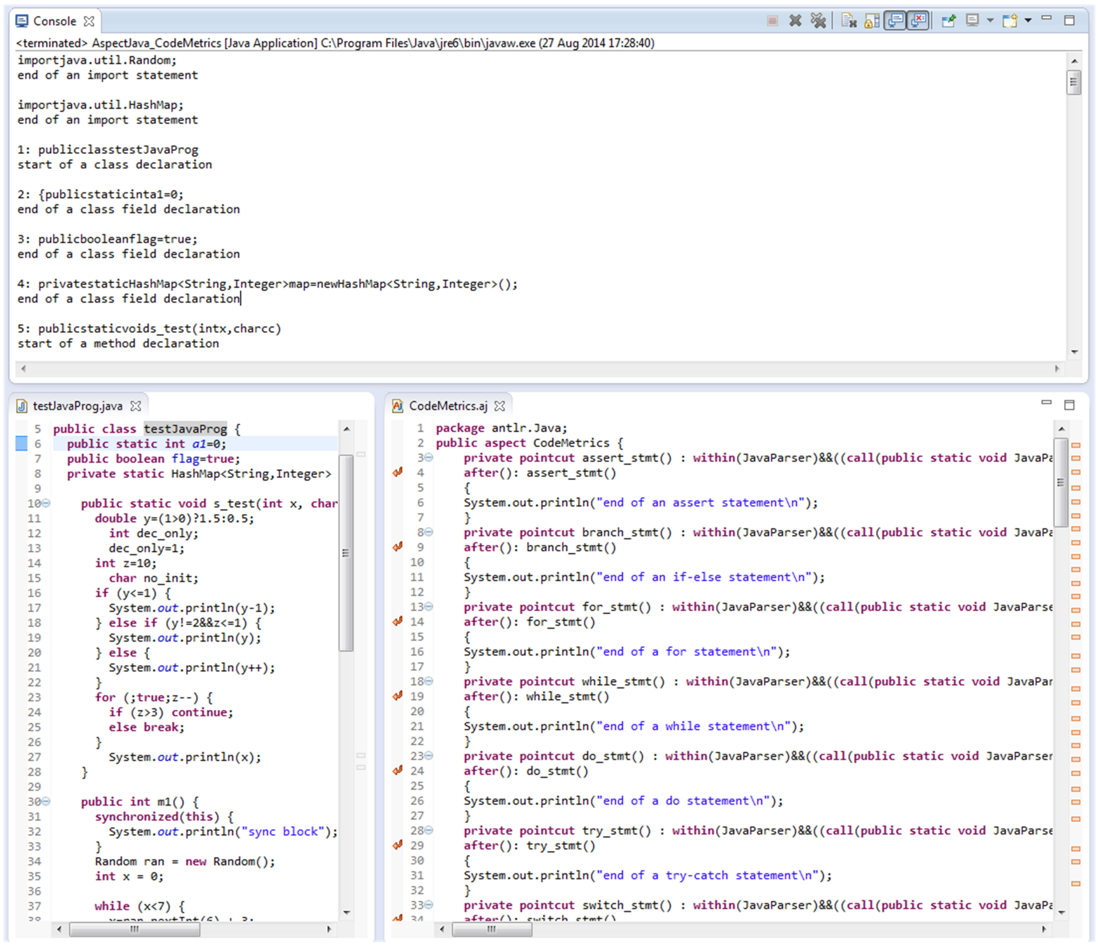
Listing 5.35: AspectJava code for the grammatical structures that we do not count

The final part of the aspect simply defines the common advice for the "lines" we count, i.e. increasing the counter by one and print the parsed text at the point. The corresponding AspectJava code snippet is shown in Listing 5.36.

```
    /* <begin: code metrics - line of code> */
    pointcut print_result=%BeforeMainExit;

    int idx = 0;
    int counter = 0;

    public void count(){
        counter++;
    }

    after : print_result() {
        System.out.println("Current loc: "+counter);
    }

    pointcut countable=countable_stmt()||countable_sub_stmt
   ();
    before : countable() {
        count();
        String currentStr=%GetParsedText(-1);
        System.out.println(counter+": "+currentStr.substring
   (idx));
        idx=currentStr.length();
    }

    pointcut uncountable=uncountable_stmt();
    before : uncountable() {
        String currentStr=%GetParsedText(-1);
        System.out.println(currentStr.substring(idx));
        idx=currentStr.length();
    }
    /* <end: code metrics - line of code> */
}
```

LISTING 5.36: Final part of the `LOCCounter` aspect

In the same test environment for the last aspect, we test the LOC counter aspect with two Java programs. The first one is our contrived test program in the test of the last aspect, as it is short enough that we can manually verify its result. Figure 5.4 shows the test result in Eclipse, that the LOC of the input program is 51.



```
<terminated> AspectJava_CodeMetrics [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (27 Aug 2014 17:28:40)
importjava.util.Random;
end of an import statement

importjava.util.HashMap;
end of an import statement

1: publicclasstestJavaProg
start of a class declaration

2: {publicstaticinta1=0;
end of a class field declaration

3: publicbooleanflag=true;
end of a class field declaration

4: privatestaticHashMap<String,Integer>map=newHashMap<String,Integer>();
end of a class field declaration

5: publicstaticvoids_test(intx,charcc)
start of a method declaration
```

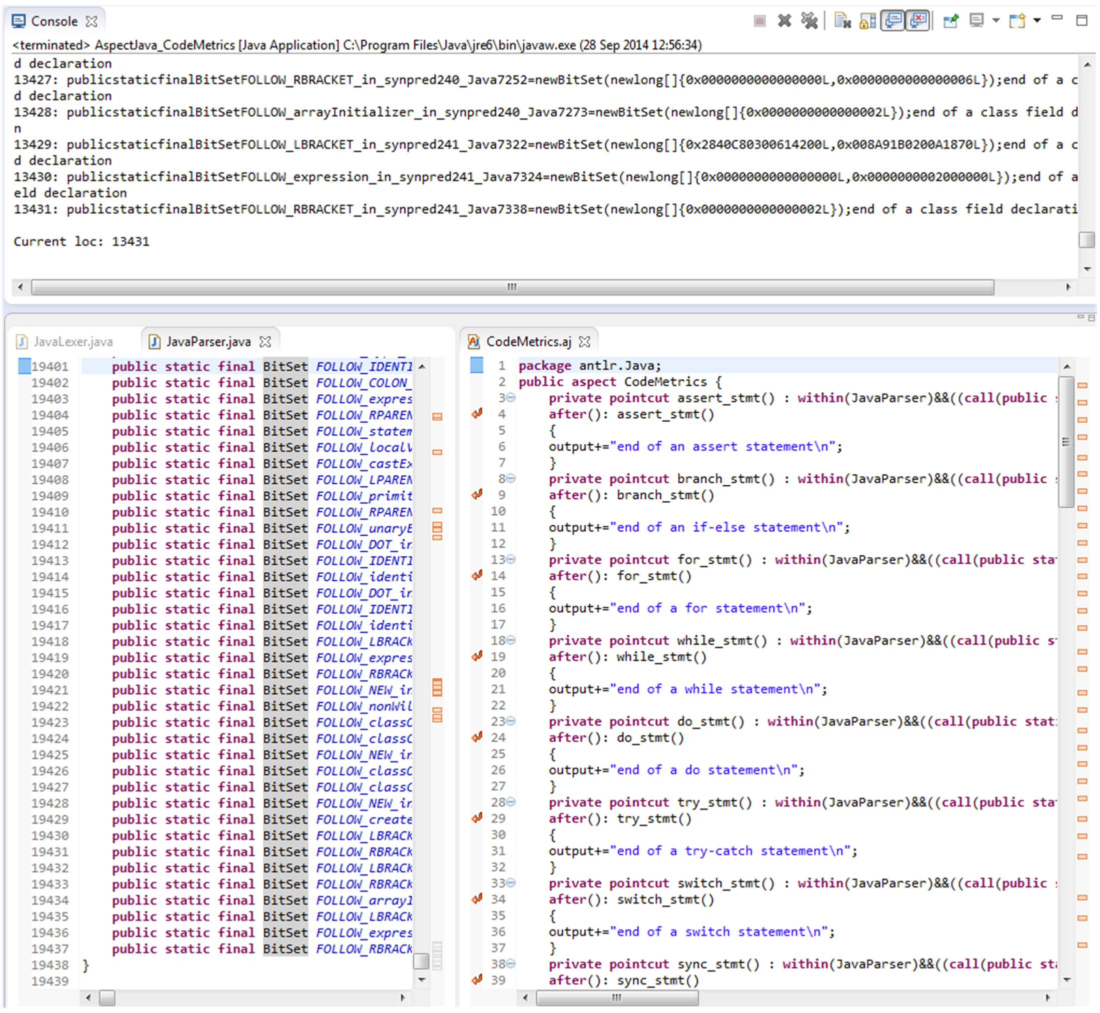
```
public class testJavaProg {
  public static int a1=0;
  public boolean flag=true;
  private static HashMap<String,Integer>

    public static void s_test(int x, char
      double y=(1>0)?1.5:0.5;
        int dec_only;
        dec_only=1;
      int z=10;
        char no_init;
      if (y<=1) {
        System.out.println(y-1);
      } else if (y!=2&&z<=1) {
        System.out.println(y);
      } else {
        System.out.println(y++);
      }
      for (;true;z--) {
        if (z>3) continue;
        else break;
      }
        System.out.println(x);
    }

    public int m1() {
      synchronized(this) {
        System.out.println("sync block");
      }
      Random ran = new Random();
      int x = 0;

      while (x<7) {
```

```
package antlr.Java;
public aspect CodeMetrics {
    private pointcut assert_stmt() : within(JavaParser)&&((call(public static void JavaPa
    after(): assert_stmt()
    {
    System.out.println("end of an assert statement\n");
    }
    private pointcut branch_stmt() : within(JavaParser)&&((call(public static void JavaPa
    after(): branch_stmt()
    {
    System.out.println("end of an if-else statement\n");
    }
    private pointcut for_stmt() : within(JavaParser)&&((call(public static void JavaParse
    after(): for_stmt()
    {
    System.out.println("end of a for statement\n");
    }
    private pointcut while_stmt() : within(JavaParser)&&((call(public static void JavaPar
    after(): while_stmt()
    {
    System.out.println("end of a while statement\n");
    }
    private pointcut do_stmt() : within(JavaParser)&&((call(public static void JavaParser
    after(): do_stmt()
    {
    System.out.println("end of a do statement\n");
    }
    private pointcut try_stmt() : within(JavaParser)&&((call(public static void JavaParse
    after(): try_stmt()
    {
    System.out.println("end of a try-catch statement\n");
    }
    private pointcut switch_stmt() : within(JavaParser)&&((call(public static void JavaPa
```

FIGURE 5.4: The output of parsing `testJavaProg.java` with the `LOCCounter`-woven JavaParser.

The second test program is the generated JavaParser class itself, which has 19438 lines of text in it, including comments and blank lines. The result the LOC of the input program is 13431 in accordance with our counting rules. The test result in Eclipse is shown in Figure 5.5.

In this chapter, we elaborated on our experiments to modify the ANTLR generated parsers with our DSCG extension approach. In particular, we use the AspectRERS



```
<terminated> AspectJava_CodeMetrics [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (28 Sep 2014 12:56:34)
d declaration
13427: publicstaticfinalBitSetFOLLOW_RBRACKET_in_synpred240_Java7252=newBitSet(newlong[]{0x0000000000000000L,0x0000000000000006L});end of a c
d declaration
13428: publicstaticfinalBitSetFOLLOW_arrayInitializer_in_synpred240_Java7273=newBitSet(newlong[]{0x0000000000000002L});end of a class field d
n
13429: publicstaticfinalBitSetFOLLOW_LBRACKET_in_synpred241_Java7322=newBitSet(newlong[]{0x2840C80300614200L,0x008A91B0200A1870L});end of a c
d declaration
13430: publicstaticfinalBitSetFOLLOW_expression_in_synpred241_Java7324=newBitSet(newlong[]{0x0000000000000000L,0x0000000002000000L});end of a
eld declaration
13431: publicstaticfinalBitSetFOLLOW_RBRACKET_in_synpred241_Java7338=newBitSet(newlong[]{0x0000000000000002L});end of a class field declarati

Current loc: 13431
```

```
    public static final BitSet FOLLOW_IDENTI
    public static final BitSet FOLLOW_COLON_
    public static final BitSet FOLLOW_expres
    public static final BitSet FOLLOW_RPAREN
    public static final BitSet FOLLOW_statem
    public static final BitSet FOLLOW_localV
    public static final BitSet FOLLOW_castEx
    public static final BitSet FOLLOW_LPAREN
    public static final BitSet FOLLOW_primit
    public static final BitSet FOLLOW_RPAREN
    public static final BitSet FOLLOW_unaryE
    public static final BitSet FOLLOW_DOT_ir
    public static final BitSet FOLLOW_IDENTI
    public static final BitSet FOLLOW_identi
    public static final BitSet FOLLOW_DOT_ir
    public static final BitSet FOLLOW_IDENTI
    public static final BitSet FOLLOW_identi
    public static final BitSet FOLLOW_LBRACK
    public static final BitSet FOLLOW_expres
    public static final BitSet FOLLOW_RBRACK
    public static final BitSet FOLLOW_NEW_ir
    public static final BitSet FOLLOW_nonWil
    public static final BitSet FOLLOW_classC
    public static final BitSet FOLLOW_classC
    public static final BitSet FOLLOW_NEW_ir
    public static final BitSet FOLLOW_classC
    public static final BitSet FOLLOW_classC
    public static final BitSet FOLLOW_NEW_ir
    public static final BitSet FOLLOW_create
    public static final BitSet FOLLOW_LBRACK
    public static final BitSet FOLLOW_RBRACK
    public static final BitSet FOLLOW_LBRACK
    public static final BitSet FOLLOW_RBRACK
    public static final BitSet FOLLOW_arrayI
    public static final BitSet FOLLOW_LBRACK
    public static final BitSet FOLLOW_expres
    public static final BitSet FOLLOW_RBRACK
}
```

```
package antlr.Java;
public aspect CodeMetrics {
    private pointcut assert_stmt() : within(JavaParser)&&((call(public :
    after(): assert_stmt()
    {
    output+="end of an assert statement\n";
    }
    private pointcut branch_stmt() : within(JavaParser)&&((call(public :
    after(): branch_stmt()
    {
    output+="end of an if-else statement\n";
    }
    private pointcut for_stmt() : within(JavaParser)&&((call(public sta
    after(): for_stmt()
    {
    output+="end of a for statement\n";
    }
    private pointcut while_stmt() : within(JavaParser)&&((call(public s
    after(): while_stmt()
    {
    output+="end of a while statement\n";
    }
    private pointcut do_stmt() : within(JavaParser)&&((call(public stat:
    after(): do_stmt()
    {
    output+="end of a do statement\n";
    }
    private pointcut try_stmt() : within(JavaParser)&&((call(public sta
    after(): try_stmt()
    {
    output+="end of a try-catch statement\n";
    }
    private pointcut switch_stmt() : within(JavaParser)&&((call(public :
    after(): switch_stmt()
    {
    output+="end of a switch statement\n";
    }
    private pointcut sync_stmt() : within(JavaParser)&&((call(public st
    after(): sync_stmt()
```

FIGURE 5.5: The output of parsing `JavaParser.java` with the `LOCCounter`-woven JavaParser.

experiment to discuss the implementation details. Besides, we did two additional experiments with ANTLR generated Java parser to further explore the reusability of our approach. In the next chapter, we continue our experiments with a quite similar domain but different code generator.

Chapter 6

DSCG Extension for CUP

As a third case study, we test our approach within another CFG parsing domain, LALR(1) parsing. On one hand, it shares much common ground with the LL(*) parsing domain, such as the concepts of terminals/nonterminals, prediction based production rule applications, etc. On the other hand, they have significant difference in the parsing process, such as the reverse production rule applications, etc. In this case study, we select *CUP* as the target code generator, which is a Java version of its well-known predecessor *YACC*.

6.1 CUP Parser Generator and LALR Parsing

In this section, we first introduce the CUP Parser Generator. We then discuss the LR and LALR parsing algorithms, in particular their difference from the LL parsing algorithms. Finally, we illustrate the meta-model of the LALR(1) parsing domain.

6.1.1 YACC and CUP

In 1970, Stephen C. Johnson developed a LALR parser generator, named *Yet Another Compiler Compiler* (YACC) [130, 131]. It works on the Unix operating system and generates an LALR parser on the input of an LALR grammar. YACC becomes so popular that there are so many variants of it with additional features and for various languages, such as *Berkeley Yacc* (BYACC) [132] and *GNU Bison* (commonly known as *Bison*) [133]. Among these variants, we choose *Constructor of Useful Parsers* (CUP) [17, 134] as our target DSCG code generator, because it shares two points in common with ANTLR. First, unlike BYACC and Bison that are both implemented in C, the CUP parser generator is implemented in Java, the same implementation language as ANTLR. This could help us collate the way we instrument different code generators to produce sentinels so we may discover a generative method. Second, as the output language in our ANTLR case study is Java, the fact that CUP also generates parsers in Java makes it more convenient to compare the tracing strategies to use, which is coupled closely with the output language.

Unlike ANTLR generating lexer and parser at the same time, CUP and many YACC variant tools only generate parsers. Thus they always have to work together with external lexer tools. For example, YACC has to work with Lex [135], Bison with Flex [136], CUP with JFlex [137]. They are functionally identical, to scan the input string and translate it into a stream of recognized tokens. There is only subtle difference among them, due to the different implementation of their partner LALR parser generators. Again, we only focus on the parser generation process in our discussion. Besides, although the current CUP maintainers (Technical University of Munich) are working on CUP2, it is still not ready according to their development page [138]. Thus we still use the last stable release of the original CUP, i.e. "cup-11a" in our experiment.

6.1.2 LR Parsing and LALR Parsing

The parsers generated by CUP are LALR parsers. To explain the LALR parsing algorithm, we first need to introduce the LR parsing algorithm, since the LALR parsing algorithm is an enhanced variant of the original LR parsing algorithm.

LR Parsing is a bottom-up parsing algorithm. The *L* here still refers to "**L**eft to right" scanning, while the *R* stands for "**R**ightmost derivation". As mentioned in Section 5.1.1, CFG parsing algorithms mainly include *LL* parsing and *LR* parsing. In contrast to LL parsing, LR parsing constructs parse trees in a "bottom-up" pattern, i.e. from leaves to roots. As an LR parser scans the input token stream from left to right, it tries to recognize the *Right Hand Side* (RHS) of the proper production rule, so that it can apply it in a reverse way, i.e. to reduce its RHS to the nonterminal in its LHS. This step is often called *reduce*. If the currently parsed token stream cannot match the RHS of any production rule, the parser simply takes in the next token to see if any luck. This step is often called a *shift*.

Similar to the original LL parsing algorithm, the original LR parsing algorithm also lacks an effective strategy to "smartly" choose the proper production to apply according to the currently parsed tokens. Therefore, the lookahead strategy is used again to allow predictions in LR parsing. But a primary difference here is that at any point in an LL parsing process, each token in the string to the left of the next token has already found its proper position on the parse tree, while in LR parsing, these tokens often still wait aside to be put on the parse tree after the current reduction, or "handle pruning" by D. Knuth [139]. *Handle* here is defined as "the leftmost set of adjacent leaves forming a complete branch". Thus the major problem in LR parsing becomes how to iteratively search for the current *handle*. Similar to LL(k) parsing, a table-driven method is involved to solve the searching problem. However, the uncertainty caused by the "unstructured" token string on the left of the current handle makes the parse table here more complicated. The keys of the parse table are two-dimensional pairs of the tokens and states within the state space of the *deterministic finite automatons* (DFAs) built from the grammar, instead of one-dimensional tokens. The values in the table cells are actions over the production rules, such as "shift", "accept", or "reduce".

More details about the table/DFA constructions are beyond the scope of our research. We only need to understand that we are able to somehow build the tables to tell us what action we should do in a certain state on the recognition of a specific token. Besides, like LL parsing, LR(0) parsing can be enhanced with predicting ability, which are collectively called *LR(k)* (k>0) parsing algorithms. However, as a grammar getting more complicated, the state space of the DFA derived from it grows very fast, which will finally lead to a steep increase of the LR(k) look-up table size. To reduce the size of parse table, several variations of LR(0) parsing are developed. For example, *Simple LR* (SLR) is introduced by identifying the current handle more strictly.

LALR Parsing is a commonly used variant of LR parsing, which merges the states in the LR(k) DFAs that are identical except for their lookahead sets. The *LookAhead LR (LALR)* is defined by DeRemer [140, 141] to enhance the LR parsing algorithms with the token peeking ability, and keeps the size growth of the LR DFAs internal state space under control. In other words, LALR parsing sacrifices a finer grained parser internal state description so we can create DFAs with realistic size. In our case study, we particularly focus on an LALR parsing variant with only one lookahead token, i.e. LALR(1) parsing.

6.1.3 LALR(1) Specific Domain Meta-Model

In Section 5.1, we have discussed the general parsing domain, which includes the common concepts shared by LL(*) parsing domain and the LALR(1) parsing domain, e.g. terminal, nonterminal, parse rule, alternative, etc. In this section, we focus on the classes and properties in the domain meta-model that are uniquely related to the LALR(1) parsing.

Shifting and Reducing In both LL(*) parsing and LALR(1) parsing, the parser keeps a cursor that tracking the parsing progress, which divides the input token string into two substrings: the left parsed part and the right unparsed part. The major difference is the way to move the cursor. The LL(*) parser tries to parse in a "predict/match" pattern, i.e. try to parse the following token with a certain production rule that is predicted according to the current lookahead token, until an error might occur and then backtrack. In other words, the cursor may shift either to the left (when backtracking) or to the right, but it will finally reach the right end to finish the parsing process.

The LALR(1) parser, on the other hand, treats the left string as its work area, where all handles get reduced at the right end of it. Its cursor only shift to the right, as each symbol in the left string is part of a handle. Therefore, in LALR(1) parsing, there are no longer steps like "predicting" or "matching". Instead, the basic parsing operations are "shifting" and "reducing". "Shifting" refers to moving a token across the cursor from the right to the left for reduction. It is similar to the notion "matching a token" in LL(*) parsing. The difference is that when a LALR(1) parser shifts, it is moves the very first unparsed token to the end of the left string, which means it is "ready for reduction". "Reducing" refers to reducing the current handle to the proper nonterminal, i.e. the application of reverse production rule. In detail, it consists of three logical steps. First, the parser pops out all the terminals and/or nonterminals of the current handle. It then replaces the handle into a single nonterminal. Finally, it pushes the reduced nonterminal back to the parse stack.

Handle Detecting and Pruning As a bottom-up parsing algorithm, LALR(1) tries to reduce the predicted parse tree by always trimming the left most subtree from leaf level. As the detecting of each handle has been encoded in the states of the pre-built DFA, it does not reflect directly in the parse table. On detecting a handle, the instruction indicated by the parse table may be either to shift or to reduce. Therefore, the parser cannot tell the inception of the parsing of a certain parse rule on recognizing a specific terminal. In other words, we cannot pinpoint the specific time point when the parser starts parsing a specific production rule during parsing. The pruning of handle, on the contrary, is much easier to detect, as it is the major step in the "reducing" action. It transforms a group of symbols (either terminal or nonterminal) into one nonterminal according to a specific parse rule alternative.

Parse Stack In LALR(1) parsing process, the parsed input is stored in a stack structure, which is often referred to as *Parse Stack*. It serves the detecting and pruning of handles. More precisely, if the action according to the parse table is to "shift", the parser simply pushes the current token into the stack. If the action is to "reduce", the stack pops out the symbols (either terminal or nonterminal), so that the parser can reduce them into a single nonterminal and then push it back to the stack. Obviously, the elements in the stack can be either terminals or nonterminals.

Embedded Action In LL(*) parsing, the "predict/match" pattern allows the parser to be aware of both the commencement and completion of parsing a certain parse rule. Therefore, LL(*) grammar allows the users to include customized code to these location. The code are called *embedded action.* In LALR(1) parsing, however, the users can only insert their code on finishing the parsing of a certain production rule. In other words, embedded actions are allowed only at the very end of the second step of "reducing", i.e. handle pruning. Another limitation due to the invisibility of handle detection is that there is no tracking on parse rule scale. For example, the users cannot set a flag at the beginning of a rule parsing process, which may get modified through the parsing, and check it on finishing the parsing process. Thus the LALR(1) grammar does not support parse rule attribute in embedded action.

By including the XSD module we created in Section 5.2.3, we define the LALR(1) grammar domain meta-model in another XSD module called `LALR_1_Domain`, which is shown in Appendix B.21.

6.2 Extending CUP Parser Generation

As an comparative experiment, we start our case study by modifying the "RERS" parser generated by CUP. We write an LALR(1) grammar for the "RERS" language that we used with in our ANTLR experiment in Section 5.3.1. Strictly speaking, the grammar consists of two separate files: `RERS.flex` for terminal definition only and `RERS.cup` for the rest of the grammar. The output are three separate files: the lexer `MyScanner.java` generated by JFlex from `RERS.flex`, the parser `Parser.java` generated by CUP from `RERS.cup`, and an auxiliary class `sym.java`. As we only focus on the parser generation process, we refer to the generated `Parser.java` by "base code".

The specifications of the CUP grammars are easy to understand. The general structure look similar to that of the ANTLR grammars, except that they require explicit declarations of nonterminals and allow the definitions of precedences and associativity of the operators. The input model, i.e. the `RERS.cup`, is shown in Appendix B.22. As for the production rules, CUP grammars do not support rule parameters or attributes. Instead, they only support a built-in return value for each rule, which can be referred to as RESULT in the embedded actions.

6.2.1 Extension Requirements Analysis

The extension requirement in this experiment is quite similar to that we mentioned in Section 5.3.2, to develop a finer grained and more flexible monitoring functions to the generated parsers, e.g. to monitor the token matching events. However, due to the different parsing mechanisms of the LL(*) grammars and the LALR grammars, we can identify the following differences. First, as the applications of production rules can only be detected when they are finished, we can only monitor the exiting of a production rule, just like the embedded actions allowed in CUP. Second, as CUP does not support rule attribute, we no longer need to monitor the attribute setting events. Third, we want to monitor the “shift” action. Fourth, we want to monitor the handle reduction, in terms of the handle detection and the handle pruning.

The Effective Meta-Model According to the four events to monitor in the above extension requirement, which include “token matching”, “invoking rule embedded action”, “shift action”, “handle reduction”, we can identify four join point classes in the LALR(1) domain meta-model, i.e. class “terminalType”, class “embeddedAction”, class “shiftAction”, and class “reduceAction”. The first two classes are instance level join point classes. The last two classes have no properties, and are class level join point classes. This difference will affect the expansion of the corresponding PCD production rules in the DSAL template, which will be explained in Section 6.2.2.

The DSAL Template Selection Given the fact that the differences between the modification requirements in the AspectRERS ANTLR experiment and that in the AspectRERS CUP experiment are all domain specific, we can simply reuse the DSAL template we used in Section 5.4.1.

6.2.2 Generation of AspectRERS

In this experiment, we still call the DSAL to generate *AspectRERS*. The generation of AspectRERS is essentially an expansion of the selected DSAL template in accordance with the effective meta-model, especially the creation of the four domain specific PCD definitions corresponding to the four join point classes.

Token Matching Pointcuts describe the join points focusing on the points tokens are matched, i.e. before or after them. In LR parsing, the beginning of applying a production rule is difficult to detect. In other words, it is difficult to tell whether a token is matched within the application of a specific production rule. Therefore, we no longer trace the token matching events within the rule application frames. Instead, we regard them as standalone events. For example, pointcut after(#9) captures all “slots” just after matching a token with internal index value 9. Besides, we also support an unary operator ! to get the complement, which means “any ... other than”. For example, pointcut before(!#9) refers to the “slots” just before matching any token, except for the one with internal index value 9. The ANTLR grammar definition of the token matching pointcuts is shown in Listing 6.1.

```
token_filter
    :   BEF LPAREN tn=token_indicator RPAREN
        -> beforeTokenFilter1(locationModifier={"Begin"},
   tokenIdx={$tn.code})
```

```
    |   BEF LPAREN NOT tn=token_indicator RPAREN
        -> beforeTokenFilter2(locationModifier={"Begin"},
   tokenIdx={$tn.code})
    |   AFT LPAREN tn=token_indicator RPAREN
        -> afterTokenFilter1(locationModifier={"End"},
   tokenIdx={$tn.code})
    |   AFT LPAREN NOT tn=token_indicator RPAREN
        -> afterTokenFilter2(locationModifier={"End"},
   tokenIdx={$tn.code})
    ;


location_modifier returns [String code]
    :   BEF {$code = "Begin";}
    |   AFT {$code = "End";}
    ;

token_indicator returns [String code]
    :   TKN INTLITERAL {$code=$INTLITERAL.text;}
    |   TKN {$code="-1";}
    ;
```

Listing 6.1: ANTLR definition of the token matching pointcuts

Embedded Action Pointcuts describe the join points before or after an embedded action that is executed after a certain production rule is applied. For example, pointcut before(@temporal : 2) captures the "slots" just before executing the embedded action that follows the application of the second alternative of rule @temporal. Similar to the branch pointcuts we described in Section 5.4.1, both columns of the embedded action pointcuts support the ! operator. For instance, (@temporal :!2) refers to the application of any alternative of rule temporal except the second one, whereas (!@temporal : 2) refers to the application of the second alternative of any rule except temporal. The ANTLR grammar definition of the embedded action pointcuts is shown in Listing 6.2.

```
reduction_action_filter
    :   lm=location_modifier LPAREN rn=rule_indicator COLON
   bi=branch_indicator RPAREN
        -> reductionActionFilter1(locationModifier={$lm.code
   }, ruleNameValue={$rn.value}, altIndex={$bi.code})
    |   lm=location_modifier LPAREN rn=rule_indicator COLON
   NOT bi=branch_indicator RPAREN
```

```
        -> reductionActionFilter2(locationModifier={$lm.code
   }, ruleNameValue={$rn.value}, altIndex={$bi.code})
    |   lm=location_modifier LPAREN NOT rn=rule_indicator
   COLON bi=branch_indicator RPAREN
        -> reductionActionFilter3(locationModifier={$lm.code
   }, ruleNameValue={$rn.value}, altIndex={$bi.code})
    |   lm=location_modifier LPAREN NOT rn=rule_indicator
   COLON NOT bi=branch_indicator RPAREN
        -> reductionActionFilter4(locationModifier={$lm.code
   }, ruleNameValue={$rn.value}, altIndex={$bi.code})
    ;

branch_indicator returns [String code]
    :   INTLITERAL {$code=$INTLITERAL.text;}
    |   {$code="*";}
    ;

rule_indicator returns [String value]
    :   RUL IDENTIFIER
        {$value=$IDENTIFIER.text;}
    |   {$value="*";}
    ;
```

Listing 6.2: ANTLR definition of the embedded action pointcuts

Shift Action Pointcuts describe the join points before or after the "shift" action, which is an internal state transition of the parser. As the corresponding join point class "shiftAction" is a class level join point class, which means join point instances do not relate to any domain element, such as token or production rule, we define two special pointcuts, BeforeShift and AfterShift, which respectively capture the beginning and ending slots of *ANY* shift action. The corresponding ANTLR definition is shown in Listing 6.5.

```
    BEFORE_SHIFT = 'BeforeShift';
    AFTER_SHIFT = 'AfterShift';
shift_filter
    :   BEFORE_SHIFT
        -> shiftFilter1()
    |   AFTER_SHIFT
        -> shiftFilter2()
    ;
```

Listing 6.3: ANTLR definition of the shift action pointcuts

To distinguish the pointcut corresponding to the class level join points, we define a percent character before the PCD names. The corresponding ANTLR definition is shown in Listing 6.4.

```
macro_filter
    :   PERCENT m=macro_filter_expression
        -> copyStr(f={$m.st})
    ;

macro_filter_expression
    :   bme=before_main_exit
        -> copyStr(f={$bme.st})
    |   hf=handle_filter
        -> copyStr(f={$hf.st})
    |   sf=shift_filter
        -> copyStr(f={$sf.st})
    ;
```

Listing 6.4: ANTLR definition of the class level pointcuts

Handle Reduction Pointcuts describe the join points before or after the operations related to the "reduce" action, i.e. handle detection and handle pruning, which are also internal actions of the LR parser. Similar to the shift action pointcuts, it also correspond to a class level join point class "reduceAction". Similarly, we define two special pointcuts FoundHandle and PrunedHandle. The corresponding ANTLR definition is shown in Listing 6.5.

```
    FOUND_HANDLE = 'FoundHandle';
    PRUNED_HANDLE = 'PrunedHandle';
handle_filter
    :   FOUND_HANDLE
        -> handleFilter1()
    |   PRUNED_HANDLE
        -> handleFilter2()
    ;
```

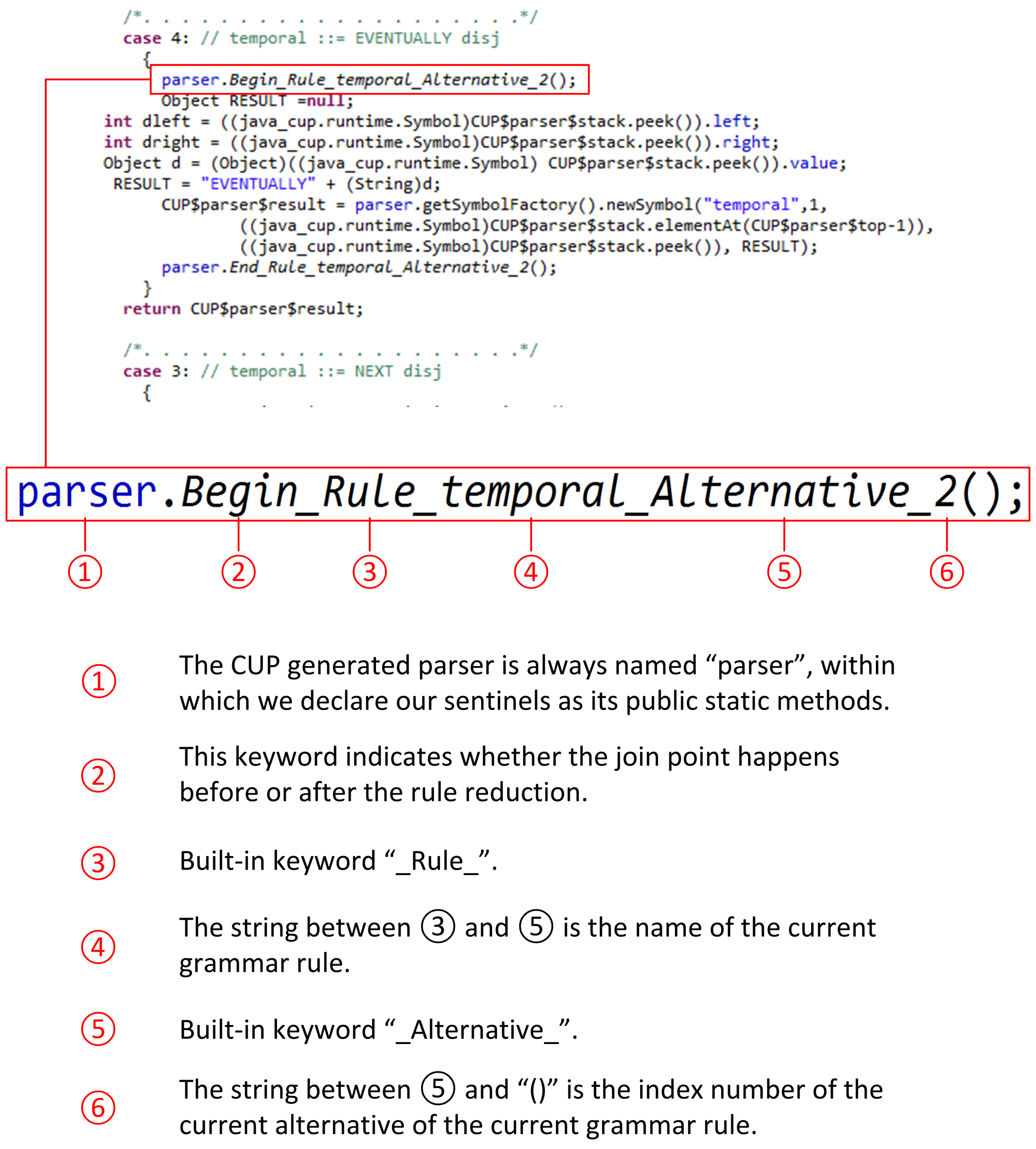
Listing 6.5: ANTLR definition of the handle reduction pointcuts

We can see that all above pointcuts are pinpoint pointcuts. The LR parsers focus on exploring handles in the parse stack, instead of explicitly tracing the applications of production rules. Concepts, like the control flow of rule application, are thus of little interest in the extension. Therefore, we do not define any "range" pointcuts, such as cflow in our AspectRERS. The AspectRERS here supports the same advice patterns, i.e. *before* and *after* advice, while it still does not support any parameter or return values bound to a certain pointcut. Similar to the four DSESs defined in Section 5.6.2, we define six DSESs as shown below, which will be syntactically taken as normal Java statements.

- **GetParsedText** takes in no parameter and returns a java.lang.String typed object containing the parsed token string.
- **GetStackToken** takes in an int typed parameter representing the index of the target token in the parse stack, and returns a java_cup.runtime.Symbol typed object containing the target token. The indices are 0 based.
- **GetCurrentToken** takes in no parameter and returns a java_cup.runtime.Symbol typed object containing the current token.
- **GetCurrentBranch** takes in no parameter and returns a java.lang.String typed object representing the production rule and its alternative indices, e.g. literal_f : 1.
- **GetCurrentBranchText** takes in no parameter and returns a java.lang.String typed object representing the full text of the production rule and its alternative indices, e.g. literal_f ::= LPAR temporal RPAR.
- **PrintHandleTokens** takes in no parameter and use standard output stream to print out a java.lang.String typed object for each token that comprises the current handle.

6.2.3 Traceable Domain Meta-Model

Due to the similar model traceability requirements as that in Section 5.4.2, we reuse most part of the tracing strategy used in the previous experiment. For example, Figure 6.1 illustrates a sentinel for the beginning of the embedded actions of applying rule temporal.



```
/*. . . . . . . . . . . . . . . . . . . .*/
case 4: // temporal ::= EVENTUALLY disj
  {
    parser.Begin_Rule_temporal_Alternative_2();
    Object RESULT =null;
int dleft = ((java_cup.runtime.Symbol)CUP$parser$stack.peek()).left;
int dright = ((java_cup.runtime.Symbol)CUP$parser$stack.peek()).right;
Object d = (Object)((java_cup.runtime.Symbol) CUP$parser$stack.peek()).value;
 RESULT = "EVENTUALLY" + (String)d;
    CUP$parser$result = parser.getSymbolFactory().newSymbol("temporal",1,
            ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt(CUP$parser$top-1)),
            ((java_cup.runtime.Symbol)CUP$parser$stack.peek()), RESULT);
    parser.End_Rule_temporal_Alternative_2();
  }
return CUP$parser$result;

/*. . . . . . . . . . . . . . . . . . . .*/
case 3: // temporal ::= NEXT disj
  {
```

```
parser.Begin_Rule_temporal_Alternative_2();
```

① The CUP generated parser is always named "parser", within which we declare our sentinels as its public static methods.

② This keyword indicates whether the join point happens before or after the rule reduction.

③ Built-in keyword "_Rule_".

④ The string between ③ and ⑤ is the name of the current grammar rule.

⑤ Built-in keyword "_Alternative_".

⑥ The string between ⑤ and "()" is the index number of the current alternative of the current grammar rule.

FIGURE 6.1: The beginning sentinel marking the embedded actions of rule temporal.

It is worth noting that our sentinels are marking the beginnings and endings of the embedded actions after applying a rule, not the beginnings and endings of a rule application.

Code Generator Customization With the generated traceable domain meta-model as formal guide, we start the customization of the latest CUP package `cup-11a.jar`, in which the parser generation function is mainly implemented in two classes: `emit.java` and `lr_parser.java`. The `lr_parser.java` class is the base class of the generated `Parser.java` class, which contains the core functionalities of LR parsing, such as token matching, "shift" and "reduce" actions. The related sentinels can thus be inserted by directly modifying this class. For example, the sentinels for token matchings and "shift" actions are inserted as shown in Figure 6.2.

```
/* decode the action -- > 0 encodes shift */
if (act > 0)
  {
    /* Sentinel insertion: shift&reduction */
  Begin_Shift();

    /* shift to the encoded state by pushing it on the stack */
    cur_token.parse_state = act-1;
    cur_token.used_by_parser = true;
    stack.push(cur_token);
    tos++;

    /* Sentinel insertion: token matching */
    Begin_Match_Token();
    /* advance to the next Symbol */
    cur_token = scan();
    /* Sentinel insertion: token matching */
    End_Match_Token();

    /* Sentinel insertion: shift&reduction */
  End_Shift();
  }
/* if its less than zero, then it encodes a reduce action */
else if (act < 0)
```

FIGURE 6.2: Part of the modified code in `lr_parser.java` to generate sentinels for the token matchings and "shift" actions.

The `emit.java` class is responsible to assemble the text string of the source code of the `Parser.java` class and get it printed. We modify this class mainly for generating the sentinels of the rule embedded actions. For example, we use a Hashtable<String, Integer> object to maintain the mapping between each alternative of a given production rule and its index, so that we can fetch the correct alternative index information whenever we need to concatenate the sentinel strings. More details are shown in Figure 6.3.

It is worth noting that we add a comment starting with "/* Sentinel insertion:" before each piece of code that we modify to generate the sentinels. After the modification for all the sentinels in the CUP package, we recompile the whole package and create a new CUP parser generator, `CUPS.jar` which stands for "CUP customized for Sentinels".

```
      // TUM 20060327 added SymbolFactory aware constructor
out.println();
out.println("  /** Constructor which sets the default scanner. */");
out.println("  public " + parser_class_name +
      "(java_cup.runtime.Scanner s, java_cup.runtime.SymbolFactory sf) {super(s,sf)
  }

  /* Sentinel insertion: for calc alternative index */
  for (Enumeration p = production.all(); p.hasMoreElements(); ) {
    production prod = (production)p.nextElement();
    String label = prod._lhs._the_symbol.name();
    if (alternatives.containsKey(label)) {
      alternatives.put(label, alternatives.get(label) + 1);
    } else {
      alternatives.put(label, 1);
    }
  }

  /* Sentinel insertion: sentinel declaration insertion here */
  out.println("  /** sentinel declaration */");
  for (Enumeration<String> r = alternatives.keys(); r.hasMoreElements(); ) {
    String rule_lhs = r.nextElement();
    for (int i=1; i <= alternatives.get(rule_lhs); i++) {
      out.println("  public static void Begin_Rule_" + rule_lhs + "_Alternative_"
      + i + "() {ruleTrace.push(\""+rule_lhs+"\"); branchTrace.push(\""+rule_lhs
      +":"+i+"\");}");
      out.println("  public static void End_Rule_" + rule_lhs + "_Alternative_"
      + i + "() {branchTrace.pop(); ruleTrace.pop();}");
    }
  }

  /* Sentinel insertion: rule&branch tracking stack declaration insertion here */
  out.println("  /** rule tracking stack declaration */");
  out.println("  public static Stack<String> ruleTrace = new Stack<String>();");
  out.println("  /** branch tracking stack declaration */");
  out.println("  public static Stack<String> branchTrace = new Stack<String>();");

  /* emit the various tables */
  emit_production_table(out);
  do_action_table(out, action_table, compact_reduces);
  do_reduce_table(out, reduce_table);

  /* instance of the action encapsulation class */
  out.println("  /** Instance of action encapsulation class. */");
  out.println("  protected " + pre("actions") + " action_obj;");
  out.println();
```

FIGURE 6.3: Part of the modified code in `emit.java` to declare the sentinel functions of the rule embedded actions.

6.2.4 Generation of AspectRERS Weaver

In this experiment, we still build the AspectRERS aspect weaver as a translator from AspectRERS into AspectJ. With the above ANTLR grammar of AspectRERS, we define the corresponding string templates to rewrite the recognized AspectRERS aspects into the corresponding AspectJ aspects. All these string template definitions are encapsulated in `AspectRERS.stg`, as in the AspectRERS in the ANTLR extension experiment. However, there are some difference in details. First, our aspects no longer need the names of the input grammar as parameters, as parser class generated by CUP are always named `Parser.java`. For example, an AspectRERS aspect definition public aspect Sample {...} simply remains the same in its translation. Second, different pointcuts and underlying sentinels require different ways of constructing the translated AspectJ strings.

Token Matching Pointcut Translation Because of the implementation of the CUP code generation process, we can only insert general token matching sentinels, i.e. call(public void java_cup.runtime.lr_parser.[Begin/End]_Match_Token(). This means we can no longer generate a simple "call" pointcut for the matching of a specific token, as the sentinels for all tokens are the same. Therefore, we use an extra "if" pointcut in Aspect to indicate the target token. For example, if we want to capture the beginning of the matchings of token OV (index value 20), we define a pointcut as shown below.

```
    pointcut bef_OV=before(#20);
```

The AspectJ translation of the above pointcut is as follows.

```
    private pointcut bef_OV() : (call(public void java_cup.
   runtime.lr_parser.Begin_Match_Token())&&if(java_cup.
   runtime.lr_parser.getCurToken().sym==20));
```

Listing 6.6 shows the definitions of the string templates behind the above translation.

```
beforeTokenFilter1(locationModifier, tokenIdx) ::= <<
call(public void java_cup.runtime.lr_parser.<
   locationModifier>_Match_Token())&&if(java_cup.runtime.
   lr_parser.getCurToken().sym==<tokenIdx>)
>>

beforeTokenFilter2(locationModifier, tokenIdx) ::= <<
call(public void java_cup.runtime.lr_parser.<
   locationModifier>_Match_Token())&&if(java_cup.runtime.
   lr_parser.getCurToken().sym!=<tokenIdx>)
>>

afterTokenFilter1(locationModifier, tokenIdx) ::= <<
call(public void java_cup.runtime.lr_parser.<
   locationModifier>_Match_Token())&&if(java_cup.runtime.
   lr_parser.getPrevToken().sym==<tokenIdx>)
>>

afterTokenFilter2(locationModifier, tokenIdx) ::= <<
call(public void java_cup.runtime.lr_parser.<
   locationModifier>_Match_Token())&&if(java_cup.runtime.
   lr_parser.getPrevToken().sym!=<tokenIdx>)
>>
```

Listing 6.6: String template definitions behind the translation of the token matching pointcuts

Embedded Action Pointcut Translation As we still generate alternative specific sentinels for the embedded action join points, we can reuse our translation strategy in our ANTLR experiment. For specific pointcuts, we simply construct the declarations of the corresponding sentinels. For example,

```
    pointcut bef_act=before(@literal_f:1);
```

is translated into

```
    private pointcut bef_act() : (call(public void Parser.
  Begin_Action_Rule_literal_f_Alternative_1(String)));
```

For generic pointcuts, we use ∗ to replace the specific alternative index in the translated string. For example,

```
    pointcut aft_act=after(@input:);
```

is translated into

```
    private pointcut aft_act() : (call(public void Parser.
  End_Action_Rule_input_Alternative_*(String)));
```

The definitions of the string templates behind the above translation are shown in Appendix B.23.

Special Pointcut Translation The last two kinds of pointcuts, i.e. the shift action pointcuts and the handle reduction pointcuts, are all special pointcuts. They are translated directly into predefined AspectJ pointcuts. The definitions of the string templates involved here are shown in Appendix B.24.

DSES Translation Similar to the translation of the special pointcuts, we directly translate the DSESs into predefined strings. The related string template definitions are shown in Listing 6.7.

```
getParsedText() ::= <<
java_cup.runtime.lr_parser.getParsedText()
>>

getStackSymbol(index) ::= <<
((Parser)thisJoinPoint.getTarget()).getStackSymbol(<index>)
>>

getCurrentSymbol() ::= <<
java_cup.runtime.lr_parser.getCurToken()
```

```
>>

getCurrentBranch() ::= <<
((Parser)thisJoinPoint.getTarget()).branchTrace.peek()[0]
>>

getCurrentBranchText() ::= <<
((Parser)thisJoinPoint.getTarget()).branchTrace.peek()[1]
>>

printHandleSymbols() ::= <<
for (int i = ((Parser)thisJoinPoint.getTarget()).
   getStackSize()-((Parser)thisJoinPoint.getTarget()).
   currentHandleSize; i \< ((Parser)thisJoinPoint.getTarget
   ()).getStackSize(); i++) { System.out.print(((Parser)
   thisJoinPoint.getTarget()).getStackSymbol(i).toString());
    } System.out.println()
>>
```

Listing 6.7: String template definitions behind the translation of the advice macros

6.2.5 Customization of CUP Parser with AspectRERS Aspect Weaving

Now that we have implemented a complete parser generation extension system based on AspectRERS, we test it with the same RERS program shown in Listing 5.25, which we used in our ANTLR test. For the sake of convenience, it is also shown below.

```
(! oZ WU (oU & ! oZ))
(G (! iC | (F oZ)))
((G ! oW) | (F (oW & (F oU))))
(G (! iE | (F oY)))
(G (! (iB & ! oU) | (! oV WU oU)))
(! oV WU oX)
((G ! oW) | (F (oW & (F oU))))
(! oU WU (oX & ! oU))
```

As mentioned in Section 6.2.1, our extension purpose is to provide a finer grained and more flexible monitoring system over the parsing process. In our AspectRERS aspect, we define four pairs of pointcuts to respectively capture the beginnings and endings of the four types of parsing events that we are interested in, as well as their corresponding advice for proper operations.

To test the monitoring of the token matching events, we capture the beginning of matching token OV (index value 20) and the ending of matching token IB (index value 14). We then print the parsed text, the third token from the top of the parse stack, and the current token. The corresponding code block in our test aspect is shown in Appendix B.25.

To test the monitoring of the embedded actions, we capture the beginning of the action after a specific rule alternative, i.e. literal_f : 1, and the ending of the action after any alternative of rule input. Apart from the above advice operations, we also print the current branch index and its full text, as well as the handle that has just been reduced. More details are shown in Appendix B.26.

We then test the capturing of the beginning and ending of the "shift" action, as well as the "reduce" action, where we respectively print the parsed text, and the top token in the parse stack, plus the current handle. More details are shown in Appendix B.27.

With regard to the test aspect in our previous AspectRERS ANTLR experiment, we also import its first pointcut to count the occurrence of the "GLOBALLY" expression. Unfortunately, since it is difficult to detect the beginning of the rule application in LR parsing, we cannot support the range pointcuts. Therefore we cannot import its second pointcut here. The detailed aspect code block is shown in Listing 6.8.

```
 // counting "GLOBALLY" expression
 int GLOBALLY_expression_counter=0;
 pointcut GLOBALLY_expression=after(@temporal:3);
 after : GLOBALLY_expression() {
     GLOBALLY_expression_counter++;
 }

 pointcut summary=%BeforeMainExit;
 after : summary() {
     System.out.println("There are " +
GLOBALLY_expression_counter + " \"GLOBALLY\" expressions.
");
 }
```

Listing 6.8: The code block of counting the "GLOBALLY" expression in the test aspect

The result of parsing the RERS program is shown in Listing 5.25. According to it, there 5 "GLOBALLY" expressions in the above program, which agrees with the test result of the AspectRER ANTLR experiment, which is shown in Section 5.5.

6.3 Extending Java Parser Generation

For the purpose of comparison, we aim to equip the Java parser generated by CUP the same functions as in our ANTLR AspectJava experiment. Unfortunately, due to the limitation on the range pointcuts in LR parsing, we have to abandon the functions requiring the ability to trace the parsing control flow. In other words, we lack the ability to capture the rule application under a specific context in terms of the previously applied rules. For example, we cannot capture the "equality check" in the expressions in the "if" predicates, although we can still count the "if-only" statements. As for the LOC counting function, the sub statement counting involved in it still requires the ability to trace the parsing control flow. Although we can change the counting rules to skip those situations in counting, it would not be a proper comparison any more. As a result, we continue our Java parser extension experiment with only one requirement. Such extension requirement can be fulfilled by the AspectRERS we generated in the above experiment. Therefore, in this experiment we simple reuse the entire extension of the CUP parser generation, except that we rename the DSAL as *AspectJava*.

The way we count the "if-only" statements is different here. In the ANTLR experiment, we respectively capturing all the "if" statements and all the "else" tokens, and then get the count by calculating their difference, as it is not easy to directly capture the "if-only" statements. In the CUP experiment, we download a Java grammar from the CUP official home page [134]. It explicitly defines a if_then_statement rule, which is shown in Listing 6.9.

```
statement ::=   statement_without_trailing_substatement
        |       labeled_statement
        |       if_then_statement
        |       if_then_else_statement
        |       while_statement
        |       for_statement
        |       foreach_statement
        ;
if_then_statement ::=
                IF LPAREN expression RPAREN statement
        ;
```

Listing 6.9: The CUP grammar of the "if-only" statement in Java

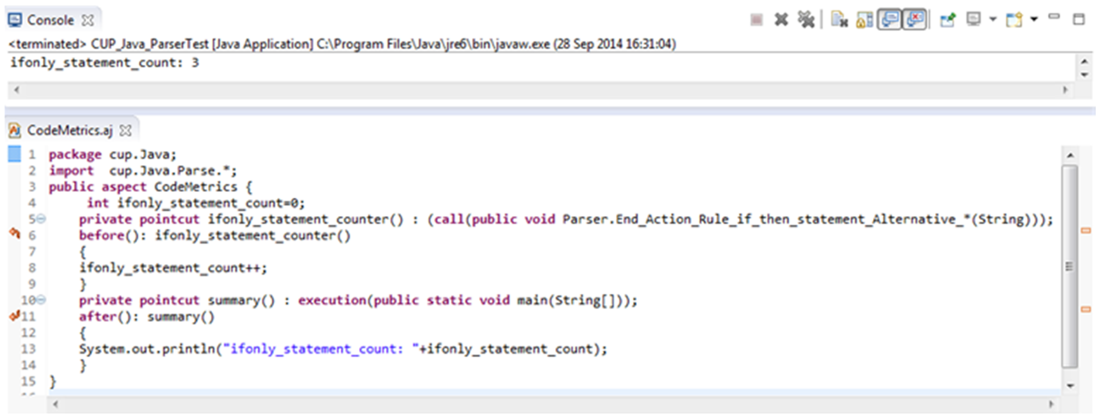
This enables us to get the count by defining a pointcut that directly captures the ending of the application of this rule, i.e. an embedded action pointcut. The corresponding AspectJava aspect code is shown in Listing 6.10.

```
package cup.Java;
import cup.Java.Parse.*;
```

```
public aspect CodeMetrics {
    int ifonly_statement_count=0;
    pointcut ifonly_statement_counter=after(
   @if_then_statement:);
    before : ifonly_statement_counter() {
        ifonly_statement_count++;
    }

    pointcut summary=%BeforeMainExit;
    after : summary() {
        System.out.println("ifonly_statement_count: "+
   ifonly_statement_count);
    }
 }
```

Listing 6.10: The AspectJava code of counting the "if-only" statement

We run the test against two input Java programs. The first program is the artificial program `testJavaProg.java` we used in Section 5.6.4. The translated AspectJ aspect and the test result is shown in Figure 6.4. The result agrees with our previous test result shown in Figure 5.3, which has been verified manually.

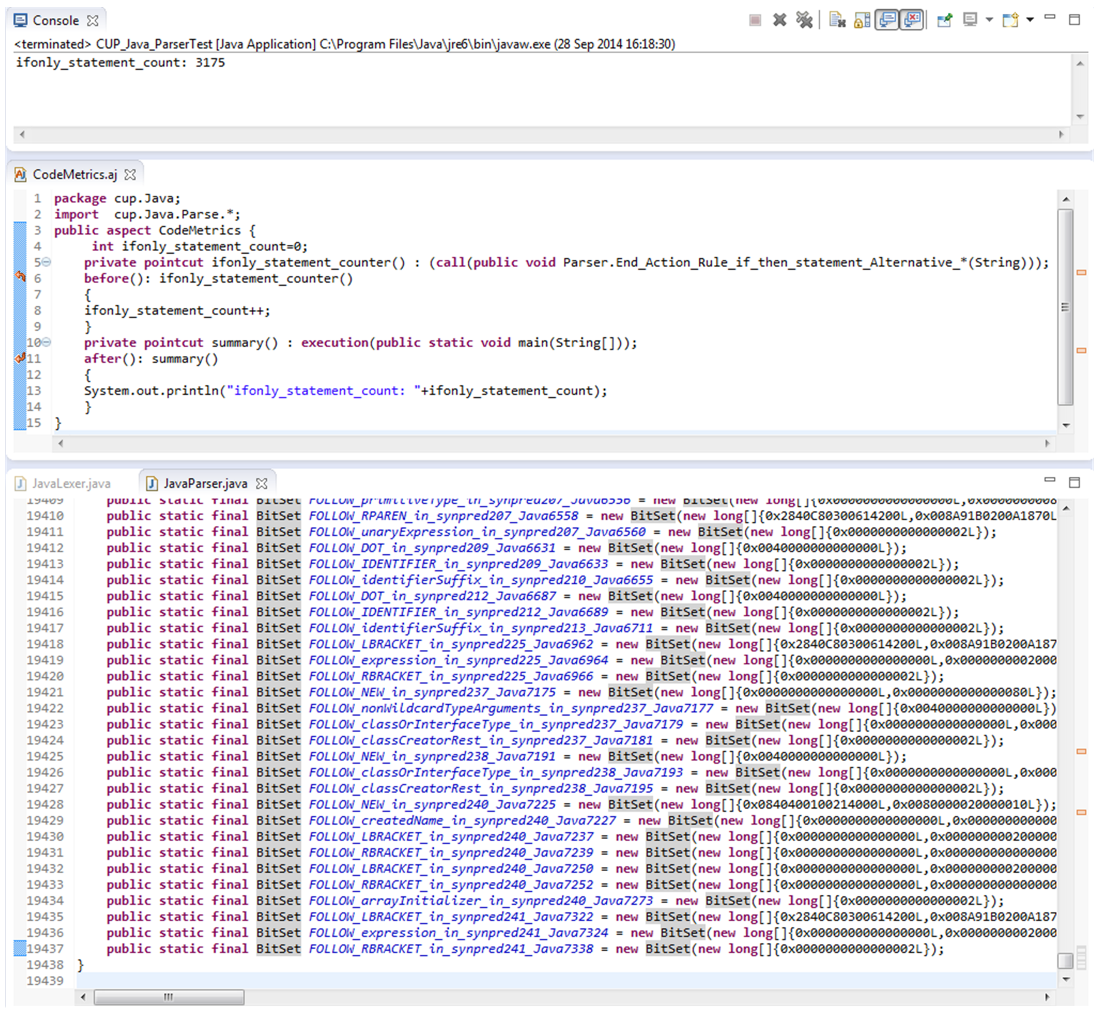


```
Console
<terminated> CUP_Java_ParserTest [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (28 Sep 2014 16:31:04)
ifonly_statement_count: 3
```

```
CodeMetrics.aj
package cup.Java;
import  cup.Java.Parse.*;
public aspect CodeMetrics {
     int ifonly_statement_count=0;
    private pointcut ifonly_statement_counter() : (call(public void Parser.End_Action_Rule_if_then_statement_Alternative_*(String)));
    before(): ifonly_statement_counter()
    {
    ifonly_statement_count++;
    }
    private pointcut summary() : execution(public static void main(String[]));
    after(): summary()
    {
    System.out.println("ifonly_statement_count: "+ifonly_statement_count);
    }
}
```

Figure 6.4: The CUP output of parsing `testJavaProg.java` with the `IfPredicateChecker`-woven JavaParser.

The second test program is the `JavaParser.java` generated in the previous LOCCounter experiment in Section 5.6.4, which has 19438 lines of text. The test result is that there are 3175 "if-only" statements in the program `JavaParser.java`, as shown in Figure 6.5.



```
<terminated> CUP_Java_ParserTest [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (28 Sep 2014 16:18:30)
ifonly_statement_count: 3175
```

CodeMetrics.aj

```
package cup.Java;
import  cup.Java.Parse.*;
public aspect CodeMetrics {
     int ifonly_statement_count=0;
    private pointcut ifonly_statement_counter() : (call(public void Parser.End_Action_Rule_if_then_statement_Alternative_*(String)));
    before(): ifonly_statement_counter()
    {
    ifonly_statement_count++;
    }
    private pointcut summary() : execution(public static void main(String[]));
    after(): summary()
    {
    System.out.println("ifonly_statement_count: "+ifonly_statement_count);
    }
}
```

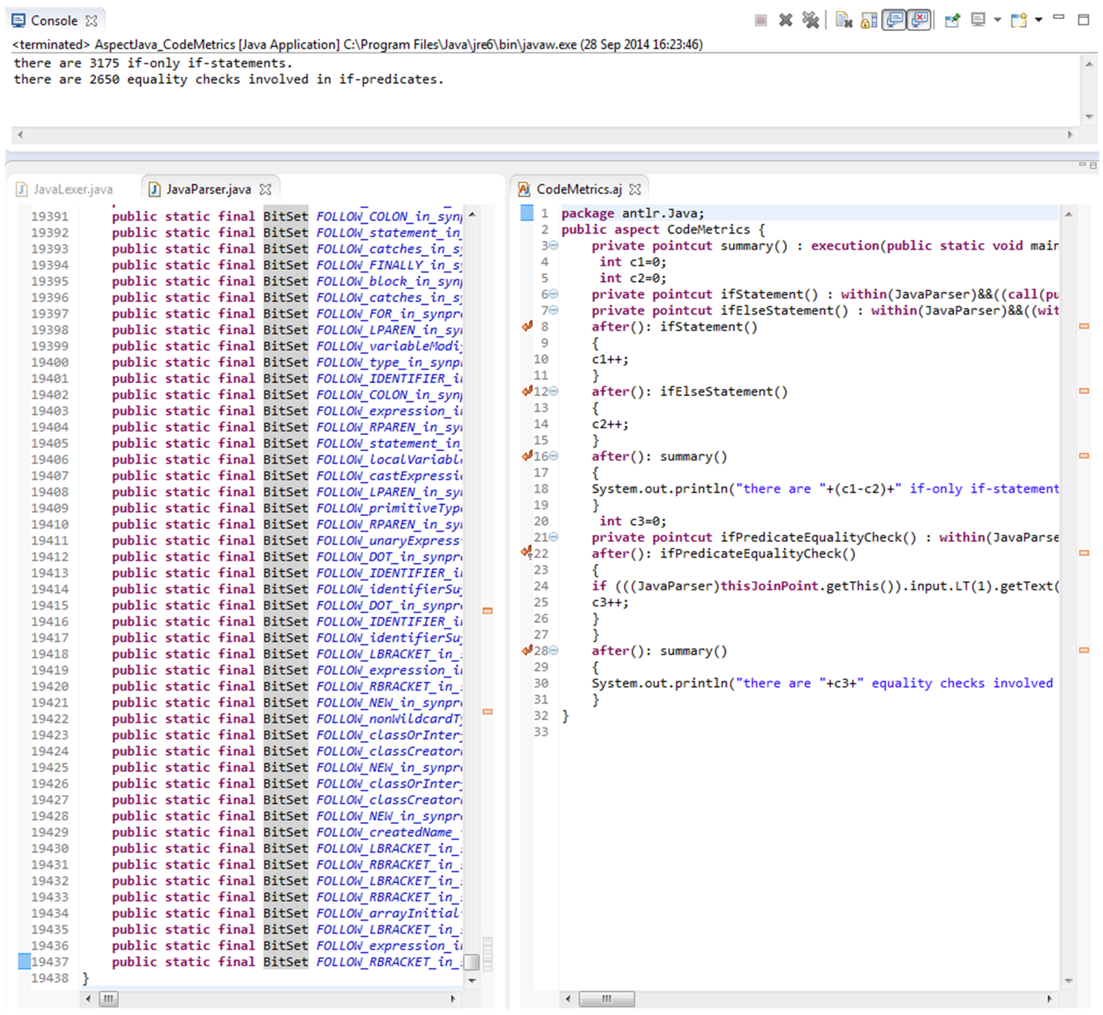
JavaLexer.java   JavaParser.java

```
    public static final BitSet FOLLOW_RPAREN_in_synpred207_Java6558 = new BitSet(new long[]{0x2840C80300614200L,0x008A91B0200A1870L
    public static final BitSet FOLLOW_unaryExpression_in_synpred207_Java6560 = new BitSet(new long[]{0x0000000000000002L});
    public static final BitSet FOLLOW_DOT_in_synpred209_Java6631 = new BitSet(new long[]{0x0040000000000000L});
    public static final BitSet FOLLOW_IDENTIFIER_in_synpred209_Java6633 = new BitSet(new long[]{0x0000000000000002L});
    public static final BitSet FOLLOW_identifierSuffix_in_synpred210_Java6655 = new BitSet(new long[]{0x0000000000000002L});
    public static final BitSet FOLLOW_DOT_in_synpred212_Java6687 = new BitSet(new long[]{0x0040000000000000L});
    public static final BitSet FOLLOW_IDENTIFIER_in_synpred212_Java6689 = new BitSet(new long[]{0x0000000000000002L});
    public static final BitSet FOLLOW_identifierSuffix_in_synpred213_Java6711 = new BitSet(new long[]{0x0000000000000002L});
    public static final BitSet FOLLOW_LBRACKET_in_synpred225_Java6962 = new BitSet(new long[]{0x2840C80300614200L,0x008A91B0200A187
    public static final BitSet FOLLOW_expression_in_synpred225_Java6964 = new BitSet(new long[]{0x0000000000000000L,0x000000002000
    public static final BitSet FOLLOW_RBRACKET_in_synpred225_Java6966 = new BitSet(new long[]{0x0000000000000002L});
    public static final BitSet FOLLOW_NEW_in_synpred237_Java7175 = new BitSet(new long[]{0x0000000000000000L,0x0000000000000080L});
    public static final BitSet FOLLOW_nonWildcardTypeArguments_in_synpred237_Java7177 = new BitSet(new long[]{0x0040000000000000L})
    public static final BitSet FOLLOW_classOrInterfaceType_in_synpred237_Java7179 = new BitSet(new long[]{0x0000000000000000L,0x000
    public static final BitSet FOLLOW_classCreatorRest_in_synpred237_Java7181 = new BitSet(new long[]{0x0000000000000002L});
    public static final BitSet FOLLOW_NEW_in_synpred238_Java7191 = new BitSet(new long[]{0x0040000000000000L});
    public static final BitSet FOLLOW_classOrInterfaceType_in_synpred238_Java7193 = new BitSet(new long[]{0x0000000000000000L,0x000
    public static final BitSet FOLLOW_classCreatorRest_in_synpred238_Java7195 = new BitSet(new long[]{0x0000000000000002L});
    public static final BitSet FOLLOW_NEW_in_synpred240_Java7225 = new BitSet(new long[]{0x0840400100214000L,0x0080000020000010L});
    public static final BitSet FOLLOW_createdName_in_synpred240_Java7227 = new BitSet(new long[]{0x0000000000000000L,0x000000000000
    public static final BitSet FOLLOW_LBRACKET_in_synpred240_Java7237 = new BitSet(new long[]{0x0000000000000000L,0x000000000200000
    public static final BitSet FOLLOW_RBRACKET_in_synpred240_Java7239 = new BitSet(new long[]{0x0000000000000000L,0x000000000000000
    public static final BitSet FOLLOW_LBRACKET_in_synpred240_Java7250 = new BitSet(new long[]{0x0000000000000000L,0x000000000200000
    public static final BitSet FOLLOW_RBRACKET_in_synpred240_Java7252 = new BitSet(new long[]{0x0000000000000000L,0x000000000000000
    public static final BitSet FOLLOW_arrayInitializer_in_synpred240_Java7273 = new BitSet(new long[]{0x0000000000000002L});
    public static final BitSet FOLLOW_LBRACKET_in_synpred241_Java7322 = new BitSet(new long[]{0x2840C80300614200L,0x008A91B0200A187
    public static final BitSet FOLLOW_expression_in_synpred241_Java7324 = new BitSet(new long[]{0x0000000000000000L,0x000000002000
    public static final BitSet FOLLOW_RBRACKET_in_synpred241_Java7338 = new BitSet(new long[]{0x0000000000000002L});
}
```

FIGURE 6.5: The CUP output of parsing `JavaParser.java` with the `IfPredicateChecker`-woven JavaParser.

As the above result is difficult to be verified manually, we weave the previous `IfPredicateChecker` aspect into the ANTLR generated Java parser, and then parse the program `JavaParser.java`. The test result is exactly the same as the above number we got. The Eclipse output is shown in Figure 6.6.

We have not found a reliable tool with the function to count "if-only" statements in Java. Thus we try to verify the result "manually". In detail, we replace certain strings with themselves to get the count of them in the program. We first count the "if" keyword as the number of all "if" statement. To avoid counting in the "if" strings in the plain text in the program, we respectively count two strings, "if(" and "if (". Similarly, we count the "else" token by counting string "else " in `JavaParser.java`. According to the results, there are 1868 "if(", 1631 "if (", and 324 "else ". With these middle results, we can calculate that the number of the "if-only" statements in `JavaParser.java` is $1868 + 1631 - 324 = 3175$. Although we still cannot guarantee that 3175 is the correct answer, the fact that all three results of the two tests and our "manual" counting are



```
<terminated> AspectJava_CodeMetrics [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (28 Sep 2014 16:23:46)
there are 3175 if-only if-statements.
there are 2650 equality checks involved in if-predicates.
```

```
    public static final BitSet FOLLOW_COLON_in_syn
    public static final BitSet FOLLOW_statement_in
    public static final BitSet FOLLOW_catches_in_s
    public static final BitSet FOLLOW_FINALLY_in_s
    public static final BitSet FOLLOW_block_in_syn
    public static final BitSet FOLLOW_catches_in_s
    public static final BitSet FOLLOW_FOR_in_synpr
    public static final BitSet FOLLOW_LPAREN_in_sy
    public static final BitSet FOLLOW_variableModi
    public static final BitSet FOLLOW_type_in_synp
    public static final BitSet FOLLOW_IDENTIFIER_i
    public static final BitSet FOLLOW_COLON_in_syn
    public static final BitSet FOLLOW_expression_i
    public static final BitSet FOLLOW_RPAREN_in_sy
    public static final BitSet FOLLOW_statement_in
    public static final BitSet FOLLOW_localVariabl
    public static final BitSet FOLLOW_castExpressi
    public static final BitSet FOLLOW_LPAREN_in_sy
    public static final BitSet FOLLOW_primitiveTyp
    public static final BitSet FOLLOW_RPAREN_in_sy
    public static final BitSet FOLLOW_unaryExpress
    public static final BitSet FOLLOW_DOT_in_synpr
    public static final BitSet FOLLOW_IDENTIFIER_i
    public static final BitSet FOLLOW_identifierSu
    public static final BitSet FOLLOW_DOT_in_synpr
    public static final BitSet FOLLOW_IDENTIFIER_i
    public static final BitSet FOLLOW_identifierSu
    public static final BitSet FOLLOW_LBRACKET_in_
    public static final BitSet FOLLOW_expression_i
    public static final BitSet FOLLOW_RBRACKET_in_
    public static final BitSet FOLLOW_NEW_in_synpr
    public static final BitSet FOLLOW_nonWildcardT
    public static final BitSet FOLLOW_classOrInter
    public static final BitSet FOLLOW_classCreator
    public static final BitSet FOLLOW_NEW_in_synpr
    public static final BitSet FOLLOW_classOrInter
    public static final BitSet FOLLOW_classCreator
    public static final BitSet FOLLOW_NEW_in_synpr
    public static final BitSet FOLLOW_createdName_
    public static final BitSet FOLLOW_LBRACKET_in_
    public static final BitSet FOLLOW_RBRACKET_in_
    public static final BitSet FOLLOW_LBRACKET_in_
    public static final BitSet FOLLOW_RBRACKET_in_
    public static final BitSet FOLLOW_arrayInitial
    public static final BitSet FOLLOW_LBRACKET_in_
    public static final BitSet FOLLOW_expression_i
    public static final BitSet FOLLOW_RBRACKET_in_
}
```

```
package antlr.Java;
public aspect CodeMetrics {
    private pointcut summary() : execution(public static void mair
     int c1=0;
     int c2=0;
    private pointcut ifStatement() : within(JavaParser)&&((call(pu
    private pointcut ifElseStatement() : within(JavaParser)&&((wit
    after(): ifStatement()
    {
    c1++;
    }
    after(): ifElseStatement()
    {
    c2++;
    }
    after(): summary()
    {
    System.out.println("there are "+(c1-c2)+" if-only if-statement
    }
     int c3=0;
    private pointcut ifPredicateEqualityCheck() : within(JavaParse
    after(): ifPredicateEqualityCheck()
    {
    if (((JavaParser)thisJoinPoint.getThis()).input.LT(1).getText(
    c3++;
    }
    }
    after(): summary()
    {
    System.out.println("there are "+c3+" equality checks involved
    }
}
```

FIGURE 6.6: The ANTLR output of parsing `JavaParser.java` with the `IfPredicateChecker`-woven JavaParser.

the same makes us more confident about the correctness of this counting functionality that derived from our AspectJava aspect weaving.

6.4 Exploring The Reusability of DSAL Aspects in Our Approach

The *AspectRERS* and *AspectJava* implemented in the CUP experiments are in fact the same DSAL. This gives us a chance to explore the possibility of reusing the same DSAL aspects to different models conforming to the same domain meta-model. In our experiments, we use two "model specific" DSAL aspects and two "general" DSAL aspects, and apply each of them respectively to the CUP parser generated from the RERS model and the Java model. By "model specific aspects", we refer to the aspects that aiming at model specific elements, in particular, not shared by the other models tested by the same aspects. The "general" aspects only include content shared by all tested models.

Theoretically, if a DSAL aspect is applied to both the target models and unrelated models, we expect its application should work properly with the related join points in the target models. In the unrelated models, since the join points expected by the DSAL aspect do not exist, the application of the aspects should either fail, or capture no join point, and thus have no effect on the unrelated models. For example, the CUP grammar of RERS shares nothing with the CUP grammar of Java but few terminals. This makes it easy to define "model specific" aspects. In Section 6.2.5, we test the DSAL aspect counting the "GLOBALLY" expressions in an RERS program. Obviously, this aspect only involves one production rule temporal : 3, which is not defined in the CUP grammar of Java. We then reuse it and apply it to the CUP grammar of Java. The compilation and weaving works normally, and the test results in parsing the `testJavaProg.java` and the `JavaParser.java` are both zero counts as expected. On the other hand, to apply a Java aspect to the CUP grammar of RERS, we reuse the "if only" statement counter aspect in Section 6.3. The compilation and weaving works well. The test of parsing the RERS program shown in Section 5.5 still returns zero count result as expected. In these two experiments, we apply irrelevant DSAL aspects to the models, i.e. the Java elements related aspect applied to the RERS grammar and the RERS elements related aspect applied to the Java grammar, without any compilation or weaving error. These two experiments show the potential of safely reusing the DSAL aspects to any model (of the target domain).

The above experiments are just preliminary exploration that the aspect reusing attempts will not bring side effect to the unrelated models. The following test applies the aspects with only the elements shared by all models. In detail, we prepare two aspects for the RERS and Java models. The first aspect is to count the "shift" actions in parsing a given program. The second aspect is to count the tokens in a given program. Since for each "shift" action by the CUP generated parser, there must be a token that has been parsed. Theoretically, the results of these two counting aspect should always be identical in the application to each model. This also helps to verify the aspect application results. The "shift" action counter aspect is shown in Listing 6.11.

```
package cup.Java;

public aspect CodeMetrics {
    int shift_action_count=0;
    pointcut shift_action_counter=%AfterShift;
    before : shift_action_counter() {
        shift_action_count++;
    }
    pointcut summary=%BeforeMainExit;
    after : summary() {
        System.out.println("shift_action_count: "+
   shift_action_count);
    }
 }
```

Listing 6.11: The DSAL aspect of counting the "shift" actions in the parsing process

We apply this aspect to the RERS model and parse the sample RERS program shown in Listing 5.25. The counting works fine and the result shows there are 110 "shift" actions occurred in parsing the program. The token counter aspect is as shown in Listing 6.12.

```
package cup.Java;

public aspect CodeMetrics {
    int token_count=0;
    pointcut token_counter=after(#);
    before : token_counter() {
        token_count++;
    }

    pointcut summary=%BeforeMainExit;
    after : summary() {
        System.out.println("token_count: "+token_count);
    }
 }
```

Listing 6.12: The DSAL aspect of counting tokens in the parsing process

As expected, the result of token counting in the sample RERS program is also 110, which agrees with the "shift" action counting result. We then apply the "shift" action counter aspect to the Java model and parse the `testJavaProg.java`. The counting works well as well, and shows that there are 480 "shift" actions in parsing the `testJavaProg.java`. Again, we use the token counter aspect to verify the result. The result is also 480.

In the above experiments, although the models contain different elements, they are defined in the same DSL, i.e. they share the same domain meta-model, which gives the common ground of reusing the aspects written in the DSAL generated in our approach.

Chapter 7

Evaluation

So far we have demonstrated our *DSCG extension approach* with three different case studies based on real-world DSCG scenarios. From these case studies, we can see that this approach allows dynamic generation of DSALs, according to the target DSCG and the modification requirements. With the DSALs, domain experts can write domain specific aspect to describe the changes they want, in which they can use both the domain specific elements and the output language statements. These aspects can then be automatically woven into the code generated from the base models to complete the modification process. In this chapter, we evaluate our approach through its comparison with three alternative approaches, including the *direct manual modification* approach, the *model round-trip engineering* approach, and the *pure model based* approach. In our evaluation, we focus on four criteria, applicability, reliability, productivity, and reusability.

7.1 Applicability

To evaluate a code modification approach, a criteria that naturally springs to mind may be its applicability, i.e. its usefulness in accommodating different changes. In particular, we are interested in two questions, *what kind of changes can be introduced to the generated code?* and *how are they described?*.

7.1.1 The Direct Manual Modification Approach

The *Direct Manual Modification* (DMM) approach might be the most expressive approach, with regard to the possible changes it can introduce to the generated code. Since the changes can be made directly in certain programming languages, i.e. the output languages of the domain specific code generators, this approach theoretically allows any change to be introduced to the base code, as long as the change does not break its compilation or interpretation. In other words, the only constraint of introducing changes in this approach is the syntactical and semantical rules of the output languages.

On the other hand, this approach often lacks high-level formal descriptions of the code changes it accommodates, in respect of their purposes and impacts. Such succinct descriptions are particularly important given the context of DSCG, where the high-level models are considered as primary products and the generated code is taken as the low-level representation of the models. Having to understand changes through their low-level descriptions is a hindrance to the MDE development. Since the manual changes are described directly in certain programming languages, it can be rather difficult for domain experts, or even programmers other than the code changers themselves, to understand the changes or to evaluate their influence. Although the changes are sometimes supplemented with extra descriptions for better understanding, like comments or documents, such additional information normally does not coheres well with the changes themselves. For example, the adjust in code changes cannot automatically update their corresponding documents, or vice verse.

7.1.2 The Model Round-Trip Engineering Approach

In the *Model Round-Trip Engineering* (MRTE) approach, an MRTE system is built to reverse the DSCG process. In detail, an MRTE system mainly consists of a detector to detect any inconsistency between the models and the code, and a model restorer, that can generate models from the modified code. This reverse transformation and the target DSCG transformation constitute the complete transformation "round-trip". In this round-trip, the changes can be made either at the model level using the target *Domain Specific Languages* (DSLs), or at the code level using the output languages of the domain specific code generators. However, any change that can be made in this approach must be reflexible in the base models. No matter what technique enables the round-trip, if the change made at the code level cannot be reflected at the model level, the round-trip will break. Even if the inconsistency between model and code can somehow be detected, the base models would remain unchanged. As a result, the changes that can be introduced to the base code in this approach are actually restricted by the target domain DSLs. In this approach, since the changes are introduced through an incremental round-trip process, there are always two descriptions for a single modification, a high-level description at the model level, and a low-level description at the code level.

7.1.3 The Pure Model Based Approach

In this approach, the changes need to be made directly to the base models, and then get reflected in the code through a regeneration from the modified models. The code changes that can be made in this approach are restricted by both the target DSLs and the internal implementation of the DSCG code generators, e.g. the string templates used in the ANTLR parser generator. In other words, this approach does not allow any change that cannot be expressed by the target DSL, or cannot be generated by the target code generator. Although we can extend the DSLs and/or the code generators to make them more expressive, the allowed changes are still restricted by the extended DSLs and/or the extended code generators. Despite the restrictions from the DSLs and the code generators, the pure model based approach guarantees that all changes that can be introduced to the base code have formal description at higher level of abstraction, i.e. in DSLs.

7.1.4 The DSCG Extension Approach

In our DSCG extension approach, the changes are described in DSALs and automatically woven into the base code by the corresponding aspect weavers. The changes that can be introduced in this approach are restricted by the DSALs and their underlying aspect weavers. This seems to be the same level of restrictions that applied in the pure model based approach. But in fact, our approach is much more flexible in describing the changes, and thus is capable of accommodating more potential changes.

First, the DSALs generated in this approach allow the expected modifications to be described with the elements from both high-level, i.e. the model level, and low-level, i.e. the code level, at the same time. For example, the DSAL generated in our "AUTOFILTER" case study allows the variables of the base models to be directly used in the statements of the output C programming language (as shown in Listing 4.22). It is worth noting that this is different from the case in the MRTE approach, where a modification can be described either completely at the model level or completely at the code level. Being capable of using the elements from both the model level and the code level at the same time makes it possible to describe the expected modifications in a more natural way.

Second, as the DSALs and their aspect weavers used in our approach are dynamically generated according to the change requirements, our approach has the potential of customizing them before they are used to describe and accommodate the expected modifications. Although the "acceptable" modifications in this approach are restricted by the DSALs and their weavers, such restrictions can potentially be removed on demand. This is different from the restrictions caused by the fixed DSLs used in the pure model based approach. Take our "AspectRERS" experiment in the ANTLR case study for example, if there is a need to monitor the complicated LTL formula based on complex "cflow_pattern" patterns, our approach allows adjust of the DSAL generation to include this new pattern, so that the newly generated "AspectRERS" will support the "cflowPattern" pointcut. Thus the same change can now be introduced with the new "AspectRERS".

In short, the modifications that can be accommodated in this approach are restricted by the dynamically generated DSALs and their weavers. For each modification in this approach, there is a formal description conforming to the generated DSAL.

7.2 Reliability

Our major research objective is that our approach should maintain the benefits of DSCG, which basically include the reliability and productivity. The reliability of DSCG mainly refers to the correctness of the generated code, which largely comes from the "correct-by-construction" guarantee provided by the domain specific code generator. In our evaluation, we look into the general steps involved in the code modification process following each approach, and analyze their effect on the correctness of the modified code. The correctness here does not only refer to the syntactical and the semantical correctness that are generally expected in the generated code, but also include the conformance with the original domain meta-models.

7.2.1 The Direct Manual Modification Approach

In DSCG, there is often great abstraction gap between the DSLs and the output programming languages. A fragile synchronization between the models and the code is maintained by the DSCG process, to guarantee the correctness of the code. A direct manual modification would break such synchronization and leave the modified code with no correctness guarantee, not to mention the fact that a manual modification itself is often tedious and error-prone. For example, the generated code may contain a lot of code-generator-specific details that cannot be easily understood, such as the *magic numbers* we see in the variable names in the code generated by AUTOFILTER. In short, there is no guarantee of the correctness of the code that updated through a direct manual modification.

7.2.2 The Model Round-Trip Engineering Approach

The changes can be introduced in two ways in the MRTE approach. The first way is to modify the base model directly, which would be automatically detected by the round-trip engineering framework and then re-trigger the DSCG process to keep the code consistent, or *synchronized*, with the modified model. We can take this modification process as "reliable", provided the DSCG code generators remain unchanged. If the code generators have been modified, e.g. to generate model traceability links during the DSCG process, the correctness of the modified code would depend on the effect of the code generator customization over the DSCG process.

The second way is to modify the generated code, which would also be detected by the round-trip engineering framework and then compute a corresponding model by the model restorer. Unfortunately, the development of the model restorers or the detectors of mode-code inconsistency is usually on ad hoc basis. There lacks the general or standardized criteria to detect or maintain the inconsistency between the domain specific models and the generated code. In brief, the reliability of this approach depends on the MRTE systems built on ad hoc basis.

7.2.3 The Pure Model Based Approach

In the pure model based approach, if the expected modifications do not require an update of the DSLs or the code generators, the “correct-by-construction” guarantee will be maintained, which ensures the correctness of the modified code. However, if an update of the DSLs or the code generators is required, the correctness of the modified code relies on the effect of the code generator customization over the DSCG process.

7.2.4 The DSCG Extension Approach

Through the generation of DSALs and their aspect weavers, the DSCG extension approach essentially extends the original DSLs at both the domain meta-model level (by the DSALs) and the code level (by the DSAL weavers). The correctness of the code modified in this approach relies on the correctness of the DSAL aspects, and that of the generated DSALs and their aspect weavers. In our evaluation, we simply assume the DSAL aspects are always written correctly, and thus only focus on the correctness of the generated DSALs and their weavers. In the previous case studies, we can see that both the DSALs and their weavers are generated based on the domain meta-models from mainly five aspects.

First, the target domain meta-models, i.e. the original DSLs, are tailored according to the modification requirements into the effective meta-models, which can be regarded as a transient extension of the DSLs within the modification systems built in our approach. Second, the effective meta-models are composed with the tracing strategies to create the traceable domain meta-models. This process can largely be automated, except that the selection of the tracing strategies requires insight of the code generators, so that the generation of the model tracing sentinels will not break the conformance of the generated code with the original domain meta-models. Third, the traceable domain meta-models serve as the formal guides to customize the code generators to enable the model traceability in the generated code. Although this is a manual process and its reliability is difficult to be evaluated, we can ensure that the code generated by the customized code generators still conforms to the original DSLs, provided the compliance with the traceable domain meta-models could be achieved in the customization process.

Fourth, the traceable domain meta-models also serve as the guides to implement the DSAL aspect weavers. From the perspective of the entire MDE tool chain involved in this approach, the traceable domain meta-models can be regarded as a protocol between the code generators and the corresponding aspect weavers. The code generators are responsible to enable the model traceability in the generate code in compliance with the protocol, and the aspect weavers interpret the generated code based on the compliance of the same protocol, to weave the DSAL aspect correctly into the code to complete the modification. Last, the compliance of the traceable domain meta-model can only guarantee the construction and interpretation of the model tracing infrastructure. The correctness of the DSAL aspect weaving also depends on the join point model of the generated DSAL. In our approach, the composition of the effective meta-model and the DSAL specification templates serves as the domain specific join point models.

In short, the reliability of our DSCG extension approach depends on the correct derivation of the target domain effective meta-models and the correct generation of the traceable domain meta-models, as they serve as the formal guide in the construction of the DSAL based modification systems. Provided their correctness can be guaranteed, we can ensure the correctness of the code modified by the DSAL aspect weaving process.

7.3 Productivity

To modify the code generated by DSCG is a very special scenario in the MDE context. A basic principle in MDE is that the models are the core products instead of the code. The code is supposed to be lower-level representations of the corresponding models. There is no need to maintain an injective function between the models and the code. Therefore, a change aiming at code level modification should be maintained independently from the primary product in MDE, i.e. the domain specific models. In other words, the code should be able to remain independent and to revert an applied change under the modification system if needed. From this perspective, the productivity of a DSCG code modification system is not only reflected in the effort required in introducing the changes, but also reflected in the effort required in reverting them. In our evaluation, we will take both into consideration. Besides, as the construction of the modification system itself is the prerequisite of the modification process, we will also take the effort it entails into consideration.

Strictly speaking, the productivity evaluation should be based on quantitative analysis over the general steps involved in the modification process following a certain approach. However, it is quite difficult to get accurate data in our evaluation. With regard to the different DSCG contexts, e.g. different DSLs and output languages, it is difficult to define a general benchmark system to assess the complexity or workload of the modification requirements. The implementors with different knowledge bases and skill sets may also lead to quite different cost for the same task. Therefore, our evaluation is based on rather loose estimation of the possible effort involved in the modification process in each approach.

7.3.1 The Direct Manual Modification Approach

For simply changes, the DMM approach seems very cost-effective, as it does not require any extra cost for building a modification system. Even for some experiments in our case studies, this approach could be more productive than our DSCG extension approach in accommodating the expected changes. However, there are three restrictions in this approach. First, manual effort may not be trivial even for simple changes, e.g. when the code blocks that need to be changed are scattered throughout the code. Adding code for tracking the update of the state vector in our AUTOFILTER experiment is exactly an example of this situation. This modification requires manual changes in the generated code at every poin that the state vector value may be updated, to insert duplicate code for monitoring purpose. Second, no matter how tiny the manual change is, it would break the synchronization between the models and the code. Even if we do not consider the maintenance of such synchronization, with more and more manual changes accumulated in the generated code, the code would become more and more "deviated" from the base models, thus difficult to understand, not to mention its maintenance. Third, the effort of introducing multiple manual changes is accumulated in a linear way, even if they are very similar to each other. Besides, this approach does not support reverting the modifications, unless with the help of external tools for version control, such as *git* [142],

7.3.2 The Model Round-Trip Engineering Approach

In this approach, considerable effort is used to develop an MRTE system, so that the code modifications can be propagated back to the base models. However, even with such a system properly built, the effort in introducing a code level change would not be evidently reduced. On the contrary, extra processes like the detection of the model-code inconsistency and model restoration have to be finished before the modification is done. These processes are introduced merely to maintain the synchronization between the base models and the code. In short, the code level modification made in this approach generally takes more time and effort than using the DMM approach. Similar to the DMM approach, the MRTE approach does not support reverting the applied modifications, unless with the help of external version control tools.

7.3.3 The Pure Model Based Approach

In the pure model based approach, if the changes can be expressed by the original DSLs, they can be introduced to the base code by modifying the models and automatically regenerating the code through DSCG. With the description at more abstract level and automated code modification, the cost of introducing a single modification in this approach is generally lower than the cost of the direct code modification, i.e. both the DMM approach and the MRTE approach. However, if the expected modifications require extension of the DSLs and/or the customization of the domain specific code generators, the cost would be again difficult to assess, due to the dependency of the implementors' expertise in the target domain meta-models and the code generators. Worse still, the extension of the DSLs and the code generators in this approach is normally introduced on ad hoc basis. It can be very simple, like adding a new property to an existing class in the domain meta-model, or rather complex, such as a thorough refactoring of the domain meta-model. Therefore, the effort and cost involved in the extension in this approach may vary greatly. Once the extension of the DSLs and the code generation is finished, the cost of a single modification is supposed to be generally lower than that in the above two approaches.

7.3.4 The DSCG Extension Approach

In the DSCG extension approach, the generated DSALs and their aspect weavers are the actual tools to introduce modifications. As we elaborated in Chapter 3, their generation is based on the model traceability in the generated code, to trace the domain elements included in the effective meta-models. To enable this model traceability, the code generators are often required to be modified. This process can be decomposed into four main tasks. The first task is the generation of the effective meta-models, including the modification requirement analysis. The second task is the modification of the code generators. The third task is the study and application of the underlying implementation technologies, e.g. to build the DSAL aspect weavers. The last task is to write different DSAL aspects representing the expected changes, apply them to the generated code, and test the modified code accordingly.

Here we show a table containing the time log of our DSCG extension experiments. Although these are not experimental data, they can serve as a guide for estimating the effort involved.

It is worth noting that we started as first-time user at the beginning of each case studies. The tasks listed above may take much less amount of time for the developers experienced in the specific area. For example, we spent four weeks on the implementation in the CUP case study. Over 80% of the time was spent on learning the underlying techniques.

	AutoFilter	ANTLR	CUP
Effective Meta-Model Generation	10 days	6 days	8 days
Code Generator Modification	4 days on modification & 6 days on bug fixing	6 days on modification & 5 days on bug fixing	4 days on modification & 3 days on bug fixing
Implementation Techniques	SDF & Stratego/XT (3 weeks)	ANTLR & LL(*) & StringTemplate (4 weeks)	CUP & LALR (4 weeks)
Aspect Tests	5 days	3 days	6 days

Table 7.1: The time log of our experiments

It is very likely that the implementors experienced in this area can reduce the time for this task significantly.

In brief, there is considerable effort involved in the DSCG extension process, including the generation of DSALs and their weavers, and the modification of the code generators. However, with regard to the possibility of requiring accommodation of further modifications that can be expressed by the generated DSALs, the cost of introducing a single modification can be greatly reduced by writing with the generated DSALs and weaving the aspects automatically by the corresponding aspect weavers. In a sentence, the considerable modification system construction effort may pay back well in the long run, provided there will be similar changes to accommodate.

7.4 Reusability

"Reusability is a measure of the ease with which those previous concepts and objects can be used in the new situation." [143] There are three main artifacts involved in the modification of the generated code, the modification systems, the modifications accommodated with them, and the base models/code to be modified. Accordingly, we can evaluate the reusability at three different levels. First, we can evaluate the reusability of a single modification. Some modifications may aim at the model level changes, e.g. change the value of a specific variable in the models. This kind of changes are naturally not supposed to be reused for other models. Some other modifications may aim at the meta-model level changes, e.g. change the value of all variables of a specific domain class in the models. Ideally, this kind of modifications should be able to be reused to the code generated from other models using the same code generator. In our following discussion about modification reuse, we always refer to this kind of modifications. Second, we can evaluate the reusability of the modification systems, i.e. more modifications can be introduced to the code generated from more models through the same modification system. As such systems normally involve considerable effort to build, it is very important that their "once-for-all" cost can be "shared" by more and more further modifications. Last, we can evaluate the reusability of the modified models/code. Strictly speaking, it should be the reusability of the existing MDE tool chain, as both of the modified models and code should still be able to reuse the existing MDE tool chain. More precisely, if the code got updated during the modification process, it should remain "synchronized" to the base models. If the models got updated during the modification process, they should still conform to the original DSLs after the modification. This level of reusability is even more important than the previous two levels, as it is the basis of maintaining the benefits of DSCG after the modifications.

7.4.1 The Direct Manual Modification Approach

The modifications introduced in this approach basically exist as code patches. It is usually difficult to apply the same patch, even if they aimed at the domain meta-model level changes, to the code generated from different models. As there is not a modification system involved, this approach does not have reusability at the modification system level. This can explain the linear growth in the time cost of introducing multiple changes in this approach. Worse still, the modified code is no longer synchronized with the base models. In brief, the reusability of the DMM approach is very poor.

7.4.2 The Model Round-Trip Engineering Approach

Similar to the case of the DMM approach, the modifications in the MRTE approach also exist as patches. The difference is that for each modification in the MRTE approach, there are two patches, a model patch and a code patch. Theoretically, their modification effect to the base models/code should be identical. Similarly, these modifications cannot be reused on different models. As for the modification system reuse, as long as the expected modification can be reflected by the DSLs, they can be accommodated by reusing the MRTE systems. After the modification, the modified code remains "synchronized" with the updated base models, and the updated base models still conform to the original domain meta-model, i.e. the DSLs. In short, this approach supports the reuse of the modification system, and maintains the existing MDE tool chain.

7.4.3 The Pure Model Based Approach

The modifications in this approach exist as the patches of models, which are generally written in terms of model specific descriptions. As a result, they cannot be reused to other models in the same DSLs, even if they aim at only the domain meta-model level changes. For example, if the modification requirement is to change the value of a specific property p of *ALL* instances of a target domain class C, and there are two instances of class C in model M_1 $(\mathsf{c}_1, \mathsf{c}_2)$, and three instances of C in model M_2 $(\mathsf{c}_3, \mathsf{c}_4, \mathsf{c}_5)$. A model patch for M_1 would look like $\mathsf{c}_1.\mathsf{p} = \mathsf{new\_value1}; \mathsf{c}_2.\mathsf{p} = \mathsf{new\_value2};$. Obviously, it cannot be reused by M_2. As for the modification system reuse, as long as a modification can be expressed by the current DSL, they can reuse the DSL and the corresponding code generator. Finally, with regard to the reuse of the existing tool chain, it basically depends on the compatibility of the DSLs. If the extended DSL is not compatible with the original DSL, neither the modified models nor the regenerated code can reuse the existing MDE tool chain that still in conformance of the original DSL.

7.4.4 The DSCG Extension Approach

From our case studies, we can observe the reusability in our approach at all three levels mentioned above.

Reuse of The Aspects In Section 6.4, we illustrate that DSAL aspects can be safely reused to different domain models, or more precisely, the code generated from them by the customized code generators, as long as they conform to the same domain meta-model. By "safely", we refer to the effect of applying the aspects. When the generated code contains the join points described by the PCDs in the aspects, the woven code should contain the expected modifications described in the aspects. When the generated code does not contain any join point involved in the aspects, the aspect weaving process ought to have no effect on the generated code, either by directly failing or resulting in no change to the generated code. In brief, our DSALs allow reusing DSAL aspects among different models conforming to the same domain meta-model. Take our experiments with "AspectRERS" for example, a pointcut $\mathsf{after}(:4)$ can be reused to pinpoint the parsing exit of the fourth alternative of any production rule, to any model written in the RERS language. It is worth noting that our DSALs simply make this level of reusability possible, not guarantee it. In the same "AspectRERS" example, if we write the aspect with a number of model-specific pointcuts, such as $\mathsf{after}(@\mathsf{temporal}:4)$ and $\mathsf{after}(@\mathsf{output}:4)$, the aspect still cannot be reused by other models.

Reuse of The DSALs As modification tools, the DSALs and the corresponding weavers are generated to be reused for accommodating any modification that can be expressed by the DSALs. We can observe many reuses of the generated DSALs in our experiments. For example, we can reuse the *AspectJava* to write and apply both the "If Predicate Checker" aspect and the "LOC Counter" aspect.

Reuse of The MDE Tool Chain Throughout the modification process in the DSCG extension approach, the base models are regarded as the only core product and remain unchanged. The code generated from them is taken as a corresponding representation of the models at the lower-level. To enable the model traceability in the code, this approach requires modification of the code generators, so that the traceability links, i.e. sentinels, can be properly inserted into the code by regeneration using the modified generators. This also maintains the "synchronization" between the models and the code. The unchanged base models and the regenerated code constitute the basis of further modifications through the DSAL aspects. In the further modifications, for each DSAL aspect, there will be a version of modified code, while the base code is always the regenerated code. In other words, the modified code is in fact synchronized with the DSAL aspect, instead of the regenerated code. This relationship is shown in the lower half of Figure 1.7. In brief, our DSCG extension approach maintain a synchronization parameterized by DSAL aspects between the base models and the modified code. From this perspective, the unchanged models, the regenerated code, and the modified code can be selectively used for reusing the existing MDE tool chain in different scenarios.

7.4.5 Summary

The results of the above comparison can be summarized into a few points as shown in Table 7.4.5.

From the above comparison table, we can see that every modification approach, except for the DMM approach, builds a reusable modification system on top of the target DSCG process, although the initial purpose might be to accommodate only a single modification of the code generated from a specific model. Without such a modification system, e.g. as the case of the DMM approach, although a direct manual modification of the generated code seems to be made in a prompt way, the cost is much greater than the gain. First, it breaks the synchronization between the models and the code, which entails the expiration of the "correct-by-construction" guarantee. Second, such modifications on ad hoc basis lose the systematicness of the modification process. This makes the process difficult to be standardized and automated, and finally damages the productivity of the complete MDE tool chain. This is the reason why our DSCG extension approach still constructs a complex DSCG extension system, despite its considerable cost.

It is not difficult to understand that these modification systems are essentially extensions of the DSLs in the target DSCG. As the expected modifications cannot be expressed by the DSLs in the first place, the DSLs have to be somehow extended to be able to describe the modifications. The PMB approach adopts a direct way to update the DSLs, which sometimes would lead to the compatibility problems. The MRTE approach implements the DSL extension indirectly using a MRTE system based on reverse engineering. Both approaches involve a code update and a model update in a single modification process. Our DSCG extension approach extends the DSLs indirectly and enhance them with the aspect-oriented perspective. Compared with the other two approaches, this extension

	The DMM Approach	The MRTE Approach	The PMB Approach	The DSCG Extension Approach
Applicability	any compilable change	restricted by DSL	restricted by DSL	restricted by DSAL
Reliability	no guarantee	depend on the MRTE systems built on ad hoc basis	depend on the extension of DSLs and the code generators	depend on: 1. the modification of the code generator 2. the generation of the DSALs and weavers (both are based on the domain meta-models)
Productivity	low cost for simple modification, linear cost growth for multiple modifications	considerable cost in building MRTE system, generally higher cost in single modification than the DMM approach	uncertain cost in the extension of the DSLs and code generators, generally lower cost in single modification than both the DMM and MRTE approach	considerable cost in building DSCG extension, similar cost in single modification as the PMB approach
Reusability	poor	modification not reusable	modification not reusable, potentially break the existing MDE tool chain	the modification, modification system, and the existing MDE tool chain are all reusable

Table 7.2: Comparison table of alternative approaches for code modification in the DSCG context.

seems more complicated, as it proposes a three-step modification process, which includes a DSAL aspect creation, a regeneration of the base code using the modified code generator, and a composition of the regenerated code and the DSAL aspect. However, the extra complexity in our approach brings about additional advantages.

First, the DSAL brings about extra expressiveness in the description of the expected modifications, in terms of the aspect-oriented perspective and the finer-grained descriptors, i.e. the statements in the lower-level output languages. Second, the base code regeneration makes the base code more extensible by the insertion of the traceability links, i.e. the sentinels, while it maintains the conformance of the base code to the original DSLs. This regeneration step can be regarded as a trade-off between the reusability of the target DSCG and its extensibility, and the regenerated code is the prerequisite of the DSAL aspect weaving process. It is worth noting that the customization of the code generators in our approach is essentially different from that in the PMB approach. The customization of the code generators in the PMB approach aims at supporting a specific kind of modifications, while the customization of the code generators in the DSCG extension approach aims at the extensibility of the generated code. With the sentinels inserted in the generated code, different DSALs can be created for different kinds of modifications.

It is worth noting that the function of these modification systems is not just to make it possible to introduce the previously inexpressible changes to the generated code. More importantly, they serve as an adapter to the existing MDE tool chain, so that the modifications accommodated through them can be accepted and reused by the existing MDE tool chains. The protocol of this adaptation is exactly the domain meta-models of the target DSCG. In other words, both the modifications and the base models/code have to conform to the same domain meta-model. This is the root reason why all three approaches, including our approach, focus on the domain meta-models. In our DSCG approach, the "protocol" meta-models are referred to as the *domain effective meta-models*.

On the other hand, there are some limitations in our approach. One major limitation is insufficient automation. In the modification process following our approach, there are several steps may require human interactions. For instance, the composition of the effective meta-models and the DSAL templates often need manual effort to finish the DSAL generation process. Another limitation in our approach is the gap between the guide of the code generator customization, i.e. the traceable domain meta-models, and the actual customization of the code generators. We will discuss about a recent work in Section 8.2.1 that seems promising to bridge this gap.

In summary, we can see that our meta-model based DSCG extension approach guarantees the reliability and reasonable productivity of the target DSCG process and the DSAL based modification process. Despite a few restrictions, this approach basically achieves our primary research objective in this thesis, namely to maintain the benefits of the DSCG when introducing the code modifications.

7.5 Other Related Works

Apart from the three alternative approaches we have compared so far, there are a few more related works that worth discussion.

7.5.1 Composition of Model Transformations

As Tisi et al. [144] argued, model transformation technologies have reached the level of maturity that model transformations can now be treated as objects. As a result, model transformations become first-class elements of the model based software systems, just like models. To address increasingly complex application that require direct manipulations of model transformation as objects, the concept of *Higher-Order Transformation* (HOT) is defined to refer to a model transformation whose input and/or output models are model transformation objects. As discussed in Seciont 2.4.3, both the DSCG process and the aspect weaving process, i.e. the composition of code and aspect, can be considered as model transformations. From this perspective, an extension of the DSCG process with aspect oriented approaches can be considered as a HOT that composes these two model transformation objects. Kurtev et al. [145] demonstrated the feasibility of HOT implementations with similar transformations. Their experiments are based on the *ATLAS Model Management Architecture* (AMMA) framework. It contains a mature implementation of the *ATL Transformation Language* (ATL), which follows the OMG QVT standard [146]. Within AMMA, a DSCG can be defined as an ATL transformation model, with its domain meta-models specified with the Gruber's definition [147]. Unfortunately, this framework does not support aspect-weaving based transformation. Thus it cannot help us to build the DSCG extension system.

7.5.2 Meta-Programming in DSAL

Although the DSALs in our approach are dynamically generated with regard to the given modification requirements, it is difficult to foresee all the potentially related modification requirements and take them into consideration during the generation of the DSALs. Once the DSALs have been generated, their syntax and semantics are fixed. If any further modification requirement is restricted by the current DSALs, the only way to accommodate the modification is to regenerate the DSALs. Regarding the high cost in the generation of the DSALs and their aspect weavers, it could be much easier if the DSAL aspect writers can tailor the language semantics in a modification specific manner. For example, when several pieces of advice need to be applied to the same join point, different orders of application may lead to different results or even conflicts. This is a typical issue in the *aspect interaction* [148], and there are several conflict resolution approaches to specify the order of advice execution [148, 149]. *Meta-Aspect Protocol* (MAP) is an extension built on top of the Groovy [150] *Meta-Object Protocol*, to realize the aspect languages implemented by the *Pluggable and OPen Aspect RunTime* (POPART) runtime. Dinkelaker et al. [151] implemented an extensible advice ordering mechanism with the MAP, which can be adapted at runtime to resolve the aspect interaction problems. Although such aspect language extension system has specific technical dependency (the POPART runtime), it shows a possible extension of our approach for better reusability of the generated DSALs. It is worth noting that MAJ is only one potential extension of the generated DSALs. There are a lot more possible extensions can be made to the DSALs to improve the reusability, e.g. the XML based *framed aspect* technique [152].

Chapter 8

Conclusion and Future Works

In the previous chapter, we evaluate our meta-model based DSCG extension approach through its comparison with three alternative approaches, in terms of its applicability, reliability, productivity and reusability. In the final chapter, we conclude all our work and list some further works that can be done to refine or extend our approach.

8.1 Conclusion

Model based software engineering approaches raise the abstraction level of software development from code to models. In particular, the *Domain Specific Code Generation* (DSCG) technique allows the domain experts to develop reliable software by directly building the domain specific models, and bridges the big gap between domain specific modelling level and code level, by automatically transforming domain specific models into code with a "correct-by-construction" guarantee. Compared with the traditional software development, it brings about many benefits in productivity and reliability. Unfortunately, such advantages are easily lost in the modification of the generated code. Traditionally, three approaches are commonly used to introduce changes to the generated code, i.e. the *Direct Manual Modification* (DMM) approach, the *Model Round-Trip Engineering* (MRTE) approach, and the *Pure Model Based* (PMB) approach. From these approaches, we identified three specific problems.

1. The changes that cannot be reflected in the models require adding unnecessary details into the modelling languages, which can lead to a breakdown of the abstraction hierarchy.

2. Enforcing a surjective or even bijective function between the models and the code deviates from the nature of the model based software engineering.

3. The existing approaches may lose track of the software when the introduced changes cannot be reflected in the models.

The primary contribution of this thesis is to propose a meta-model based DSCG extension approach to address the primary problem of losing the benefits of DSCG in the modification of the generated code, with regard to the above three specific problems. Given a target DSCG and certain modification requirements, our approach extends the DSCG by dynamically generating a *Domain Specific Aspect Language* (DSAL) and its aspect weaver, based on the analysis of the modification requirements. Thus the domain experts can describe their expected modifications as domain specific aspects in the generated DSAL, and use the generated aspect weaver to automatically weave the aspects into the code generated from certain input model, to complete the modification process. The entire DSCG extension system is constructed based on the meta-models of the target domain. In particular, an effective meta-model is generated to describe the domain classes and properties that are involved in the expected modifications. A traceable domain meta-model is generated from the effective meta-model to describe how customize the code generator to enable the model traceability in the generated code, which is the basis of the domain specific aspect weaving. As the modifications are in fact accommodated by such meta-model based extension system of the target DSCG, we call this approach as the "meta-model based DSCG extension approach", or "DSCG extension approach" for short.

In our approach, a modification that previous cannot be expressed by the original DSL can now be described in the dynamically generated DSAL at both model level and code level, and then woven into the generated code automatically. Throughout the modification process, the input model will remain unchanged. The modification is directly made to the generated code. The modified code does not need to be manipulated to maintain the synchronization with the input model. Instead, it is synchronized with a combination of the input model and the DSAL aspect applied to it, which makes more sense in respect of the principle of MDE. Code is the lower-level representation of the primary product, i.e. models. A change made to the representation should not affect the models themselves. This relationship is illustrated in the lower half of Figure 1.7. With our approach, the version controlling of the software developed with DSCG also becomes very simple. We only need to track the versions of the models and DSAL aspects. The code version can always be calculated through a combination of the model version and the aspect version. Moreover, the evaluation based on the comparison with the alternative code modification approaches shows that the DSALs and their weavers that systematically generated based on the domain meta-models can provide reasonable guarantee for the reliability and the productivity of the modification process. As a whole, the meta-model base DSCG extension approach that we propose successfully achieves our initial research objectives.

A secondary contribution of our work is to develop a generic approach to generate configurable parser enhancement tools for theoretically any programming language. The basic idea is to apply our DSCG extension approach to the language parser generation process. By generating DSALs of the language grammar domain, our approach allows configuring enhancement of the language parsers by writing and applying the corresponding grammar enhancement aspects. By applying such aspects to the parsers generated from the target language grammars, we generate the parsers with configured enhancements. For example, we build a DSAL called "AspectJava" to extend the ANTLR parser generation process. As shown in Section 5.6.4, we write an "IfPredicateChecker" aspect and an "LOCCounter" aspect in "AspectJava", and respectively weave them into the Java parser generated from the ANTLR grammer of Java, to generate two different statistics tools for Java programs, one for logging the "if-only" statements and involved "equality checks", the other for counting the *Line Of Code* (LOC), for any Java program. With similar aspects, we can conveniently configure a Java parser generated by ANTLR with many other enhancements, e.g. to measure code metrics like code coverage, etc.

8.2 Future Works

As mentioned in Section 7.4.4, there are still some limitations of our approach. In this section, we discuss some potential refinements or extensions that can be introduced to our work.

8.2.1 Guided Customization of Code Generators Using Symbol Table

The recent work by Nazari et al. [153] proposes an approach to manage customizations of template-based code generators at generation-time by reusing the *symbol table* [40] data structure. In particular, we are interested in its discussion about how to manage the template replacements dynamically at generation-time for a *guided customization*. The basic idea is to define some *hook points* in the code templates. Each hook point has a unique identity and can be bound with one or more values, which can be either a string or another template. Thus a template graph is formed through these hook points. The *symbol table* is then used as a map of the graph of the hook points and templates. Each table entry, i.e. a *symbol*, stores the name, the type, and all essential information of a model element, which can be a hook point or a template among others. Given a name and a type, the symbol table can find all associated information of the corresponding element.

In our approach, we use traceable domain meta-models to guide the customization of the DSCG code generators to insert sentinels into the corresponding locations in the generated code, so that the generated DSAL weavers can trace the domain models in the code. This process can be regarded as a “guided customization” of the DSCG code generator. Given a traceable domain meta-model, we can define a symbol table accordingly, with the join point classes as the root symbol elements, and the corresponding sentinel strings as the string values bound to the underlying hook points. During the base code regeneration, these symbols are evaluated in the template expansions and help to generate the corresponding sentinels in the regenerated code.

8.2.2 Generating Better Domain Specific Aspect Language

In our experiments with the ANTLR and CUP generators, we deliberately reuse AspectJ as the underlying aspect language of the generated DSALs. This decision helps to reduce the cost in the generation of the DSAL. But it restricts the DSALs with the limitations of AspectJ. In fact, the DSALs work like the general purpose aspect languages, in which the participants are domain specific elements. However, the “real” DSALs should work more closely with the domain meta-models. Some other “domain specific” content other than the participated domain elements, such as the domain restrictions, should be reflected in the DSALs as well. For example, if the value of domain element x should not be modified in the predict stage, any attempt to update x in the advice binding the pointcuts that capture the predict stage join points should be detected and alerted. To achieve this, the “translator+GPAL core” pattern seems not enough. Unfortunately, our investigation does not find many promising alternatives. The “library+notation” pattern is very similar to our “translator+GPAL core” pattern, and it has to build its own library for the domain semantics. The “partial evaluation+reflection” pattern demands the reflection function, and work best with object oriented programming languages.

Appendices

Appendix A

Examples

A.1 Traceability Link Examples

```
/* <DomainElementTracing type="Action" name="action1"> */
<code block generated for action1>
/* </DomainElementTracing type="Action" name="action1"> */
```

Listing A.1: A pair of comments serve as a link to trace domain specific element "action1"

```
Begin_Tracing_action1();
<code block generated for action1>
End_Tracing_action1();
```

Listing A.2: A pair of function invocations serve as a link to trace domain specific element "action1"

```
@DomainElementTracing(type="Action" name="action1")
int tracingBegin;
<code block generated for action1>
@DomainElementTracing(type="Action" name="action1")
int tracingEnd;
```

Listing A.3: A pair of annotated variable serve as a link to trace domain specific element "action1"

```
<DomainElementTracing type="Action" name="action1">
  <location>
    <start_point>
      <line_number>10</line_number>
      <column_number>10</column_number>
    </start_point>
    <end_point>
      <line_number>12</line_number>
      <column_number>10</column_number>
    </end_point>
  </location>
</DomainElementTracing>
```

Listing A.4: A node inside a standalone XML log file serves as a link to trace domain specific element "action1"

A.2 The Automata Example

```
grammar DFA;

tokens {
    DFA = 'DFA';
    ALP = 'Alphabet';
    STA = 'States';
    SST = 'StartState';
    AST = 'AcceptStates';
    TRSN = 'Transition';
}

INTLITERAL
    :   IntegerNumber
    ;

fragment
IntegerNumber
    :   '0'
    |   '1'..'9' ('0'..'9')*
    ;

INTLIST
    :   '{' INTLITERAL (',' INTLITERAL)* '}'
    ;

WHITESPACE : ( '\t' | '\n' | ' ' | '\r' | '\u000C' )+ {$
   channel = HIDDEN; };

program
```

```
    :   DFA '{' alphabet_declaration state_declaration
   transition_declaration '}' EOF
    ;

alphabet_declaration
    :   ALP INTLIST ';'
    ;

state_declaration
    :   states_decl start_state_decl accept_states_decl
    ;

states_decl
    :   STA INTLIST ';'
    ;

start_state_decl
    :   SST '(' INTLITERAL ')' ';'
    ;

accept_states_decl
    :   AST INTLIST ';'
    ;

transition_declaration
    :   (transition_decl)*
    ;

transition_decl
    :   TRSN '(' INTLITERAL ',' INTLITERAL ',' INTLITERAL ')
   ' ';'
    ;
```

Listing A.5: The ANTLR grammar of the "DFA" language

```
grammar DSAL;

options {
    backtrack=true;
    memoize=true;
    output=template;
}
```

```
tokens {
    NOT = '!';
    OR  = '||';
    AND = '&&';
}

WHITESPACE
    : ( '\t' | '\n' | ' ' | '\r' | '\u000C' )+ {$channel =
   HIDDEN; }
    ;

program
    :   'aspect' an=IDENTIFIER LBRACE (s+=aspect_statement)+
    RBRACE
    ;

aspect_statement
    :   loc_modifier pd=pcd_decl ad=adv_decl
    ;

loc_modifier
    :   'before'
    |   'after'
    ;

pcd_decl
    :   LPAREN df=disflt RPAREN
    ;

adv_decl
    :   LBRACE cc=custom_code RBRACE
    ;

custom_code
    :   STRINGLITERAL
    ;

disflt
    :   c=conflt d=disflt_f
    ;
```

```
disflt_f
    :   OR d=disflt
    |
    ;

conflt
    :   l=litflt c=conflt_f
    ;

conflt_f
    :   AND c=conflt
    |
    ;

litflt
    :   LPAREN d=disflt RPAREN
    |   af=property_filter
    |   NOT lf=litflt
    ;

/* Placeholder: property_filter
 * Use: a conditional expression of
 * the properties of join point class
 * Expansion sample:
 * "filter_property1 operator value_literal"
 * Note:
 * rule "operator" and "value_literal"
 * will be selected according to
 * the type of "filter_property1"
 */
property_filter
    :
    ;

/* Placeholder: filter_property
 * Use: the properties of join point class
 * Expansion sample:
 * "jp_class1_name COLON property1.1_name"
 * "jp_class1_name COLON property1.2_name"
 *              ...
 * "jp_class2_name COLON property1.1_name"
 *              ...
 */
filter_property
    :
    ;
```

Listing A.6: The DSAL template selected in the automata example

```
grammar DSAL;

options {
    backtrack=true;
    memoize=true;
    output=template;
}

tokens {
    NOT = '!';
    OR  = '||';
    AND = '&&';
}

WHITESPACE
    : ( '\t' | '\n' | ' ' | '\r' | '\u000C' )+ {$channel =
   HIDDEN; }
    ;


program
    :   'aspect' an=IDENTIFIER LBRACE (s+=aspect_statement)+
    RBRACE
    ;

aspect_statement
    :   loc_modifier pd=pcd_decl ad=adv_decl
    ;

loc_modifier
    :   'before'
    |   'after'
    ;

pcd_decl
```

```
    :   LPAREN df=disflt RPAREN
    ;

adv_decl
    :   LBRACE cc=custom_code RBRACE
    ;

custom_code
    :   STRINGLITERAL
    ;

disflt
    :   c=conflt d=disflt_f
    ;

disflt_f
    :   OR d=disflt
    |
    ;

conflt
    :   l=litflt c=conflt_f
    ;

conflt_f
    :   AND c=conflt
    |
    ;

litflt
    :   LPAREN d=disflt RPAREN
    |   af=property_filter
    |   NOT lf=litflt
    ;

/* Placeholder: property_filter
 * Use: a conditional expression of
 * the properties of join point class
 * Expansion sample:
 * "filter_property1 operator value_literal"
 * Note:
```

```
 * rule "operator" and "value_literal"
 * will be selected according to
 * the type of "filter_property1"
 */
property_filter
    :   filter_property1 numerical_comparer INTLITERAL
    |   filter_property2 numerical_comparer INTLITERAL
    |   filter_property3 numerical_comparer INTLITERAL
    ;

/* Placeholder: filter_property
 * Use: the properties of join point class
 * Expansion sample:
 * "jp_class1_name COLON property1.1_name"
 * "jp_class1_name COLON property1.2_name"
 *           ...
 * "jp_class2_name COLON property1.1_name"
 *           ...
 */
filter_property1
    :    'Transition' COLON 'from_state'
    ;

filter_property2
    :    'Transition' COLON 'to_state'
    ;

filter_property1
    :    'Transition' COLON 'input'
    ;
```

Listing A.7: The ANTLR grammar of "AspectDFA" generated by template expansion

Appendix B

Case Studies

B.1 The AUTOFILTER Case Study

```
   <?xml version="1.0" encoding="utf-8"?>
<xs:schema id="EstimationProblemDomainSchema" xmlns:xs="
   http://www.w3.org/2001/XMLSchema">

  <!-- <helper_types>  -->
  <xs:simpleType name="natural_number_type">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="expr_string_type">
    <xs:restriction base="xs:normalizedString"/>
  </xs:simpleType>

    <xs:simpleType name="name_string_type">
    <xs:restriction base="xs:string">
      <xs:pattern value="([A-Za-z_])+"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="mean_type">
    <xs:choice>
      <xs:element name="mean_value" type="double"/>
      <xs:element name="mean_vector_name" type="
   name_string_type"/>
```

```
    </xs:choice>
  </xs:complexType>

  <!--
  <xs:complexType name="basic_data_ref_type">
    <xs:choice>
      <xs:element name="nat_type" type="nat"/>
      <xs:element name="type" type="int"/>
      <xs:element name="type" type="double"/>
    </xs:choice>
  </xs:complexType>
  -->
  <xs:simpleType name="basic_data_ref_type">
    <xs:restriction base="xs:string">
      <xs:enumeration value="nat"/>
      <xs:enumeration value="int"/>
      <xs:enumeration value="double"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="role_type">
    <xs:restriction base="xs:string">
      <xs:enumeration value="const"/>
      <xs:enumeration value="variable"/>
      <xs:enumeration value="data"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="basic_domain_element_type">
    <xs:choice>
      <xs:element name="scalar" type="scalar_type"/>
      <xs:element name="vector" type="vector_type"/>
      <xs:element name="matrix" type="matrix_type"/>
    </xs:choice>
  </xs:complexType>
  <!-- </helper_types>  -->

  <xs:simpleType name="nat">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
    </xs:restriction>
```

```
</xs:simpleType>

<xs:simpleType name="int">
  <xs:restriction base="xs:integer"/>
</xs:simpleType>

<xs:simpleType name="double">
  <xs:restriction base="xs:decimal"/>
</xs:simpleType>

<xs:complexType name="scalar_type">
  <xs:sequence>
    <xs:element name="role" type="role_type"/>
    <xs:element name="name" type="name_string_type"/>
    <xs:element name="element_type" type="
 basic_data_ref_type"/>
    <xs:element name="init_value" type="expr_string_type"
 minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="vector_type">
  <xs:sequence>
    <xs:element name="role" type="role_type"/>
    <xs:element name="name" type="name_string_type"/>
    <xs:element name="element_type" type="
 basic_data_ref_type"/>
    <xs:element name="length" type="natural_number_type"/>
    <xs:element name="init_value" type="expr_string_type"
 minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="matrix_type">
  <xs:sequence>
    <xs:element name="role" type="role_type"/>
    <xs:element name="name" type="name_string_type"/>
    <xs:element name="element_type" type="
 basic_data_ref_type"/>
    <xs:element name="size_1" type="natural_number_type"/>
    <xs:element name="size_2" type="natural_number_type"/>
    <xs:element name="init_value" type="expr_string_type"
 minOccurs="0" maxOccurs="1"/>
   </xs:sequence>
  </xs:complexType>

  <xs:complexType name="vector_with_gaussian_distribution">
   <xs:sequence>
     <xs:element name="target_vector_name" type="
 name_string_type"/>
     <xs:element name="mean" type="mean_type"/>
     <xs:element name="standard_deviation_vector_name" type
 ="name_string_type"/>
   </xs:sequence>
  </xs:complexType>

  <xs:complexType name="equation_set_type">
   <xs:sequence>
     <xs:element name="equation" type="expr_string_type"
 minOccurs="1" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>

  <xs:complexType name="eqs_model_type">
   <xs:sequence>
     <xs:element name="name" type="name_string_type"/>
     <xs:element name="equation_set" type="
 equation_set_type"/>
   </xs:sequence>
  </xs:complexType>

  <xs:complexType name="problem_type">
   <xs:sequence>
     <xs:element name="auxiliary_element" type="
 basic_domain_element_type" minOccurs="0" maxOccurs="
 unbounded"/>
     <xs:element name="statistical_variable_declaration"
 type="vector_with_gaussian_distribution" minOccurs="0"
 maxOccurs="unbounded"/>

     <xs:element name="state_vector" type="vector_type"/>
```

```
      <xs:element name="process_noise_vector" type="
   vector_type"/>
      <xs:element name="process_noise_distribution" type="
   vector_with_gaussian_distribution"/>
      <xs:element name="process_model" type="eqs_model_type"
   />

      <xs:element name="measurement_noise_vector" type="
   vector_type"/>
      <xs:element name="measurement_noise_distribution" type
   ="vector_with_gaussian_distribution"/>
      <xs:element name="measurement_model" type="
   eqs_model_type"/>

      <xs:element name="filter" type="filter_type"/>

      <xs:element name="estimator" type="estimator_type"/>
    </xs:sequence>

    <xs:attribute name="name" type="name_string_type" use="
   required"/>
  </xs:complexType>

  <xs:complexType name="filter_type">
    <xs:sequence>
      <xs:element name="name" type="name_string_type"/>
      <xs:element name="output_parameter" type="
   basic_domain_element_type" minOccurs="0" maxOccurs="
   unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="estimator_type">
    <xs:sequence>
      <xs:element name="name" type="name_string_type"/>
      <xs:element name="steps" type="natural_number_type"
   minOccurs="0" maxOccurs="1"/>
      <xs:element name="update_interval" type="double"
   minOccurs="0" maxOccurs="1"/>
      <xs:element name="timevar" type="name_string_type"
   minOccurs="0" maxOccurs="1"/>
```

```
      <xs:element name="process_eqs" type="name_string_type"
   />
      <xs:element name="measurement_eqs" type="
   name_string_type"/>
      <xs:element name="initials" type="expr_string_type"/>
      <xs:element name="initial_covariance" type="
   expr_string_type"/>
    </xs:sequence>
  </xs:complexType>

  <!-- <xs:element name="problem" type="problem_type"/> --
   >
</xs:schema>
```

Listing B.1: The XML schema of the estimation problem domain meta-model

```
   <?xml version="1.0" encoding="utf-8"?>
<xs:schema id="KalmanFilterDomainSchema" xmlns:xs="http://
   www.w3.org/2001/XMLSchema">

  <xs:include schemaLocation="EstimationProblem_Domain.xsd"/
   >

  <!-- <helper_types> -->
  <xs:simpleType name="kalman_filter_type_type">
    <xs:restriction base="xs:string">
      <xs:enumeration value="standard"/>
      <xs:enumeration value="linearized"/>
      <xs:enumeration value="extended"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="stage_type">
    <xs:restriction base="xs:string">
      <xs:enumeration value="init"/>
      <xs:enumeration value="predict"/>
      <xs:enumeration value="update"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="scalar_calculation_type">
    <xs:sequence>
      <xs:element name="scalar" type="scalar_type"/>
      <xs:element name="stage" type="stage_type"/>
      <xs:element name="expr" type="expr_string_type"/>
      <xs:element name="var_name" type="name_string_type"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="vector_calculation_type">
    <xs:sequence>
      <xs:element name="vector" type="vector_type"/>
      <xs:element name="stage" type="stage_type"/>
      <xs:element name="expr" type="expr_string_type"/>
      <xs:element name="var_name" type="name_string_type"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="matrix_calculation_type">
    <xs:sequence>
      <xs:element name="matrix" type="matrix_type"/>
      <xs:element name="stage" type="stage_type"/>
      <xs:element name="expr" type="expr_string_type"/>
      <xs:element name="var_name" type="name_string_type"/>
    </xs:sequence>
  </xs:complexType>
  <!-- </helper_types>  -->

  <xs:complexType name="kalman_filter_type">
    <xs:complexContent>
      <xs:extension base="filter_type">
        <xs:sequence>
          <xs:element name="process_model" type="matrix_type
   "/>
          <xs:element name="control_model" type="matrix_type
   " minOccurs="0" maxOccurs="1"/>
          <xs:element name="process_noise_covariance" type="
   matrix_type"/>
          <xs:element name="measurement_noise_covariance"
   type="matrix_type"/>
          <xs:element name="observation_model" type="
   matrix_type"/>
          <xs:element name="measurement_vector" type="
   vector_type"/>
```

```
          <xs:element name="Identity_matrix" type="
   matrix_type"/>

          <xs:element name="priori_state_estimate" type="
   vector_calculation_type"/>
          <xs:element name="prior_process_covariance" type="
   matrix_calculation_type"/>
          <xs:element name="innovation" type="
   vector_calculation_type"/>
          <xs:element name="innovation_covariance" type="
   matrix_calculation_type"/>
          <xs:element name="kalman_gain" type="
   matrix_calculation_type"/>
          <xs:element name="posteriori_state_estimate" type=
   "vector_calculation_type"/>
          <xs:element name="posterior_process_covariance"
   type="matrix_calculation_type"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="solution_type">
    <xs:sequence>
      <xs:element name="kalman_filter" type="
   kalman_filter_type"/>
      <xs:element name="kalman_estimator" type="
   estimator_type"/>
    </xs:sequence>
  </xs:complexType>

  <!--<xs:element name="solution" type="solution_type"/>-->
</xs:schema>
```

Listing B.2: The meta-model of the solution domain i.e. the Kalman filter domain

```
   <?xml version="1.0" encoding="utf-8"?>
<xs:schema id="KalmanFilterDomainSchema" xmlns:xs="http://
   www.w3.org/2001/XMLSchema">

  <!-- <helpers>  -->
  <xs:simpleType name="natural_number">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="expr_string">
    <xs:restriction base="xs:normalizedString"/>
  </xs:simpleType>

  <xs:simpleType name="name_string">
    <xs:restriction base="xs:string">
      <xs:pattern value="([A-Za-z_])+"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="basic_data_ref">
    <xs:restriction base="xs:string">
      <xs:enumeration value="nat"/>
      <xs:enumeration value="int"/>
      <xs:enumeration value="double"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- </helpers>  -->

  <xs:complexType name="Scalar">
    <xs:sequence>
      <xs:element name="name" type="name_string"/>
      <xs:element name="element" type="basic_data_ref"/>
      <xs:element name="init_value" type="expr_string"
   minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Vector">
    <xs:sequence>
      <xs:element name="name" type="name_string"/>
      <xs:element name="element" type="basic_data_ref"/>
      <xs:element name="length" type="natural_number"/>
      <xs:element name="init_value" type="expr_string"
   minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
```

```
<xs:complexType name="Matrix">
  <xs:sequence>
    <xs:element name="name" type="name_string"/>
    <xs:element name="element" type="basic_data_ref"/>
    <xs:element name="size_1" type="natural_number"/>
    <xs:element name="size_2" type="natural_number"/>
    <xs:element name="init_value" type="expr_string"
 minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="AlgorithmicParticipantType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Scalar"/>
    <xs:enumeration value="Vector"/>
    <xs:enumeration value="Matrix"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="AlgorithmicStage">
  <xs:annotation>
    <xs:documentation>IJoinInstance</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="init"/>
    <xs:enumeration value="predict"/>
    <xs:enumeration value="update"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="AlgorithmicParticipant">
  <xs:annotation>
    <xs:documentation>IJoinInstance</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="name" type="name_string"/>
    <xs:element name="type" type="
 AlgorithmicParticipantType"/>
  </xs:sequence>
</xs:complexType>
```

```
</xs:schema>
```

Listing B.3: The effective meta-model of the Kalman filter domain

```
%% Kalman Filter Aspect Language

module KFAL

imports KFCS



exports
  sorts CrosscutPatternModifier ReservedString Keyword
   APIdentifier PointcutDescriptorName
   GenericPointcutDescriptor SpecificPointcutDescriptor
   PointcutDescriptorDeclaration AdviceBlock
   CustomizationBlock

  lexical syntax
    "before"                           ->
   CrosscutPatternModifier {cons("before")}
    "after"                            ->
   CrosscutPatternModifier {cons("after")}
    "around"                           ->
   CrosscutPatternModifier {cons("around")}

    "customization"                    -> ReservedString {
   cons("customization")}

    ReservedString                     -> Keyword {cons("
   ReservedString")}
    CrosscutPatternModifier            -> Keyword {cons("
   CrosscutPatternModifier")}

    Keyword                            -> Identifier {
   reject}

    "Base"                             -> Keyword {cons("
   BasecodePlaceholder")}

  context-free syntax
    "$" AlgorithmicParticipant         -> APIdentifier {
   cons("APIdentifier")}
```

```
 APIdentifier                        -> Identifier {cons(
"APApplication")}

 Identifier                          ->
PointcutDescriptorName {cons("PointcutDescriptorName")}
 AlgorithmicStage                    ->
PointcutDescriptorName {reject}

 AlgorithmicStage                                   ->
GenericPointcutDescriptor {cons("
GenericPointcutDescriptor")}
 AlgorithmicStage "$" AlgorithmicParticipant         ->
SpecificPointcutDescriptor {cons("
SpecificPointcutDescriptor")}

 "pointcut" PointcutDescriptorName ":"
GenericPointcutDescriptor            ->
PointcutDescriptorDeclaration {cons("
GenericPointcutDescriptorDeclaration")}
 "pointcut" PointcutDescriptorName ":"
SpecificPointcutDescriptor           ->
PointcutDescriptorDeclaration {cons("
SpecificPointcutDescriptorDeclaration")}

 CrosscutPatternModifier PointcutDescriptorName "()" "{"
Statement "};"      -> AdviceBlock {cons("
PointcutNameBasedAdvice")}
 CrosscutPatternModifier GenericPointcutDescriptor "()" "
{" Statement "};"   -> AdviceBlock {cons("GenericAdvice")
}
 CrosscutPatternModifier SpecificPointcutDescriptor "()"
"{" Statement "};"  -> AdviceBlock {cons("SpecificAdvice"
)}

 "customization" Identifier "{"
PointcutDescriptorDeclaration* AdviceBlock* "}"       ->
CustomizationBlock {cons("CustomizationBlock")}

 "customization" APIdentifier "{" AdviceBlock* "}"    ->
CustomizationBlock {reject}
```

```
    "Base;"                              -> Statement {cons("
   BasecodePlaceholderStatement")}


hiddens
  context-free start-symbols CustomizationBlock
```

Listing B.4: The SDF module of our DSAL template

```
%% KalmanFilterAlgorithm - effective meta-model

module KalmanFilterAlgorithm

imports Whitespace

exports
  sorts Vector Matrix AlgorithmicParticipant
      AlgorithmicStage DomainSpecificElement

  lexical syntax

    "u"                         -> ControlVector
    "x"                         -> StateVector
    "y"                         -> Innovation
    "z"                         -> MeasurementVector

    ControlVector               -> Vector
    StateVector                 -> Vector
    InnovationVector            -> Vector
    MeasurementVector           -> Vector

    "K"                         -> KalmanGain
    "P"                         -> EstimateCovariance
    "S"                         -> InnovationCovariance

    KalmanGain                  -> Matrix
    EstimateCovariance          -> Matrix
    InnovationCovariance        -> Matrix

    Vector                      -> AlgorithmicParticipant
    Matrix                      -> AlgorithmicParticipant
    "init"                    -> AlgorithmicStage {cons("
   init")}
    "predict"                 -> AlgorithmicStage {cons("
   predict")}
    "update"                  -> AlgorithmicStage {cons("
   update")}

    AlgorithmicStage          -> DomainSpecificElement
    AlgorithmicParticipant    -> DomainSpecificElement
```

LISTING B.5: Join point expansion for the KFAL generation

```
SpecificAdviceApplication(|weavingpattern,algostage,
 algoparticipant,customcode,participantlist) = ?tmp;!
 weavingpattern;
  switch id
    case equal(|"before") : !tmp;oncebu(
 SpecificAdviceApplicationBefore(|algostage,
 algoparticipant,customcode,participantlist))
    case equal(|"after")  : !tmp;oncebu(
 SpecificAdviceApplicationAfter(|algostage,algoparticipant
 ,customcode,participantlist))
    case equal(|"around") : !tmp;oncebu(
 SpecificAdviceApplicationAround(|algostage,
 algoparticipant,customcode,participantlist))
    otherwise             : fail
  end

/* SpecificAdviceApplication helper strategy section
 begins. */
SpecificAdviceApplicationBefore(|algostage,algoparticipant
 ,customcode,participantlist) :
  JoinPointSentinelStatement(JoinPointSentinel(
 JoinPointSentinelBeginTag("/*<JoinPoint-Begin",
  CommonJoinPointDescription(StageDescription(algostage),
 KeyRoleDescription(Participant(algoparticipant, apvarid))
 ),
  "/>*/"), JoinPointSentinelContent(Code),
 JoinPointSentinelEndTag("/*<JoinPoint-End",
  CommonJoinPointDescription(StageDescription(algostage),
 KeyRoleDescription(Participant(algoparticipant, apvarid))
 ),
  "/>*/"))) ->
```

```
   JoinPointSentinelStatement(JoinPointSentinel(
  JoinPointSentinelBeginTag("/*<JoinPoint-Begin",
   CommonJoinPointDescription(StageDescription(algostage),
  KeyRoleDescription(Participant(algoparticipant, apvarid))
  ),
   "/>*/"), JoinPointSentinelContent(mergedcode),
  JoinPointSentinelEndTag("/*<JoinPoint-End",
   CommonJoinPointDescription(StageDescription(algostage),
  KeyRoleDescription(Participant(algoparticipant, apvarid))
  ),
   "/>*/"))) where ctmcode := <ReplaceAPWithVar(|
  algoparticipant, apvarid, participantlist, customcode)>
  customcode;mergedcode := <MergeCode(|ctmcode,Code)> Code
```

Listing B.6: Stratego rewriting rules supporting *before*, *after* and *around* advice patterns

```
/* Code manipulation helper strategy section begins. */
MergeCode(|t1,t2) = <oncebu(?CompoundStat(t2S1,t2S2);!
 CompoundStat(CompoundStat(t1,t2S1),t2S2))> t2

MergeCode(|t1,t2) : t1 -> CompoundStat(t1,t2) where !t2;
 not(oncebu(?CompoundStat(_,_)))

ReplaceSingleAPWithVar(|ap, apvar, code) = <innermost(?Id(
 APApplication(APIdentifier(ap)));!Id(apvar))> code

ReplaceAPWithVar(|ap, apvar, pl, code) =
tmpcode := <ReplaceApplier> (pl, code); <innermost(?Id(
 APApplication(APIdentifier(ap)));!Id(apvar))> tmpcode

ReplaceApplier :
([], code) -> code

ReplaceApplier :
([Participant(ap,apvar) | aps], code1) -> code3
where code2 := <ReplaceSingleAPWithVar(|ap, apvar, code1)>
  code1;
code3 := <ReplaceApplier>(aps, code2)

StatementExtractor =
collect(not(?CompoundStat(_,_)))
```

```
StatementWrapperHelper :
([], ct) -> ct

StatementWrapperHelper :
([x|xs], ct1) -> ct3
where ct2 := CompoundStat(ct1,x);
ct3 := <StatementWrapperHelper> (xs,ct2)

StatementWrapper :
[x,y|xs] -> <StatementWrapperHelper> (xs,CompoundStat(x,y)
 )

ReplaceBasecodePlaceHolder(|basecode) :
customcode -> <StatementWrapper> ctclist where extcode :=
 <StatementExtractor> basecode;
ctc := <StatementExtractor> customcode;(befcode,aftcode) :
 = <split-fetch(?BasecodePlaceholderStatement())> ctc;
ctclist := <concat>[befcode,extcode,aftcode]

BasecodePlaceholderHandler(|basecode) :
customcode -> <ReplaceBasecodePlaceHolder(|basecode)>
 customcode where <oncebu(?BasecodePlaceholderStatement())
 > customcode
/* Code manipulation helper strategy section ends. */
```

Listing B.7: Stratego rewriting rules replacing domain element names during merging

B.2 The ANTLR Case Study

```
/*
 [The "BSD licence"]
 Copyright (c) 2005-2007 Terence Parr
 All rights reserved.
 Redistribution and use in source and binary forms, with or
   without
 modification, are permitted provided that the following
   conditions
 are met:
 1. Redistributions of source code must retain the above
   copyright
```

```
    notice, this list of conditions and the following
   disclaimer.
 2. Redistributions in binary form must reproduce the above
   copyright
    notice, this list of conditions and the following
   disclaimer in the
    documentation and/or other materials provided with the
   distribution.
 3. The name of the author may not be used to endorse or
   promote products
    derived from this software without specific prior
   written permission.
 THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY
   EXPRESS OR
 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   IMPLIED WARRANTIES
 OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
    DISCLAIMED.
 IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT,
   INDIRECT,
 INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (
   INCLUDING, BUT
 NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES
   ; LOSS OF USE,
 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
   AND ON ANY
 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
    OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF
 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
   DAMAGE.
*/
grammar ANTLRv3;

grammarDef
   : DOC_COMMENT? ( 'lexer' | 'parser' | 'tree' )
   ;

tokensSpec
   : TOKENS tokenSpec+ '}'
   ;

tokenSpec
   : TOKEN_REF ( '=' ( STRING_LITERAL | CHAR_LITERAL ) | ) '
   ;'
   ;

attrScope
   : 'scope' id ACTION
   ;

action
   : '@' ( actionScopeName '::' )? id ACTION
   ;

actionScopeName
   : id | 'lexer' | 'parser'
   ;

optionsSpec
   : OPTIONS ( option ';' )+ '}'
   ;

option
   : id '=' optionValue
   ;

optionValue
   : id | STRING_LITERAL | CHAR_LITERAL | INT | s = '*'
   ;

rule_
   : DOC_COMMENT? ( ( 'protected' | 'public' | 'private' | '
   fragment' ) )? id '!'? ( ARG_ACTION )? ( 'returns'
   ARG_ACTION )? throwsSpec? optionsSpec? ruleScopeSpec?
   ruleAction* ':' altList ';' exceptionGroup?
   ;

ruleAction
   : '@' id ACTION
   ;
```

```
throwsSpec
   : 'throws' id ( ',' id )*
   ;

ruleScopeSpec
   : 'scope' ACTION | 'scope' id ( ',' id )* ';' | 'scope'
   ACTION 'scope' id ( ',' id )* ';'
   ;

block
   : '(' ( ( optionsSpec )? ':' )? alternative rewrite ( '|'
    alternative rewrite )* ')'
   ;

altList
   : alternative rewrite ( '|' alternative rewrite )*
   ;

alternative
   : element+
   ;

exceptionGroup
   : ( exceptionHandler )+ ( finallyClause )? |
   finallyClause
   ;

exceptionHandler
   : 'catch' ARG_ACTION ACTION
   ;

finallyClause
   : 'finally' ACTION
   ;

element
   : elementNoOptionSpec
   ;

elementNoOptionSpec
```

```
    : id ( '=' | '+=' ) atom ( ebnfSuffix ) | id ( '=' | '+='
     ) block ( ebnfSuffix ) | atom ( ebnfSuffix ) | ebnf |
    ACTION | SEMPRED ( '=>' ) | treeSpec ( ebnfSuffix )
    ;

atom
    : range ( ( '^' | '!' ) ) | terminal_ | notSet ( ( '^' |
    '!' ) ) | RULE_REF ( ARG_ACTION )? ( ( '^' | '!' ) )?
    ;

notSet
    : '~' ( notTerminal | block )
    ;

treeSpec
    : '^(' element ( element )+ ')'
    ;

ebnf
    : block ( '?' | '*' | '+' | '=>' )
    ;

range
    : CHAR_LITERAL RANGE CHAR_LITERAL
    ;

terminal_
    : ( CHAR_LITERAL | TOKEN_REF ( ARG_ACTION ) |
    STRING_LITERAL | '.' ) ( '^' | '!' )?
    ;

notTerminal
    : CHAR_LITERAL | TOKEN_REF | STRING_LITERAL
    ;

ebnfSuffix
    : '?' | '*' | '+'
    ;

rewrite
```

```
   : ( '->' SEMPREDrewrite_alternative )* '->'
   rewrite_alternative
   ;

rewrite_alternative
   : rewrite_template | rewrite_tree_alternative
   ;

rewrite_tree_block
   : '(' rewrite_tree_alternative ')'
   ;

rewrite_tree_alternative
   : rewrite_tree_element+
   ;

rewrite_tree_element
   : rewrite_tree_atom | rewrite_tree_atom ebnfSuffix |
   rewrite_tree ( ebnfSuffix ) | rewrite_tree_ebnf
   ;

rewrite_tree_atom
   : CHAR_LITERAL | TOKEN_REF ARG_ACTION? | RULE_REF |
   STRING_LITERAL | '$' id | ACTION
   ;

rewrite_tree_ebnf
   : rewrite_tree_block ebnfSuffix
   ;

rewrite_tree
   : '^(' rewrite_tree_atom rewrite_tree_element* ')'
   ;

rewrite_template
   :
   ;

rewrite_template_ref
   : id '(' rewrite_template_args ')'
   ;
```

```
rewrite_indirect_template_head
    : '(' ACTION ')' '(' rewrite_template_args ')'
    ;

rewrite_template_args
    : rewrite_template_arg ( ',' rewrite_template_arg )*
    ;

rewrite_template_arg
    : id '=' ACTION
    ;

id
    : TOKEN_REF | RULE_REF
    ;


SL_COMMENT
    : '//' ( ' $ANTLR ' SRC | ~ ( '\r' | '\n' )* ) '\r'? '\n'
     -> skip
    ;


ML_COMMENT
    : '/*' ()*? '*/'
    ;


CHAR_LITERAL
    : '\'' LITERAL_CHAR '\''
    ;


STRING_LITERAL
    : '\'' LITERAL_CHAR LITERAL_CHAR* '\''
    ;


fragment LITERAL_CHAR
    : ESC | ~ ( '\'' | '\\' )
    ;


DOUBLE_QUOTE_STRING_LITERAL
    : '"' ( ESC | ~ ( '\\' | '"' ) )* '"'
    ;


DOUBLE_ANGLE_STRING_LITERAL
    : '<<' ()*? '>>'
    ;


fragment ESC
    : '\\' ( 'n' | 'r' | 't' | 'b' | 'f' | '"' | '\'' | '\\'
    | '>' | 'u' XDIGIT XDIGIT XDIGIT XDIGIT | . )
    ;


fragment XDIGIT
    : '0' .. '9' | 'a' .. 'f' | 'A' .. 'F'
    ;


INT
    : '0' .. '9'+
    ;


ARG_ACTION
    : NESTED_ARG_ACTION
    ;


fragment NESTED_ARG_ACTION
    : '[' ( NESTED_ARG_ACTION | ACTION_STRING_LITERAL |
    ACTION_CHAR_LITERAL | . )* ']'
    ;


ACTION
```

```
   : NESTED_ACTION ( '?' )?
   ;


fragment NESTED_ACTION
   : '{' ( NESTED_ACTION | SL_COMMENT | ML_COMMENT |
   ACTION_STRING_LITERAL | ACTION_CHAR_LITERAL | . )* '}'
   ;


fragment ACTION_CHAR_LITERAL
   : '\'' ( ACTION_ESC | ~ ( '\\' | '\'' ) ) '\''
   ;


fragment ACTION_STRING_LITERAL
   : '"' ( ACTION_ESC | ~ ( '\\' | '"' ) )* '"'
   ;


fragment ACTION_ESC
   : '\\\'' | '\\' '"' | '\\' ~ ( '\'' | '"' )
   ;


TOKEN_REF
   : 'A' .. 'Z' ( 'a' .. 'z' | 'A' .. 'Z' | '_' | '0' .. '9'
    )*
   ;


RULE_REF
   : 'a' .. 'z' ( 'a' .. 'z' | 'A' .. 'Z' | '_' | '0' .. '9'
    )*
   ;


OPTIONS
   : 'options' WS_LOOP '{'
   ;
```

```
TOKENS
   : 'tokens' WS_LOOP '{'
   ;


fragment SRC
   : 'src' ' ' ACTION_STRING_LITERAL ' ' INT
   ;


WS
   : ( ' ' | '\t' | '\r'? '\n' )+ -> skip
   ;


fragment WS_LOOP
   : ( WS | SL_COMMENT | ML_COMMENT )*
   ;


DOC_COMMENT
   : 'DOC_COMMENT'
   ;


PARSER
   : 'PARSER'
   ;


LEXER
   : 'LEXER'
   ;


RULE
   : 'RULE'
   ;
```

```
BLOCK
    : 'BLOCK'
    ;


OPTIONAL
    : 'OPTIONAL'
    ;


CLOSURE
    : 'CLOSURE'
    ;


POSITIVE_CLOSURE
    : 'POSITIVE_CLOSURE'
    ;


SYNPRED
    : 'SYNPRED'
    ;


RANGE
    : 'RANGE'
    ;


CHAR_RANGE
    : 'CHAR_RANGE'
    ;


EPSILON
    : 'EPSILON'
    ;


ALT
```

```
    : 'ALT'
    ;


EOR
    : 'EOR'
    ;


EOB
    : 'EOB'
    ;


EOA
    : 'EOA'
    ;

// end of alt

ID
    : 'ID'
    ;


ARG
    : 'ARG'
    ;


ARGLIST
    : 'ARGLIST'
    ;


RET
    : 'RET'
    ;


LEXER_GRAMMAR
```

```
    : 'LEXER_GRAMMAR'
    ;


PARSER_GRAMMAR
    : 'PARSER_GRAMMAR'
    ;


TREE_GRAMMAR
    : 'TREE_GRAMMAR'
    ;


COMBINED_GRAMMAR
    : 'COMBINED_GRAMMAR'
    ;


INITACTION
    : 'INITACTION'
    ;


LABEL
    : 'LABEL'
    ;


TEMPLATE
    : 'TEMPLATE'
    ;


SCOPE
    : 'scope'
    ;


SEMPRED
    : 'SEMPRED'
    ;


GATED_SEMPRED
    : 'GATED_SEMPRED'
    ;


SYN_SEMPRED
    : 'SYN_SEMPRED'
    ;


BACKTRACK_SEMPRED
    : 'BACKTRACK_SEMPRED'
    ;


FRAGMENT
    : 'fragment'
    ;


TREE_BEGIN
    : '^('
    ;


ROOT
    : '^'
    ;


BANG
    : '!'
    ;


RANGE2
    : '..'
    ;
```

```
REWRITE
   : '->'
   ;
```

Listing B.8: The ANTLR grammar of the ANTLR grammar specification language

```
   <?xml version="1.0" encoding="utf-8"?>
<xs:schema id="generalParsingDomainSchema" xmlns:xs="http://
   www.w3.org/2001/XMLSchema">

  <xs:simpleType name="terminalNameConvention">
    <xs:restriction base="xs:string">
      <!-- by convention -->
      <xs:pattern value="([A-Z_])+"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="nonterminalNameConvention">
    <xs:restriction base="xs:string">
      <!-- by convention -->
      <xs:pattern value="([a-z_])+"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="terminalType">
    <xs:sequence>
      <xs:element name="name" type="terminalNameConvention"/
   >
      <xs:element name="value" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="nonterminalType">
    <xs:sequence>
      <xs:element name="name" type="
   nonterminalNameConvention">
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="singleNonterminalType">
```

```
    <xs:sequence>
      <xs:element name="nonterminal" type="nonterminalType"/
 >
    </xs:sequence>
  </xs:complexType>

  <!--
  <xs:simpleType name="nonZeroUnsignedInt">
    <xs:restriction base="xs:unsignedInt">
      <xs:minInclusive value="1"/>
    </xs:restriction>
  </xs:simpleType>
  -->

  <xs:complexType name="alternativeType">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="terminal" type="terminalType"/>
      <xs:element name="nonterminal" type="nonterminalType"/
 >
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="rhsType">
    <xs:sequence>
      <xs:element name="alternative" type="alternativeType"
 minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ruleType">
    <xs:sequence>
      <xs:element name="lhs" type="singleNonterminalType"/>
      <xs:element name="rhs" type="rhsType"/>
    </xs:sequence>
  </xs:complexType>


  <!-- independent type definition facilitates modular
   extension -->
  <xs:complexType name="GeneralGrammarType">
    <xs:sequence>
```

```
      <xs:element name="symbols">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="terminal" type="terminalType"
   minOccurs="1" maxOccurs="unbounded"/>
            <xs:element name="nonterminal" type="
   nonterminalType" minOccurs="1" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>

      <xs:element name="root" type="singleNonterminalType"/>

      <xs:element name="rules">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="rule" type="ruleType"
   minOccurs="1" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>

    <xs:attribute name="name" type="xs:string" use="required
   "/>
    <xs:attribute name="type" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:pattern value="LL_STAR"/>
          <xs:pattern value="LALR_1"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>

</xs:schema>
```

Listing B.9: The general CFG domain meta-model

```
   <?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:include schemaLocation="General_Parsing_Domain.xsd"/>
```

```
<xs:redefine schemaLocation="General_Parsing_Domain.xsd">
  <xs:complexType name="ruleType">
    <xs:complexContent>
      <xs:extension base="ruleType">
        <xs:sequence>
          <xs:element name="attributes" type="
attributesType" minOccurs="0" maxOccurs="1"/>
          <xs:element name="parameters" type="
parametersType" minOccurs="0" maxOccurs="1"/>
          <xs:element name="rule_actions" type="
ruleActionType" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="alternativeType">
    <xs:complexContent>
      <xs:extension base="alternativeType">
        <xs:sequence>
          <xs:element name="alternative_action" type="
alternativeActionType" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:redefine>

<!-- <domain elements that do not need to be involved in a
  domain-specific model, i.e., a LL* grammar> -->
<xs:complexType name="peekAction">
  <xs:sequence>
    <xs:element name="peek_token" type="terminalType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="matchAction">
  <xs:sequence>
    <xs:element name="match_token" type="terminalType"/>
  </xs:sequence>
```

```
</xs:complexType>

<xs:complexType name="predictAction">
  <xs:sequence>
    <xs:element name="target_symbol" type="nonterminalType
 "/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="actionType">
  <xs:choice>
    <xs:element name="peek" type="peekAction"/>
    <xs:element name="match" type="matchAction"/>
    <xs:element name="predict" type="predictAction"/>
  </xs:choice>
</xs:complexType>
<!-- </domain elements that do not need to be involved in
 a domain-specific model, i.e., a LL* grammar> -->

<!-- <extension for rule attribute> -->
<xs:simpleType name="attributeTypeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="String"/>
    <xs:enumeration value="int"/>
    <xs:enumeration value="float"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="attributeType">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="type" type="attributeTypeType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="attributesType">
  <xs:sequence>
    <xs:element name="attribute" type="attributeType"
 minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

```
  <!-- </extension for rule attribute> -->

  <!-- <extension for rule parameter> -->
  <xs:complexType name="parametersType">
    <xs:sequence>
      <xs:element name="parameter" type="attributeType"
   minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <!-- </extension for rule parameter> -->

  <!-- <extension for rule action> -->
  <xs:simpleType name="alternativeActionType">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>

  <xs:complexType name="ruleActionType">
    <xs:sequence>
      <xs:element name="before_rule_action" type="xs:string"
    minOccurs="0" maxOccurs="1"/>
      <xs:element name="after_rule_action" type="xs:string"
   minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
  <!-- </extension for rule action> -->

  <xs:element name="grammar" type="GeneralGrammarType"/>
</xs:schema>
```

Listing B.10: The *LL(*)* grammar domain meta-model

```
prog
  : (temporal {System.out.println($temporal.esbmc);}) + EOF
  ;

temporal returns [String esbmc, int a, int c]
  : NEXT disj {$esbmc = "X ".concat($disj.esbmc);$a=1;}
  | EVENTUALLY disj {$esbmc = "F ".concat($disj.esbmc);$c
   =3;}
  | GLOBALLY disj {$esbmc = "G ".concat($disj.esbmc);$a=3;$c
   =3;}
  | disj temporal_f[$disj.esbmc] {$esbmc = $temporal_f.esbmc
   ;}
  ;

temporal_f[String left] returns [String esbmc, int b]
  : UNTIL disj {$esbmc = $left.concat(" U ").concat($disj.
   esbmc);}
  | WEAKUNTIL disj
    {$esbmc = "(".concat($left).concat(" U ").concat($disj.
   esbmc).concat(")
    || (G ").concat($left).concat(")");$b=-2;}
  | RELEASE disj {$esbmc = $left.concat(" R ").concat($disj.
   esbmc);}
  | {$esbmc = $left;}
  ;

disj returns [String esbmc]
  : conj disj_f {$esbmc = $conj.esbmc.concat($disj_f.esbmc)
   ;}
  ;

disj_f returns [String esbmc]
  : OR d=disj {$esbmc = " || ".concat($d.esbmc);}
  | {$esbmc = "";}
  ;

conj returns [String esbmc]
  : literal conj_f {$esbmc = $literal.esbmc.concat($conj_f.
   esbmc);}
  ;

conj_f returns [String esbmc]
  : AND c=conj {$esbmc = " && ".concat($c.esbmc);}
  | {$esbmc = "";}
  ;

literal returns [String esbmc]
  : NOT LPAR temporal RPAR
    {$esbmc = "!(".concat($temporal.esbmc).concat(")");}
  | LPAR temporal RPAR
    {$esbmc = "(".concat($temporal.esbmc).concat(")");}
  | NOT atom["!="] {$esbmc = $atom.esbmc;}
  | atom["=="] {$esbmc = $atom.esbmc;}
  ;

atom[String op] returns [String esbmc, int c]
  : input[$op] {$esbmc = $input.esbmc;$c=1;}
  | output[$op] {$esbmc = $output.esbmc;$c=2;}
  ;

input[String op] returns [String esbmc]
  : 'iA' {$esbmc = "{input ".concat($op).concat(" 1}");}
  | 'iB' {$esbmc = "{input ".concat($op).concat(" 2}");}
  | 'iC' {$esbmc = "{input ".concat($op).concat(" 3}");}
  | 'iD' {$esbmc = "{input ".concat($op).concat(" 4}");}
  | 'iE' {$esbmc = "{input ".concat($op).concat(" 5}");}
  | 'iF' {$esbmc = "{input ".concat($op).concat(" 6}");}
  ;

output[String op] returns [String esbmc, int c]
  : 'oU' {$esbmc = "{output ".concat($op).concat(" 21}");$c
   =11;}
  | 'oV' {$esbmc = "{output ".concat($op).concat(" 22}");$c
   =12;}
  | 'oW' {$esbmc = "{output ".concat($op).concat(" 23}");$c
   =13;}
  | 'oX' {$esbmc = "{output ".concat($op).concat(" 24}");$c
   =14;}
  | 'oY' {$esbmc = "{output ".concat($op).concat(" 25}");$c
   =15;}
  | 'oZ' {$esbmc = "{output ".concat($op).concat(" 26}");$c
   =16;}
  ;
```

Listing B.11: Production rules of the RERS grammar

```
grammar AspectRERS;

options {
    backtrack=true;
    memoize=true;
    output=template;
}

tokens {
    ASP = 'aspect';
    PCD = 'pointcut';

    RUL = '@';
    TKN = '#';
    ATR = '$';

    BEF = 'before';
    AFT = 'after';
    MTH = 'match';
    SET = 'set';

    WTN = 'within';
    CFW = 'cflow';

    NOT = '!';
    OR  = '||';
    AND = '&&';

    BEFORE_MAIN_EXIT = 'BeforeMainExit';
}
program
    :   pd=packageDeclaration ad=aspect_declaration EOF
    ;

packageDeclaration
    :   PACKAGE qn=qualifiedName SEMI
    ;

aspect_declaration
    :   am=access_modifier ASP an=IDENTIFIER LPAREN grn=
   IDENTIFIER RPAREN LBRACE asl=aspect_statement_list RBRACE
    ;

access_modifier returns [String code]
    :   PUBLIC {$code = $PUBLIC.text;}
    |   PROTECTED {$code = $PROTECTED.text;}
    |   PRIVATE  {$code = $PRIVATE.text;}
    |   {$code = "private";}
    ;

aspect_statement_list
```

```
    :   (s+=aspect_statement)+
    ;

aspect_statement
    :   md=memberDecl
    |   pd=pcd_decl
    |   ad=adv_decl
    ;

pcd_decl
    :   am=access_modifier pd=pointcut_descriptor SEMI
    ;

pointcut_descriptor
    :   PCD IDENTIFIER EQ p=pcd_body
    ;

pcd_body
    :   df=disflt
    |   mf=macro_filter
    ;

adv_decl
    :   lm=location_modifier COLON d=disflt b=block
    ;

disflt
    :   c=conflt d=disflt_f
    ;

disflt_f
    :   OR d=disflt
    |
    ;

conflt
    :   l=litflt c=conflt_f
    ;

conflt_f
    :   AND c=conflt
    |
    ;

litflt
    :   LPAREN d=disflt RPAREN
    |   af=atomic_filter
    |   NOT lf=litflt
    ;

atomic_filter
    :   pf=pointcut_filter
    |   abf=advice_bound_filter
    ;
advice_bound_filter
    : IDENTIFIER LPAREN RPAREN
    ;

/* Java grammar block below */
```

Listing B.12: The *AspectRERS* template skeleton

```
pointcut_filter
    :   pf=pinpoint_filter
    |   rf=range_filter
    ;

pinpoint_filter
    :   af=attribute_filter
    |   tf=token_filter
    |   nf=nonterminel_filter
    |   bf=branch_filter
    ;

macro_filter
    :   PERCENT m=macro_filter_expression
    ;

macro_filter_expression
    :   b=before_main_exit
    ;

before_main_exit
    :   BEFORE_MAIN_EXIT
    ;

attribute_filter
    :   lm=location_modifier LPAREN rn=rule_indicator ai=
   alt_indicator_for_within COLON an=attr_indicator RPAREN
    |   lm=location_modifier LPAREN rn=rule_indicator ai=
   alt_indicator_for_within COLON NOT an=attr_indicator
   RPAREN
    |   lm=location_modifier LPAREN NOT rn=rule_indicator ai
   =alt_indicator_for_within COLON an=attr_indicator RPAREN
    |   lm=location_modifier LPAREN an=attr_indicator RPAREN
    |   lm=location_modifier LPAREN NOT an=attr_indicator
   RPAREN
    ;

location_modifier returns
    :   BEF
    |   AFT
    ;

rule_indicator returns
    :   RUL IDENTIFIER
    |
    ;

alt_indicator_for_within
    :   COLON ai=INTLITERAL
    |   COLON NOT ai=INTLITERAL
    |
    ;

attr_indicator returns
    :   ATR IDENTIFIER
    |   ATR
    ;

token_filter
    :   lm=location_modifier LPAREN rn=rule_indicator ai=
   alt_indicator_for_within COLON tn=token_indicator RPAREN
    |   lm=location_modifier LPAREN rn=rule_indicator ai=
   alt_indicator_for_within COLON NOT tn=token_indicator
   RPAREN
    |    lm=location_modifier LPAREN NOT rn=rule_indicator ai
   =alt_indicator_for_within COLON tn=token_indicator RPAREN
    |    lm=location_modifier LPAREN tn=token_indicator
   RPAREN
    |    lm=location_modifier LPAREN NOT tn=token_indicator
   RPAREN
    ;

token_indicator returns [String code]
    :    TKN IDENTIFIER {$code=$IDENTIFIER.text;}
    |    TKN {$code="*";}
    ;

nonterminel_filter
    :    lm=location_modifier LPAREN rn=rule_indicator ai=
   alt_indicator_for_within COLON srn=subrule_indicator
   RPAREN
    |    lm=location_modifier LPAREN rn=rule_indicator ai=
   alt_indicator_for_within COLON NOT srn=subrule_indicator
   RPAREN
    |    lm=location_modifier LPAREN NOT rn=rule_indicator ai
   =alt_indicator_for_within COLON srn=subrule_indicator
   RPAREN
    ;

subrule_indicator returns [String code]
    :    RUL IDENTIFIER {$code=$IDENTIFIER.text;}
    |    RUL {$code="*";}
    ;

branch_filter
    :    lm=location_modifier LPAREN rn=rule_indicator COLON
   bi=branch_indicator RPAREN
    |    lm=location_modifier LPAREN rn=rule_indicator COLON
   NOT bi=branch_indicator RPAREN
    |    lm=location_modifier LPAREN NOT rn=rule_indicator
   COLON bi=branch_indicator RPAREN
    |    lm=location_modifier LPAREN NOT rn=rule_indicator
   COLON NOT bi=branch_indicator RPAREN
    |    lm=location_modifier LPAREN rn=rule_indicator RPAREN
    |   lm=location_modifier LPAREN NOT rn=rule_indicator
   RPAREN
    ;

branch_indicator returns [String code]
    :   INTLITERAL {$code=$INTLITERAL.text;}
    |   {$code="*";}
    ;

range_filter
    :   wf=within_filter
    |   cf=cflow_filter
    ;

within_filter
    :   WTN LPAREN rn=rule_indicator ai=
   alt_indicator_for_within RPAREN
    |   WTN LPAREN NOT rn=rule_indicator ai=
   alt_indicator_for_within RPAREN
    ;

cflow_filter
    :   CFW LPAREN rn=rule_indicator ai=
   alt_indicator_for_cflow RPAREN
    |   CFW LPAREN NOT rn=rule_indicator ai=
   alt_indicator_for_cflow RPAREN
    ;

alt_indicator_for_cflow
    :   COLON ai=INTLITERAL
    |   COLON NOT ai=INTLITERAL
    |
    ;
```

Listing B.13: The *AspectRERS* template skeleton

```
attribute_filter
    :   lm=location_modifier LPAREN rn=rule_indicator ai=
   alt_indicator_for_within COLON an=attr_indicator RPAREN
        -> attrFilter1(locationModifier={$lm.code},
   ruleNameValue={$rn.value}, ruleNamePattern={$rn.pattern},
    alt={$ai.st}, attributeName={$an.code})
    |   lm=location_modifier LPAREN rn=rule_indicator ai=
   alt_indicator_for_within COLON NOT an=attr_indicator
   RPAREN
        -> attrFilter2(locationModifier={$lm.code},
   ruleNameValue={$rn.value}, ruleNamePattern={$rn.pattern},
    alt={$ai.st}, attributeName={$an.code})
    |   lm=location_modifier LPAREN NOT rn=rule_indicator ai
   =alt_indicator_for_within COLON an=attr_indicator RPAREN
        -> attrFilter3(locationModifier={$lm.code},
   ruleNameValue={$rn.value}, ruleNamePattern={"\\\\w*"},
   alt={$ai.st}, attributeName={$an.code})
    |   lm=location_modifier LPAREN an=attr_indicator RPAREN
        -> attrFilter1(locationModifier={$lm.code},
   ruleNameValue={"*"}, alt={""}, attributeName={$an.code})
    |   lm=location_modifier LPAREN NOT an=attr_indicator
   RPAREN
        -> attrFilter2(locationModifier={$lm.code},
   ruleNameValue={"*"}, alt={""}, attributeName={$an.code})
    ;

location_modifier returns [String code]
    :   BEF {$code = "Begin";}
    |   AFT {$code = "End";}
    ;

rule_indicator returns [String value, String pattern]
    :   RUL IDENTIFIER
        {$value=$IDENTIFIER.text;$pattern=$IDENTIFIER.text;}
    |   {$value="*";$pattern="\\\\w*";}
    ;

alt_indicator_for_within
    :   COLON ai=INTLITERAL -> altFilterForWithin(altIndex={
   $ai.text})
    |   COLON NOT ai=INTLITERAL -> reverseAltFilterForWithin
   (altIndex={$ai.text})
    |   -> eptStr()
    ;

attr_indicator returns [String code]
    :   ATR IDENTIFIER {$code=$IDENTIFIER.text;}
    |    ATR {$code="*";}
    ;
```

LISTING B.14: String templates attrFilterN to translate attribute setting pointcuts

```
altFilterForWithin(altIndex) ::= <<
&&if(<grammarName>Parser.branchTrace.peek().matches("<
   ruleNamePattern>:<altIndex>"))
>>

reverseAltFilterForWithin(altIndex) ::= <<
&&if(!<grammarName>Parser.branchTrace.peek().matches("<
   ruleNamePattern>:<altIndex>"))
>>

/* default behaviour of @rule:alt:$attr is set($attr)&&
   within(@rule:alt) */
attrFilter1(locationModifier, ruleNameValue, ruleNamePattern
   , alt, attributeName) ::= <<
call(public static void <grammarName>Parser.<
   locationModifier>_Set_Rule_<ruleNameValue>_Attribute_<
   attributeName>())<alt>
>>

attrFilter2(locationModifier, ruleNameValue, ruleNamePattern
   , alt, attributeName) ::= <<
call(public static void <grammarName>Parser.<
   locationModifier>_Set_Rule_<ruleNameValue>_Attribute_*())
   &&!call(public static void <grammarName>Parser.<
   locationModifier>_Set_Rule_<ruleNameValue>_Attribute_<
   attributeName>())<alt>
>>

attrFilter3(locationModifier, ruleNameValue, ruleNamePattern
   , alt, attributeName) ::= <<
call(public static void <grammarName>Parser.<
   locationModifier>_Set_Rule_*_Attribute_<attributeName>())
   &&!call(public static void <grammarName>Parser.<
   locationModifier>_Set_Rule_<ruleNameValue>_Attribute_<
   attributeName>())<alt>
>>
```

LISTING B.15: Definition of the string templates for attribute_filter in AspectRERS.stg

```
parExpression
    :   '(' expression ')'
    ;

expressionList
    :   expression
        (',' expression
        )*
    ;


expression
    :   conditionalExpression
        (assignmentOperator expression
        )?
    ;


assignmentOperator
    :   '='
    |   '+='
    |   '-='
    |   '*='
    |   '/='
    |   '&='
    |   '|='
    |   '^='
    |   '%='
    |    '<' '<' '='
    |    '>' '>' '>' '='
    |    '>' '>' '='
    ;


conditionalExpression
    :   conditionalOrExpression
        ('?' expression ':' conditionalExpression
        )?
    ;

conditionalOrExpression
```

```
    :   conditionalAndExpression
        ('||' conditionalAndExpression
        )*
    ;

conditionalAndExpression
    :   inclusiveOrExpression
        ('&&' inclusiveOrExpression
        )*
    ;

inclusiveOrExpression
    :   exclusiveOrExpression
        ('|' exclusiveOrExpression
        )*
    ;

exclusiveOrExpression
    :   andExpression
        ('^' andExpression
        )*
    ;

andExpression
    :   equalityExpression
        ('&' equalityExpression
        )*
    ;

equalityExpression
    :   instanceOfExpression
        (
            (   '=='
            |   '!='
            )
            instanceOfExpression
        )*
    ;

instanceOfExpression
    :   relationalExpression
```

```
        ('instanceof' type
        )?
    ;

relationalExpression
    :   shiftExpression
        (relationalOp shiftExpression
        )*
    ;

relationalOp
    :    '<' '='
    |    '>' '='
    |   '<'
    |   '>'
    ;

shiftExpression
    :   additiveExpression
        (shiftOp additiveExpression
        )*
    ;


shiftOp
    :    '<' '<'
    |    '>' '>' '>'
    |    '>' '>'
    ;


additiveExpression
    :   multiplicativeExpression
        (
            (   '+'
            |   '-'
            )
            multiplicativeExpression
         )*
    ;
```

```
multiplicativeExpression
    :
        unaryExpression
        (
            (   '*'
            |   '/'
            |   '%'
            )
            unaryExpression
        )*
    ;
```

Listing B.16: Production rules involved in the application of parExpression in Java ANTLR grammar

```
import java.util.Random;
import java.util.HashMap;


public class testJavaProg {
  public static int a1=0;
  public boolean flag=true;
  private static HashMap<String,Integer> map = new HashMap<
   String,Integer>();

    public static void s_test(int x, char cc) {
      double y=(1>0)?1.5:0.5;
        int dec_only;
        dec_only=1;
      int z=10;
        char no_init;
      if (y<=1) {
        System.out.println(y-1);
      } else if (y!=2&&z<=1) {
        System.out.println(y);
      } else {
        System.out.println(y++);
      }
      for (;true;z--) {
        if (z>3) continue;
        else break;
      }
        System.out.println(x);
```

```
}

public int m1() {
  synchronized(this) {
    System.out.println("sync block");
  }
  Random ran = new Random();
  int x = 0;

  while (x<7) {
    x=ran.nextInt(6) + 3;
  }

  do {
    x=ran.nextInt(6) + 3;
  } while (x<7);

  switch (x) {
    case 1: System.out.println(1);break;
  case 5: System.out.println(x+1);break;
  default: System.out.println("default");
  }

  return x;
}

public static int s_m2() {
  testJavaProg tjp = null;
  try{
    if(tjp.flag==true) tjp=new testJavaProg();
    if(true) tjp.flag=false;
    if(tjp.m1()==2) System.out.println("nothing");
    throw new Exception("bla");
  } catch(ArrayIndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
  } catch(IOException  e) {
    System.out.println(e.getMessage());
  } catch(FileNotFoundException  e) {
    System.out.println(e.getMessage());
  } finally {
        System.out.println("finally");
        }
      return tjp.m1();
    }

    public static void main(String[] args) {
        int tmp=2;
        s_test(((tmp)));
        s_test(tmp);
    }
}
```

Listing B.17: Our artificial test Java program testJavaProg.java

```
    /* <begin: primitive statement> */
    pointcut assert_stmt=after(@statement:2)||after(
   @statement:3);
    after : assert_stmt() {
        System.out.println("end of an assert statement\n");
    }
    pointcut branch_stmt=after(@statement:4);
    after : branch_stmt() {
        System.out.println("end of an if-else statement\n");
    }
    pointcut for_stmt=after(@statement:5);
    after : for_stmt() {
        System.out.println("end of a for statement\n");
    }
    pointcut while_stmt=after(@statement:6);
    after : while_stmt() {
        System.out.println("end of a while statement\n");
    }
    pointcut do_stmt=after(@statement:7);
    after : do_stmt() {
        System.out.println("end of a do statement\n");
    }
    pointcut try_stmt=after(@statement:8);
    after : try_stmt() {
        System.out.println("end of a try-catch statement\n")
   ;
    }
    pointcut switch_stmt=after(@statement:9);
    after : switch_stmt() {
        System.out.println("end of a switch statement\n");
```

```
 }
 pointcut sync_stmt=after(@statement:10);
 after : sync_stmt() {
     System.out.println("end of a synchronized statement\
n");
 }
 pointcut return_stmt=after(@statement:11);
 after : return_stmt() {
     System.out.println("end of a return statement\n");
 }
 pointcut throw_stmt=after(@statement:12);
 after : throw_stmt() {
     System.out.println("end of a throw statement\n");
 }
 pointcut break_stmt=after(@statement:13);
 after : break_stmt() {
     System.out.println("end of a break statement\n");
 }
 pointcut cont_stmt=after(@statement:14);
 after : cont_stmt() {
     System.out.println("end of a continue statement\n");
 }
 pointcut norm_stmt=after(@statement:15);
 after : norm_stmt() {
     System.out.println("end of a normal statement\n");
 }
 pointcut label_stmt=after(@statement:16);
 after : label_stmt() {
     System.out.println("end of a label statement\n");
 }
 pointcut empty_stmt=after(@statement:17);
 after : empty_stmt() {
     System.out.println("end of an empty statement\n");
 }

 pointcut countable_primitive_stmt=assert_stmt()||
return_stmt()||throw_stmt()||norm_stmt();
 pointcut uncountable_primitive_stmt=branch_stmt()||
for_stmt()||while_stmt()||do_stmt()||try_stmt()||
switch_stmt()||sync_stmt()||break_stmt()||cont_stmt()||
label_stmt()||empty_stmt();
```

```
 /* <end: primitive statement> */
```

Listing B.18: AspectJava code to identify the Java primitive statements

```
 /* <begin: countable statement> */
 /* <begin: variable/field declaration> */
 pointcut local_var_decl=after(@variableDeclarator:
@variableInitializer)&&cflowpattern(
@localVariableDeclarationStatement:1,
@localVariableDeclaration:1,@variableDeclarator:1);
 after : local_var_decl() {
     System.out.println("end of a local variable
declaration\n");
 }
 pointcut class_var_decl=after(@memberDecl:1);
 after : class_var_decl() {
     System.out.println("end of a class field declaration
\n");
 }
 /* <end: variable/field declaration> */

 /* <begin: method declaration> */
 pointcut class_method_name_decl_1=before(
@methodDeclaration:@blockStatement);
 pointcut class_method_name_decl_2=before(
@methodDeclaration:@block);
 pointcut class_method_name_decl=class_method_name_decl_1
()||class_method_name_decl_2();
 after : class_method_name_decl() {
     System.out.println("start of a method declaration\n"
);
 }
 /* <end: method declaration> */

 /* <begin: class declaration> */
 pointcut class_name_decl=before(@normalClassDeclaration:
@classBody);
 after : class_name_decl() {
     System.out.println("start of a class declaration\n")
;
 }
 /* <end: class declaration> */
 pointcut countable_stmt=countable_primitive_stmt()||
local_var_decl()||class_var_decl()||
class_method_name_decl()||class_name_decl();
```

Listing B.19: AspectJava code for the declarations to count

```
 /* <begin: countable sub-statement> */
 /* <begin: branch condition> */
 pointcut if_predicate=after(@statement:@parExpression)&&
within(@statement:4);
 after : if_predicate() {
     System.out.println("end of a branch condition (if
predicate)\n");
 }
 /* <end: branch condition> */

 /* <begin: for condition> */
 pointcut for_condition=after(@forstatement:#RPAREN);
 after : for_condition() {
     System.out.println("end of a for condition\n");
 }
 /* <end: for condition> */

 /* <begin: while condition> */
 pointcut while_condition=after(@statement:@parExpression
)&&within(@statement:6);
 after : while_condition() {
     System.out.println("end of a while condition\n");
 }
 /* <end: while condition> */

 /* <begin: do while condition> */
 pointcut do_while_condition=after(@statement:
@parExpression)&&within(@statement:7);
 after : do_while_condition() {
     System.out.println("end of a do-while condition\n");
 }
 /* <end: do while condition> */

 /* <begin: try catch finally> */
 pointcut trystmt_catch=after(@catchClause:#RPAREN);
 after : trystmt_catch() {
     System.out.println("end of a catch clause condition\
n");
 }
 pointcut try_clause_group=trystmt_catch();
 /* <end: try catch finally> */

 /* <begin: switch condition> */
 pointcut switch_condition=after(@statement:
@parExpression)&&within(@statement:9);
 after : switch_condition() {
     System.out.println("end of a switch condition\n");
 }
 /* <end: switch condition> */

 /* <begin: synchronized resource> */
 pointcut sync_source=after(@statement:@parExpression)&&
within(@statement:10);
 after : sync_source() {
     System.out.println("start of a synchronized block\n"
);
 }
 /* <end: synchronized resource> */

 pointcut countable_sub_stmt=if_predicate()||
for_condition()||while_condition()||do_while_condition()
||try_clause_group()||switch_condition()||sync_source();
 /* <end: countable sub-statement> */
```

Listing B.20: AspectJava code for the sub statements to count

B.3 The CUP Case Study

```
   <?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:include schemaLocation="General_Parsing_Domain.xsd"/>

  <xs:complexType name="shiftAction">
    <xs:sequence>
      <xs:choice>
        <xs:element name="before_shift" type="xs:string"/>
        <xs:element name="after_shift" type="xs:string"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="embeddedAction">
    <xs:sequence>
      <xs:element name="rule" type="ruleType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="actionType">
    <xs:choice>
      <xs:element name="shift" type="shiftAction"/>
      <xs:element name="reduce" type="reduceAction"/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="reduceAction">
    <xs:choice>
      <xs:element name="handle_detection" type="xs:string"/>
      <xs:element name="handle_pruning" type="xs:string"/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="LALR1GrammarType">
    <xs:complexContent>
      <xs:extension base="GeneralGrammarType"/>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="grammar" type="LALR1GrammarType"/>
</xs:schema>
```

Listing B.21: The *LALR(1)* grammar domain meta-model

```
// CUP specification for RERS
package cup.RERS;
import java_cup.runtime.*;

/* Terminals (tokens returned by the scanner). */
terminal           NOT, AND, OR;
terminal           NEXT, EVENTUALLY, GLOBALLY;
terminal           UNTIL, WEAKUNTIL, RELEASE;
terminal             LPAR, RPAR;
terminal             IA, IB, IC, ID, IE, IF;
terminal             OU, OV, OW, OX, OY, OZ;

/* Non-terminals */
non terminal         prog, temporal, temporal_f, disj, disj_f
   , conj, conj_f;
non terminal         literal, literal_f, atom, input, output;

/* Precedences */
precedence nonassoc AND, OR;
precedence right NEXT, EVENTUALLY, GLOBALLY, UNTIL,
   WEAKUNTIL, RELEASE;
precedence right NOT;

/* The grammar */
prog         ::= prog:p temporal:t
                 {: RESULT = (String)p + (String)t + "\n"; :}
                 |
                 {: RESULT = ""; :}
                 ;

temporal     ::= NEXT disj:d
                 {: RESULT = "NEXT" + (String)d; :}
                 |
                 EVENTUALLY disj:d
                 {: RESULT = "EVENTUALLY" + (String)d; :}
                 |
                 GLOBALLY disj:d
                 {: RESULT = "GLOBALLY" + (String)d; :}
                 |
                 disj:d temporal_f:t
                 {: RESULT = (String)d + (String)t; :}
                 ;

temporal_f   ::= UNTIL disj:d
                 {: RESULT = "UNTIL" + (String)d; :}
                 |
                 WEAKUNTIL disj:d
                 {: RESULT = "WEAKUNTIL" + (String)d; :}
                 |
                 RELEASE disj:d
                 {: RESULT = "RELEASE" + (String)d; :}
                 |
                 {: RESULT = ""; :}
                 ;

 disj        ::= conj:c disj_f:d
                 {: RESULT = (String)c + (String)d; :}
                 ;

 disj_f      ::= OR disj:d
                 {: RESULT = "|" + (String)d; :}
                 |
                 {: RESULT = ""; :}
                 ;

 conj        ::= literal:l conj_f:c
                 {: RESULT = (String)l + (String)c; :}
                 ;

 conj_f      ::= AND conj:c
                 {: RESULT = "&&" + (String)c; :}
                 |
                 {: RESULT = ""; :}
                 ;

 literal     ::= NOT literal_f:l
                 {: RESULT = "!" + (String)l; :}
                 |
                 literal_f:l
                 {: RESULT = (String)l; :}
                 ;

 literal_f   ::= LPAR temporal:t RPAR
                 {: RESULT = "(" + (String)t + ")"; :}
                 |
                 atom:a
                 {: RESULT = (String)a; :}
                 ;

 atom        ::= input:i
```

```
                {: RESULT = (String)i; :}
                |
                output:o
                {: RESULT = (String)o; :}
                ;

input        ::= IA {: RESULT = "iA"; :}
                |
                IB {: RESULT = "iB"; :}
                |
                IC {: RESULT = "iC"; :}
                |
                ID {: RESULT = "iD"; :}
                |
                IE {: RESULT = "iE"; :}
                |
                IF {: RESULT = "iF"; :}
                ;

output       ::= OU {: RESULT = "oU"; :}
                |
                OV {: RESULT = "oV"; :}
                |
                OW {: RESULT = "oW"; :}
                |
                OX {: RESULT = "oX"; :}
                |
                OY {: RESULT = "oY"; :}
                |
                OZ {: RESULT = "oZ"; :}
                ;
```

Listing B.22: The CUP grammar for RERS

```
reductionActionFilter1(locationModifier, ruleNameValue,
   altIndex) ::= <<
call(public void Parser.<locationModifier>_Action_Rule_<
   ruleNameValue>_Alternative_<altIndex>(String))
>>

reductionActionFilter2(locationModifier, ruleNameValue,
   altIndex) ::= <<
```

```
call(public void Parser.<locationModifier>_Parse_Rule_<
   ruleNameValue>_Alternative_*(String))&&!call(public void
   Parser.<locationModifier>_Action_Rule_<ruleNameValue>
   _Alternative_<altIndex>(String))
>>

reductionActionFilter3(locationModifier, ruleNameValue,
   altIndex) ::= <<
call(public void Parser.<locationModifier>_Parse_Rule_*
   _Alternative_<altIndex>(String))&&!call(public void
   Parser.<locationModifier>_Action_Rule_<ruleNameValue>
   _Alternative_<altIndex>(String))
>>

reductionActionFilter4(locationModifier, ruleNameValue,
   altIndex) ::= <<
call(public void Parser.<locationModifier>_Action_Rule_*
   _Alternative_*(String))&&!call(public void Parser.<
   locationModifier>_Action_Rule_*_Alternative_<altIndex>(
   String))&&!call(public void Parser.<locationModifier>
   _Action_Rule_<ruleNameValue>_Alternative_*(String))
>>
```

Listing B.23: String template definitions behind the translation of the embedded action pointcuts

```
beforeMainExit() ::= <<
execution(public static void main(String[]))
>>

shiftFilter1() ::= <<
call(public void java_cup.runtime.lr_parser.Begin_Shift())
>>

shiftFilter2() ::= <<
call(public void java_cup.runtime.lr_parser.End_Shift())
>>

handleFilter1() ::= <<
call(public void java_cup.runtime.lr_parser.Begin_Reduction
   ())
>>
```

```
handleFilter2() ::= <<
call(public void java_cup.runtime.lr_parser.End_Reduction())
>>
```

Listing B.24: String template definitions behind the translation of the special pointcuts

```
 pointcut bef_OV=before(#20);
 pointcut aft_IB=after(#14);
 after : bef_OV() {
      System.out.println("========begin before token OV
========");
      System.out.println("Parsed text: "+%GetParsedText())
;
      System.out.println(%GetStackToken(2));
      System.out.println("Cur token: "+%GetCurrentToken())
;
      System.out.println("========end before token OV
========");
 }
 before : aft_IB() {
      System.out.println("========begin after token IB
========");
      System.out.println("Parsed text: "+%GetParsedText())
;
      System.out.println(%GetStackToken(2));
      System.out.println("Cur token: "+%GetCurrentToken())
;
      System.out.println("========end after token IB
========");
 }
```

Listing B.25: An AspectRERS aspect snippet to test the token matching pointcuts

```
 pointcut bef_act=before(@literal_f:1);
 pointcut aft_act=after(@input:);
 after : bef_act() {
      System.out.println("========begin before the
embedded action of literal_f:1========");
      System.out.println("Parsed text: "+%GetParsedText())
;
      System.out.println(%GetStackToken(4));
      System.out.println("Cur token: "+%GetCurrentToken())
;
```

```
     System.out.println("Cur branch: "+%GetCurrentBranch
());
     System.out.println("Cur branch text: "+%
GetCurrentBranchText());
     System.out.println("========");
     System.out.print("Cur handle: ");
     %PrintHandleTokens();
     System.out.println("========");
     System.out.println("========end before the embedded
action of literal_f:1========");
 }
 before : aft_act() {
     System.out.println("========begin after act input
:========");
     System.out.println("Parsed text: "+%GetParsedText())
;
     System.out.println(%GetStackToken(2));
     System.out.println("Cur token: "+%GetCurrentToken())
;
     System.out.println("Cur branch: "+%GetCurrentBranch
());
     System.out.println("Cur branch text: "+%
GetCurrentBranchText());
     System.out.println("========");
     System.out.print("Cur handle: ");
     %PrintHandleTokens();
     System.out.println("========");
     System.out.println("========end after act input
:========");
 }
```

Listing B.26: The code block of testing the rule embedded action pointcuts in the test aspect

```
 pointcut bef_sft=%BeforeShift;
 pointcut aft_sft=%AfterShift;
 after : bef_sft() {
     System.out.println("========begin before shift
========");
     System.out.println("Parsed text: "+%GetParsedText())
;
     System.out.println("========end before shift========
");
```

```
 }
 before : aft_sft() {
     System.out.println("========begin after shift
========");
     System.out.println("Parsed text: "+%GetParsedText())
;
     System.out.println("========end after shift========"
);
 }

 pointcut bef_red=%FoundHandle;
 pointcut aft_red=%PrunedHandle;
 after : bef_red() {
     System.out.println("========begin before reduction
========");
     System.out.println("Parsed text: "+%GetParsedText())
;
     System.out.println(%GetStackToken(0));
     %PrintHandleTokens();
     System.out.println("========end before reduction
========");
 }
 before : aft_red() {
     System.out.println("========begin after reduction
========");
     System.out.println("Parsed text: "+%GetParsedText())
;
     System.out.println(%GetStackToken(0));
     %PrintHandleTokens();
     System.out.println("========end after reduction
========");
 }
```

Listing B.27: The code block of testing the "shift" and "reduce" action pointcuts in the test aspect

Bibliography

[1] S. Kent, "Model Driven Engineering," in *Proceedings of the 3rd International Integrated Formal Methods Conference, IFM 2002, May 15-18, 2002, Turku, Finland* (M. J. Butler, L. Petre, and K. Sere, eds.), vol. 2335 of *Lecture Notes in Computer Science*, pp. 286–298, Springer, 2002.

[2] R. B. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Proceedings of the 29th International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA* (L. C. Briand and A. L. Wolf, eds.), pp. 37–54, IEEE Computer Society, 2007.

[3] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. Wiley-IEEE Computer Society Press, 2008.

[4] E. Denney and B. Fischer, "Certifiable program generation," in *Proceedings of the 4th International Generative Programming and Component Engineering Conference, GPCE 2005, September 29 - October 1, 2005, Tallinn, Estonia* (R. Glück and M. R. Lowry, eds.), vol. 3676 of *Lecture Notes in Computer Science*, pp. 17–28, Springer, 2005.

[5] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[6] M. Antkiewicz, "Round-trip engineering of framework-based software using framework-specific modeling languages," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006, September 18-22, 2006, Tokyo, Japan* [154], pp. 323–326.

[7] U. Aßmann, "Automatic roundtrip engineering," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 5, pp. 33–41, 2003.

[8] T. Hettel, M. Lawley, and K. Raymond, "Model synchronisation: Definitions for round-trip engineering," in *Proceedings of the 1st International Theory and Practice of Model Transformations Conference, ICMT 2008, July 1-2, 2008, Zürich, Switzerland* (A. Vallecillo, J. Gray, and A. Pierantonio, eds.), vol. 5063 of *Lecture Notes in Computer Science*, pp. 31–45, Springer, 2008.

[9] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.

[10] A. Mehmood and D. N. A. Jawawi, "Aspect-oriented model-driven code generation: A systematic mapping study," *Information & Software Technology*, vol. 55, no. 2, pp. 395–411, 2013.

[11] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, 2003.

[12] Y. Lin, J. Zhang, and J. Gray, "Model comparison: A key challenge for transformation testing and version control in model driven software development," in *Control in Model Driven Software Development. OOPSLA/GPCE: Best Practices for Model-Driven Software Development*, pp. 219–236, Springer, 2004.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *11th European Conference on Object-Oriented Programming, ECOOP 97, June 9-13, 1997, Jyväskylä, Finland*, vol. 1241, pp. 220–242, Springer, 1997.

[14] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented programming: Introduction," *Communications of the ACM*, vol. 44, pp. 29–32, Oct 2001.

[15] J. Whittle and J. Schumann, "Automating the Implementation of Kalman Filter Algorithms," *ACM Transactions on Mathematical Software*, vol. 30, pp. 434–453, Dec 2004.

[16] T. J. Parr and R. W. Quong, "ANTLR: A Predicated- *LL(k)* Parser Generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[17] S. E. Hudson, F. Flannery, C. S. Ananian, D. Wang, and A. W. Appel, "CUP parser generator for java," *Princeton University*, 1999.

[18] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[19] C. Atkinson and T. Kühne, "Model-driven development: A metamodeling foundation," *IEEE Software*, vol. 20, no. 5, pp. 36–41, 2003.

[20] B. Hailpern and P. Tarr, "Model-driven development: The good, the bad, and the ugly," *IBM systems journal*, vol. 45, no. 3, pp. 451–461, 2006.

[21] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, "Model-driven architecture," in *Proceedings of the Advances in Object-Oriented Information Systems, OOIS 2002 Workshops, Montpellier, France, September 2, 2002* (J.-M. Bruel and Z. Bellahsene, eds.), vol. 2426 of *Lecture Notes in Computer Science*, pp. 290–297, Springer, 2002.

[22] Object Management Group (OMG) Architecture Board, "MDA specifications," 2001. http://www.omg.org/mda/specs.htm.

[23] S. Beydeda, M. Book, and V. Gruhn, *Model-driven software development*. Springer Verlag, 2005.

[24] Object Management Group (OMG) Architecture Board, "MDA Guide V1.0.1," 2001. http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[25] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: the Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[26] P. Laforcade, B. Zendagui, and V. Barré, "A domain-specific-modeling approach to support scenarios-based instructional design," in *Proceedings of the 3rd European Conference on Technology Enhanced Learning, EC-TEL 2008, September 16-19, 2008, Maastricht, The Netherlands* (P. Dillenbourg and M. Specht, eds.), vol. 5192 of *Lecture Notes in Computer Science*, pp. 185–196, Springer, 2008.

[27] Aspect-Oriented Software Association Steering Committee, "Aspect-Oriented Software Development Community & Conference," 2001. http://www.aosd.net/.

[28] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, pp. 1–17, Citeseer, 2003.

[29] S. Choi, J. wook Jang, S. Mohanty, and V. K. Prasanna, "Domain-specific modeling for rapid energy estimation of reconfigurable architectures," *The Journal of Supercomputing*, vol. 26, no. 3, pp. 259–281, 2003.

[30] M. A. Simos, "Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle," in *Proceedings of the ACM SIGSOFT Symposium on Software Reusability, SSR 95, April 23-30, 1995, Seattle, Washington, USA, Co-Located with The 17th International Conference on Software Engineering, ICSE 1995, May 15-18, 2002* (M. H. Samadzadeh and M. K. Zand, eds.), vol. 20 of *ACM SIGSOFT Software Engineering Notes, Special Issue, August 1995*, pp. 196–205, ACM, 1995.

[31] K. Czarnecki, "Generative programming: Methods, techniques, and applications tutorial abstract," *Software Reuse: Methods, Techniques, and Tools*, pp. 477–503, 2002.

[32] T. R. Gruber, "The role of common ontology in achieving sharable, reusable knowledge bases," in *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR 91), April 22-25, 1991, Cambridge, MA, USA* (J. F. Allen, R. Fikes, and E. Sandewall, eds.), pp. 601–602, Morgan Kaufmann, 1991.

[33] M. Musen, "Ontology-oriented design and programming," *Knowledge Engineering and Agent Technology*, vol. 52, pp. 3–16, 2000.

[34] M. I. Sergeevitch, "Information systems metamodels developing for providing structural and semantic interoperability," in *Proceedings of the 2nd International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX 2007), June 14-16, 2007, Szklarska Poreba, Poland*, pp. 59–64, IEEE Computer Society, 2007.

[35] M. Grüninger and J. Lee, "Ontology Applications and Design - Introduction," *Communications of the ACM*, vol. 45, no. 2, pp. 39–41, 2002.

[36] D. Djuric, D. Gasevic, and V. Devedzic, "Ontology modeling and MDA," *Journal of Object Technology*, vol. 4, no. 1, pp. 109–128, 2005.

[37] Object Management Group (OMG), "OMG's Meta Object Facility." http://www.omg.org/mof/.

[38] G. Booch, A. Brown, S. Iyengar, J. Rumbaugh, and B. Selic, "An MDA Manifesto," *Business Process Trends/MDA Journal*, 2004.

[39] W. H. Harrison, "A new strategy for code generation - the general-purpose optimizing compiler," *IEEE Transactions on Software Engineering*, vol. 5, no. 4, pp. 367–373, 1979.

[40] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, vol. 1009. Pearson/Addison-Wesley, 2007.

[41] W. J. Ray and A. Farrar, "Object model driven code generation for the enterprise," in *Proceedings of the 12th IEEE International Workshop on Rapid System Prototyping, RSP 2001, June 25-27, 2001, Monterey, CA, USA*, pp. 84–89, IEEE Computer Society, 2001.

[42] A. V. Aho and R. Sethi, "How hard is compiler code generation?," in *Proceedings of the Fourth Colloquium of Automata, Languages and Programming, July 18-22, 1977, University of Turku, Finland* (A. Salomaa and M. Steinby, eds.), vol. 52 of *Lecture Notes in Computer Science*, pp. 1–15, Springer, 1977.

[43] M. E. Stickel, R. J. Waldinger, M. R. Lowry, T. Pressburger, and I. Underwood, "Deductive composition of astronomical software from subroutine libraries," in *Proceedings of the 12th International Conference on Automated Deduction, June 26 - July 1, 1994, Nancy, France* (A. Bundy, ed.), vol. 814 of *Lecture Notes in Computer Science*, pp. 341–355, Springer, 1994.

[44] C. C. Cleaveland and J. C. Cleaveland, *Program Generators with XML and Java with CD-ROM.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[45] F. Pfenning and C. Elliott, “Higher-order abstract syntax,” in *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation (PLDI), June 22-24, 1988, Atlanta, Georgia, USA* (R. L. Wexelblat, ed.), pp. 199–208, ACM, 1988.

[46] Z. Hemel, L. C. L. Kats, D. M. Groenewegen, and E. Visser, “Code generation by model transformation: a case study in transformation modularity,” *Software and System Modeling*, vol. 9, no. 3, pp. 375–402, 2010.

[47] E. Visser, E. Bouwers, E. Bouwers, K. T. Kalleberg, and M. Bravenboer, “Stratego Program Transformation Language,” 2010. http://strategoxt.org/.

[48] J. Herrington, *Code Generation in Action*. Greenwich, CT, USA: Manning Publications Co., 2003.

[49] K. Czarnecki and U. W. Eisenecker, “Components and generative programming,” in *Software Engineering - ESEC/FSE 99, Proceedings of the 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 1999, Toulouse, France* (O. Nierstrasz and M. Lemoine, eds.), vol. 1687 of *Lecture Notes in Computer Science*, pp. 2–19, Springer, 1999.

[50] ISO/IEC, “Information technology - Programming languages - C, 9899:2011.” http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.

[51] D. Hutchins, “Partial evaluation + reflection = domain specific aspect languages,” in *Proceedings of the 1st Domain-Specific Aspect Languages Workshop held in conjunction with the 5th International Conference on Generative Programming and Component Engineering, GPCE 2006, October 22-26, 2006, Portland, Oregon, USA*, 2006.

[52] J. Radatz, A. Geraci, and F. Katki, “IEEE standard glossary of software engineering terminology,” *IEEE Std*, vol. 610, p. 121, 1990.

[53] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni, “Model traceability,” *IBM Systems Journal*, vol. 45, no. 3, pp. 515–526, 2006.

[54] M. W. Whalen and M. P. E. Heimdahl, “An approach to automatic code generation for safety-critical systems,” in *Proceedings of the 14st IEEE/ACM International Conference on Automated Software Engineering, ASE 1999, October 12-15, 1999, Cocoa Beach, Florida, USA*, pp. 315–318, 1999.

[55] J. P. A. Almeida, P. van Eck, and M. Iacob, “Requirements traceability and transformation conformance in model-driven development,” in *Proceedings of the Tenth IEEE International Enterprise Distributed Object Computing Conference, EDOC*

2006, October 16-20, 2006, Hong Kong, China, pp. 355–366, IEEE Computer Society, 2006.

[56] E. Denney and B. Fischer, “A verification-driven approach to traceability and documentation for auto-generated mathematical software,” in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, ASE 2009, November 16-20, 2009, Auckland, New Zealand*, pp. 560–564, IEEE Computer Society, 2009.

[57] J. Richardson and J. Green, “Automating traceability for generated software artifacts,” in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering, ASE 2004, September 20-25, 2004, Linz, Austria*, pp. 24–33, IEEE Computer Society, 2004.

[58] M. Antkiewicz, T. T. Bartolomei, and K. Czarnecki, “Automatic extraction of framework-specific models from framework-based application code,” in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2007, Atlanta, Georgia, USA, November 5-9, 2007* (R. E. K. Stirewalt, A. Egyed, and B. Fischer, eds.), pp. 214–223, ACM, 2007.

[59] D. Rebernak, M. Mernik, H. Wu, and J. G. Gray, “Domain-specific aspect languages for modularising crosscutting concerns in grammars,” *IET Software*, vol. 3, no. 3, pp. 184–200, 2009.

[60] G. Kiczales, “AspectJ: Aspect-Oriented Programming in Java,” in *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, October 7-10, 2002, Erfurt, Germany, Revised Papers* (M. Aksit, M. Mezini, and R. Unland, eds.), vol. 2591 of *Lecture Notes in Computer Science*, p. 1, Springer, 2002.

[61] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An Overview of AspectJ,” in *Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP 2001, June 18-22, 2001, Budapest, Hungary* (J. L. Knudsen, ed.), vol. 2072 of *Lecture Notes in Computer Science*, pp. 327–353, Springer, 2001.

[62] B. Harbulot and J. R. Gurd, “A join point for loops in AspectJ,” in *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, March 20-24, 2006, Bonn, Germany* (R. E. Filman, ed.), pp. 63–74, ACM, 2006.

[63] N. Ali and A. Rashid, “A state-based join point model for AOP,” in *Proceedings of the 1st ECOOP Workshop on Views, Aspects and Role, VAR 05, in the19th European Conference on Object-Oriented Programming, ECOOP 05, Glasgow, Scotland*, 2005.

[64] M. Shonle, K. J. Lieberherr, and A. Shah, "Xaspects: an extensible system for domain-specific aspect languages," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA* (R. Crocker and G. L. S. Jr., eds.), pp. 28–37, ACM, 2003.

[65] H. Masuhara, G. Kiczales, and C. Dutchyn, "Compilation semantics of aspect-oriented programs," in *Foundations of Aspect-Oriented Languages Workshop, FOAL 2002, at the 1st International Conference on Aspect-Oriented Software Development, AOSD 2002, April 22-26, 2002, University of Twente, Enschede, The Netherlands*, pp. 17–25, 2002.

[66] K. Gybels and J. Brichau, "Arranging language features for more robust pattern-based crosscuts," in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development AOSD 2003, March 17–21, 2003, Boston, MA, USA*, (New York, NY, USA), pp. 60–69, ACM, 2003.

[67] M. Stoerzer and S. Hanenberg, "A classification of pointcut language constructs," in *Workshop on Software-engineering Properties of Languages and Aspect Technologies (SPLAT), held in conjunction with the Fourth International Conference on Aspect-Oriented Software Development, AOSD 2005, March 1418, 2005, Chicago, Illinois, USA*, 2005.

[68] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with AspectJ," *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, 2001.

[69] D. Orleans and K. J. Lieberherr, "DJ: dynamic adaptive programming in java," in *Proceedings of the 3rd International Conference of Metalevel Architectures and Separation of Crosscutting Concerns, REFLECTION 2001, September 25-28, 2001, Kyoto, Japan* (A. Yonezawa and S. Matsuoka, eds.), vol. 2192 of *Lecture Notes in Computer Science*, pp. 73–80, Springer, 2001.

[70] C. Koppen and M. Störzer, "Pcdiff: Attacking the fragile pointcut problem," in *Proceedings of the 1st European Interactive Workshop on Aspects in Software, EIWAS 2004, September, 23-24, 2004, Berlin, Germany*, 2004.

[71] G. Kiczales and M. Mezini, "Separation of concerns with procedures, annotations, advice and pointcuts," in Black [155], pp. 195–213.

[72] K. Ostermann, M. Mezini, and C. Bockisch, "Expressive pointcuts for increased modularity," in Black [155], pp. 214–240.

[73] M. Störzer and F. Forster, "Detecting precedence-related advice interference," in *Proceedings of the 21st IEEE/ACM International Conference on Automated*

Software Engineering, ASE 2006, September 18-22, 2006, Tokyo, Japan [154], pp. 317–322.

[74] D. Stein, S. Hanenberg, and R. Unland, "A uml-based aspect-oriented design notation for aspectj," in *AOSD*, pp. 106–112, 2002.

[75] M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger, and E. Kapsammer, "A survey on uml-based aspect-oriented design modeling," *ACM Comput. Surv.*, vol. 43, no. 4, p. 28, 2011.

[76] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson, "Survey of aspect-oriented analysis and design approaches," 2015.

[77] W. H. Harrison, H. L. Ossher, and P. L. Tarr, "Asymmetrically vs. symmetrically organized paradigms for software composition," tech. rep., RESEARCH REPORT RC22685, IBM THOMAS J. WATSON RESEARCH, 2002.

[78] C. V. Lopes, "Aspect-oriented programming: An historical perspective (what's in a name?)," 2002.

[79] S. Clarke and E. L. A. Baniassad, *Aspect-oriented analysis and design - the theme approach*. Addison Wesley object technology series, Addison-Wesley, 2005.

[80] C. Driver, S. Reilly, É. Linehan, V. Cahill, and S. Clarke, "Managing embedded systems complexity with aspect-oriented model-driven engineering," *ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 2, p. 21, 2010.

[81] D. Simmonds, "Aspect-oriented approaches to model driven engineering," in *Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008, July 14-17, 2008, Las Vegas Nevada, USA, 2 Volumes* (H. R. Arabnia and H. Reza, eds.), pp. 356–361, CSREA Press, 2008.

[82] J. Bennett, K. Cooper, and L. Dai, "Aspect-oriented model-driven skeleton code generation: A graph-based transformation approach," *Science of Computer Programming*, vol. 75, no. 8, pp. 689 – 725, 2010. Designing high quality system/software architectures.

[83] M. A. Wehrmeister, C. E. Pereira, and F. J. Rammig, "Aspect-oriented model-driven engineering for embedded systems applied to automation systems," *IEEE Transactions on Industrial Informatics*, vol. 9, pp. 2373–2386, Nov 2013.

[84] B. Baudry, F. Fleurey, R. France, and R. Reddy, "Exploring the relationship between model composition and model transformation," in *Proceedings of the AOM Workshop at MODELS'05*, (Montego Bay, Jamaica), Oct. 2005.

[85] J. Bézivin, S. Bouzitouna, M. D. D. Fabro, M. Gervais, F. Jouault, D. S. Kolovos, I. Kurtev, and R. F. Paige, "A canonical scheme for model composition," in *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings* (A. Rensink and J. Warmer, eds.), vol. 4066 of *Lecture Notes in Computer Science*, pp. 346–360, Springer, 2006.

[86] F. Fleurey, B. Baudry, R. B. France, and S. Ghosh, "A generic approach for automatic model composition," in *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers* (H. Giese, ed.), vol. 5002 of *Lecture Notes in Computer Science*, pp. 7–15, Springer, 2007.

[87] S. Sen, N. Moha, B. Baudry, and J. Jézéquel, "Meta-model pruning," in *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings* (A. Schürr and B. Selic, eds.), vol. 5795 of *Lecture Notes in Computer Science*, pp. 32–46, Springer, 2009.

[88] J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, 1970.

[89] F. DeRemer, "Simple LR(k) Grammars," *Communications of the ACM*, vol. 14, no. 7, pp. 453–460, 1971.

[90] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel, "Textbased modeling," *CoRR*, vol. abs/1409.6623, 2014.

[91] Object Management Group (OMG), "UML home page." `http://www.uml.org/`.

[92] World Wide Web Consortium (W3C), "XML Scheme Language Specification." `http://www.w3.org/TR/xmlschema-0/`.

[93] Visual Paradigm International, "Visual Paradigm home page." `https://www.visual-paradigm.com/`.

[94] Microsoft Corporation, "Visual Studio - Microsoft Developer Tools." `http://www.visualstudio.com/`.

[95] M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM journal of research and development*, vol. 3, no. 2, pp. 114–125, 1959.

[96] T. J. Parr, "The ANTLR parser generator." `http://www.antlr.org/`.

[97] T. J. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[98] L. M. Garshol, "BNF and EBNF: What are they and how do they work," *acedida pela última vez em*, vol. 16, 2003.

[99] E. Visser, *Syntax Definition for Language Prototyping.* Eelco Visser, 1997.

[100] D. Lohmann, G. Blaschke, and O. Spinczyk, "Generic Advice: On the Combination of AOP with Generative Programming in AspectC++," in *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering, GPCE 2004, October 24-28, 2004, Vancouver, Canada* (G. Karsai and E. Visser, eds.), vol. 3286 of *Lecture Notes in Computer Science*, pp. 55–74, Springer, 2004.

[101] T. J. Parr, "StringTemplate." http://www.stringtemplate.org/.

[102] P. M. Lewis and R. E. Stearns, "Syntax-directed transduction," *Journal of the ACM*, vol. 15, no. 3, pp. 465–488, 1968.

[103] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[104] M. Grewal and A. Andrews, *Kalman Filtering: Theory and Practice using MATLAB.* Wiley-IEEE Press, 2011.

[105] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," tech. rep., University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1995.

[106] A. H. Jazwinski, *Stochastic Processes and Filtering Theory.* Courier Dover Publications, 2007.

[107] S. J. Julier and J. K. Uhlmann, "A new extension of the Kalman filter to nonlinear systems," in *The Proceedings of AeroSense: The 11th International Symposium on Aerospace/Defense Sensing, Simulation and Controls. Orlando, Florida, USA.*, vol. 3068, pp. 182–193, SPIE, 1997.

[108] N. Wirth and D. Gries, *Programming in MODULA-2*, vol. 4. Springer, 1985.

[109] R. C. Martin, *Clean Code: a Handbook of Agile Software Craftsmanship.* Pearson Education, 2008.

[110] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers, "The syntax definition formalism SDF - reference manual," *SIGPLAN Notices*, vol. 24, no. 11, pp. 43–75, 1989.

[111] J. Visser and J. Scheerder, "A Quick Introduction to SDF," *Community Works Institute (CWI), April*, vol. 25, 2000.

[112] M. G. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier, "Efficient Annotated Terms," *Software: Practice and Experience*, vol. 30, pp. 259–291, Mar. 2000.

[113] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT Manual, Part II. The XT Transformation Tools, Chapter 4: Annotated Terms." http://releases.strategoxt.org/strategoxt-manual/unstable/manual/chunk-chapter/tutorial-aterms.html.

[114] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "XT / As Fix." http://www.program-transformation.org/Tools/AsFix.

[115] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT Manual, Part II. The XT Transformation Tools, Chapter 9: Pretty Printing with GPP." http://releases.strategoxt.org/strategoxt-manual/unstable/manual/chunk-chapter/generic-pretty-printing.html#box.

[116] N. Chomsky, *Syntactic Structures*. Walter de Gruyter, 2002.

[117] A. V. Aho and S. C. Johnson, "LR parsing," *ACM Computing Surveys*, vol. 6, no. 2, pp. 99–124, 1974.

[118] T. J. Parr and K. Fisher, "Ll(*): the foundation of the ANTLR parser generator," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, June 4-8, 2011, San Jose, California, USA* (M. W. Hall and D. A. Padua, eds.), pp. 425–436, ACM, 2011.

[119] T. J. Parr, "The Aggregate: PCCTS, Purdue Compiler Construction Tool Set." http://aggregate.org/PCCTS/.

[120] T. J. Parr, "ANTLR Version 3.5.2." https://github.com/antlr/antlr3.

[121] F. Jouault, "The antlr grammar domain metamodel." http://git.eclipse.org/c/mmt/org.eclipse.atl.tcs.git/plain/dsls/ANTLR/Metamodel/ANTLR.km3.

[122] F. Jouault and J. Bézivin, "KM3: A DSL for metamodel specification," in *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14-16, 2006, Proceedings* (R. Gorrieri and H. Wehrheim, eds.), vol. 4037 of *Lecture Notes in Computer Science*, pp. 171–185, Springer, 2006.

[123] P. Wolper, "Temporal logic can be more expressive," in *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science, October 28-30, 1981, Nashville, Tennessee, USA*, pp. 340–348, IEEE Computer Society, 1981.

[124] W3C, "XSL Transformations (XSLT) Version 1.0." http://www.w3.org/TR/xslt/.

[125] E. V. Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering, WCRE 2002, October 28 - November 1, 2002, Richmond, VA, USA* (A. van Deursen and E. Burd, eds.), p. 97, IEEE Computer Society, 2002.

[126] Y. Jiang, "Java Grammar for ANTLR." http://www.antlr3.org/grammar/list.html/.

[127] R. S. Pressman, *Software Engineering: a Practitioners Approach*. McGrow-Hill International Edition, 2005.

[128] C. Guindon, "Eclipse - Juno Simultaneous Release." http://www.eclipse.org/juno/.

[129] A. Eisenberg and M. Chapman, "AspectJ Development Tools (AJDT)." http://www.eclipse.org/ajdt/.

[130] S. C. Johnson, *YACC: Yet Another Compiler-Compiler*. Bell Laboratories, 1975.

[131] S. C. Johnson, "Yacc - home page." http://dinosaur.compilertools.net/yacc/.

[132] T. E. Dickey and R. Corbett, "BYACC: BERKELEY YACC." "http://invisible-island.net/byacc/byacc.html".

[133] GNU Project, "Bison - GNU parser generator." http://www.gnu.org/software/bison/.

[134] S. E. Hudson, "CUP Parser Generator." http://www2.cs.tum.edu/projects/cup/.

[135] T. Mason and D. Brown, *lex & yacc*. O'Reilly, 1992.

[136] J. Levine, *flex & bison*. O'Reilly, 2009.

[137] G. Klein, "JFlex - The Fast Scanner Generator for Java." http://www.jflex.de/.

[138] S. Hudson, "CUP2 User Manual." http://www2.in.tum.de/~petter/cup2/.

[139] D. E. Knuth, "On the translation of languages from left to rigth," *Information and Control*, vol. 8, no. 6, pp. 607–639, 1965.

[140] F. L. DeRemer, "Practical translators for LR (k) languages," *Massachusetts Institute of Technology, PhD Thesis*, 1969.

[141] F. DeRemer and T. J. Pennello, "Efficient computation of LALR(1) look-ahead sets," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 4, pp. 615–649, 1982.

[142] The Git Community, "Git home page." https://git-scm.com/.

[143] R. Prieto-Diaz and P. Freeman, "Classifying software for reusability," *IEEE Software*, vol. 4, no. 1, pp. 6–16, 1987.

[144] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the use of higher-order model transformations," in *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings* (R. F. Paige, A. Hartman, and A. Rensink, eds.), vol. 5562 of *Lecture Notes in Computer Science*, pp. 18–33, Springer, 2009.

[145] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez, "Model-based DSL frameworks," in *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA* (P. L. Tarr and W. R. Cook, eds.), pp. 602–616, ACM, 2006.

[146] Object Management Group (OMG), "QVT 1.2." `http://www.omg.org/spec/QVT/1.2/`.

[147] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing?," *Int. J. Hum.-Comput. Stud.*, vol. 43, no. 5-6, pp. 907–928, 1995.

[148] R. Douence, P. Fradet, and M. Südholt, "A framework for the detection and resolution of aspect interactions," in Batory *et al.* [156], pp. 173–188.

[149] J. Brichau, K. Mens, and K. D. Volder, "Building composable aspect-specific languages with logic metaprogramming," in Batory *et al.* [156], pp. 110–127.

[150] T. software foundation, "The groovy programming language." `http://groovy.codehaus.org/`.

[151] T. Dinkelaker, M. Mezini, and C. Bockisch, "The art of the meta-aspect protocol," in *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, (New York, NY, USA), pp. 51–62, ACM, 2009.

[152] N. Loughran and A. Rashid, "Framed aspects: Supporting variability and configurability for AOP," in *Software Reuse: Methods, Techniques and Tools: 8th International Conference, ICSR 2004, Madrid, Spain, July 5-9, 2009. Proceedings*, vol. 3107 of *Lecture Notes in Computer Science*, pp. 127–140, Springer, 2004.

[153] P. M. S. Nazari, A. Roth, and B. Rumpe, "Management of guided and unguided code generator customizations by using a symbol table," in *Proceedings of the Workshop on Domain-Specific Modeling, DSM@SPLASH 2015, Pittsburgh, PA, USA, October 27, 2015* (J. Gray, J. Sprinkle, J. Tolvanen, and M. Rossi, eds.), pp. 37–42, ACM, 2015.

[154] *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, IEEE Computer Society, 2006.

[155] A. P. Black, ed., *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, vol. 3586 of *Lecture Notes in Computer Science*, Springer, 2005.

[156] D. S. Batory, C. Consel, and W. Taha, eds., *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*, vol. 2487 of *Lecture Notes in Computer Science*, Springer, 2002.