Making Aspect-Orientation Accessible through Syntax-based Language Composition

Victor Winter  
Department of Computer Science  
University of Nebraska at Omaha  
Omaha, Nebraska USA 68182  
vwinter@mail.unomaha.edu

Günter Kniesel  
Dept. of Computer Science III  
University of Bonn  
Römerstr. 164, D-53117 Bonn Germany  
gk@cs.uni-bonn.de

Harvey Siy  
Department of Computer Science  
University of Nebraska at Omaha  
Omaha, Nebraska USA 68182  
hsiy@mail.unomaha.edu

Mansour Zand  
Department of Computer Science  
University of Nebraska at Omaha  
Omaha, Nebraska USA 68182  
mzand@mail.unomaha.edu

Abstract

A generic syntax-based approach is presented by which a fixed set of aspect-oriented features belonging to an aspect language family $L_A$ can be applied to a DSL. The approach centers on the construction of a grammar in which a predefined and fixed set of abstract join points and join point environments are linked with their concrete counterparts within the DSL. This connection enables the behavior of static weaving to be expressed in a generic manner. The resulting framework is one in which aspect-orientation is accessible to non-experts across a wide spectrum of abstractions.

1 Introduction

Recent times have seen a growing demand, across the industrial-academic spectrum, for a more significant and focused application of generalized computer science to a variety of disciplines and problem domains such as bioinformatics, healthcare, and information assurance. This has resulted in an increase of interdisciplinary research in which computer science is seen as the enabling technology. In order for interdisciplinary research to be successful and yield results that are of practical value, it is essential for domain experts to be engaged to the fullest extent possible. A precondition to such engagement is the existence of a language allowing suitable communication between the domain expert and the computer scientist. This motivates the wide-spread use of domain-specific languages.

Definition 1 A domain-specific language (DSL) is a language in which abstractions, both static and dynamic, that are intrinsic to a problem domain are given syntactic prominence.

Definitions similar to the one above can be found across the literature [20, 23, 18] and the issues raised are part of a larger, well-known discussion concerning itself with the relationship between language and thought. In the context of this article, the construction of a DSL represents a necessary first step towards maximizing the engagement of domain experts in interdisciplinary research. Loosely stated, a DSL can be seen as providing the “interface” from a
problem domain to the general body of computer science knowledge. When cast in these terms, the importance of a
DSL cannot be overstated.

In this article, the terms artifact and program will be used to refer to a member of a DSL. Depending on the DSL,
an artifact/program can refer to a wide range of abstractions including: specifications, design documents, security
policies, clinical guidelines, and software.

A well-designed DSL should be consistent with the fundamental design principles underlying modern programming
languages. The rationale being that as the information content in an artifact increases, one encounters the same
complexity-related barriers that programming languages have been designed to try to mitigate or overcome. It can
be argued that mechanisms for abstraction, scope, hierarchy, and cross-referencing represent universal constructs
common to all formalisms whose goal is to capture complexity in a decompositional manner.

Aspect-orientation represents a fundamental mechanism for mitigating a particular kind of complexity that nat-
urally accumulates in a traditional decompositional framework. As a result, it becomes highly desirable to also
incorporate aspect-orientation into the framework of a DSL. In general, when considering the intersection of aspect-
orientation and domain specificity, several combinations are meaningful. The space of possibilities is highlighted in
Table 1.

<table>
<thead>
<tr>
<th></th>
<th>DSAL</th>
<th>GPAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPL</td>
<td>√</td>
<td>−</td>
</tr>
<tr>
<td>DSL</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 1: The intersection of domain specificity and aspect-orientation

The most common [20] composition, which we denote $GPL \circ DSAL$, arises when applications in a specific problem
domain are implemented using a general purpose language (GPL). In this composition, a domain-specific aspect
language (DSAL) provides support for the modularization of cross-cutting concerns specific to the problem domain.
For example, ERTSAL [25] is a DSAL intended to facilitate the analysis of embedded real-time systems that are
implemented in C++.

In the $DSL \circ DSAL$ composition, programs are written in a DSL and again a DSAL serves as the modularization
mechanism for cross-cutting concerns. For example, AspectLISA [20] is a DSAL providing an aspect-oriented exten-
sion to the specification language LISA. LISA is a specification language in which regular expressions, BNF grammars
and attribute grammars can be combined to specify the syntax and semantics of a given programming language $L$. A
LISA specification can be used to automatically generate a compiler for $L$. The join points recognized by AspectLISA
are static and describe places in a LISA specification where semantic rules can be added. Environments describing
places where join points reside within a LISA specification include the language and rule modules.

A more universal approach towards the $DSL \circ DSAL$ composition is supported by the XAspects [23] system
which provides a general framework in which domain-specific aspect and (non-aspect) component languages can be
integrated.
Subject to certain constraints, the compositional approach presented in this article is applicable across the entire spectrum of categories defined in Table 1. Our approach assumes (presently) that an abstract (but fixed) set of aspect-oriented concepts have been defined (i.e., agreed upon) and that join point matching as well as weaving can be expressed at the syntactic level. Subject to these constraints, our approach is generic in the sense that it can be used to incorporate the agreed upon aspect-oriented concepts into any base language, be it a DSL or a GPL.

In order to make our ideas more understandable, our approach is presented, in a language-neutral fashion, in terms of abstractions of widely-known general purpose aspect-oriented concepts (e.g., abstractions of set/get join points and within environments). The reason for this is to make our ideas more easily accessible to any language engineer, no matter what their special domain of interest might be. As a result, we primarily illustrate the category $\text{DSL} \circ \text{GPAL}$ denoting the composition of a DSL with a general-purpose aspect language (GPAL). This choice also has an added benefit because this category (to our knowledge) has not been explored before. A sensible example of $\text{DSL} \circ \text{GPAL}$ is then presented in Section 4.2.

On a more technical level, the focus of this paper is on a syntax-based approach supporting the automated synthesis of $\text{DSL} \circ \text{GPAL}$. The goal of this syntax-based language composition is to produce a composite language in which weaving can be performed using a generic weaver $W_A$. By accomplishing this, the task of implementing a weaver is factored out of the effort needed to create the language composition. Thus, the effort of incorporating aspect-orientation into a DSL is significantly eased, and as a result more accessible to technically-oriented domain experts. In addition, we also claim that the exploration of the $\text{DSL} \circ \text{GPAL}$ composition is consistent with the goal of providing “simpler use of aspects in specialized domains” [9]. Supplementing this claim, a more concrete rationale for the practical utility of a $\text{DSL} \circ \text{GPAL}$ language is given in Section 1.1. The rationale presented here presents a context and storyline motivating this particular direction of research. As such, informal conclusions drawn from the rationale should not be used to supplant the scientific validation provided by a proper case study.

Shonle et. al. [23] have conjectured that there will always be a gap between domain-specific aspect languages and general-purpose aspect languages with respect to the clarity and conciseness of pointcut specifications and the efficiency of their implementations. The intuition behind this statement stems from the belief that, in general, a concern can crosscut a system in an infinite number of ways. This suggests a continuing need for DSALs, in particular, DSALs well-suited to capturing specific domain-dependent concerns. While we agree with this point, we also take the position that it is orthogonal to the rationale motivating the need of the $\text{DSL} \circ \text{GPAL}$ combination explored in this article.

1.1 Motivation

The focus of this article is on how to incorporate a basic set of aspect-oriented abstractions into a domain-specific language (DSL) in a manner that is accessible to technically-oriented domain experts. In this article, we assume that technically-oriented domain experts have a background in GPL programming, but that their background in language
and compiler design is limited. We also assume that the applications being developed by these technically-oriented
domain experts involve special-purpose computing platforms. Examples of such platforms include one-of-a-kind
state-of-the-art systems as well as systems making use of sunset technologies (i.e., legacy technologies). For these
systems, commercial compilers and analysis tools are generally not available [1].

The assumptions stated in the previous paragraph are based on personal experiences obtained from working
with industry. In fact, the stated assumptions closely reflect a funded project that is currently underway. In this
environment, the development of a DSL is accomplished through a close collaboration between technically-oriented
domain experts and language experts. As a result, it is reasonable to assume that the resulting DSL conforms to
good language design principles. It is also assumed that minor maintenance associated with the DSL will be the
responsibility of the technically-oriented domain experts and that language experts will only be engaged when major
changes to the DSL are considered. Under these assumptions, we want to make it possible for a technically-oriented
domain expert to create a DSL ◦ GPAL composition.

Our experiences suggest that under the assumptions described here, DSLs often have syntactic and semantic
similarities to general-purpose mainstream programming languages. These similarities are not accidental and are in
fact the result of premeditated efforts. The objective is to lower acceptance barriers associated with learning how to
write programs in the DSL by (future) team members and to lower efforts associated with the maintenance of the
DSL.

Given this perspective, it can be argued that a close approximation to a DSL can be realized through a special
purpose library written in a general-purpose language (GPL). A GPL library would implement (and thereby identify)
a set of data types that are well-suited to a specific problem domain. These abstractions would closely support the
kinds of analysis and reasoning in which technically-oriented domain experts engage during problem solving. RG
[17] is a classic example of a system implemented in this fashion.

In practice however, simply developing a suitable GPL library is generally not a viable approach in this context
for a variety of reasons. One reason, previously stated, is that commercial tools (e.g., compilers) are generally not
available for the job at hand. A second argument against a GPL library-based approach is based on reducing the
effort associated with developing a suitable custom compiler. For example, consider a GPL library developed in C++.
Developing even the parsing component of such a compiler can represent a daunting task. Furthermore, it may be
the case that only a modest subset of C++ is actually needed and therefore the full power of C++’s abstractions
need not be supported.

A third argument against a GPL library-based approach is that the syntactic overhead associated with using
the library may be high. It is not uncommon for these kinds of DSLs to have certain idiosyncrasies leading to an
awkward syntactic fit within a GPL framework. Even though, from a theoretical perspective, this represents nothing
more than “syntactic sugar”, a significant portion of the rational behind the decision to develop a DSL often is based
on smoothing out the syntactic differences between the abstractions of the problem domain and the abstraction
mechanisms of the GPL.

1.2 An Example-Driven Overview

Let us now take a look at a small example highlighting the major issues surrounding the syntactic-based approach to the base-language \( \circ \) aspect-language composition proposed in this article. Conceptually speaking, the approach described in this article consists of three parts.

**Part 1.** Formalize a set of grammar conventions so that join points and join point environments within an artifact \( p \) can be statically recognized simply by inspecting the parse tree corresponding to \( p \). For base languages whose semantics permit this, two important aspect-oriented concepts, the join point and the join point environment, can be explicitly embedded into the syntax of the language. Note that how this explicit embedding is to be accomplished is described in a language independent fashion and is captured in the form of a set of grammar conventions. The result is a set of join point and join point environment recognition primitives for the base language.

**Part 2.** Given a set of join point and join point environment recognition primitives, the semantics of a pointcut language can now be constructed in which one is capable of specifying particular join points (e.g., \texttt{set}) occurring within certain environments (e.g., \texttt{within}). In other words, one is able to evaluate an arbitrary parse tree, conforming to our grammar conventions, and correctly identify those join points satisfying a given pointcut specification.

**Part 3.** A generic grammar for an aspect language can be developed in which a given set of aspect concepts are abstractly modelled. For example, the notion of advice constrained by pointcut expressions can be modelled. Similarly, the notion of a composition type (e.g., \texttt{before}, \texttt{after}, and \texttt{around}) associated with an advice can also be abstractly modelled. In this article, we will refer to this language as a GPAL.

The usefulness of the constructions described in Parts 1-3 rests on the observation that a weaver can be developed in which knowledge of join points and join point environments is solely based on the set of grammar conventions mentioned in Part 1. As a consequence, such a weaver is language independent. Therefore, it can be used for any language provided the language adheres to the stated grammar conventions. Indeed, we have verified this claim by building such a generic weaver. Though its details lie beyond the scope of this article, we would like to mention that the weaver has been implemented in the transformation language TL [32, 29] within the HATS environment [30]. The technical details describing its implementation can be found in [31].

For the remainder of this section, we would like to give a more conceptual overview of the intent behind the grammar conventions upon which this entire approach rests. We will do this by looking at a parse tree belonging to an artifact \( p \) written in a small language called Toy. Figure 1 gives a fragment of the syntax of Toy and the source code of \( p \). We claim that, along some important dimensions, Toy is conceptually similar to some of the real-world DSLs that we have encountered. For example, Toy is block-structured and computation is expressed in
It should be noted that, in Toy, identifiers can occur in a variety of contexts including as l-values and r-values.

<table>
<thead>
<tr>
<th>Toy</th>
<th>The Artifact p</th>
</tr>
</thead>
<tbody>
<tr>
<td>class_def ::= “class” id class_body</td>
<td>class A {</td>
</tr>
<tr>
<td>class_body ::= [ “extends” id ] “{” dec_list “}”</td>
<td>int f(int x) {</td>
</tr>
<tr>
<td>dec_list ::= dec [ “;” ] dec_list</td>
<td>int y;</td>
</tr>
<tr>
<td>dec ::= method_def</td>
<td>( “int”</td>
</tr>
<tr>
<td>method_def ::= ( “int”</td>
<td>“bool”) id method_body</td>
</tr>
<tr>
<td>method_body ::= “(“ arg_list “)” block</td>
<td></td>
</tr>
<tr>
<td>stmt_list ::= ( block</td>
<td>assign</td>
</tr>
<tr>
<td>block ::= “{” stmt_list “}”</td>
<td></td>
</tr>
<tr>
<td>assign ::= id “=” id</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: The syntax of Toy and an artifact $p$ belonging to Toy

We freely acknowledge that not all DSLs are of this nature. However, we do claim that, in the real world, the number of DSLs with conceptual similarities to Toy is not negligible.

Conceptually, a base-language ◦ aspect-language composition requires that the aspect-oriented abstractions in the aspect-language be mapped onto the base-language and that this mapping take place at least at the syntactic level. For example, in Toy, suppose we would like class and method bodies to be recognized as environments in which join points can reside and which can be discriminated by the pointcut language of the aspect-language. We also would like identifiers occurring on the left and right hand side of an assignment (i.e., l-values and r-values) to be recognized as distinct join point types (e.g., set versus get). In this example, we assume that identifiers occurring in other contexts (e.g., in declarations and as formal parameters) fall outside of our join point model and can be ignored.

Figure 2(a) shows the parse tree for the artifact $p$ whose source was shown in Figure 1. This parse tree contains a number of occurrences of identifiers, only some of which are meaningful with respect to the join point model described in the previous paragraph. Figure 2(b) shows a slightly modified parse tree for $p$ in which join point environments and join points have been concretely marked within the parse tree of $p$. Grammar productions in Toy can also be correspondingly marked. Figure 2(c) shows an additionally modified parse tree for $p$ in which abstract markers, belonging to the grammar of the aspect-language have been inserted, and appropriately linked with concrete markers belonging to Toy. It is with respect to this last structure that generic weaving becomes possible.

1.3 Technical Overview

A generic approach is presented where a fixed set of aspect-oriented abstractions belonging to a GPAL can be incorporated into a DSL yielding the composite language $DSL ◦ GPAL$. In the discussion that follows, $\mathcal{L}$ denotes a base DSL and $\mathcal{G}_\mathcal{L}$ denotes the grammar of $\mathcal{L}$. The syntax of GPAL is distributed across two grammars: (1) a generic grammar $\mathcal{G}_A$ in which the core of the GPAL is defined (e.g., advice and pointcut expressions), and (2) a grammar
Figure 2: Parse trees denoting the artifact $p$ given in Figure 1
\( G_{JP} \) enumerating a fixed set of abstractions describing static join points, join point environments (that is, the static contexts in which join points can occur), and advice/join point compositions.

We say that the grammar \( G_A \) is generic because it is completely free of terminal symbols, except for the definition of wildcard identifiers. As a result, \( G_A \) can be understood, at a conceptual level, as describing a family \( L_A \) of aspect languages. Elements of \( L_A \) share a common set of aspect-oriented abstractions. The syntactic details characterizing a particular element of \( L_A \) can be resolved by instantiating \( G_A \). In this context, instantiation concerns itself with making all nonterminals in \( G_A \) generating\(^1\).

For the language \( DSL \circ GPAL \), static weaving is possible through a sequence of discrete generic weaving steps. Given an advice \( \alpha \in G_A \), an associated pointcut \( pc \in G_A \), and a candidate join point \( jp \in G_{JP} \), a weaving step proceeds as follows: a match between \( pc \) and \( jp \) is attempted and if successful, \( jp \) and \( \alpha \) are abstractly composed.

For a given term \( t \), generic weaving is the transitive closure of weaving steps over the cartesian product of join points and advice.

Three key insights make this approach possible: (1) the use of abstract marker productions, defined in \( G_{JP} \), to model join points and join point environments in a language-independent fashion, (2) the introduction of concrete marker productions into the grammar of the DSL (i.e., the base language) in order to expose language-dependent join points and join point environments, and (3) the use of grammar superimposition as a way of linking abstract marker productions with concrete marker productions in a manner that the abstractions in \( G_{JP} \) are still recognizable by a weaver within \( L_A \). As a final step in the composition of \( G_A \) with \( G_L \), the grammar \( G_A \) is instantiated creating a dialect \( G_A' \) in which advice bodies are elements belonging to \( G_L \) and whose “look-and-feel” has been adapted to \( G_L \). The result is the composition \( L_{G_A \circ G_L} \) that supports the aspect-oriented features of the language family \( L_A \) on base programs written in \( L_L \).

Our paper makes the following contributions to the state-of-the-art of domain specific aspect language engineering:

1. From the perspective of the aspect-oriented language component.
   (a) We give a general approach to defining a family of aspect languages. We then construct a generic grammar \( G_A \) for a particular aspect language family \( L_A \). This language family supports advice definition and a variety of identifier-based pointcut expressions.
   (b) We describe a grammar \( G_{JP} \) in which generic join points and join point environments are modelled via abstract marker productions. The grammar \( G_{JP} \) also contains abstract representations of advice/join point compositions, which we refer to as residuals.

2. From the perspective of the domain-specific base language component.
   (a) Within the context of a DSL grammar \( G_L \), we introduce concrete marker productions as a construct for exposing join points and join point environments.

\(^1\)A nonterminal symbol \( X \) is generating if \( X \rightarrow^* w \) where \( w \) consists solely of terminal symbols.
(b) We introduce grammar superimposition as a technique by which abstract marker productions and concrete marker productions can be linked. It is through linking that a generic, language independent, weaver \( W_A \) is able to recognize, within artifacts belonging to \( L_C \), the aspect-oriented abstractions in \( G_{JP} \) with respect to which its weaving behavior is defined. We also present a grammar superimposition algorithm.

The rest of the paper is organized as follows: Section 2 introduces the generic grammar \( G_A \) for aspect languages that support advice and identifier-based pointcuts. Section 3 describes our approach to syntax-based language composition and explains the challenges that it raises, the solutions that we propose, and the assumptions that we make to enable our solutions. Section 4 presents two concrete applications of our techniques, showing how \( G_A \) and \( G_{JP} \) can be mapped to two very different non-toy DSLs and how the corresponding grammar compositions are derived.

Section 5 compares our approach to AspectJ-like languages. Section 6 discusses related work, and Section 7 concludes.

## 2 A Generic Grammar for a Simple Aspect Language Family

In this section, we introduce a generic grammar \( G_A \) describing a simple family of aspect languages \( L_A \). We say that \( G_A \) (Figure 3) is generic because all symbols occurring in \( G_A \), except for \{id, num, "*", \( \epsilon \)\}, are nonterminal symbols. We say that the captured aspect language family is simple because it supports a minimal set of aspect-oriented concepts (just basic advice and pointcuts) and therefore cannot be compared to full-blown aspect languages.

```
01  A ::= aspect_def_list
02  aspect_def_list ::= aspect_def aspect_def_list | \( \epsilon \)
03  aspect_def ::= ASPECT id BEGIN_ASPECT advice_def_list END_ASPECT
04
05  advice_def_list ::= advice_def advice_def_list | \( \epsilon \)
06  advice_def ::= initial_advice | final_advice
07  initial_advice ::= ADVICE [ INLINE ] id composition_type pc ADVICE_BODY
08
09  composition_type ::= BEFORE | AFTER | AROUND
10
11  pc ::= match_sequence
12  match_sequence ::= static_env jp_designator
13  static_env ::= static_env env_designator DOT | \( \epsilon \)
14
15  jp_designator ::= jp_type LPAR wild_id RPAR
16  env_designator ::= env_type LPAR wild_id RPAR
17  jp_type ::= JP_TYPE1 | JP_TYPE2 | JP_TYPE3
18  env_type ::= ENV_TYPE1 | ENV_TYPE2
19  wild_id ::= ( ( id | num ) ] [ "*" ] )
```

Underlining indicates the uninstantiated category that will be mapped to a category of \( G_C \). Capital spelling indicates uninstantiated categories whose instantiation is independent of \( G_C \). The nonterminal id stands for an identifier and num stands for an integer (see Section 3.5).

Figure 3: The grammar \( G_A \) describing the aspect language family \( L_A \).
2.1 Aspects and Advice

An aspect program belonging to $L(G_A)$, the language defined by the grammar $G_A$, consists of a list of aspect definitions as shown in Figure 3 by the production in line 02. The body of an aspect definition (line 05) consists of a list of named advice definitions (unlike in AspectJ-like languages). An (initial) advice definition (line 07) contains (1) an optional in-lining directive, (2) an unparameterized before/after/around join point composition type (defined in line 09), and (3) an anonymous, unparameterized pointcut (defined in line 11).

2.2 Wildcards, Designators and Pointcuts

In $G_A$, the wildcard symbol is denoted * and may only occur within a wildcard identifier. A wildcard identifier is a description of an identifier in which zero or more wildcard symbols may appear (e.g., $x*1$ and $x*1*$). Wildcard identifiers can be used in designators for join points and join point environments and their syntax is given on line 19 of Figure 3.

A join point designator (line 15) is the mechanism of the aspect language family $L_A$ for referencing join points in the base language $L_L$. Similarly, an environment designator (line 16) is the mechanism of $L_A$ for specifying the static context in which join points occur in $L_L$ (e.g., that a call join point occurs in a particular method of a particular class). Both categories of designators are currently based on identifier matching, not on signatures (i.e., type information is not used). They only contain (1) a reference (line 17) to a particular join point type (e.g., get), and (2) a wildcard identifier (e.g., $x*1$).

Note that $G_A$ specifies a fixed number of join point and environment types but makes neither assumptions about the concrete keywords used nor about their semantics, that is the elements of $G_L$ that they designate. The expressions get($x*$), set($x*$), and call($f*$) would be legal join point designators and in_class($A*$) and in_method($f*1$) would be legal environment designators in an instantiation of $G_A$ (Section 2.4) containing the following productions:

\[
\begin{align*}
\text{JP_TYPE1} & ::= \text{`get'} \\
\text{JP_TYPE2} & ::= \text{`set'} \\
\text{JP_TYPE3} & ::= \text{`call'} \\
\text{ENV_TYPE1} & ::= \text{`in_class'} \\
\text{ENV_TYPE2} & ::= \text{`in_method'}
\end{align*}
\]

A pointcut (line 11 in Figure 3) is a join point designator (line 12) preceded by a possibly empty sequence of environment designators (line 13). For human readability designators can be separated by a punctuation symbol represented by the (language independent) DOT nonterminal. An example of a derivation beginning with the nonterminal pc is as follows:

\[
\begin{align*}
\text{pc} & \Rightarrow \text{env_designator}_1 \text{ DOT} \cdots \text{ DOT} \text{ env_designator}_n \text{ DOT} \text{ jp_designator}
\end{align*}
\]
2.3 Aspect Language Genericity

A generic grammar $G_A$ describing a family of aspect languages must define aspect-oriented abstractions with sufficient detail to enable the construction of a generic weaver while remaining sufficiently general in order to be adaptable to different base languages. This is a challenge because the specification of advice definition and pointcut expressions both reference the underlying base language whose elements are the subject of weaving.

In $G_A$, genericity is achieved by leaving several syntactic categories *undefined*. That is, $G_A$ contains syntactic categories whose abstract meaning is known to the weaver but whose structural details have not yet been concretely defined in terms of grammar productions. For example, in $G_A$ (initial) advice is defined generically in terms of the following production

$$\text{initial_advice ::= } \text{ADVICE [ INLINE ] id composition_type pc ADVICE_BODY}$$

leaving the nonterminals $\text{ADVICE}$, $\text{id}$ and $\text{ADVICE_BODY}$ undefined.

Uninstantiated syntactic categories serve two purposes: First, a weaving function defined in terms of uninstantiated syntactic categories will be generic by construction. Second, it provides syntactic flexibility when constructing $G_A \circ G_L$, e.g. for adapting the “look-and-feel” of the aspect language fragment to a particular DSL (see 2.4).

Note that we are talking here about genericity of the language specification, not of the language itself. A *generic aspect language* is an aspect language that provides genericity to its users by supporting meta-variables that range over syntactic categories of the base language (types, methods, fields, statements, etc.) [13]. In this paper we specify a class of non-generic aspect languages, but specify this class generically. Extending our technique to generic aspect languages is a matter of future research.

2.4 Instantiation of $G_A$

The resolution of undefined syntactic categories is deferred to the time when $G_A$ is composed with a particular DSL. It is through the definition of these categories that an *instance* $G'_A$ of $G_A$ is created.

The essential uninstantiated category in $G_A$ is $\text{ADVICE_BODY}$, which must be mapped to a category of $G_L$. The instantiation of $\text{ADVICE_BODY}$ reflects a major design decision, characterizing the nature of the advice code that can be composed with a join point during weaving. For example, when composing $G_A$ with a *block-structured DSL* whose grammar defines a block production, the following instantiation of $\text{ADVICE_BODY}$ may be appropriate:

$$\text{ADVICE_BODY ::= } \text{block}$$

In addition to the syntactic category $\text{ADVICE_BODY}$, the definition of $G_A$ shown in Figure 3 leaves uninstantiated all syntactic categories marked by capital spelling. These do not need to be mapped to categories of the base language but are simply placeholders for tokens that define the keywords and punctuation symbols of the aspect language.

By deferring their instantiation we can provide the aspect language with different “skins”. Although this does
not increase the expressiveness of the resulting DSAL it might be important for providing a “look-and-feel” that is familiar to users of the base DSL, lowering the acceptance barrier. For instance, in a DSL whose syntax resembles Java, C++ or C the following instantiations will make sense, among others:

```plaintext
ASPECT ::= "aspect"
BEGIN_ASPECT ::= "{
END_ASPECT ::= "}
LPAR ::= "(
RPAR ::= ")"
```

3 Syntax-Based Language Composition

The primary aim of our work is the automated synthesis of a grammar $G_A \circ G_L$, given just the generic grammar $G_A$ of an aspect language family, the grammar $G_L$ of a concrete DSL and some information about correspondences between elements of the different grammars but no knowledge about the semantics of the languages. In particular, the composition $G_A \circ G_L$ should be synthesizable without depending on the semantics of the DSL. We call this approach syntax-based language composition. The rationale behind a syntax-based restriction is that the incorporation of language semantics can raise complex weaving-related issues. Weaving is significantly eased if semantic issues can be factored out.

DSLs belonging to the domain of a generic weaving function must have certain aspect-oriented commonalities. It is with respect to these commonalities that generic weaving is defined. In our approach, commonality is captured through syntactic abstractions and is collected in two generic grammar fragments, $G_{JP}$ and $G_{W}$. The grammar fragment $G_{JP}$, discussed in Section 3.1, generically defines a fixed set of join point types and join point environment types. The grammar fragment $G_{W}$, discussed in Section 3.6, generically defines join-point/advice compositions.

3.1 Abstract Markers for Join Point Types

The generic aspect grammar $G_A$ specifies a fixed number of join point types and environment types but makes neither assumptions about the concrete keywords used nor about their semantics, that is the elements of $G_L$ that they designate (Figure 3 and Section 2.2).

The weaver $W_A$ however, must be able to match pointcuts against concrete elements of any DSL $L_L$ to which it should be applicable. Thus, it must possess an abstract representation of the join points and join point environments designated by a pointcut that is sufficiently general to be mapped to concrete elements of any $L_L$.

Our approach to solving this challenge is syntactic matching. That is, $W_A$ matches join points only based on the syntax tree of an artefact, because we cannot depend on the semantics of individual languages. The identification of particular types of join points and join point environments in the syntax tree is based on the occurrence of particular non-terminals in the derivation. Such non-terminals are called markers and their defining productions are called abstract marker productions.
A production rule such as “get marker ::= id” could be a trivial marker production indicating that id is a variable access join point (e.g., an r-value). However, marker productions that only have an identifier on the right-hand-side would strongly restrict the structure of join points that the weaver can identify. On the other hand, we cannot allow arbitrary right-hand-sides since the generic weaver needs to know at least from which of possibly many elements on the RHS to derive the join point marked by the production. We adopt a middle way by requiring that for each of the \( n \) join point types and \( m \) join point environment types that the generic weaver must support there is a specific marker production in \( G_L \) that has the structure

\[
\begin{align*}
\text{jp}_i \text{marker} & ::= \text{jp}_i \text{left} \text{id} \text{jp}_i \text{right} \quad \text{for } i = 1 \ldots n \\
\text{env}_j \text{marker} & ::= \text{env}_j \text{left} \text{id} \text{env}_j \text{right} \quad \text{for } j = 1 \ldots m
\end{align*}
\]

In each marker production \text{id} defines the identifier associated with the join point and the \text{jp}_i \text{left} and \text{jp}_i \text{right} categories generically capture syntactic context that may precede or follow the join point identifier. The same principle applies to the marker productions for join point environments.

\[
\begin{align*}
\text{jp}_1 \text{marker} & ::= \text{jp}_1 \text{left} \text{id} \text{jp}_1 \text{right} \\
\text{env}_1 \text{marker} & ::= \text{env}_1 \text{left} \text{id} \text{env}_1 \text{right} \\
\text{jp}_2 \text{marker} & ::= \text{jp}_2 \text{left} \text{id} \text{jp}_2 \text{right} \\
\text{env}_2 \text{marker} & ::= \text{env}_2 \text{left} \text{id} \text{env}_2 \text{right} \\
\text{jp}_3 \text{marker} & ::= \text{jp}_3 \text{left} \text{id} \text{jp}_3 \text{right}
\end{align*}
\]

Figure 4: The grammar fragment \( G_{JP} \) defines abstract join points and join point environments.

The resulting set of abstract marker productions forms the grammar fragment \( G_{JP} \) on which the matching functionality of the generic weaver \( W_A \) is based. For the aspect grammar fragment from Figure 3, \( G_{JP} \) is defined as shown in Figure 4. Note that \( G_{JP} \) models two join point environment types and three join point types but could easily be extended to model any number of join point environments and join points. In such a case, the weaver \( W_A \) would need to be extended accordingly. Both of these extensions are straightforward and could easily be mechanized.

### 3.2 Concrete Marker Productions

One of the things that must occur during the composition \( G_A \circ G_L \) is that the abstractions in \( G_{JP} \) must be appropriately linked with elements in \( G_L \). Such linking is facilitated if, in \( G_L \), join points and join point environments are identified in terms of concrete marker productions having the same structural characteristics of those in \( G_{JP} \). For example, the a variable access might have been directly defined in \( G_L \) in terms of the production \text{expr} ::= \text{id}. To re-express this in terms of a concrete marker production \( G_L \) would need to be modified as follows:

\[
\begin{align*}
\text{expr} & ::= \text{varaccess}_\text{marker} \\
\text{varaccess}_\text{marker} & ::= \text{varaccess}_\text{marker}_\text{left} \text{id} \text{varaccess}_\text{marker}_\text{right} \\
\text{varaccess}_\text{marker}_\text{left} & ::= \epsilon \\
\text{varaccess}_\text{marker}_\text{right} & ::= \epsilon
\end{align*}
\]

Currently, the task of “massaging” \( G_L \) in order to describe all join points and join point environments in terms of concrete marker productions is left to the language engineer. However, given sufficiently powerful tools that hide most of the complexities of grammars behind a “programming-by-example” metaphor, such a modification could be
done in the future even by users of a DSL. On a more technical side of things, a concern may be voiced that such grammar modifications may produce grammars that lie beyond the reach of fixed lookahead parsers (e.g., LL(k)). This concern is side-stepped by our generic weaver due to the fact that it makes use of GLR parsing technology. More specifically, the parser used by $W_A$ is capable of performing backtracking and therefore has arbitrary lookahead capabilities.

3.3 Linking Abstract and Concrete Marker Productions

If the generic weaver built on the basis of $G_{JP}$ is to be reusable, a challenge is to devise an approach for composing the abstract marker productions in $G_{JP}$ with the concrete marker productions arising within a specific DSL grammar $G_L$. Recall, that the behavior of the weaver $W_A$ is defined with respect to $G_{JP}$. A noteworthy property of $W_A$ is that it recognizes abstract join points and abstract join point environments in terms of grammar derivations rather than fixed term structures. For example, let $w$ denote a code fragment corresponding to a concrete join point in $G_L$.

If $G_{JP}$ and $G_L$ can be linked in such a way that, in the resulting grammar, the derivation of $w$ has the following form

$$
jp_{marker1} \Rightarrow jpl_{left} \ id \ jpl_{right} \Rightarrow w$$

then $W_A$ will recognize $w$ as a join point of the type $jp_{type1}$ associated to $jp_{marker1}$.

Let us take a closer look at an instance of this problem. In the example below, we want to establish a link between the abstract marker production (line 01) and the concrete marker production (line 02).

01 $jp_{marker} ::= jpl_{left} \ id \ jpl_{right} \in G_{JP}$
02 $dsl_{marker} ::= dsl_{left} \ id \ dsl_{right} \in G_L$

We use a technique we call grammar superimposition to establish such a link. In this example, the link established through grammar superimposition yields the following set of productions.

$$
dsl_{marker} ::= \quad pj_{marker}$$
$$jp_{marker} ::= \quad jpl_{left} \ id \ jpl_{right}$$
$$jp_{left} ::= \quad dsl_{left}$$
$$jp_{right} ::= \quad dsl_{right}$$

Note that this replacement preserves derivations that are possible from $dsl_{marker}$ to the join point identifier $id$, as well as derivations from $jp_{marker}$ to the join point identifier $id$. We would like to mention that an approach to linking that is exclusively based on substituting one marker production for another does not work. The primary reason for this is that (1) the abstract marker production must remain because this is what $W_A$ expects (this orients the direction of the substitution from concrete to abstract), and (2) a link specification is a mapping from abstract marker productions onto concrete marker productions.

Overall, separating abstract and concrete marker productions, rather than embedding abstract marker productions directly into the grammar of a DSL, serves several purposes. First, the mapping that connects abstract and concrete marker productions can be viewed as a specification of how join points will be treated in a given DSL by the
generic weaver $W_A$. Second, factoring out the embedding-related concern facilitates experimentation with mappings. Third, the separation of abstract and concrete marker productions enables linking to be performed automatically. Automated linking is important because mappings can be constructed where multiple abstract production markers are mapped onto a single concrete production marker. In such a situation, the manual embedding of abstract marker productions into the grammar of a DSL is somewhat non-trivial and error prone.

We define a linking specification $M_{link}$ as an onto function\(^2\) from abstract marker productions to concrete marker productions. An example of a linking specification having a one-to-one mapping and involving the grammar $G_{JP}$ is shown in Figure 5. The figure illustrates that the same DSL element (in this case, $dsl_2$) can play a double role, as a join point environment and a join point. This can occur in uniformly generic aspect languages [13] such as LogicAJ [21]. For instance, LogicAJ, treats classes as the join points of generic introductions and at the same time as the static environment of other join points.

\[
\begin{align*}
env_1\text{.marker} &::= env_1\text{.left} \text{ id} env_1\text{.right} \quad \rightarrow \quad dsl_1\text{.marker} &::= dsl_1\text{.left} \text{ id} dsl_1\text{.right} \\
env_2\text{.marker} &::= env_2\text{.left} \text{ id} env_2\text{.right} \quad \rightarrow \quad dsl_2\text{.marker} &::= dsl_2\text{.left} \text{ id} dsl_2\text{.right} \\
jp_1\text{.marker} &::= jp_1\text{.left} \text{ id} jp_1\text{.right} \quad \rightarrow \quad dsl_3\text{.marker} &::= dsl_3\text{.left} \text{ id} dsl_3\text{.right} \\
jp_2\text{.marker} &::= jp_2\text{.left} \text{ id} jp_2\text{.right} \quad \rightarrow \quad dsl_4\text{.marker} &::= dsl_4\text{.left} \text{ id} dsl_4\text{.right} \\
jp_3\text{.marker} &::= jp_3\text{.left} \text{ id} jp_3\text{.right} \quad \rightarrow \quad dsl_3\text{.marker} &::= dsl_3\text{.left} \text{ id} dsl_3\text{.right}
\end{align*}
\]

Figure 5: An example of a linking specification $M_{link}$ between $G_{JP}$ and a DSL

To complete the discussion of linking we would like to mention that like most aspect languages [12, 26, 3, 21] we assume a weaving model in which advice bodies are also subject to weaving, just like any other base language code\(^3\). Extending weaving to advice bodies does not require any special treatment in our approach due to the fact that advice bodies are defined in terms of the DSL grammar $G_L$. From the perspective of $W_A$, this makes advice bodies indistinguishable from other DSL code fragments.

### 3.4 A Grammar Composition Algorithm

Here we present an algorithm that, when given the instantiated aspect grammar fragment $G'_A$, the DSL grammar $G_L$, the join point grammar fragment $G_{JP}$, and a mapping specification $M_{link}$, will yield the composed grammar $G'_A \circ G_L$.

Let $G_L = (V_L, T_L, P_L, S_L)$ denote a context-free grammar describing the DSL $L_L$ where $V_L$ denotes the set of nonterminal symbols, $T_L$ denotes the set of terminal symbols, $P_L$ denotes the set of production rules, and $S_L$ denotes the start symbol.

\(^2\)Recall that a function $F : X \rightarrow Y$ is onto if and only if \(\forall y \in Y, \exists x \in X : F(x) = y\).

\(^3\)At this time of this writing, the treatment of this in AspectC++ has been identified as a known bug.
Let \( \mathcal{G}_{\mathcal{J}P} = (V_{\mathcal{J}P}, T_{\mathcal{J}P}, P_{\mathcal{J}P}, S_{\mathcal{J}P}) \) denote a context-free grammar abstractly describing a fixed set of join points and join point environments as discussed in Section 3.1. Let \( \mathcal{G}_A = (V_A, T_A, P_A, S_A) \) denote a context-free grammar describing the aspect language family \( \mathcal{L}_A \).

1. Input:
   (a) \( \mathcal{G}_L = (V_L, T_L, P_L, S_L) \) a grammar for a DSL \( \mathcal{L} \)
   (b) \( \mathcal{G}'_A = (V_A', T_A', P_A', S_A') \) an instance of \( \mathcal{G}_A \)
   (c) \( \mathcal{G}_{\mathcal{J}P} = (V_{\mathcal{J}P}, T_{\mathcal{J}P}, P_{\mathcal{J}P}, S_{\mathcal{J}P}) \)
   (d) \( M_{\text{link}} \) – a mapping from the set of abstract marker productions in \( \mathcal{G}_{\mathcal{J}P} \) to \( P_{\text{weavable}} \) where \( P_{\text{weavable}} \subseteq P_L \) is the set of concrete marker productions in \( \mathcal{G}_L \).

**Precondition:** Except for \( \text{id} \), assume that the symbols in \( \mathcal{G}_L, \mathcal{G}'_A, \) and \( \mathcal{G}_{\mathcal{J}P} \) are disjoint.

2. Construct:
   (a) \( P_{\text{initial}} = P_L \cup P_A' \cup P_{\mathcal{J}P} \)
   (b) \( P'_L = \text{build}(M_{\text{link}}, P_{\text{initial}}) \) where \( \text{build} \) is defined in Figure 6

3. Output: \( \mathcal{G}'_A \circ \mathcal{G}_L = (V_L \cup V_A' \cup V_{\mathcal{J}P}, T_L \cup T_A' \cup T_{\mathcal{J}P}, P'_L, S_L) \)

**Postcondition:** For all \( w \in L(\mathcal{G}'_A \circ \mathcal{G}_L) \), the parse tree for \( w \) contains join points and join point environments that can be recognized by the weaver \( \mathcal{W}_A \) in a fashion that is consistent with the specification \( M_{\text{link}} \).

```plaintext
build({(env_i.marker ::= env_i.left id env_i.right) \rightarrow (B ::= \alpha \ id \ \beta)}) \cup M_{\text{link}}, P) = build(\sigma(M_{\text{link}}), P')

where \( \sigma \) is the substitution \([B ::= \alpha \ id \ \beta] \rightarrow (env_i.marker ::= env_i.left id env_i.right)]\)
and \( P' = (P - \{B ::= \alpha \ id \ \beta\}) \cup \{B ::= env_i.marker, env_i.left ::= \alpha, env_i.right ::= \beta\} \)

build({(jp_i.marker ::= jp_i.left id jp_i.right) \rightarrow (B ::= \alpha \ id \ \beta)}) \cup M_{\text{link}}, P) = build(\sigma(R'), P')

where \( \sigma \) is the substitution \([B ::= \alpha \ id \ \beta] \rightarrow (jp_i.marker ::= jp_i.left id jp_i.right)]\)
and \( P' = (P - \{B ::= \alpha \ id \ \beta\}) \cup \{B ::= jp_i.context, jp_i.left ::= \alpha, jp_i.right ::= \beta\} \)

build({}, P) = P
```

Figure 6: A superimposition algorithm for linking abstract and concrete marker productions

The \( \text{build} \) algorithm in Figure 6 implements the superimposition principle informally explained in Section 3.3.

We would like to mention that the algorithm shown in Figure 6 has been implemented in the strategic programming language TL and grammar compositions of the form \( \mathcal{G}'_A \circ \mathcal{G}_L \) for both the examples given in Section 4 have been
automatically generated using this algorithm. This has facilitated experimentation with grammar construction and linking. We would also like to mention that, though we have not done so, checking the disjointness property (explained below) could also be automated.

### 3.5 Composition Constraints

There are two syntactic properties the grammar $G_L$ of a DSL must satisfy in order for the composition $G_A \circ G_L$ to be syntactically well-defined and weavable by $W_A$. First, the concrete marker productions in $G_L$ must share structural characteristics with the abstract marker productions in $G_{JP}$ as discussed in Section 3.2. Second, the grammars $G_A$, $G_{JP}$, and $G_L$ should have a certain degree of disjointness.

A sufficient (though not necessary) disjointness property, in order for $G_A \circ G_L$ to be syntactically well-formed, is that the grammars $G_A$, $G_{JP}$ and $G_L$ be strictly disjoint, except for the id production defined in $G_L$.

This constraint can be formally expressed as follows. Let $G_L = (V_L, T_L, P_L, S_L)$ denote a context-free grammar describing the DSL $L_L$ where $V_L$ denotes the set of nonterminal symbols, $T_L$ denotes the set of terminal symbols, $P_L$ denotes the set of production rules, and $S_L$ denotes the start symbol. Let $G_{JP} = (V_{JP}, T_{JP}, P_{JP}, S_{JP})$ denote a context-free grammar abstractly describing a fixed set of join points and join point environments as discussed in Section 3.1. Let $G_A = (V_A, T_A, P_A, S_A)$ denote a context-free grammar describing the aspect language family $L_A$.

1. **Constraints on Nonterminals**
   - (a) $V_A \cap V_L = V_A \cap V_{JP} = V_{JP} \cap V_L = \{ \text{id}, \text{num} \}$

2. **Constraints on Terminals**
   - (a) $T_A \cap T_L = T_A \cap T_{JP} = T_{JP} \cap T_L = \emptyset$

3. **Constraints on Productions**
   - (a) $\{ \text{id} ::= \ldots, \text{num} ::= \ldots \} \in P_L$
   - (b) $\{ \text{final_advice} ::= \ldots \} \in P_W$
   - (c) $P_A \cap P_L = P_A \cap P_{JP} = P_{JP} \cap P_L = \emptyset$

Note that in practice it is possible for a variety of nonterminals to be shared between $G_A$, $G_{JP}$, and $G_L$ without causing problems. For example, keyword-like terminals such as “{” and “}” can be shared without impacting $W_A$. However, the formal characterization of such sharing introduces some complexity into the discussion and is tangential to the topic of this paper. For this reason it has been omitted from this discussion.

### 3.6 Weaving

Join points participate in the weaving process in two ways: (1) a pointcut can be matched with a join point, and (2) a join point can be composed with advice. The use of join points in matching is captured through the abstract/concrete
marker productions and their superimposition described previously. How join point participate in a composition is
described next.

A general requirement for any weaver is that it be language-preserving in the sense that the woven code that it
creates is still in the base language $L$. If this were not the case the woven code could not be handled anymore by
existing interpreters or compilers for $L$. This presents a significant challenge to generic weaving and cannot be fully
resolved within the framework of a syntax-based approach to generic weaving.

Still our generic weaver can perform several useful operations that ease later steps. In particular, it can capture
the distinction between before, after and around advice, as well as the distinction between inlined weaving and
forwarding-based weaving. It maps them all to a single composition operator, $[+]$, which itself is a language construct
falling outside of the original DSL $L$.

Figure 7: The Grammar $G_W$

Figure 7 shows the grammar fragment that describes abstractly the syntax of the intermediate language that is
the target of weaving.

The use of join points in composition is modelled using context productions (line 1 – 3). In particular, each
abstract marker production modelling a join point is accompanied by a context production of the form:

\[
\text{jp}_i \text{context ::= \text{jp}_i \text{marker} | \text{jp}_i \text{composition}}
\]

Context productions provide the syntactic context in which the replacement of a join point by an advice/join
point composition can occur. This replacement is the essence of weaving.

The composition rules capture the implementation of before, after and around advice by different argument orders for the composition operator (line 5 – 15).

Each of these three forms has two sub-forms: (1) the composition contains a reference to a final advice definition (line 17), which we call forwarding, or (2) the composition explicitly contains the advice body (line 18), which we call inlining.

Final advice (line 20) is the definition of an advice method that remains after pointcut matching and interpretation of inlining and composition_type directives (see Figure 3 in Section 2).

Advice/join point compositions are defined in terms of concrete symbols (i.e., terminal symbols – line 22). The rationale behind this is that WA must generate the concrete representation of compositions during the weaving process. However, it is relatively straightforward to parameterize WA on the set of concrete symbols it should use when generating compositions. We have implemented a prototype of such a weaver, but its discussion lies beyond the scope of this article.

WA expresses advice/join point compositions in a form we call a residual (line 5 – 15). For example, let jp denote a join point and let refadvice denote a reference to an advice element having bodyadvice as its body. Figure 8 shows the set of possible residuals that can be created involving jp, refadvice, and bodyadvice.

<table>
<thead>
<tr>
<th>forwarding before</th>
<th>produces the residual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[ refadvice [+] jp ]</td>
</tr>
<tr>
<td>inlining before</td>
<td>produces the residual</td>
</tr>
<tr>
<td></td>
<td>[ bodyadvice [+] jp ]</td>
</tr>
<tr>
<td>forwarding after</td>
<td>produces the residual</td>
</tr>
<tr>
<td></td>
<td>[ jp [+] refadvice ]</td>
</tr>
<tr>
<td>inlining after</td>
<td>produces the residual</td>
</tr>
<tr>
<td></td>
<td>[ jp [+] bodyadvice ]</td>
</tr>
<tr>
<td>forwarding around</td>
<td>produces the residual</td>
</tr>
<tr>
<td></td>
<td>[ refadvice ]</td>
</tr>
<tr>
<td>inlining around</td>
<td>produces the residual</td>
</tr>
<tr>
<td></td>
<td>[ bodyadvice ]</td>
</tr>
</tbody>
</table>

Figure 8: The Set of Possible Residuals Involving jp, refadvice, and bodyadvice.

A language dependent post-weaving phase is needed to translate residuals into a form belonging to L-len. Depending on the constructs provided by L-len as well as the desired semantics of residuals, a post-weaving phase may be straightforward or complex.

3.7 Constructing the Target Language

The grammar fragment G-W just describes an abstract view of the intermediate language that is the target of weaving. For a given DSL’s grammar G-len it is necessary to produce a variant of G-len that incorporates the rules of G-W.

This is easy, given that the superimposition of G-len and G-len performed by the build algorithm of Figure 6 already takes care of providing the necessary hooks. It creates rules that already contain references to the jp_context
productions that are the entry points into the $G_W$ fragment. Thus, the construction of a grammar for the intermediate language reduces to creating the union of $G_W$ with the result of superimposing $G_{JP}$ on $G_L$.

4 Application

In this section we give two examples of DSLs. For each DSL $G_L$ we describe how the composition $A \circ L$ can be achieved.

4.1 Composing $A$ with a Vanilla DSL

The example shown here involves a composition of $G_A$ with a small imperative DSL that has been extended with a minimal form of classes. We call this DSL $V$. In the realm of DSLs, $V$ is not particularly interesting. However, $V$ serves as a good vehicle for showing the aspect-oriented features that are present in the language resulting from the composition $G_A \circ G_V$.

A grammar $G_V$ defining the syntax of the DSL $L_V$ is given in Figure 9. Note that in $G_V$ all join points and join point environments are (already) defined in terms of concrete marker productions. In order to apply $L_A$ (whose syntax is given in Figure 3) to $L_V$, we must do the following:

1. A specification $M_{link}$ must be constructed linking the abstract marker productions in $G_{JP}$ (Figure 4) with the concrete marker productions in $G_L$. Such a linking specification is shown in Figure 10.

2. An instance $G'_A$ of $G_A$ must be created in which the mapping of advice bodies to blocks has been specified by adding the production `ADVICE_BODY ::= block` (Figure 11) and all syntactic categories relating to “look-and-feel” have been defined. The latter definitions are given in Figure 12. The instance $G'_A$ is obtained by adding the productions shown in Figure 11 and 12 to the productions of $G_A$.

3. Apply the build algorithm (described in 3.4) with inputs $G'_A$, $G_V$, $G_{JP}$, and $M_{link}$ to produce the composition $G'_A \circ G_V$.

4. Create the union of $G'_A \circ G_V$ with $G_W$ to obtain a specification of the language that is the target of weaving $L_{G'_A \circ G_V}$ programs.

From the perspective of aspect-orientation, the highlights of the DSAL $L_{G'_A \circ G_V}$ are as follows:

1. Class definitions are treated as join point environments (i.e., a class environment can be referenced by a pointcut and is taken into account by the pointcut matching algorithm). Within a pointcut, the designator `in_class` denotes this type of join point environment.

2. Method definitions are treated as join point environments. Within a pointcut, the designator `in_method` denotes this type of join point environment.
3. Lvalues are treated as join points. Within a pointcut, the designator set denotes this type of join point.

4. Rvalues are treated as join points. Within a pointcut, the designator get denotes this type of join point.

5. Method calls are treated as join points. Within a pointcut, the term call denotes the type of this join point.
6. Advice/join point compositions are expressed in terms of residuals (see Section 3.6).

7. Aspects are unaffected by the weaving process and remain in the (tangled) program output by the weaver. We could have easily extended our approach to weaving so that, in this case, aspects would be transformed into classes and advice would be transformed into method calls. However, experimentation leads us to believe that such minor changes are better left to the post-weaving process which is needed to handle residuals.

Figure 13 gives a concrete example showing the results of weaving for the DSAL $G'_A \circ G_V$ using the weaver $W_A$.

![Table showing Original Form and Woven Form](image)

Figure 13: A Weaving Example for the DSAL $L_{A\circ V}$

### 4.2 A Requirements Specification DSL

We now illustrate the composition with a nonprocedural language. We have designed a DSL for specifying requirements for software product lines [28, 6]. A product line is a set of products having a common base. Specifying requirements for a product line involves developing a customizable family of requirements that can be instantiated for a given product. Customization is needed for many reasons. For example, a certain subset of products may need some special requirements. Conversely, a requirement may apply to all but a few products. Furthermore, some other requirements may be different for every product. The specification of these customizations in the product line are expressed via parameters and aspects. Creating a product-specific set of requirements is done by parameter instantiation and aspect weaving. The original language and the issues surrounding its creation are discussed in [24].

In this article, we show how we modified it to make it suitable for composition with the generic aspect language $G_A$. 

22
We call the base language $\mathcal{R}$. In the base language, requirements are written in natural language sentences. Language constructs are used to group together related requirements. Requirements can be organized hierarchically through these groupings. These requirements are also organized into viewpoints. (A viewpoint provides a list of requirements that are of concern to a particular stakeholder.) Requirements can also be parameterized to make them more general. Figure 14 shows a part of the grammar of $\mathcal{L}_\mathcal{R}$ that is relevant for our purposes.

![Figure 14: The Syntax of $\mathcal{R}$](Image)

The specification of natural language requirements into viewpoints and requirements has also been done elsewhere, e.g., Arcade [19]. While Arcade uses XML, it is possible to perform a straightforward transformation from XML to $\mathcal{L}_\mathcal{R}$.

Note that viewpoints and requirements have identifiers. Viewpoints are identified by their corresponding stakeholder. On the other hand, the requirement identifier is a tag which describes the requirement. Such a tag might be systematically derived from a semantic analysis of the requirement text as in [5]. Aspect composition is performed
primarily by matching pointcuts with viewpoint and requirement identifiers.

The grammar $G_R$ defining the syntax of the DSL $L_R$ is given in Figure 14. Note that in $G_R$ all join points and join point environments are defined in terms of concrete marker productions. In order to apply $L_A$ (whose syntax is given in Figure 3) to $L_R$ we follow the same sequence of steps that were used in the previous example to construct $G'_A \circ G_V$. An interesting thing to note is that the $M_{\text{link}}$ specification in Figure 15 maps both an abstract join point environment and an abstract join point onto the concrete marker production for requirement_def.

From the perspective of aspect-orientation, the highlights of the DSAL $L_{G'_A \circ G_R}$ are as follows:

1. Viewpoint definitions are treated as join point environments. Within a pointcut, the designator in_viewpoint denotes this type of join point environment.

2. Requirement definitions are treated as join point environments. Within a pointcut, the designator in_req denotes this type of join point environment.

3. Requirement definitions are also treated as join points. Within a pointcut, the designator req_jp denotes this type of join point environment.

4. Parameter references are treated as join points. Within a pointcut, the designator par_jp denotes this type of join point environment. This enables the weaving of advice to replace the parameter with an actual value.

Figure 18 gives an example showing the results of weaving for the DSAL $L_{G'_A \circ G_R}$.

5 Comparison to AspectJ

Although AspectJ is certainly not a domain-specific aspect language, comparing our approach to AspectJ is instructive nevertheless. It sheds light on the challenges that several known aspectual abstractions pose to syntax-based grammar superimposition. These insights are of value for language engineers considering similar abstractions in their DSAL. When compared to an industrial-strength language such as AspectJ [12, 11], notable restrictions on $A$ include a lack of support for the following features:

1. unparameterized pointcut abstractions

2. boolean operators (e.g., $||$, $&&$, $!$) within pointcut expressions,

3. the wildcard “..”,

4. declaration of fields within an aspect,

5. parameterized pointcut abstractions,

6. pointcut matches based on type (i.e., signatures),
The system must be able to retrieve a customer upon demand.

The system must allow a customer to update personal profile information.

The system must provide a way for authorized employees to lookup a customer personal health record.

The retrieval process must take less than five seconds.

The system must allow a customer to update personal profile information.

The retrieval process must take less than five seconds.

The system must provide a way for authorized employees to lookup a customer personal health record.

The system must be able to retrieve a customer personal health record upon demand.

The retrieval process must take less than five seconds.

The system must allow a customer to update personal profile information.

The retrieval process must take less than five seconds.

The aspect language family $\mathcal{L}_A$ presented can be easily extended to include the unsupported items 1, 2, and 3 listed above. Support for item 4 is also not particularly difficult. However, items 5 – 9 reach more deeply into the semantics of the DSL $\mathcal{L}_L$ with which $\mathcal{L}_A$ is to be composed and their inclusion in $\mathcal{L}_A$ is nontrivial.

6 Related Work

Gray and Roychoudhury [7] have considered the idea of using transformation as a way to develop industrial-strength weavers for languages used in legacy systems. They identify the development of language and platform-independent weaving as a long-term goal. To demonstrate the benefits of aspect-orientation in the context of legacy systems, they
have implemented a transformation-based weaver in DMS [2] for a commercial distributed application implemented in ObjectPascal. The approach taken was to identify and remove crosscutting concerns from the legacy code and re-implement these as aspects. The restructuring resulted in a significant improvement to the code base.

In [10], an interesting variation on the relationship between transformation and aspect-orientation is explored. In particular, a DSAL called AspectStratego is developed for the strategic programming language Stratego [27]. Domain-specific crosscutting concerns that have been identified in the context of term rewriting include origin tracking and rewriting with layout. A consequence of the idea of aspect-oriented extensions to transformation systems, such as the Stratego system, is that a transformation-based weaver can be itself specified within an aspect-oriented framework. This may have interesting implications with respect to how the goals of generic weaving could be expressed within a transformational framework.

In [22], an approach is described in which transformation-based weaving is based on metamodels describing both an aspect language and a language used by a particular transformation engine. The goal is to create a framework in which issues surrounding weaver construction can be separated from idiosyncrasies of the underlying transformation engine, thereby sidestepping the “accidental complexities” associated with a particular transformation system. In effect, a transformation-based “compiler” written in ATL (ATLAS Transformation Language) is used to translate aspects into transformations that when applied to a base language program realize the appropriate weaving actions. In this setting, the use of a metamodel describing an aspect language is similar to our use of $G_A$ to describe the aspect language family $L_A$.

In [8], a syntax-based approach to generic weaving is described based on the Reuseware Composition Framework, a tool for engineering context-free grammars and executing compositions (i.e., rewrites). The foundation of the Reuseware system is based on the ideas of invasive software composition, an approach to composition of program fragments (also called partial programs) based on hooks, which are abstractly seen as the variation points of a program fragment. Aspect-oriented concepts can be generically modelled in Reuseware using three variation points: slots, hooks, and anchors. These are similar to the marker productions we use. In Reuseware, the modelling of variation points requires language extension, a non-oblivious technique that makes variation points explicit through the introduction of concrete symbols (i.e., terminals) into the core language (which we call the base language). In contrast, our approach to such identification takes place at a more abstract level and does not require the insertion of terminal symbols into the source program in order to recognize join points and join point environments (i.e., thus obliviousness is achieved in our approach). In Reuseware, weaving is realized through the binding of slots and hooks. The quantification associated with weaving is achieved via fragment queries which is also a mechanism for modularizing program fragments. Advice/joint point composition is performed in a language dependent fashion using a composition function which appears to be syntax-based.

ELIDE [4] it a tool that supports explicit programming in Java. When compared to AOP, a distinguishing feature of explicit programming is that join points are not matched by pointcuts. A pointcut language is not needed.
Instead, join points, which correspond to points in the program text, are decorated with modifiers. Modifiers provide a mechanism for modularizing non-local design concepts. The semantics of modifiers is transformation-based and defined in terms of Java classes. This allows modifiers to be used in the implementation of other modifiers.

XAspects [23] is an extensible system in which domain-specific aspect and component languages can be integrated within a framework based on AspectJ. DSLs and DSALs are implemented as plug-ins that are extensions to AspectJ. Weaving semantics is defined with respect to the Java bytecode representation of an artifact. This enables weaving semantics to be very expressive. The price that must be paid is that the development of a plug-in is a non-trivial task.

The Mjølner System [16, 15] is a development environment for the BETA programming language. An important component of the Mjølner System is the Fragment System which is responsible for managing modularization of BETA programs. In the Mjølner System, constructs supporting modularization are seen as being orthogonal to the constructs of BETA. As a result, modularization is expressed using a special-purpose language called the fragment language. Conceptually speaking, the fragment language extends the BETA syntax thereby enabling BETA programs to be represented as a collection of modules called fragments. This fragment-based modularization centers around two concepts: (1) forms, and (2) slots. The Fragment System itself can be seen as having a weaver-like functionality capable of converting a fragment-based representation of a BETA program into a whole (unfragmented) BETA program.

Fragment-based modularization serves several purposes: It is a decomposition mechanism facilitating program understanding, and it provides a means for partitioning a program into independently compilable units. With respect to program understanding, fragment-based modularization can be used for a variety of purposes including: (1) separation of interfaces and implementations, (2) information hiding, and (3) AOP-like program composition/decomposition [14]. In the discussion that follows, the fragment language is described with an eye towards aspect-orientation.

Conceptually speaking, there are three essential pieces of information associated with a fragment: (1) the name, \( Id \), of the fragment, (2) the syntactic category, \( A \), of the fragment, and (3) the body, \( ff \), of the fragment which is also called a form. Let \( L \) denote a programming language (such as BETA) and \( G_L \) denote a grammar defining the syntax of \( L \). In order for a fragment \((Id, A, ff)\) to be well-formed, the derivation \( A \Rightarrow ff \) must be possible in \( G_L \).

There are two essential pieces of information associated with a slot: (1) a fragment name, \( Id \), and (2) a syntactic category \( A \). When combined, the tuple \((Id, A)\), called a slot, uniquely identifies a fragment.

Conceptually speaking, a fragment \( F_1 \) can be decomposed into two smaller fragments, \( F'_1 \) and \( F_2 \), as follows:

1. Select a portion \( ff \) of source code belonging to \( F_1 \).

2. Create a new fragment \((F_2, A, ff)\), where \( F_2 \) is the fragment name and \( A \Rightarrow ff \).

3. Obtain \( F'_1 \) from \( F_1 \) by replacing \( ff \) with the slot \((F_2, A)\).

\[4\]BETA is a strongly-typed object-oriented language originating from the Scandinavian school of object-orientation whose ancestry includes SIMULA.
From an aspect-oriented perspective, one can consider the fragment set \( \{F_1\} \) to represent the tangled form of the program whose source is the body of \( F_1 \). Similarly, one can consider the fragment set \( \{F'_1, F_2\} \) to represent the untangled form of the program. It is this observation that provides the basis for attributing a weaver-like functionality to the Fragment System. While such an interpretation is valid, it is worth analyzing the similarities as well as the differences between weaving within the Fragment System and the understanding of weaving (e.g., AspectJ) that is commonly held within the aspect-oriented community.

In the fragment language, join points are modelled by slots and must be made explicit within the program. Advice is modelled by fragments. The information present in slots and fragments subsume the need for a pointcut language, but as a result also correspondingly impact the functionality of weaving. Though the forms-based theory underlying fragments does not impose such a restriction, the Fragment System, as currently implemented, only supports a 1-1 relation between slots and fragments. This means that fragments cannot “crosscut” programs. This also admits situations resulting in unnecessary duplication of code\(^5\).

When comparing the fragment language to our approach it is important to make a clear distinction between (1) the function of _grammar composition_ and (2) the function of _weaving_. The goal of grammar composition is to combine a domain-specific language (DSL) with a general-purpose aspect language (GPAL). In this article, we do not present or view grammar composition as a form of weaving. Rather, we see weaving as a function that can be performed on source code programs belonging to a given grammar composition (i.e., weaving is performed on \( p \) where \( p \in DSL \circ GPAL \)). Nevertheless, when viewed abstractly, grammar composition and weaving have some similarities. As a result, the functionality of the fragment system can be compared to and contrasted with both (1) the functionality of grammar composition, as well as (2) the functionality of weaving.

From the perspective of weaving, our work is built on a theoretical foundation having similarities to the theoretical foundation underlying the fragment language. In particular, both approaches are based on sentential forms arising within a context-free grammar. However, rather than requiring our join points to be explicit, we instead embed them in the parse-tree of the source program. This is done at the BNF-level and therefore creates a join point model for all source code programs (even for base-code programs that were written prior to this grammar composition). A BNF-level composition is accomplished by linking abstract and concrete marker productions through a process we call grammar superimposition. The semantics of our composition mechanism is left unresolved and in return our weaver is applicable to not just one language, but to all languages belonging to a particular language family.

A comparison can also be made between our approach to grammar composition and the fragment language. Specifically, consider the situation where grammars themselves are viewed as the objects of modularization. In this case, meta-slots would be places in a grammar where forms (i.e., concrete productions) could be inserted. However, there are some problems with such a comparison. For example, it is unclear how to model, in the fragment language, situations where multiple join point types and/or join point environment types are mapped to the same concrete

\(^5\)It is conjectured that such situations are rare and somewhat artificial in nature.
production in the grammar.

7 Conclusion

In this paper, a syntax-based approach is presented in which aspect-orientation can be added to domain-specific languages. The approach makes minor assumptions about the syntax of the DSL being targeted and thus is essentially language independent. A key part of the approach is that join point and join point environments are abstractly modelled via marker productions. This enables grammar superimposition to be used to align such models with corresponding models belonging to the target DSL. A significant portion of the grammar composition process is automated.

A weaver function can be specified with respect to abstract marker productions, and as a result is applicable to any DSL on which such productions can be superimposed. Though not discussed in great detail here, a prototype of such a generic weaver has been implemented in TL and is described in [31]. Because abstract weaving is possible under these circumstances, incorporating aspect-orientation into the family of DSLs described in this article reduces to a grammar modification problem. Our experiences with industrially funded projects lead us to believe that such skills lie within the reach of technically-oriented domain experts. As a result, we believe that the approach described in the paper makes the exploration of extending DSLs with aspect-orientation practical.

Because weaving is syntax-based and generic, issues remain concerning semantic well-formedness of advice/join point compositions. At present, such issues must be addressed in a post-weaving stage.

References


