A Model-Driven Framework for Aspect Weaver Construction

Suman Roychoudhury[1], Jeff Gray[2], and Frédéric Jouault[3]

[1]Tata Research Development and Design Center, Pune 411013, India
suman.roychoudhury@tcs.com

[2]University of Alabama, Department of Computer Science
Tuscaloosa, AL 35487, USA
gray@cs.ua.edu

[3]AtlanMod (INRIA & EMN), Nantes, France
frederic.jouault@inria.fr

Abstract. Aspect orientation has been used to improve the modularization of crosscutting concerns that emerge at different levels of software abstraction. Although initial research was focused on imparting aspect-oriented (AO) capabilities to programming languages, the paradigm was later on extended to software artifacts that appear at higher levels of abstraction (e.g., models). In particular, the Model-Driven Engineering (MDE) paradigm has largely benefitted from the inclusion of aspect-oriented techniques. In a converse way, we believe it may also be productive to investigate how MDE techniques can be adopted to benefit the development of aspect-oriented tools. The main objective of this paper is to show how MDE techniques can be used to improve the construction of aspect weavers for General-Purpose Languages (GPLs) through reusable models and transformations. The approach described in the paper uses models to capture the concepts of various Aspect-Oriented Programming (AOP) language constructs at a metamodeling level. These models are then mapped to concrete weavers for GPLs through a combination of higher-order model transformation and program transformation rules. A generic extension to the framework further supports reusability of artifacts among weavers during the construction process. Aspect weavers for FORTRAN and Object Pascal have been constructed using the framework, and their features evaluated against several case study applications.

Keywords: model transformation, program transformation, model engineering, aspect-oriented software development, generative programming

1 Introduction

The history of software development paradigms suggests that a new paradigm often has its genesis in programming languages and then moves up to design and analysis (e.g., structured programming preceded structured design and analysis, and object-oriented programming predated object-oriented design and modeling). This same pattern can also be observed with respect to aspect orientation. Most of the early work on aspects was heavily concentrated on issues at the coding phase of the software

lifecycle [1], and gradually propagated to other phases (e.g., requirement, design / modeling [2, 3, 4]). Similar to the benefits that aspects can offer to modeling, we believe there are distinct advantages that Model-Driven Engineering (MDE) [5] can provide to impart aspect-oriented capabilities to programming languages. Specifically, we have been constructing aspect weavers for various programming languages using a program transformation approach. From our experience, we have found that MDE provides a capability to isolate the dependence on specific transformation engines and enable the generation of aspect weavers from high-level models. The next subsection outlines the challenges of creating aspect weavers that we have observed from our experience.

1.1 Challenges of aspect weaver construction

As a result of programming language research over the past fifty years, a veritable "Tower of Babel" exists with multiple billions of lines of legacy code maintained in hundreds of different languages [6]. In fact, legacy languages are estimated to account for a large percentage of existing production software [7]. Yet, the majority of Aspect-Oriented Programming (AOP) [1] research is focused on just a few modern languages, such as Java. A generalized approach that brings aspects to legacy software is still missing. A naïve proposal would attempt to migrate legacy code into a modern object-oriented language like Java, such that existing tools (e.g., AspectJ [8]) could be applied. However, such a proposition is often not possible due to technical, cultural and political concerns within the institution that owns the legacy code [7]. Rather than bringing the code to existing Java-based weavers, an alternative is to take AOP principles to the legacy languages and tool environments. Given the large number of languages in use, a solution that mitigates the effort needed to create each new weaver is more desirable than an approach that manually recreates a weaver from scratch for each legacy language.

There are several key challenges towards providing an initial methodology that allows usage of aspects in languages other than Java. We have identified four main obstacles toward adoption of aspects for legacy software. The first two challenges (Challenge C1 and C2) are not the primary contribution of this paper and have been addressed in the past with existing technologies. A summary of these challenges are:

- ***Challenge C1 - The Parser Construction Problem:*** Building a parser for a toy language, or a subset of an existing language, is not difficult. But, designing a parser that is capable of handling millions of lines of production legacy code is an onerous task. As observed by Lämmel and Verhoef [6], the dominant factor in producing a renovation tool is constructing the parser. Software developers who want to explore modern restructuring capabilities in legacy systems will require industrial-scale parsers to allow them to evaluate the feasibility of adoption within their organization. Incomplete parsers for small research prototypes will not scale and may leave a negative first impression of aspects.
- ***Challenge C2 - The Weaver Construction Problem:*** When a new program restructuring or modularization idea is conceived (e.g., AOP), it is often desired to impart the idea to older legacy applications. In order to realize such an objective, a capability is needed to perform the underlying transformations and

rewrites on a syntax-tree or on an abstract model. This requires considerable effort to provide a sound and scalable infrastructure for program transformation.

Challenge C1 and *Challenge C2* can be addressed by using program transformation techniques [9, 12]. Firstly, full-fledged parsers available in program transformation frameworks can be reused to assist in constructing aspect weavers. Secondly, program transformation engines generally have support for low-level rewriting (i.e., by using term-rewriting or graph-rewriting) that can be used to construct aspect weavers for multiple GPLs.

However, it is often the case that the integration efforts to support a core set of transformations are repeated for each language to which the new idea is applied. Such repetition of effort is unfortunate and strongly suggests the need for further generalization of transformation objectives. Moreover, the abstraction level at which most transformation systems operate is too low for software developers who are familiar with the concepts of AOP, but unaware of the accidental complexities of program transformation. In our previous work [12], we provided initial solutions to the first two challenges related to parsing and weaving for a specific language (i.e., Object Pascal). The main contributions of this paper focus on the two additional challenges that are introduced as follows:

- ***Challenge C3 – Accidental Complexity of Transformation Specifications:*** An inherent difficulty associated with using program transformation engines is the low-level of abstraction at which a transformation rule is specified. Transformation rules typically quantify over the grammar elements (e.g., terminals, production rules, non-terminals) of a programming language rather than the conceptual elements in the language domain (e.g., objects, methods). Therefore, it is highly desirable to hide the accidental complexities of program transformation systems from AOP end-users and instead provide a conceptual aspect layering on top of an underlying program transformation system.
- ***Challenge C4 – Language-Independent Generalization of Transformation Objectives:*** Although most program transformation engines provide a general toolkit with pre-existing parsers, the transformation rules that actually perform the desired restructuring are encoded to the productions of a specific concrete syntax (i.e., grammar of base language). Thus, all of the effort that is placed into creating the transformations to enable weaving for a given language cannot be reused for another language. A key research contribution of this paper is an approach that adopts an abstract syntax to increase the level of reuse of aspect transformation rules across multiple languages. The contribution uses higher-order transformations to evolve abstract transformations into more specialized versions that are specific to a particular programming language.

In Section 2, *Challenge C3* is addressed by providing an aspect layering on top of a program transformation system. This aspect layering is realized by using model-driven language engineering techniques [21]. In particular, a model-driven front-end is built on top of a program transformation engine that transforms an aspect model to a model that represents the concept of program transformation rules. Such a convenience layer assists in removing the idiosyncrasies of the underlying

transformation technology. Section 3 shows how the model-driven front-end is generalized to accommodate a family of related GPLs. This is achieved by using specific techniques like metamodel extension [22]. Within the context of the generalized framework of Section 3, *Challenge C4* is addressed by constructing a library of generic higher-order model transformations that make use of the concrete syntax of the base language to generate lower-order program transformation rules.

We applied our approach by constructing aspect weavers for languages like Object Pascal (a language popularized by Borland's Delphi) and FORTRAN 90. Both weavers share a generic front-end and the core set of aspect weaving transformations was reused during the construction process for each weaver.

Currently, the framework addresses the domain representing the imperative class of languages and is not evaluated against logic based or functional languages like Prolog or ML. From our understanding, the generalization should be restricted within a particular class of languages (e.g., imperative, functional, or logic based) to extract maximum commonality. The results presented in this research are based on the assumption that these languages share a certain degree of commonality and should be regarded as a stepping stone for a more comprehensive solution that requires extensive validation.

1.2 Overview of paper contents

The next section of the paper introduces our model-driven framework for constructing aspect weavers. Section 3 discusses generalization of the model-driven weaving framework. In Section 4, a series of case studies demonstrating Aspect Pascal and Aspect FORTRAN illustrate what can be achieved with the approach. A brief comparison of related work is summarized in Section 5, with concluding remarks, lessons learned, and future work appearing in Section 6.

2 Model-Driven Aspect Weaving Framework

This section introduces an extensible and scalable model-driven framework for developing aspect weavers for general-purpose programming languages (GPLs). The framework can be used to construct weavers for object-oriented languages (e.g., C++, Java or Object Pascal) and older legacy languages (e.g., C, FORTRAN or COBOL). Figure 1 represents an overview of the model-driven aspect weaving framework.

The scalability of the framework is provided by using a powerful program transformation engine (PTE); namely, the Design Maintenance System (DMS) [9], which represents the back-end of our framework (*item 3* in *Figure 1*). DMS provides support for mature language tools (e.g., lexers, parsers, and analyzers) for more than a dozen programming languages. It has been used to parse several million lines of code written in any of these languages. The adoption of DMS as a back-end provides a solution to *Challenge C1* (*parser construction problem*) through immediate availability of industrial-scale parsers. DMS also provides functionality for transforming a program after it has been parsed. Through transformation rules and a rich API of transformation functions, DMS offers a solution to *Challenge C2 (weaver construction problem)*. Thus, *Challenge C1* and *Challenge C2* as enumerated in Section 1.1 are gratuitously resolved through adoption of a mature program transformation engine into the weaver construction framework [12]. However, the

low-level representation of transformation rules introduces new accidental complexities that make it difficult for programmers to specify aspects at this level.

A model-driven front-end (*item 1* in *Figure 1*) raises the level of abstraction and hides the accidental complexities that are generally associated with a program transformation system. The core of the framework is a higher-order transformation rule generator (*item 2* in *Figure 1*) that produces program transformation rules from an aspect program. The aspect code is initially parsed by the front-end and later processed by the program transformation rule generator. The result is a set of generated program transformation rules that accomplish weaving of the source aspect for a specific language. As shown in Figure 1, the program transformation back-end takes the generated transformation rules and the source program as its inputs and weaves aspects into the source program to produce the transformed target program. The model-driven front-end offers a solution to *Challenge 3 (accidental complexity)* because it hides the complexity of using a program transformation system in its native form.

The following subsections discuss each of the key components (shown as items 1, 2, and 3) of this framework in detail, including their primary benefits and internal mechanisms. The subsections also outline the reasons behind choosing each of these components and explain why it is desirable to follow an MDE philosophy to construct aspect weavers for GPLs. The following subsection introduces the technical details of a weaver built from a program transformation engine.

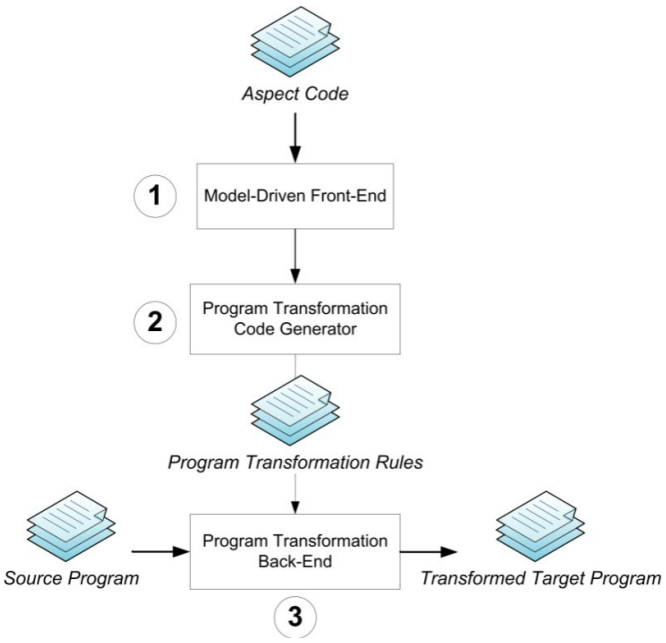


Figure 1: Overview of our model-driven aspect weaver framework

2.1 Background - Program transformation back-end

Fradet and Südholt were among the first to observe that aspect weaving can be performed using a general transformation framework for a specific programming language [10]. Similarly, Aßmann and Ludwig provided an early demonstration of aspect weaving using graph rewriting [11]. Most PTEs support a term-rewriting or graph-rewriting engine such that transformation rules can be constructed that realize the weaving of aspects into a source program. In our previous work, we demonstrated how a PTE can be used to construct an aspect weaver for Object Pascal [12]. Instead of re-inventing a new weaving engine for each new programming language of interest, our objective is to leverage a PTE that provides powerful pattern matching, and term-rewriting capabilities required for aspect weaving.

There are several program transformation engines available with each having their own advantages and disadvantages. In addition to DMS, other popular examples include ASF+SDF [13] and TXL [14]. We chose DMS for the back-end of our framework because of the maturity of the tool (e.g., DMS has been used on several large-scale industrial software renovation projects [15]) and the immediate availability of a large collection of pre-constructed *domains* (i.e., lexers, parsers, and analysis tools) for various programming languages. From our survey of the available transformation tools, DMS was the only tool to supply an Object Pascal and FORTRAN domain that was ready for immediate use to support our experimentation.

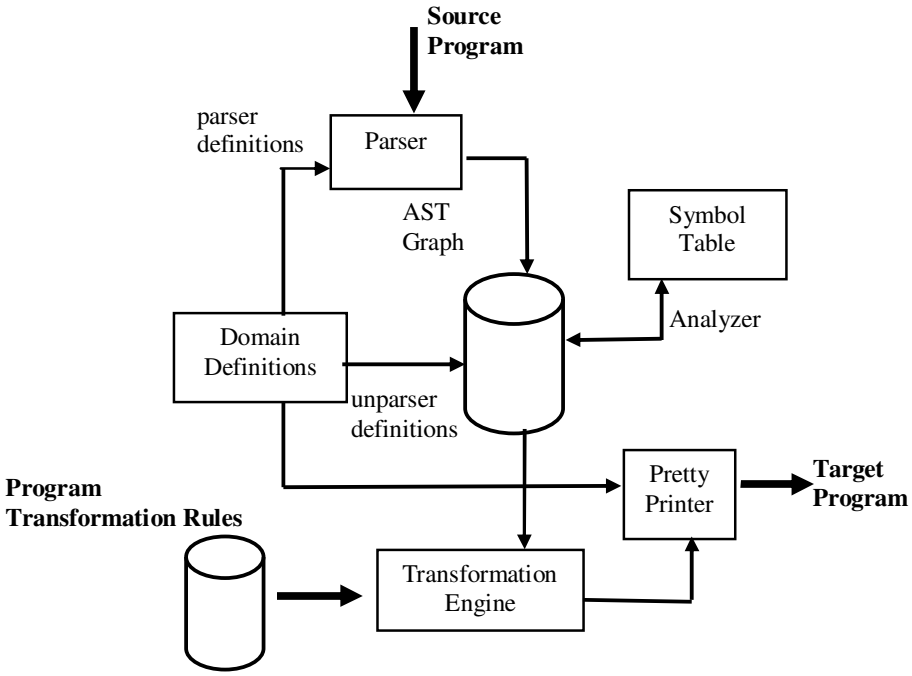


Figure 2: Overview of back-end transformation process

Figure 2 presents an overview of the back-end transformation process (previously shown as item 3 in Figure 1). The program transformation rule (shown in general in Figure 2 with a specific example in Listing 1) is written in the DMS Rule Specification Language (RSL) and processed by the back-end transformation engine to perform the actual rewriting. RSL provides basic primitives for describing the numerous transformations that are to be performed across the entire code base of an application. An RSL program consists of declarations of patterns, rules, conditions, and rule sets using the external form (i.e., concrete syntax) defined by a language domain.

```
default base domain ObjectPascal.
private rule insert_probe(stmt_list: statement_list):
function_body  →  function_body
= "begin \stmt_list end"  →
  "begin WriteLn(\"Entering Method\"); \stmt_list end".
public ruleset TraceAllFunctions = {insert_probe}
```

Listing 1: A simple example of a program transformation rule that traces function executions

Although term-rewriting has several application domains (e.g., code migration, code refactoring or program refinement), the particular example in Listing 1 highlights an aspect-oriented style. The first line of this transformation rule resolves the *domain* (i.e., language) to which the rule can be applied. In this case, a tracing probe is inserted before the execution of all functions written in Object Pascal. The statement list that appears inside of a function body is passed as a parameter to this rule. Note that a rule is typically used as a rewrite specification that maps from a left-hand side (source) syntax tree expression to a right-hand side (target) syntax tree expression (syntactically denoted by "→" in RSL). The `insert_probe` rule matches all function body declarations in the source program and adds a `WriteLn` statement before the execution of the original statement list. Rules can be combined into rule sets that form a transformation strategy by defining a collection of transformations that can be applied to a syntax tree. As shown in Figure 2, these transformations along with the source program are syntactically checked and statically analyzed to ensure the expected weaving behavior. However, RSL rules are typically hardcoded and dependent on the grammar of the base language. For instance, all text highlighted in bold in Listing 1 corresponds either to terminal or to non-terminal symbols in the Object Pascal grammar.

Challenges of program transformation engine usage

Program transformation engines (PTEs) offer several advantages, especially with respect to reusable parsers and a weaving engine. However, to provide advanced aspect weaving capabilities (e.g., like that of AspectJ), the underlying rewrite rules can become significantly more complex than what is shown in Listing 1. For

example, to provide reflective capabilities like `thisJoinPoint` or to perform signature matching with wildcards, more complicated transformation rules are required. Such rules generally use exit functions to do static analysis on the underlying AST [12]. This requires a thorough understanding of the various term rewriting semantics specific to a particular PTE. Moreover, the rewrite rules are often tied to the grammar of the base language (as highlighted in bold in Listing 1), which impedes reusability when the base language changes. Thus, using a tool like DMS to construct aspect weavers requires knowledge of the base language grammar (concrete syntax), and of the core machinery provided by DMS. However, the design decision to use a transformation engine is particularly useful to *challenge C1* and *C2*. Hence, a suitable solution is desired such that program transformation systems can be used not only by language researchers, but by a larger audience through mainstream software development. In our previous research in constructing an aspect weaver for Object Pascal using DMS [12], we observed these broader challenges and recognized that an appropriate front-end support alongside a systematic code generator was needed to bring program transformation systems closer to mainstream software development. The proper selection of an appropriate front-end and program transformation rule generator can hide the accidental complexity associated with PTEs. Nevertheless, aspect weavers can still leverage the power of PTEs to perform the lower-level complex code transformation. In the following subsection, we introduce our investigation into a model-driven front-end and discuss the primary benefits offered by MDE in the overall context of the framework.

2.2 Model-driven Front-end

There are many ways to design the front-end of an aspect language. In some examples, the language format is expressed in raw XML [18], but in other cases it is expressed in a more sophisticated declarative language [19]. Through our investigation in the design of various aspect languages, we realized that the declarative nature of expressing aspects (e.g., as popularized by pointcuts in AspectJ) has a common language-independent characteristic. For example, the concepts of join points, pointcuts and advice can be adapted to many aspect language designs within the same paradigm. Metamodels can precisely capture these concepts and their relations.

In addition, a model-driven front-end (item 1 in Figure 1) is well-suited for abstracting the various semantics associated with PTEs. MDE provides an abstraction layer that can be mapped down to lower-level transformation rules. Combining the technical spaces of MDE and PTE offers more possibilities than each considered separately.

Figure 3 shows an excerpt of the abstract syntax of an aspect language in the form of a metamodel represented as a collection of three class diagrams. This metamodel illustrates the specification of Aspect Pascal, which is an aspect language we defined for Object Pascal. An aspect described in this language consists of `Pointcuts` and `Advice`. They together constitute the fundamental elements for defining an aspect-oriented language (influenced by the asymmetric AspectJ style). As evident in Figure 3, an aspect can have multiple `pointcuts` and multiple `advice`.

An `Advice`, defined as an abstract class in the metamodel, can be further categorized as a `BeforeAdvice`, `AfterAdvice` or `AroundAdvice`. An advice can have advice parameters and an advice body (i.e., a list of statements). Every advice parameter has a type and an associated name that is used for passing the context information. Every advice statement conforms to the grammar of the base language. Because the back-end program transformation engine already has the parser/analyzer available for managing the base language, the body of the advice is typically delegated to the back-end for further processing. Such delegation of an advice body reduces the complexity of the metamodel by not including every possible program construct that belongs to the base GPL. These program fragments are referenced in the front-end metamodel as `OpaqueStatements` (i.e., statements that are not handled by the front-end). In addition to `OpaqueStatement`, there are other special statements: the `loop` statement and the `proceed` statement. The `proceed` statement is generally used in the bodies of an `around` advice and the `loop` statement is a new join point that captures additional weaving operations (e.g., monitoring timing statistics around a FORTRAN "`do loop`" for performance evaluation). An example of a loop statement is given in Section 4.

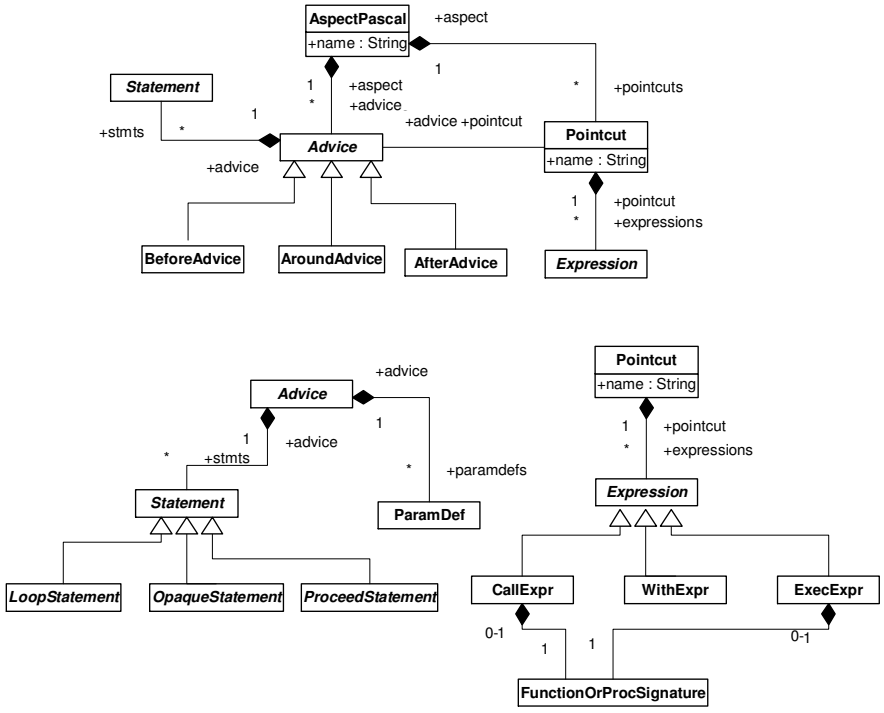


Figure 3: Subset of Aspect Pascal metamodel represented as a class diagram

`Pointcuts` consist of pointcut expressions, which can, for instance, be further expressed as *call* expressions, *with* expressions or *execution* expressions. Pointcut expressions form the key for pattern matching in the lower-level transformation rule. All pointcut expressions are derived from the abstract `Expression` class. As seen in Figure 3, both the `call` and `exec` expressions are derived from `Expression` and both reference the type pattern `FunctionOrProcSignature`, which identifies the prototype declaration (i.e., signature) for a function or procedure defined in Object Pascal. This type pattern is particularly useful for pattern matching. Although `call` and `exec` are the two most common forms of pointcut expressions, new expressions can be experimented with and derived from the base `Expression` class template. For example, Object Pascal allows the definition of `with` expressions that are used to control the execution of code within a specific context. Other pointcut expressions available in the join point model of the base language can be similarly added to the metamodel of the front-end aspect language. *Wildcards* are also allowed and examples are given in Section 4.

The pointcut expressions are translated to RSL patterns or rules that do the actual pattern matching. The front-end AOP layer is simply a façade to the back-end PTE. It helps to hide the accidental complexity associated with PTEs (*Challenge C3*) and also provides a platform to experiment with new AOP language constructs that can be suitably translated to back-end rewrite rules. The translation mechanism that generates the back-end RSL rules from the front-end aspect language is explained in detail in Section 2.3 and Section 3.2.

Implementing the front-end aspect language

The first step in implementing a front-end is to create a metamodel that defines the abstract syntax of the aspect language. The KM3 [20] (Kernel MetaMetaModel) is used for this purpose. Although other MDE tools can be used to define the metamodel, we chose KM3 because it has the added advantage of being independent of the concrete MDE technology (e.g., the Eclipse Modeling Framework or OMG's Meta-Object Facility). The example snippet in Figure 4 demonstrates how KM3 is used to define the Aspect Pascal metamodel.

Figure 4 shows a snippet of the KM3 specification used to implement the metamodel introduced in Figure 3. The `AspectPascal` class contains references to other classes; namely, the core elements `Pointcut` and `Advice`. The `oppositeOf` construct is used to maintain reverse navigational links for efficient traversal purposes required during model transformation (Section 2.3).

In addition to the abstract syntax shown as a metamodel in KM3, the concrete syntax of the aspect language is specified using a grammar-like notation - TCS [23] (Textual Concrete Syntax). Figure 5 illustrates how the concrete syntax of different metamodel elements (e.g., *Aspects*, *Pointcuts,* and *Advice*) is expressed in TCS. In TCS, every class represented in the KM3 specification has its corresponding template definition. It also introduces other terminal tokens like separators, brackets and semicolons that are required to describe the concrete syntax of the aspect language but are not captured in the abstract syntax of the metamodel. Thus, TCS gives the structure of the source aspect language. In addition, context information can also be passed and stored in the symbol table for further analysis. The choice of the particular MDE technology described in this paper (e.g., TCS, KM3 and ATL are part of the

overall AMMA platform) is provided only as a proof of concept. Alternative model-driven technologies may be used to implement the aspect metamodel.

```
class AspectPascal extends LocatedElement {
   attribute name : String;
   reference domain container : Domain;
   reference pointcuts[1-*] container : Pointcut oppositeOf aspect;
   reference advice[1-*] container : Advice oppositeOf aspect;
}
class Pointcut extends Element {
   attribute name : String;
   reference aspect : AspectPascal oppositeOf pointcut;
   reference paramdefs[*] container : ParameterDef;
   reference exprs[1-*] container : Expression oppositeOf pointcut;
}
abstract class Advice extends LocatedElement {
   reference aspect : AspectPascal oppositeOf advice;
   reference pointcut : Pointcut;
   reference paramdefs[*] container : ParameterDef;
   reference stmts[1-*] container : Statement;
}
```

Figure 4: KM3 specification (snippet) for Aspect Pascal

```
template AspectPascal main
       :  "aspect" name "{" pointcut advice "}"
       ;
template Pointcut context addToContext
       :  "pointcut" name "(" paramdefs{separator = ","} ")"
          ":" exprs {separator = "&&"} ";"
       ;
template Advice abstract;

template BeforeAdvice
       :  "before" "(" paramdefs {separator = ","} ")" ":"
          ...
       ;
template AfterAdvice
       :  "after"  "(" paramdefs {separator = ","} ")" ":"
           ...
       ;
```

Figure 5: TCS specification (snippet) for Aspect Pascal

The front-end would be incomplete without appropriate code generators that transform the front-end aspect language to its corresponding target language. In our model-driven framework, the back-end is the transformation language of the PTE; specifically, the DMS RSL in our case. The following section demonstrates how RSL transformation rules are generated from the front-end aspect specification.

2.3 Program transformation rule generator

The program transformation rule generator (shown as item 2 in Figure 1) represents the core of the framework and embodies a higher-order transformation (i.e., a model transformation rule is used to generate program transformation rules). As mentioned earlier, the front-end aspect language is only a façade to the back-end PTE and all pointcut declarations and advice code present in the source aspect language are eventually translated to target RSL code that consists of RSL patterns, external conditions and rewrite rules. Therefore, the goal of the program transformation rule generator is to synthesize transformation engine specific weaving code (RSL) from the front-end representation defined by a higher-order aspect specification.

Target metamodel for RSL

In order to realize a systematic translation from a higher-order aspect language to a lower-order transformation language, it is necessary to define a metamodel for the back-end program transformation engine. The target RSL metamodel serves two basic purposes: firstly, it allows experimenting with new aspect languages (e.g., Aspect Ruby or Aspect FORTRAN) and new aspect constructs (e.g., loops) without changing the model for the back-end PTE. In this case, the commonalities of different aspect languages for various GPLs can be generalized in a generic aspect metamodel. The differences can be captured using metamodel extensions; however, no change is required for the target metamodel. This helps to improve the generality of the framework.

Secondly, instead of an ad hoc technique, a metamodel allows more sophisticated translations where complex pointcut expressions and join point shadows (areas in the source where join points may emerge) [40] from the front-end aspect language could be mapped correspondingly to patterns and rules in the back-end RSL language. The presence of a target metamodel provides an internal representation of the back-end transformation language (RSL) that can be used to validate the generated lower-order transforms. For future experimental purposes, the presence of a RSL metamodel may also permit bidirectional mappings (currently, the mapping is unidirectional, from Aspect-to-RSL). In such a scenario, given a generated RSL program as input, the corresponding aspect specification for a different GPL may be recovered, provided a mapping exists between RSL and the GPL.

To capture the essential concepts of the DMS RSL, an RSL metamodel has been created in KM3, illustrated by a class diagram in Figure 6. As noted earlier, RSL consists of elements like patterns, rules, conditions, and rule-sets, which are captured in this metamodel. The complete KM3 and TCS specification for the RSL metamodel is available at [30]. It should be noted that the target metamodel defines the essence (i.e., concepts and relations) of a domain without concern for semantics. In our case, the semantics of the various components of the source aspect metamodel is captured in the mapping to RSL defined as an ATL transformation. ATL [24] is the model transformation language of AMMA. The semantics of the aspect language is thus captured in terms of the semantics of RSL, which is in turn processed by DMS. Case studies are presented in Section 4, where complete scenarios describing this model-to-model transformation are explained with concrete examples.

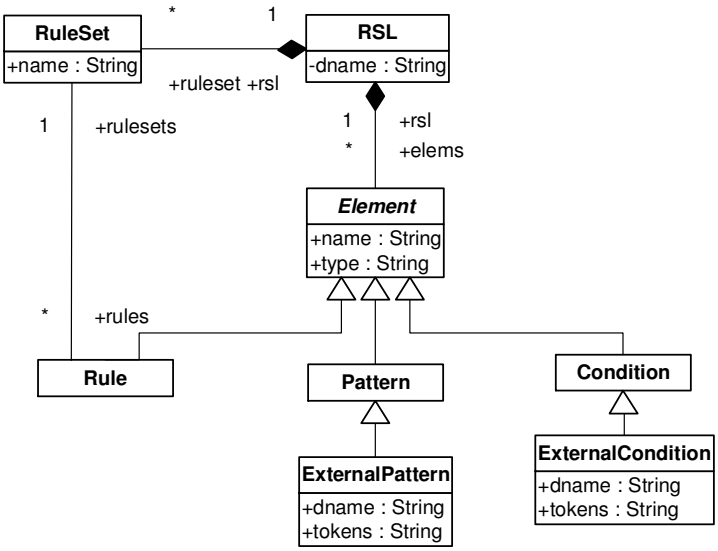


Figure 6: Subset of the RSL metamodel (as a class diagram)

Model transformation using ATL

Given the definition of the source and target metamodels, it is possible to generate RSL program transformation rules from an aspect program using model transformations. Figure 7 explains the complete model transformation scenario in our framework. In this figure, M1, M2, and M3 are the three modeling levels in the grammarware [28] and MDE technical spaces (TS). From the grammarware TS, the front-end aspect source file is initially injected into a source aspect model using TCS. The aspect model is then transformed into a target RSL model using a model transformation defined in ATL. This ATL transformation forms the core of the program transformation rule generation process. After translation, the generated RSL model belonging to the MDE TS is extracted (using TCS) into the target RSL program in the grammarware TS.

To modularize the RSL generation process, the framework defines a library of ATL transformations with each transformation corresponding to a primitive pointcut specification (e.g., `call`, `execution`). For a given aspect, the corresponding ATL transformation rule is automatically invoked depending on the pointcut specification used in the aspect. The higher-order ATL transformation generates the lower-order RSL transformation that eventually performs the aspect weaving. The collective set of all higher-order model transformations is assembled in a transformation library that implements the semantics of the source aspect language.

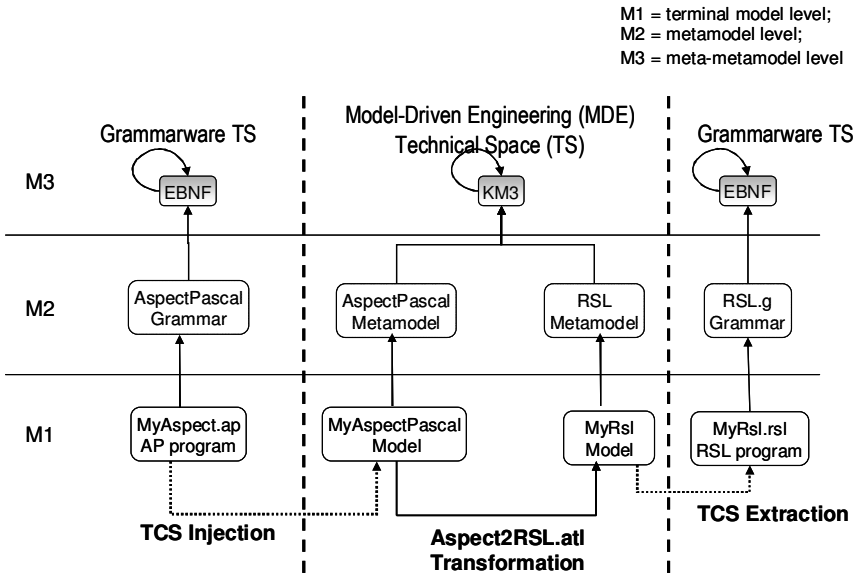


Figure 7: Model transformation scenario for generating RSL rules from aspects

Figure 8 depicts a snippet of a sample ATL transformation from the core transformation library. This particular transformation evaluates a `call` expression in the source aspect, and generates the corresponding RSL transformation rule. The ATL helper function `EvalCallExpr` is used for this purpose. The transformation maps individual elements from the source aspect metamodel to the target RSL metamodel. For example, Aspect Pascal model elements like advice (Line 10, Figure 8) and pointcuts (Line 11, Figure 8) are mapped to RSL elements like patterns, conditions and rules (i.e., RSL elements in Figure 6). Similarly, *before* advice statements (Line 25, Figure 8) from the source aspect language are mapped to RSL patterns. The relationships between the source aspect model elements to the target RSL model elements can be one-to-one, one-to-many, many-to-one or many-to-many. This depends on the type of pointcut expressions used in the source aspect program.

It should be noted that the source metamodel to describe these pointcut expressions (see Section 2.2) is completely independent of the target RSL language. In addition, it is structurally and semantically similar to a traditional AOP language, like AspectJ. This metamodel captures all the essential concepts of AOP (as influenced by AspectJ) - join points, pointcuts and advice. The actual transformation on the source code is performed using RSL rules that are generated from the higher-order aspect language using ATL. These ATL transformations implement the semantics of the source aspect language and all corresponding mapping information from source to target are embedded in the ATL specifications.

```
1. module AspectPascal2RSL;
2. create OUT : RSL from IN : APascal;
3. rule APascal2RSL {
4.     from
5.         s : APascal!APascal
6.     to
7.         t : RSL!RSL (
8.             dname <- 'ObjectPascal',
9.             elems <- Sequence {
10.                         s.advice,
11.                         s.pointcut->collect(e |
12.                            thisModule.EvalCallExpr(e)
13.                         ),
14.                        ...
15.             },
16.             ruleset <- rs
17.         ),
18.       rs : RSL!RuleSet (
19.             name <- s.name,
20.             rules <- s.pointcut->collect(e|e.name)
21.         )
22. }
23. rule BeforeAdvice2Pattern {
24.     from
25.         s : APascal!BeforeAdvice
26.     to
27.         t : RSL!Pattern (
28.             name  <- 'before_advice_stmt_list'
29.             ptype <- 'statement_list',
30.             ptext <-  spt
31.         ),
32.       spt : RSL!SimplePatternText (
33.             ptext <- s.stmts->iterate(...)
34.         )
35. }
-- [original code omitted for brevity]
```

Figure 8: ATL transformation (snippet) from Aspect Pascal to RSL

The generated RSL is not shown here because it is internal to the framework (i.e., users of the framework do not see any of the intermediate transformation rules); however, interested readers who want to view the generated artifacts may refer to the GenAWeave website [30], which represents the project webpage and includes video demonstrations, papers, and all of the source. In addition to the website reference, the case studies presented in Section 4 also serve as specific examples for describing the complete transformation scenario illustrated in this section.

Benefits of using a Model-Driven front-end

There are several advantages of using a model-driven front-end layered on top of a program transformation engine. PTEs typically work at a lower-level of abstraction and are a useful research tool for language researchers. Software developers who are not familiar with PTEs and willing to experiment with AspectJ- like languages will

find it increasingly difficult to express aspects in the form of transformation (rewrite) rules. For example if one looks into the generated transformation rules as presented in [12, 30], it is obvious that the complexity of these rules is orders of magnitude higher than the declarative aspect specification that corresponds to the same rewrite rule. Moreover, the transformation rule forces the user to have detailed knowledge of the underlying grammar production rules and the associated parsing techniques. In other words, these minute details are the core accidental complexity that alienates PTEs from mainstream software development. In this research, one of the primary challenges (*Challenge C3*) has been to make PTEs available for general-purpose software maintenance and development. An MDE based approach helped us to realize this goal. By capturing the key concepts associated with AOP as higher-order models (e.g., as expressed in KM3/TCS) and using model transformation rules (ATL) to generate lower-order program transformation rules (RSL), the technique hides the accidental complexities of PTEs, but still leverages the powerful transformation capabilities required to carry out weaving. Thus, the end-user is oblivious to the low-level program transformation machinery and only works at a conceptual level he or she is familiar with. A model-driven front-end helped us to bridge the gap between PTE and AOP.

Remaining challenges to be addressed in the framework

The model-driven weaver generation framework presented in this section offers a solution to the challenge of using a program transformation engine to implement an aspect weaver. The section provided a discussion of the key parts of the framework, including the front-end aspect language, the transformation rule generator and the back-end weaving engine. The context of the discussion was centered on the creation of a weaver for a single base language, such as Object Pascal, to address *Challenge C3 (accidental complexity)* from Section 1.1. However, an additional challenge remains. As mentioned in the beginning of this section, a program written in RSL or any other term-rewriting engine is typically tied to the grammar of the source program (i.e., the RSL example in Listing 1 has Object Pascal grammar productions appearing throughout the transformation rule). Moreover, there are variations in design from one aspect language to another, even if a common generic part is shared. Unless carefully designed, the front-end, the core transformation libraries, and the back-end modules are rendered unusable when constructing a new weaver in another context (i.e., a new aspect language or a new base programming language). The goal of any extensible framework is to not construct a single fixed solution (i.e., constructing each new weaver from scratch) after enough knowledge, time, and effort have been spent. The next section discusses how this framework was made more generic to support reuse in new contexts. Thus, instead of building a new weaver from scratch, the benefit from the experience gained in a previous construction can be reused and applied toward the construction of a new weaver for a programming language even in a different paradigm.

3 Extending to a Generic Framework

Generalizing the framework presented in Section 2 to accommodate a broad range of GPLs is challenging due to the dissimilarities among various programming languages.

Yet, many languages in the same paradigm (e.g., structured or object-oriented) may share common concepts at an abstract level such that parts of the framework can be reused. Unfortunately, most aspect weavers are built from scratch with little emphasis on reusing the existing knowledge or framework already available for constructing a weaver for a particular GPL. Previous research towards constructing aspect weavers for multiple languages has been based on the following approaches:

- **Common Intermediate Language:** Weave.NET is a load-time weaver that allows aspects and components to be written in a variety of .Net based languages [18]. It takes an existing .Net binary component as input with crosscutting specifications provided in an XML file. The behavior (i.e., implementation-specific advice code) of an aspect is provided separately in another .Net assembly. Weave.NET recreates the input assembly, but in this regenerated version, join points are bound to behavior in the aspect assembly as specified in the XML aspect file. Because all transformations are done at the intermediate language (IL), it serves as a language-independent weaver. In addition to Weave.NET, Loom.Net [41] is another aspect weaving tool that targets the .Net framework. It uses metadata and reflection mechanisms to weave into the .Net assemblies.
- **Generic Source Model:** SourceWeave.Net uses a generic architecture that is built on top of CodeDOM, which is the .NET standard for representing source code models [17]. Using SourceWeave.NET, a developer can write base and aspect components in standard C#, VB.NET and J#. An XML descriptor file is used to specify the interaction between the aspects and representative components. The technique uses a mapping to identify join point shadows and uses a "pointcut-to-join point binding" to isolate parts of the source.

Comparative discussion of AOP frameworks to support multiple GPLs

Each of these representative approaches provides a distinguishing set of strengths and weaknesses. For example, Weave.Net offers a strong solution to *Challenge C1 (parser construction problem)* because of the availability of pre-existing industrial scale parsers within the .Net Framework. SourceWeave.Net is weak on *Challenge C1* due to the limited availability of CodeDOM providers beyond a handful of languages (e.g., mainly C#, J#, and VB.Net). However, both approaches are weak on *Challenge C3 (accidental complexity)* because of the reduced expressiveness (or increased verbosity) of raw XML to specify aspects in each of the frameworks.

The representation of the underlying abstract source model also contributes to several differences affecting the solutions to each challenge. Because of its reliance on CodeDOM, SourceWeave.Net has limitations in terms of expressiveness. C# maps reasonably well to CodeDOM, but that is not true for all GPLs. It remains to be determined if either CodeDOM or .Net CLI are applicable to a large class of legacy languages (e.g., COBOL, FORTRAN, C, or Object Pascal) whose language definitions are very different from the expectations of CLI or CodeDOM. Moreover, a considerable engineering effort would be required if all programming languages were forced to conform to a generic source model or compiled to a common IL. Further,

such an approach would ignore all of the effort that has already been spent in constructing lexers, parsers, analyzers and other tools for these languages.

Another interesting tool that supports the use of language-independent abstractions of the base program is the Compose* compiler [44]. During compilation, the language-independent model of the base program is generated, which is then processed by the compiler for weaving aspects. The model is finally mapped back to the base program. Although the Compose* compiler does not aim to support multiple language front-ends, the same core can be used for multiple base languages.

While investigating a generic aspect weaving framework, we realized these challenges and discovered a solution whereby the model-driven weaver framework uses the existing parsers of DMS, but extracts the commonalities among these languages. Although such a framework does not automate all the tasks involved in creating an aspect weaver (i.e., making it language-independent), our generalized framework can considerably reduce the weaver construction effort by reusing the shared or common parts among different aspect weavers through abstract models and corresponding model transformations (please see Section 4.3 for a discussion of experimental results).

Moreover, because DMS provides support for 23 different programming languages (including legacy languages like COBOL, FORTRAN, and C), a generic front-end with a reusable code generator that translates our front-end aspect language to DMS RSL can make use of all the parsers and analyzers that are already available within each of the language domains supported by DMS. In addition, we may also consider changing the back-end if another PTE supports other languages that we would like to use. The solution approach introduced in this section addresses the obstacles toward weaver construction enumerated in *Challenge C4 (generalization of transformation objectives)*.

3.1 Support for a Generic Front-End

The first step towards a generalized model-driven weaver framework is to design a generic aspect front-end that can be shared among various GPLs. If the AspectJ definition of an aspect is used as a focus point for discussion, every language that is integrated into the framework must define the meaning of a join point model (JPM), pointcuts, and advice within the language context. Such a notion can be defined abstractly such that each new aspect language inherits and extends this common definition.

Reconsidering the Aspect Pascal metamodel of Figure 3, it can be observed that metamodel elements such as pointcut, advice, abstract expressions, and abstract statements are actually generic in the Aspect Pascal metamodel. Thus, instead of modeling these elements as part of the Aspect Pascal metamodel, they can be extracted to a common generic core. However, there may be differences in the concrete syntax of certain model elements. For example, concrete statements and expressions may vary from one GPL to another. In such cases, the differences can be captured in individual metamodel extensions [22] and commonality can be shared using a general metamodel. To explain this concept, this subsection will summarize the construction of aspect weavers for two different GPLs (i.e., Object Pascal and

FORTRAN) using the framework. The example shows how languages across different paradigms can share AOP concepts through metamodel extension.

Figure 9[1] shows the class diagram representing the new Aspect Pascal metamodel that extends the core GAspect (Generic Aspect) metamodel. The latter captures all of the essential concepts that are intrinsic to any aspect-oriented language influenced by the asymmetric AspectJ style. For example, the core model elements such as pointcuts and advice belong to GAspect. There are also abstract placeholders for expressions and statements in GAspect. Although the figure does not show a metamodel for a Join Point Model, a further enhancement in this direction could be made in the future to define different JPMs.

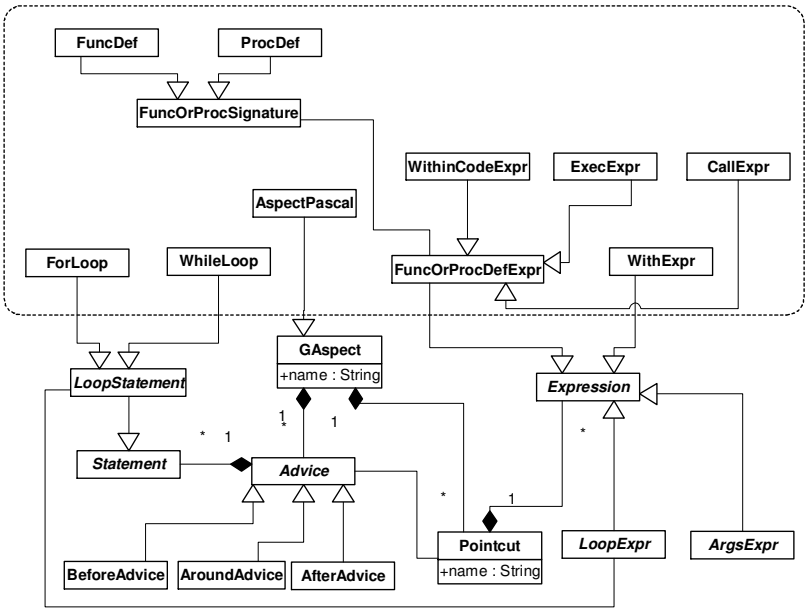


Figure 9: Class Diagram (snippet) of Aspect Pascal (top) extending from a common Generic Aspect metamodel (bottom)

Every language-specific expression and statement must extend from these abstract definitions. For example, a concrete `execution` expression join point or a `call` expression join point for any aspect-oriented language (AOL) must be derived from the abstract `expression` join point of GAspect. In Figure 9, the `call` expression and `exec` expression of Aspect Pascal inherits from `FuncOrProcDefExpr` (which itself is derived from the abstract `Expression` class) and references the

[1] Typically in UML, inheritance is drawn from top to bottom. However, the inheritance relationship for Aspect Pascal is shown in reverse to denote its extension from the generic aspect metamodel and also to accommodate all of the elements in the limited space. However, this does not affect the common understanding of the UML notation used.

`FuncOrProcSignature` type pattern. The type pattern captures the concrete syntax (i.e., signature) for expressing functions or procedures in Object Pascal and is dependent on the grammar of the base language. For every new language, the concrete syntax of type pattern varies. The dotted rectangle in Figure 9 depicts all those points of variability that are specific to Aspect Pascal.

Because most programming languages have some form of support for loops, we have introduced the notion of a loop execution join point in the generic metamodel. Concrete loop statements belonging to the base AOL must be derived from the abstract `LoopStatement` of GAspect. The Aspect Pascal metamodel shows support for `while` loop and `for` loop join points that are extended from the abstract loop execution join point present in GAspect. The concept of a loop execution join point is not present in AspectJ, but has been found to be useful for monitoring high-performance scientific applications [25].

Furthermore, a join point for capturing `with` expressions in Object Pascal is introduced in the Aspect Pascal metamodel. An example of a crosscutting concern based on a `with` expression join point is given in [12]. In a similar way, the entire join point model for an AOL can be constructed by adding concrete extensions from the abstract GAspect metamodel. Moreover, the technique allows experimentation with new features (e.g., `loop` execution join point) to be added to an existing AOL. Such an addition is beneficial if the aspect language evolves. The Aspect Pascal metamodel shown here is only a snippet of the original. The complete KM3 and TCS specifications are available at [30].

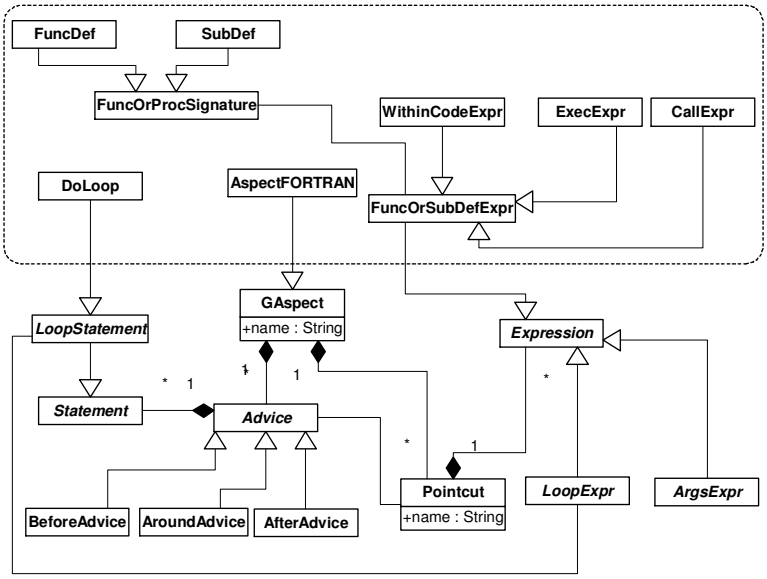


Figure 10: Metamodel (snippet) of Aspect FORTRAN (top) conforming to a common Generic Aspect metamodel (bottom)

Figure 10 shows the corresponding metamodel for Aspect FORTRAN. Similar to Aspect Pascal, the Aspect FORTRAN metamodel is extended from the generic core

GAspect. However, the points of variability (shown by the enclosing dotted rectangle) for this metamodel exist in their concrete syntax. In the case of Aspect FORTRAN, the `call`, `exec` and `withincode` expressions reference subroutine/function definitions unlike the procedure/function definitions in the Aspect Pascal metamodel. Moreover, the concrete function definitions for Aspect FORTRAN and Aspect Pascal are different due to the dissimilarity in their underlying grammar. The TCS specification in Figure 11 shows this variability of concrete syntax for the two metamodels.

```
1. template FuncDef
2. :  "FUNCTION" name "(" paramdefs{separator = ","} ")"
3. ;
4. template SubDef
5. :  "SUBROUTINE" name "(" paramdefs{separator = ","} ")"
6. ;
```

```
1. template FuncDef
2. :  "function" (isDefined(classifier) ? classifier ".")
3.     name "(" paramdefs{separator = ";"} ")"
4. ;
5. template ProcDef
6. :  "procedure" (isDefined(classifier) ? classifier ".")
7.     name "(" paramdefs{separator = ";"} ")"
8. ;
```

Figure 11: TCS specification showing differences in concrete syntax for Aspect FORTRAN (top) and Aspect Pascal metamodel (bottom)

Figure 11 shows the differences in the concrete syntax of function/subroutine/procedure declarations for Aspect FORTRAN and Aspect Pascal. The upper half represents the TCS specification of Aspect FORTRAN function/subroutine declaration whereas the bottom half shows the corresponding function/procedure declaration for Aspect Pascal. All points of variation between the two metamodels are captured in their corresponding extended metamodel (dotted rectangle) whereas the commonality is captured in the generic aspect metamodel. Generally, the point of variation between two aspect languages will be in their formal syntactic representation.

In addition, GAspect also captures certain program fragments belonging to a GPL that may not be analyzed or parsed by the front-end. Instead, these program fragments are delegated to the back-end PTE for parsing and analysis. Such fragments typically appear in the body of advice code and are referenced as `OpaqueStatement`s. This considerably reduces the complexity of the aspect metamodel as several language constructs of the base language need not be parsed or analyzed by the front-end. Instead, the back-end PTE that already has the capability (parser/analyzer) to process the base language (Object Pascal / FORTRAN) can handle such program fragments. An example of using `OpaqueStatement` is shown in the case study of Section 4.

The construction of a generic aspect metamodel helps to generalize the commonalities among distinct aspect languages. Each common concept may be refined using language-specific metamodel extensions. Furthermore, an extension of GAspect may categorize commonalities within a paradigm that can be reused (e.g., a

metamodel named Object-Oriented that extends GAspect with common OO concepts, which is then extended by concrete OO languages). This was one of the important lessons learned in using MDE during the course of this research and can significantly improve the genericity of the metamodel. The issue of additional specialization of the GAspect metamodel is further discussed in Section 6.1 as a future enhancement.

3.2 Generalizing the Rule Generator Design

The goal of the program transformation rule generator (item 2 in Figure 1) is to translate a given aspect to a corresponding program transformation rule (e.g., RSL). This role is handled by an assembly of transformation libraries written in ATL. In the context of a generic framework, it is desirable to reuse as much of the transformation library code as possible when constructing an aspect weaver for a new GPL. To realize this objective, the transformation libraries must follow a general guideline (similar to a generic API) that ensures maximum reusability.

The guideline ensures that every transformation rule that captures the semantics of a particular weaving intent must conform to a generic rule interface. For example, an RSL rule that captures the semantics of a method invocation join point (i.e., to capture a particular method call and trigger advice) should conform to a generic method invocation rule interface that the back-end transformation engine expects. Generally, the back-end rewrite rules are parameterized and expect a set of parameters that would be used during the transformation. Thus, it is important to normalize the parameter structure to a common interface specification. These parameters are determined by the semantics of the joinpoint that is to be encoded in a program transformation rule. By conforming to this generic interface, model transformation libraries written for various GPLs may share a generalized common pattern. For example, to represent a method call join point, the back-end program transformation rule should conform to a generic rule interface called `generic_advice_call`, which accepts the following five named-parameters: `program_root_`, `method_id_`, `proceed_call_`, `before_advice_` and `after_advice_`. The first two named parameters refer to the abstract syntax of the program under consideration, `program_root_` captures the root of the tree and `method_id_` captures the current method being advised. The rest of the named parameters capture the semantics of this join point. Every named parameter has a type associated with it, which is determined by the concrete syntax (grammar) of the base GPL. For example, for a FORTRAN 90 program, this generic function should be encoded as follows:

```
{name → type}
generic_advice_call  (
{ program_root_  → Fortran90_program },
{ method_id_     → Name},
{ proceed_       → Name},
{ before_advice_ → execution_part_construct_list },
{ after_advice_  → execution_part_construct_list },
) → Fortran90_program
```

One may note that although the types (shown in *italics*) are concrete, the rule interface is abstract. This generalization is necessary to address *Challenge C4* (*language-independent generalization of transformation objectives*) and facilitate the ATL rule generator to program to a common rule interface that can be reused among various GPLs. At this point, one may recollect from Listing 1 how RSLs or any term-rewrite rules are tied to the concrete syntax of the base programming language. The `proceed_` is internally used to determine if the advice is an around advice that makes a call to proceed. Similarly, for an Object Pascal program, the `generic_advice_call` is encoded as follows:

```
{name → type}
generic_advice_call  (
{ program_root_  → ObjectPascal },
{ method_id_     → IDENTIFIER},
{ proceed_       → IDENTIFIER},
{ before_advice_ → statement_list },
{ after_advice_  → statement_list },
) → ObjectPascal
```

For every join point in our AOP language model, we have developed a set of formal rule interfaces to which each corresponding ATL transformation must conform (i.e., there is a separate rule interface for method execution or loop execution join point). The generic rule interfaces not only enforce the code generators for different aspect weavers to adhere to a known abstract interface, but also considerably reduce the development time and effort to transfer knowledge from one rule generator to another.

Figure 12 and Figure 13 show comparative snippets of the higher-order model transformation rules (ATL specification) for translating a method call join point written in Aspect Pascal or Aspect FORTRAN to a corresponding lower-order program transformation rule (RSL rewrite specification). Each of the ATL specifications (Figures 12 and 13) consist of several smaller ATL rules that together perform the actual transformation on the metamodel. For example, the rules `AfterAdvice2Pattern`, `BeforeAdvice2Pattern` and `PointCutToExternalPattern` (as shown in Figures 12 and 13) are used to construct the ATL specification for translating a *method call join point*. However, this is only a subset; the complete ATL specification is available at [30]. The individual rules (e.g., `AfterAdvice2Pattern`, `BeforeAdvice2Pattern`) are fired whenever a corresponding model element (e.g., model elements like `BeforeAdvice`, `AfterAdvice` in the Aspect Pascal metamodel) in the source metamodel is reached.

Both of these higher-order transformation rules conform to an abstract structure (generic rule interface) that drives the ATL rule generator. As a direct benefit of forcing the ATL transformations to conform to a common structure or interface, the model transformation rules presented in Figures 12 and 13 appear distinctly similar. For example, all of the three rules (i.e., `AfterAdvice2Pattern`, `BeforeAdvice2Pattern` and `PointCutToExternalPattern`) have the same *left-hand-side (LHS)*, such that the main difference lies in their concrete syntax (i.e., the grammar of the two languages).

```
rule BeforeAdvice2Pattern {
      from
              s : APascal!BeforeAdvice
      to
              t : RSL!Pattern (
                      phead  <- ph,
                      ptoken <- 'statement_list',
                      ptext  <- spt
              ),
              ph : RSL!PatternHead (
                      name <- 'before_advice_stmt'
              ),
               ...
              )
}
rule AfterAdvice2Pattern {
      from
              s : APascal!AfterAdvice
      to
              t : RSL!Pattern (
                      phead  <- ph,
                      ptoken <- 'statement_list',
                      ptext  <- spt
              ),
              ph : RSL!PatternHead (
                      name <- 'after_advice_stmt'
              ),
             ...
}
lazy rule PointCutToExternalPattern {
      from
              s : APascal!Pointcut
      to
              t : RSL!ExternalPattern (
                      dname  <- 'ObjectPascal',
                      eptext <- 'around_advice_call',
                      ptoken <- 'ObjectPascal',
                      phead  <-  ph
              ),
              ph : RSL!PatternHead (
                      name   <- 'around_advice_call',
                      params <-  Sequence {pp1,pp2,pp3,pp4,pp5,pp6}
              ),
              pp1 : RSL!PatternParameter (
                      name    <- 'program',
                      referTo <- 'ObjectPascal'
              ),
              pp2 : RSL!PatternParameter (
                      name    <- 'method_name',
                      referTo <- 'IDENTIFIER'
              ),
              pp3 : RSL!PatternParameter (
                      name    <- 'proceed_call',
                      referTo <- 'IDENTIFIER'
              ),
               ...
}
```

Figure 12: ATL specification (snippet) used to generate lower-order transformation rules (RSL) for weaving an Object Pascal source program

```
rule BeforeAdvice2Pattern {
      from
              s : AFortran!BeforeAdvice
      to
              t : RSL!Pattern (
                      phead  <- ph,
                      ptoken <- 'execution_part_construct_list',
                      ptext  <- spt
              ),
              ph : RSL!PatternHead (
                      name <- 'before_advice_stmt'
              ),
               ...
}

rule AfterAdvice2Pattern {
      from
              s : AFortran!AfterAdvice
      to
              t : RSL!Pattern (
                      phead  <- ph,
                      ptoken <- 'execution_part_construct_list',
                      ptext  <- spt
              ),
              ph : RSL!PatternHead (
                      name <- 'after_advice_stmt'
              ),
               ...
}

lazy rule PointCutToExternalPattern {
      from
              s : AFortran!Pointcut
      to
              t : RSL!ExternalPattern (
                      dname  <- 'FORTRAN',
                      eptext <- 'around_advice_call',
                      ptoken <- 'Fortran90_program',
                      phead  <- ph
              ),
              ph : RSL!PatternHead (
                      name   <-   'around_advice_call',
                      params <- Sequence {pp1,pp2,pp3,pp4,pp5,pp6}
              ),
              pp1 : RSL!PatternParameter (
                      name    <- 'program',
                      referTo <- 'Fortran90_program'
              ),
              pp2 : RSL!PatternParameter (
                      name    <- 'method_name',
                      referTo <- 'NAME'
              ),
              pp3 : RSL!PatternParameter (
                      name    <- 'proceed_call',
                      referTo <- 'NAME'
              ),
              ...
}
```

Figure 13: ATL specification (snippet) used to generate lower-order transformation rules (RSL) for weaving a FORTRAN source program

For example, in the ATL rule `BeforeAdvice2Pattern`, the *before advice* in the source aspect metamodel is mapped to a *RSL pattern* in the target RSL metamodel that consists of a pattern head (`phead`), a pattern token (`ptoken`) and the pattern text (`ptext`). Similarly, a *RSL external pattern* is translated from a source *pointcut* specification and has the same LHS signature (`dname`, `eptext`, `ptoken`, `phead`) for both Object Pascal and FORTRAN generators. The main difference lies in the concrete syntax (*right-hand side)* of the base language grammar as referred in the transformation rules (e.g., an `execution_part_construct_list` in FORTRAN is similar to a `statement_list` in Object Pascal). Obviously, there are other non-terminal and terminal tokens in both the Object Pascal and FORTRAN grammar that have similar structural representation and meaning, but differ by name in their grammar form. The strategy is always to follow a common abstract structure (or substructure) to translate a particular join point from an aspect description to RSL. However, in certain cases, where the difference in signature or concrete syntax between two language grammars differs significantly, it may not be directly possible to map to a generic interface. Instead, the mapping can then conform to sub-structures or sub-interfaces.

Steps to Construct a New Weaver

Using the current methodology, in order to construct a new weaver for a language that is not supported by the framework, the following steps have to be performed:

- The aspect metamodel extension for the new language must be designed. Firstly, the new weaver should extend the language-specific aspect metamodel for the new language by inheriting from the generic aspect metamodel provided by the framework. Figures 9 and 10 show how this extension is done for Object Pascal and FORTRAN. This metamodel represents the abstract syntax of the newly constructed aspect language and is implemented using KM3, as shown in Figure 4. However, in some situations, representing common concepts between aspect languages may require the extension of the generic aspect metamodel.
- After the abstract syntax is defined, the corresponding concrete syntax that defines the syntactic structure of the aspect language should be designed. In our case, this was done using TCS, as shown in Figure 5. The choice of the metamodeling technology may slightly alter these first two steps.
- Apart from the generic metamodel, the framework provides a library of model transformation rules (i.e., ATL rules) that need to be customized based on the design of the new aspect language. Figures 12 and 13 show snippets of ATL transformation rules that are customized for Object Pascal and FORTRAN. In general, the model transformation rules differ in their concrete syntactical representation from one language to another and this difference is shown in Figure 17. In our future work, we plan to automate this customization by capturing the differences using a model mapper, thereby reducing the accidental complexity associated in this step.
- No changes are required for the back-end rules metamodel. After the model transformation rules are defined, the corresponding program transformation rules can be generated, as shown in Figure 7. The generated program

transformation rules perform the actual weaving on the source code. The weaver constructor can automate this entire process by providing appropriate Ant scripts [30] such that the end-user is fully oblivious to the two-level transformations (i.e., first model transformation and then program transformation) taking place. This virtually completes the tasks for constructing a new weaver.

- The end-user only has to write their aspect program as shown in Listing 3 or Listing 5; they are shielded from the internal complexities. The opaque statements appearing in the aspect program thrust some accidental complexity on the end-user, which is a limitation of the current implementation, but is intended to reduce the burden of the front-end parser.

As an alternative approach to model-to-model (M2M) transformation followed by TCS extraction, an interesting technique that can be used is model-to-text (M2T) transformation [37]. In the M2T approach, models of particular software solutions are refined and transformed into source code (e.g., Java, C++). Such transformations generally make use of "templates." A template is a text sequence interspersed with commands that extract information from a model. The Jet or Acceleo template languages can be used for such a purpose [37]. We recognize that this is an interesting solution and could serve as an alternative approach towards constructing the RSL rule generator. However, using M2T, we may lose the precise concept mapping between the source and the target model, and rely on mapping concepts to strings. Nevertheless, many alternative approaches can still benefit from the technique described in this paper.

In the following section, we present two case studies that use our model-driven aspect weaving framework to construct aspect weavers for two different GPLs. In particular, we construct aspect weavers for Object Pascal and FORTRAN and make comparative studies of reuse of their front-end, the rule generator and the back-end. The observations made in the case studies help to illustrate the potential of the techniques presented so far. It also reveals some of the limitations of the current implementation of the framework and offers lessons learned during the process that can be applied for future improvements.

4 Case Studies – Object Pascal and FORTRAN

In order to experiment with the approach presented in the previous sections, we constructed two aspect weavers – one for Object Pascal and another for FORTRAN – using our generic model-driven framework. A subset (e.g., primitive pointcuts like call, execution, loop) of standard AOP features was built into both weavers in an AspectJ-like style. The FORTRAN weaver was constructed after the completion of the Aspect Pascal weaver and reused several functionalities, code and knowledge from the previous construction without much alteration to the core artifacts. For example, both weavers shared the generic front-end, which constituted around 50% of the overall front-end lines of code (LOC) written in KM3 and TCS. Moreover, the FORTRAN weaver reused 30% of the Object Pascal rule generator code without any alteration, and another 25% with minor customization. Most of the time and effort on building the FORTRAN weaver was spent on understanding the concrete syntax of the language and on the conceptual design of the weaver. The rest of the section is

devoted to evaluating the basic functionalities of these weavers through sample case study applications.

4.1 Object Pascal Weaver Examples

The initial experimentation towards evaluating our Aspect Pascal weaver was realized within the scope of a commercial distributed application written in Object Pascal. The case study application and all the examples discussed here were first introduced in [12]. One specific application used for evaluation was a utility that assisted in upgrading a database after a schema change. The first example presented in this section is concerned with updating a processing dialog meter within the schema evolution tool. The second example relates to synchronization between various database error-handlers.

4.1.1 Processing Dialog Meter

Utilities such as a schema evolution tool provide feedback to the user in the form of a processing dialog, or meter, which indicates the progress of the overall task. The updating of the progress meter represents a crosscutting concern because the code to increment the meter is spread across the methods that perform much of the functionality (e.g., deleting database triggers, compiling new stored procedures, and other evolution tasks). Lines 2-8 in Listing 2 contain a redundant code fragment that appears in 62 different places of the schema evolution utility. This code is necessary to update the processing dialog after each database evolution task is completed. Technically, this happens after every call to the predefined `Inc` procedure.

```
1.  Inc(TotalInsertionsPerformed);
2.  if not ProcDlg1.Process(TotalInsertionsPerformed /
3.           TotalInsertionsCalculated) then
4.    begin
5.      ProcDlg1.Canceled := True;
6.      Result := True;
7.    exit;
8.  end; // if not Process
```

Listing 2: Progress meter updating

Listing 3 shows the `UpdateProgressMeter` aspect that encapsulates the crosscutting concern shown in Listing 2. The pointcut `IncrCall_` captures all calls to procedure `Inc`. The advice code shown between Lines 5-13 is triggered after the "procedure call join point" is reached. It may be noted that the entire conditional 'if statement' (defined internally as an `OpaqueStatement` between `<!` and `!>`) is not parsed by the front-end but delegated to the back-end parser.

```
1.  aspect UpdateProgressMeter
2.    begin
3.      pointcut IncrCall_() :
              call(procedure  *.Inc(Integer));
4.
5.     after() : IncrCall_()
6.     begin
7.       <!if not ProcDlg1.Process(TotalInsertionsPerformed /
8.              TotalInsertionsCalculated) then
9.        begin
10.         ProcDlg1.Canceled := True;
11.         Result := True;
12.       exit;
13.    end;!>
13.     end
14. end
```

Listing 3: Aspect to capture progress meter updating

Following TCS injection on the above source program (Listing 3), the corresponding Aspect Pascal model is generated (shown in Figure 14). The model (represented in XMI format) conforms to the APascal and GAspect metamodels introduced in Section 3 (this representation is never seen by the end-user). After applying an ATL transformation (*method call*) to this Aspect Pascal model, the resulting RSL model is generated that conforms to the target RSL metamodel. Finally, the lower-order RSL transformation rule is extracted from the RSL model using TCS extraction. The complete RSL model and the RSL transformation rule are available at [30]. Note that the complete transformation scenario was initially introduced in Section 2.3 of the paper (also refer to Figure 7) and is fully automated using Ant scripts.

```
<APascal xmlns="APascal" xmlns:_1="GAspect" name="UpdateProgressMeter">
  <domain name="ObjectPascal"/>
  <pointcut name=" IncrCall_">
    <pctexpr xsi:type="CallExpr">
      <funcOrProcSig xsi:type="ProcedureDef" name="Inc" classifier="*">
        <paramdefs name="*" type="Integer"/>
      </funcOrProcSig>
    </pctexpr>
  </pointcut>
  <advice xsi:type="_1:BeforeAdvice" pctname="//@pointcut.0">
    <advStmt xsi:type="_1:OpaqueStatement" stmt="..."/>
  </advice>
</APascal>
```

Figure 14: Aspect Pascal model (snippet) generated from Aspect Pascal source program

The next example in our case study shows how a synchronization aspect is captured using the Aspect Pascal weaver constructed from our model-driven framework.

4.1.2 Database Error Handler Synchronization

Often, a commercial application must support databases from several different vendors (e.g., Oracle, Interbase, and SQL Server). In such a situation, exception handling of database errors is a major difficulty because each database has its own way of raising exceptions. The same conceptual error (e.g., a null value in a required field) may be raised in completely different ways with dissimilar error codes. Moreover, the exception handling code must be thread-safe because numerous clients may access the database at the same time. The addition of this concurrency concern resulted in a manual invasive change to over 20 classes in the schema evolution utility [12]. An example error handler is shown in Listing 4. In this listing, lines 3-4 and 6-8 represent a single synchronization concern. Furthermore, this exact code is replicated in all of the entry and exit points of each type of error handler. Line 5 represents the actual database error handing code which is omitted for brevity.

```
1. function TExNullField.Handle(ServerType: TServerType;
                                E : EDBEngineError) : Integer;
2.  begin
3.     TExHandleColl(Collection).LockHandle;
4.     try
5.       <database error handling code omitted here>
6.       finally
7.         TExHandleColl(Collection).UnLockHandle;
8.     end;
9.  end;
```

Listing 4: Synchronization in a database error handler

Listing 5 shows the aspect to support a synchronization concern as stated above. The pointcut `funcHandler_` captures execution of all database handler functions.

```
1. aspect SyncDBErrHandler
2.   begin
3.    pointcut funcHandler_() :
                execution(function *.Handle(..));
4.    void around() : funcHandler_()
5.      begin
6.       <!TExHandleColl(Collection).LockHandle;!>
7.       try
8.          proceed ();
9.          finally
10.           <!TExHandleColl(Collection).UnLockHandle;!>
11.      end;
12.     end
13.   end
```

Listing 5. Aspect to capture synchronization in a database error handler

Synchronization is realized by an around advice that wraps calls to the `LockHandle` and `UnlockHandle` methods inside a `try/finally` block. The `proceed` statement allows the database error handling code to execute normally within the

synchronization aspect. We applied the same steps as in the previous example to separate this concern from the main code base. The example shows another special case of using opaque statements that are not part of the aspect metamodel. Such statements are not parsed by the front-end and instead delegated to the back-end transformation engine for further processing. This may add some accidental complexity for the end-user who needs to have prior knowledge about which concrete syntax are supported by the metamodel and which are actually delegated as opaque statements. This is a limitation in the implementation of the current approach but intended to keep the syntax of the aspect language separated from the syntax of the base language. The idea is to parse the base language syntax using the already available back-end parser (i.e., without having to extend them with new constructs) and implement the front-end parser incrementally to handle aspect-specific constructs. We recognize that the use of notation <!..!> raises the accidental complexity for the end-user and is not a desirable solution. In future implementation, we intend to improve the representation of opaque statements such that it is oblivious to the end-user.

The Aspect Pascal model shown in Figure 15 is obtained by applying TCS injection on the above source program. The complete ATL transformation (*method exec*) and generated RSL code is available at [30]. It should be noted that it is this lower-order RSL code that does the actual weaving on the base program, but the general user of this framework is oblivious to its presence. Instead, the front-end aspect language acts as a façade to the back-end PTE and hides all the accidental complexity associated with it (*Challenge C3*).

The XMI (Figure 14 and Figure 15) is only an internal representation of the Aspect Pascal model and is used for analyzing and transforming the aspect specification. A software developer does not see this internal representation. However, the information is useful for more advanced users (e.g., weaver constructor) for debugging and analysis purposes.

```
<APascal xmlns="APascal" xmlns:_1="GAspect" name="SyncDBErrHandler">
  <domain name="ObjectPascal"/>
  <pointcut name="funcHandler_">
    <pctexpr xsi:type="ExecExpr">
      <funcOrProcSig xsi:type="FunctionDef" name="Handle" classifier="*">
        <paramdefs name="*" type="*"/>
      </funcOrProcSig>
    </pctexpr>
  </pointcut>
  <advice xsi:type="_1:AroundAdvice" pctname="//@pointcut.0">
    <advStmt xsi:type="_1:OpaqueStatement" stmt="..."/>
    <advStmt xsi:type="_1:TryCatchFinallyStatement">
      <stmts xsi:type="_1:ProccedStatement"/>
      <finallyStmts xsi:type="_1:OpaqueStatement" stmt="..."/>
    </advStmt>
  </advice>
</APascal>
```

Figure 15: Aspect Pascal model (snippet) generated from Aspect Pascal source program

4.2 FORTRAN Weaver Examples

Although most AOP research is centered around Java, we believe several numerical and scientific computing applications that are written in legacy languages like FORTRAN can benefit from AOP. There has been prior AOP / metaprogramming research conducted in the area of parallel programming [45, 47], especially with optimization of FORTRAN code [46]. To evaluate our framework in this regard, we constructed a FORTRAN weaver and was able to reuse a majority of the code generator libraries that were previously written for Object Pascal. The front-end of the FORTRAN weaver is based on the same Generic Aspect Metamodel that was used by the Object Pascal weaver. We evaluated our weaver in a FORTRAN application using the Message Passing Interface (MPI) [29] written for high-performance scientific computing. The first example shows how a security concern can be weaved into such applications and the second example illustrates how to monitor and weave an aspect around loops.

4.2.1 Security Aspect

MPI is a library specification for message-passing and is largely used in high-performance scientific computing applications [29]. MPI provides more than 125 core functions that include all the basic functionalities to assist in writing parallel programs. There are several implementations of MPI written in various languages (e.g., C, FORTRAN, C++ and Java). In order to provide security to FORTRAN-based MPI applications, it is often required to encrypt/decrypt messages while they are sent or received across the network. Listing 6 shows a snippet of a FORTRAN MPI program, in which lines 9 and 12 illustrate how a security concern (i.e., a call to the `encrypt` function) is added before each call to `MPI_SEND`. The implementation of the security concern is scattered over the entire code base for all messages that require encryption during `MPI_SEND`.

Listing 7 shows the aspect program required to enable security for all messages during MPI message send and receive. The pointcut captures all calls to `MPI_SEND` and passes the message to be encrypted as an argument. In a similar way, security to messages may be enabled during calls to `MPI_RECV`. The internal representation of the generated artifacts (e.g., Aspect FORTRAN model, RSL model and RSL transformation rule) is not shown here but the transformation process is similar to previous descriptions.

```
1.    program send_recv_with_MPI
2.    ...! original code
3.    real :: a_msg
4.    real :: b_msg
5.    ...! original code
6.    allocate (a_msg(msg_len))
7.    allocate (b_msg(msg_len))
8.    ...
9.    call encrypt(a_msg)
10.   call MPI_SEND(a_msg,...)
11.   ...
12.   call encrypt(b_msg)
13.   call MPI_SEND(b_msg,...)
14.   ...
15.   deallocate (a_msg)
16.   deallocate (b_msg)
17.   ...! original code
18.   End
```

Listing 6. Encryption of messages during MPI_SEND

```
1.  aspect enable_encryption
2.    pointcut mpi_send_(real :: orig_msg) ::
3.      call(MPI_SEND(real,*)) && args(orig_msg)
4.    before(real :: orig_msg):: mpi_send_(orig_msg)
5.      call encrypt(orig_msg)
6.    endbefore
7.  endaspect
```

Listing 7. Aspect to enable encryption during MPI Calls

4.2.2 Join Point for Loops

It is often desired to monitor the performance of loops for some high-performance scientific applications. Harbulot et. al. first introduced this concept in an extension to AspectJ [25]. We borrowed from their definition and added this feature into our FORTRAN and Object Pascal weavers. According to our definition, the join point for a loop has the following signature:

```
<loop_name>(init::<val>, exit::<val>, stride::<val>)
```

`Init` specifies the loop initialization value, `exit` specifies the loop termination value and `stride` specifies the loop increment counter. Listing 8 shows an implementation of `MPI_GATHER` written in FORTRAN.

```
1.    program gather_vector
2.    ... ! original code
3.    parameter(niters=10)
4.    parameter(xmax=100,ymax=100)
5.    parameter (totelem=xmax*ymax)
6.    ...
7.    ! start timer
8.    time_begin = MPI_Wtime()
9.    do iter = 1,niters
10.     ...
11.       do i=1,totelem
12.          ...
13.       enddo
14.   enddo
15.   ! stop timer
16.   time_end = MPI_Wtime()
17.   ... ! original code
18.   end
```

Listing 8: Adding a timer around do loops

```
aspect AddTimerAroundLoops
  pointcut loop_timer_() ::
    execution(do(init::1,exit::10,stride:*))
  around():: loop_timer_()
    time_begin = MPI_Wtime()
    call proceed()
    time_end = MPI_Wtime()
  endaround
endaspect
```

Listing 9: Aspect to add a timer around do Loops

In MPI, messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather). Often, it is required to measure timing statistics around critical parts of program execution. One such case is shown in Listing 8. Lines 9-14 show the execution of the outer `do` loop, which has initial value as 1, exit value as 10 and a default stride as 1. In a manual approach, it is required to invasively add the timer information (Lines 8 and 16 shown in *italics*) and change the source program at every place whenever the program runs into the execution of a loop join point that matches the loop conditions.

Listing 9 shows the aspect program that can automatically add the timing functions during the execution of the loop join point. The join point for loops matches any loop expression in the base program that satisfies the loop initialization value, the loop finalization value (exit) and the loop stride value. The wildcard '*' may be interpreted as 'any.' Currently, both integer and string value types are supported, but future extensions can support other value types. However, as a side effect, the behavior of a base program may be altered if there are logical errors (`init=1`, `exit=1`, `stride=2`) in the loop expression and there is a corresponding match. Such a situation may be avoided in the future by adding semantic validations to the existing pattern matching functionality.

```
<AFortran xmlns="AFortran" xmlns:_1="GAspect" name="AddTimerAroundLoops">
 <domain name="FORTRAN"/>
 <pointcut name="loop_timer_">
  <pctexpr xsi:type="_1:LoopExpr">
   <loopStmt xsi:type="DoLoop">
      <loopInitCond xsi:type="1:IntLoopInitCond" condition="1"/>
      <loopExitCond xsi:type="1:IntLoopExitCond" condition="10"/>
      <loopStrideCond xsi:type="1:StringLoopStrideCond" condition="*"/>
   </loopStmt>
  </pctexpr>
 </pointcut>
 <advice xsi:type="_1:AroundAdvice" pctname="//@pointcut.0">
  <advStmt xsi:type="_1:OpaqueStatement" stmt="time_begin = MPI_Wtime()"/>
  <advStmt xsi:type="_1:ProceedStatement">
  <advStmt xsi:type="_1:OpaqueStatement" stmt="time_end = MPI_Wtime()"/>
  </advice>
</AFortran>
```

Figure 16: Aspect FORTRAN model generated from source aspect program

The Aspect FORTRAN model (XML format) corresponding to the aspect specification (Listing 9) is shown in Figure 15. The model conforms to the Aspect FORTRAN metamodel from Figure 10. The corresponding ATL transformation for loops and the generated RSL code can be found at the project website [30].

4.3 Discussion of Experimental Results

In terms of reusability, all the examples listed in Section 4 reuse the same generic aspect metamodel (GAspect). Moreover, the ATL transformation for translating a particular join point reveals non-trivial reuse among weavers constructed for different GPLs. This was illustrated in Section 3.2 through Figures 12 and 13 (i.e., an ATL transformation for translating a `method call join point` in FORTRAN and Object Pascal). In that particular example, 230 lines of model transformation code (out of 280 LOC) were reused without any modification. The remaining 50 LOC were reused with minor customization.

Similarly, for translating a `loop execution join point` in FORTRAN and Object Pascal, 265 LOC out of 305 were reused without any modification, while the remaining 40 LOC were reused with minor customization. Examples of an ATL rule for translating a `loop execution join point` for Object Pascal is shown in [30].

A visual comparison between the ATL rules (`loop execution join point`) for Object Pascal and FORTRAN weavers is shown in Figure 17[2], which suggests the level of reuse among the two ATL rules. This level of reuse is a direct benefit of using the framework, which enforces the model transformation rules to conform to a common abstract structure. The yellow lines show the difference between two rules, which is mostly due to the dissimilarity in the grammars (terminal and non-terminal symbols) of the two languages (Object Pascal and FORTRAN). Although these parts seem to be tangled in the current implementation (weak copy/paste reusability), a future improvement (strong reuse) would capture the

[2] In the right-hand side of Figure 17, there is an extra carriage return on line 220 that caused the gray line to appear in the left-hand side of the figure.

mapping information (i.e., the concrete syntax or grammars of the base languages) in a model weaver [39] and apply the mapping information to automatically generate part of this model transformation library. From our own experience in constructing aspect weavers using the generalized framework, we realized that a large part of the generic front-end and program transformation rule generator is reusable across languages with little customization. A comparative analysis between other ATL rules for the Aspect Pascal and Aspect FORTRAN weaver is available at the GenAWeave website [30].

```
199         pp1 : RSL!PatternParameter (
200             name <- 'program',
201             referTo <- 'ObjectPascal'
202         ),
203         pp2 : RSL!PatternParameter (
204             name <- 'proceed_bef',
205             referTo <- 'statement_list'
206         ),
207         pp3 : RSL!PatternParameter (
208             name <- 'proceed_after',
209             referTo <- 'statement_list'
210         ),
211         pp4 : RSL!PatternParameter (
212             name <- 'withincode',
213             referTo <- 'IDENTIFIER'
214         ),
215         pp5 : RSL!PatternParameter (
216             name <- 'init',
217             referTo <- 'NATURAL_NUMBER'
218         ),
219         pp6 : RSL!PatternParameter (
220             name <- 'exit',
221             referTo <- 'NATURAL_NUMBER'
222         )
223 }
```

```
199         pp1 : RSL!PatternParameter (
200             name <- 'program',
201             referTo <- 'Fortran90_program'
202         ),
203         pp2 : RSL!PatternParameter (
204             name <- 'proceed_bef',
205             referTo <- 'execution_part_construct_list'
206         ),
207         pp3 : RSL!PatternParameter (
208             name <- 'proceed_after',
209             referTo <- 'execution_part_construct_list'
210         ),
211         pp4 : RSL!PatternParameter (
212             name <- 'withincode',
213             referTo <- 'NAME'
214         ),
215         pp5 : RSL!PatternParameter (
216             name <- 'init',
217             referTo <- 'LABEL'
218         ),
219         pp6 : RSL!PatternParameter (
220
221             name <- 'exit',
222             referTo <- 'LABEL'
223
224         )
225 }
```

Figure 17: A comparative analysis of model transformation rules

Likewise, the front-end of all weavers share a generic metamodel (i.e., GAspect). Out of 550 LOC used for defining the front-end metamodel (KM3 and TCS specifications), nearly 280 LOC were shared among the two weavers. However, it should be noted that the current weavers have limited functionalities and the reuse may decrease with mutually exclusive functionalities (e.g., *with* join point is present only in Object Pascal and not in FORTRAN). Nevertheless, the purpose of the Aspect Pascal and Aspect FORTRAN weavers were to experimentally evaluate the generality of the model-driven framework for building aspect weavers. The main objective was to evaluate the reusability of features that can be shared among multiple weavers without writing them from scratch. In the current stage of our investigation, we have adopted a simple join point model (a subset of AspectJ) with primitive pointcuts like `call`, `execution`, `loop`, `withincode`, `with`, `within` and `args` and advice declarations like `before`, `after` and `around`. It was observed that the Aspect FORTRAN weaver that was constructed after the completion of the Aspect Pascal weaver reused a majority of the available front-end artifacts (e.g., generic metamodel and ATL specifications).

In addition to front-end reuse, the framework provides a reusable library of back-end external functions that can be used to provide low-level transformation support for new aspect weaver construction. These functions provide efficient tree traversal strategies in addition to AST manipulation [12]. Currently, there are 11 such functions that are shared by the Object Pascal and FORTRAN weavers. However, not all external functions are reusable or shared, especially, the ones that are dependent on

the syntax of the base language. In such cases, the functions adopted by multiple weavers generally use identical algorithms and conform to a common abstract structure to increase their reusability.

Figure 18 shows the reusability summary for the FORTRAN and Object Pascal weavers. It can be observed that the front-end reusability is considerably larger than the back-end reusability. Overall, approximately 55-65% of the artifacts were reused between the two weavers.

FRONT-END REUSABILITY				
METAMODEL	KM3+TCS (LOC)	Shared LOC		Percentage
Aspect Pascal	565	280		**49.5**
Aspect FORTRAN	550			**50.1**
MODEL TRANSFORMATION	ATL (LOC)	Shared LOC		Percentage
Aspect Pascal	1890	1290		**68.2**
Aspect FORTRAN	1585			**81.3**
BACK-END REUSABILITY				
PARLANSE FUNCTIONS	Total LOC	Shared LOC	No. of Shared Functions	Percentage
Aspect Pascal	873	310	11	**35.5**
Aspect FORTRAN	775			**40**
OVERALL REUSABILITY				
OVERALL	LOC	Shared LOC		Percentage
Aspect Pascal	3328	1880		**56.4**
Aspect FORTRAN	2910			**64.6**

Figure 18: Reusability summary for FORTRAN and Object Pascal weavers

4.4 Limitations to Current Framework

Although more advanced pointcuts like control flow (`cflow`) and reflection (`thisJoinPoint`) were omitted from the current investigation due to limited control flow analysis in DMS for Object Pascal and FORTRAN, future research aims to introduce them at a later stage. It should be noted that the join point model in our implementation is AST based, where join points are mapped to specific nodes or control points in the AST. Using the underlying AST based infrastructure, a control flow graph for a specific language in DMS can be implemented by using an attribute grammar evaluator that propagates control points around the AST and assembles those control points into a completed graph. Additional context information like `thisJoinPoint` can be added at specific nodes or control points in the AST. One can construct these evaluators by using a DMS Control Flow graph domain and a supporting library of control flow graph facilities that it provides. These evaluators are currently available off-the-shelf for the more popular C++, C and Java domains. Since building an attribute evaluator for Object Pascal and FORTRAN is purely engineering and less scientific in nature, we chose to introduce them at a later stage.

However, we understand that with the addition of reflective capabilities and more advanced pointcut mechanisms, there might be a reduction in the overall reusability of model transformation rules in another language or platform context, but the goal of this research has been to show a technique to construct reusable aspect weavers by utilizing most of the software artifacts (e.g., existing parsers and analysis engines) that are already available for a variety of legacy and modern programming languages. The science and theory to construct such tools are already well-established and it would require considerable engineering effort to build them from scratch without gaining any additional scientific knowledge. On the other hand, new language-independent techniques like .Net CLI / CodeDOM are not always feasible to support various legacy languages like FORTRAN, COBOL and Object Pascal due to their non-conformance to the .Net specification. Unless those languages are forced to comply with a language-independent CLI specification, new experimentation to impart AOP features to such languages is very challenging. Currently, DMS provides more mature analysis engines for languages like C++ and Java. As part of possible future extensions, we plan to experiment with such advanced pointcut mechanisms (`cflow`, `reflection`) for these two languages. Another limitation in our current aspect metamodel is the absence of inter-type declaration (ITD). This was primarily due to the reason that we intended to experiment with a smaller subset of AOP language features and introduce others gradually at a later stage. This was more of a design choice than a limitation of the framework or the underlying program transformation engine, which has the necessary machinery to support ITDs.

Language-specific weavers will have a distinct advantage in their initial implementation of new features because they are direct extensions of their base language (low complexity). However, if one has to repeat those features in another language and platform context, there is no technique to support reuse of such features (low reusability). In our framework, the complexity is high on the part of the weaver constructor because of multiple levels of transformation. However, it is possible to pass on already implemented features from one language to another (high reusability). From the end-user's point of view, the complexity is similar for both language-specific and language-independent techniques, except the fact that a better mechanism is needed to handle opaque statements (i.e., statements that are not parsed by the front-end).

5 Related Work

In addition to Weave.Net [18], SourceWeave.Net [17], and Compose* [44] (presented in Section 3), another framework that aims toward language-independent AOP is Aspicere [16]. Currently, Aspicere's weaver transforms a C program by manipulating an XML-representation of its Abstract Syntax Tree (AST), but future extensions aim to combine Aspicere with the Gnu Compiler Collection (GCC). The technique plans to introduce two new intermediate representations: GENERIC and GIMPLE trees. Each different language front-end produces a forest of GENERIC trees, which are then turned into GIMPLE for optimizations and eventually fed to the back-ends. Aspicere aims at expressing the weaver's semantics in terms of generic trees that can eventually lead to language-independent AOP.

An initial prototype that brings aspects to COBOL was developed through a collaboration of academic and industrial partners [31]. The implementation reuses a

pre-existing COBOL front-end to construct an AST that is persisted as XML. The Aspect COBOL weaver operates on the XML representation using a DOM-based approach. The weaver has similar semantics to AspectJ join points [8], but uses an imperative language that is closer to COBOL syntax. The weaver provides ad hoc type analysis (e.g., use-to-def site navigation) for more sophisticated data join points.

A related challenge emerging from representing the source model in the form of an XML representation (as seen in Weave.Net, SourceWeave .Net, and AspectCobol) concerns the issue of scalability. The verbosity of an XML code representation may hamper the size of an application that can be weaved. It has been reported that an XML representation is up to 50 times larger than other internal representations and much slower to transform [32]. The verbosity of XML may influence the ability of SourceWeave.Net, Aspicere and AspectCobol to handle very large applications.

More recently, Heidenreich et al. showed a generic approach for implementing aspect orientation for arbitrary languages using invasive software composition. However, their technique is more useful for declarative DSLs than for GPLs [26].

Morin et al. presented a generic aspect-oriented modeling framework to represent aspects that can be adapted to any modeling domain [27]. Although our work tends to capture the generality of aspect languages and not individual aspects, nevertheless, it can gain interesting insights from such an approach.

Within the AOSD-Europe project, a metamodel for aspect-oriented programming languages was developed [42]. The metamodel presented in that project consisted of four different metamodels; namely, the join point metamodel, join point selector metamodel, selector advice binding metamodel and advice metamodel. We believe this to be useful information and may be used to enhance the metamodel described in this paper.

A recent addition to the class of language extension tools is MetaBorg [33], which provides an ability to embed domain-specific languages into general-purpose languages. However, the embedding permitted by MetaBorg is focused on localized adaptations, and cannot accommodate the global effects of aspects. MetaBorg also requires specific transformation rules to be written for each GPL. Based on the MetaBorg approach, an extensible kernel language for multi-language AOP called Reflex was developed [43].

In [34], advanced pattern detection techniques are suggested by applying a logic-based query language that uses concrete source code templates to match against a combination of structural and behavioral program representations, including call-graphs, points-to analysis results and abstract syntax trees. This is similar to the rule specification language available in DMS that is used for pattern matching. RSL also provides external patterns and conditions that make calls to external functions written in DMS PARLANSE (a parallel language for manipulating symbolic expressions) for more advanced program analysis.

Ramos et al. propose a framework for expressing patterns as *model-snippets* and show how pattern matching can be performed with model-snippets for any given metamodel [35]. In our current framework, all pattern matching and analysis is done through the back-end, where the metamodel is used to express the front-end aspect language and its generic extensions. All of the higher-order aspect specifications are translated to lower-order back-end program transformation code that does the actual weaving.

6 Conclusion

The research presented in this paper raised several key challenges (identified in Section 1.1) in designing a generic framework to construct aspect weavers. In particular, the paper describes a model-driven framework that combines program transformation with model-driven engineering to construct aspect weavers for modern and legacy programming languages. We showed how *Challenges C1 (parser construction problem)* and *C2 (weaver construction problem)* were resolved through adoption of a mature program transformation engine as the back-end of the framework. The paper also illustrated how accidental complexities (*Challenge C3*) that are generally associated with a program transformation system can be reduced using a model-driven front-end. The last challenge (*Challenge C4*) deals with generality, reusability, and transfer of knowledge from one weaver to another. In our opinion, this is the most difficult challenge of the four. We demonstrated that by making the front-end generic, along with a systematic program transformation rule generator, significant inroads have been made to address this challenge. To evaluate the usefulness of the generic framework, two aspect weavers were constructed for Object Pascal and FORTRAN. The FORTRAN weaver was built after the successful construction of the Object Pascal weaver. When constructing the second weaver, we observed that we could reuse more than 50% of the artifacts (generic front-end and rule generator) that were created during the construction of the Aspect Pascal weaver.

The current approach is focused on a simple join point model. More advanced pointcuts like control flow and reflective techniques like `thisJoinPoint` are currently not available. However, with the availability of a mature control flow analysis engine for Object Pascal and FORTRAN in DMS, we can extend the weavers to support advanced aspect language features. Note that most of the analysis and pattern matching is realized through the back-end program transformation engine and the front-end only acts as a wrapper to the back-end. If the back-end PTE can support advanced program analysis, it is possible to wrap those features through the front-end, avoiding all the accidental complexities (*Challenge C3*) that are generally associated with complex PTEs.

In summary, this research provides an initial solution to several challenges listed in Section 1 by reusing most of the existing software artifacts (e.g., lexers, parsers, analyzers, evaluators) that are already available for a variety of GPLs. Thus, our framework enables new experimentation with advanced software engineering principles like AOP for existing legacy languages. The research also addresses new challenges that arise from the usage of complex PTEs like DMS by providing a suitable front-end that hides most of the accidental complexity.

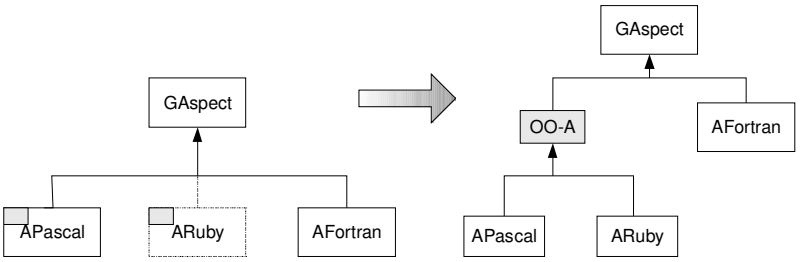
6.1 Lessons Learned

In this section, we summarize the seven main lessons that we have learned while working on the research presented in this paper. These lessons are enumerated below:

- ***Lesson 1 - Generalizing the front-end:*** We realized that parts of the aspect language front-end can be reused by making it generic. By generalizing the front-end metamodel, several aspect languages can extend a single core (e.g.,

GAspect) while the differences can be captured within their specific part. The solution can be achieved using MDE techniques like metamodel extension.

- ***Lesson 2 - Improving the generic metamodel:*** The current generic metamodel (i.e., GAspect) generalizes what is common between APascal and AFortran (i.e., the aspect languages for Object Pascal, and FORTRAN). Figure 19a shows this current design. If we consider the construction of ARuby (i.e., an aspect language for Ruby) using our framework, this new language could directly extend GAspect as shown in Figure 19a. However, it is expected that both ARuby and APascal will have some commonality (e.g., related to the object paradigm) not shared with AFortran. Figure 19b shows the structure of the new improved design. The commonalities between APascal and ARuby are extracted into OO-A (the common object-oriented constructs of ARuby and APascal), which extends GAspect. In [36], a proposal for typing models as a collection of interconnected objects is discussed. The formalism described there is an extension to object-oriented typing, but suitable to a model-oriented context. Our approach of defining an abstract metamodel and its conformance between other metamodels via metamodel extensions is similar to the concept described in [36].



a) Current design of front-end metamodel

b) Improved design of front-end metamodel

Figure 19: Improving the front-end metamodel design

- ***Lesson 3 – Use of generic interfaces in the rule generator:*** The concept of generic interface was introduced in Section 3.2 to generalize the design of the rule generator. As a result, the rule generator library can be reused across languages with minimum customization.
- ***Lesson 4 - Modeling can be suitably applied to PTEs:*** From our research, we realized that it is possible to model and transform program transformation rewrite-rules using MDE. The combination of both technical spaces offers more possibilities than each considered separately.
- ***Lesson 5 - Changing the target PTE:*** The source aspect metamodel need not be altered even if one chooses to opt for a different target PTE (e.g., ASF+SDF). In such a case, a new PTE metamodel needs to be developed, as well as a new rule generator for this new target. We expect that it may be possible to generalize part of the transformation code by introducing a PTE pivot metamodel that abstracts common properties of many PTEs.

- ***Lesson 6 - Changing the source language:*** Conversely, for every new aspect language, one needs to add the appropriate metamodel extensions to the GAspect metamodel, but no change to the target metamodel is needed.
- ***Lesson 7 - Automation of rule generator:*** We realized that most of the time and effort on building a new weaver is spent on understanding the concrete syntax of the base language. We believe that it should be possible to extract the join point model from transformation rules, and model it in terms of the concrete syntax. Then, a significant part of the transformations could be automatically generated.

6.2 Future Work and Possible Extensions

The following represent areas of future investigation that can extend the capabilities of the model-driven weaver construction techniques introduced in this paper:

- ***Improving reusability:*** We would like to improve the reusability of features among aspect weavers by further enhancements to the existing design of our framework. For example, we would like to create a generic metamodel for object-oriented constructs, from which the weavers constructed for object-oriented languages can inherit. Similarly, we would like to create a generic metamodel for Join Point Models (JPMs). All weavers can inherit from the generic JPM, and (if required) add new join point extensions to their specific JPM.
- ***Constructing weavers for other GPLs:*** Another possible extension of our work is to construct aspect weavers for other GPLs including object-oriented scripting languages like Ruby, JavaScript and Python. We can also experiment with adding new types of join points (e.g., loops and conditional statements) to existing general-purpose programming languages like Java and C++.
- ***Applying the approach to DSALs:*** Although the majority of research in the AOSD community focuses on general-purpose aspect languages (e.g. AspectJ), there have been a number of influential investigations on domain-specific aspect languages (DSALs) (e.g., COOL for concurrency management and RIDL for serialization [38]). So far, we have only considered the construction of general-purpose aspect languages using our framework, but it would be interesting to investigate how the framework can also accommodate the development of weavers targeting DSALs.

Acknowledgement

This work is supported in part by an NSF CAREER award (CCF-1052616) and by the OpenEmbeDD project.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J.: Aspect-Oriented Programming. *European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag LNCS 1241, Jyväskylä, Finland, June 1997, pp. 220-242.
2. Clarke, S., Harrison, W., Ossher, H., Tarr, P.: Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Denver, CO, October 1999, pp. 325-339.
3. Jacobson, I., and Ng, P.: *Aspect-Oriented Software Development with Use Cases*, Addison-Wesley, 2005.
4. France, R., Ray, I., Georg, G., and Ghosh, S.: Aspect-Oriented Approach to Design Modeling. *IEE Proceedings - Software* (Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design), vol. 151, no. 4, August 2004, pp. 173-185.
5. Schmidt, D.: Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, vol. 39, no. 2, February 2006, pp. 25-31.
6. Lämmel, R., and Verhoef, C.: Cracking the 500 Language Problem. *IEEE Software*, November/December 2001, pp. 78-88.
7. Ulrich, W., *Legacy Systems: Transformation Strategies*, Prentice-Hall, 2002.
8. Kiczales, G., Hilsdale. E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.: Getting Started with AspectJ. *Communications of the ACM*, October 2001, pp. 59-65.
9. Baxter, I., Pidgeon, C., and Mehlich, M.: DMS: Program Transformation for Practical Scalable Software Evolution. *International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004, pp. 625-634.
10. Fradet, P., and Südholt, M.: Towards a Generic Framework for Aspect-Oriented Programming. *AOP Workshop, ECOOP '98 Workshop Reader*, Springer-Verlag LNCS 1543, Brussels, Belgium, July 1998, pp. 394-397.
11. Aßmann, U., and Ludwig, A.: Aspect Weaving with Graph Rewriting. *International Symposium on Generative and Component-Based Software Engineering*, Erfurt, Germany, September 1999, pp. 24-36.
12. Gray, J., and Roychoudhury, S.: A Technique for Constructing Aspect Weavers using a Program Transformation Engine. *International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, March 2004, pp. 36-45.
13. van den Brand M., Heering, J., Klint, P., and Olivier, P.: Compiling Rewrite Systems: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, July 2002, pp. 334-368.
14. Cordy, J.: The TXL Source Transformation Language. *Science of Computer Programming*, vol. 61, no. 3, August 2006, pp. 190-210.
15. Akers, R., Baxter, I., Mehlich, M., Ellis, B., and Luecke, K.: Case Study: Re-engineering C++ Component Models via Automatic Program Transformation. *Information & Software Technology*, vol. 49, no. 3, 2007, pp. 275-291.

16. Adams, B.: Language-independent Aspect Weaving. *Summer School on Generative and Transformational Techniques in Software Engineering,* Braga, Portugal, July 2005.
17. Jackson, A., and Clarke, S.: SourceWeave.NET: Cross-Language Aspect-Oriented Programming. *Generative Programming and Component Engineering (GPCE)*, Springer-Verlag LNCS 3286, Vancouver, Canada, October 2004, pp. 115-135.
18. Lafferty, D., and Cahill, V.: Language-Independent Aspect-Oriented Programming. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, CA, October 2003, pp. 1-12.
19. Lämmel, R.: Declarative Aspect-Oriented Programming. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Antonio, TX, January 1999, pp. 131-146.
20. Jouault, F., and Bézivin, J.: KM3: a DSL for Metamodel Specification. *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Springer-Verlag LNCS 4037, Bologna, Italy, June 2006, pp. 171-185.
21. Kurtev, I., Bézivin, J., Jouault, F., and Valduriez, P.: Model-based DSL Frameworks. *Object-Oriented Programming, Systems, Languages and Applications*, Portland, OR, October 2006, pp. 602-616.
22. Barbero, M., Jouault, F., Gray, J., and Bézivin, J.: A Practical Approach to Model Extension. *European Conference on Model Driven Architecture Foundations and Applications (ECMDA)*, Haifa, Israel, June 2007, pp. 32-42.
23. Jouault, F., Bézivin, J., and Kurtev, I.: TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. *Generative Programming and Component Engineering (GPCE)*, Portland, OR, October 2006, pp. 249-254.
24. Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I.: ATL: A Model Transformation Tool. *Science of Computer Programming*, vol. 72, nos. 1-2, 2008, pp. 31-39.
25. Harbulot, B., and Gurd, J.: Using AspectJ to Separate Concerns in Parallel Scientific Java Code. *International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, March 2004, pp. 122-131.
26. Heidenreich, F., Johannes, J., and Zschaler, S.: Aspect Orientation for Your Language of Choice. *11th International Workshop on Aspect-Oriented Modeling*, Nashville, TN, September 2007.
27. Morin, B., Barais, O., Jézéquel, J., and Ramos, R.: Towards a Generic Aspect-Oriented Modeling Framework. *Workshop on Models and Aspects (ECOOP)*, Berlin, Germany, July 2007.
28. Klint, P., Lämmel, R., and Verhoef, C.: Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 3, 2005, pp. 331-380.
29. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, vol. 22, no. 6, 1996, pp. 789-828.
30. *GenAWeave project website*, http://www.cis.uab.edu/softcom/genaweave
31. Lämmel, R., and De Schutter, K.: What Does Aspect-Oriented Programming Mean to Cobol? *International Conference on Aspect-Oriented Software Development (AOSD)*, Chicago, IL, March 2005, pp. 99-110.

32. Germon, R.: Using XML as an Intermediate Form for Compiler Development. *XML Conference and Exposition*, Orlando, FL, December 2001.
33. Bravenboer, M., and Visser, E.: Concrete Syntax for Objects: Domain-specific Language Embedding and Assimilation without Restrictions. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA),* Vancouver, Canada, October 2004, pp. 365-383.
34. De Roover, C., Brichau, J., Noguera, C., D'Hondt, T., and Duchiena L.: Behavioural Similarity Matching using Concrete Source Code Templates in Logic Queries. *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'07)*, Nice, France, January 2007, pp. 92-101.
35. Ramos, R., Barais, O., and Jézéquel., J.: Matching Model-Snippets. *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*, Nashville, TN, pp. 121-135.
36. Steel, J., and Jézéquel, J.: On Model Typing. *Journal of Software and Systems Modeling (SoSyM)*, vol. 6, no. 4, December 2007, pp. 452-468.
37. *Eclipse Model to Text (M2T) project*, http://www.eclipse.org/modeling/m2t/.
38. Lopes, C.: "D: A Language Framework for Distributed Programming," *Ph.D. Dissertation*, College of Computer Science, Northeastern University, December 1997 (1998).
39. Jossic, A., Del Fabro, M., Lerat, J., Bézivin, J., and Jouault, F.: Model Integration with Model Weaving: A Case Study in System Architecture. *International Conference on Systems Engineering and Modeling*, Haifa, Israel, March 2007, pp. 79-84.
40. Hilsdale, E., and Hugunin, J.: Advice Weaving in AspectJ. *International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, March 2004, pp. 26-35.
41. Schult, W., and Tröger, P.: Loom.NET - An Aspect Weaving Tool. *Workshop on Aspect-Oriented Programming, ECOOP'03*, Darmstadt 2003.
42. Brichau, J., Mezini, M., Noyé, J., Havinga, W., Bergmans, L., Gasiunas, V., Bockisch, C., Fabry, J., D'Hondt, T.: An Initial Metamodel for Aspect-Oriented Programming Languages, *Deliverable D39*, AOSD-Europe, 2006.
43. Tanter, E.: An Extensible Kernel Language for AOP. *AOSD Workshop on Open and Dynamic Aspect Languages*, Bonn, Germany, 2006.
44. de Roo, A., Hendriks, M., Havinga, W., Durr, P., Bergmans, L.:Compose*: A Language- and Platform-Independent Aspect Compiler for Composition Filters. *Workshop on Advanced Software Development Tools and Techniques*, Pahpos, Cyprus, July 2008.
45. Chalabine, M., and Kessler, C.: Parallelization of Sequential Programs by Invasive Composition and Aspect Weaving. *Advanced Parallel Processing Technologies*, Book Chapter, Springer Berlin / Heidelberg, 2005.
46. Irwin, J., Loingtier, J., Gilbert, J., Kiczales, G., Lamping, J., Mendhekar, A., and Shpeisman, T.: Aspect-Oriented Programming of Sparse Matrix Code. *International Scientific Computing Object-Oriented Parallel Environments*, Springer-Verlag LNCS 1343, 1997, pp. 249-256.
47. Herrmann, C., and Lengauer, C.: Using Metaprogramming to Parallelize Functional Specifications, *Parallel Processing Letter*, Vol. 12, Issue 2, 2002, pp. 193-210.