Model-based Aspect Weaver Construction

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

[1] University of Alabama at Birmingham, Birmingham, AL 35294
{roychous, gray}@cis.uab.edu

[2] ATLAS group (INRIA & LINA), University of Nantes, France
frederic.jouault@univ-nantes.fr

Abstract. This paper describes an approach that combines model engineering with program transformation techniques to construct aspect weavers for general-purpose programming languages. The front-end of the weaver uses a high-level language (i.e., an aspect language) to specify aspects and is designed with a metamodel-based approach using the AMMA toolsuite. The back-end of the weaver uses transformation rules, which are generated from these high-level aspect specifications, to perform the actual low-level weaving. Both the aspect specification (front-end) and transformation rules (back-end) are captured as models within the AMMA platform. The translation from source aspect model to target transformation model is achieved using ATL, which is a model transformation language.

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

1 Introduction

Program transformation systems have been used in the past to construct aspect weavers for programming languages [1]. Program transformation engines (PTEs) such as the Design Maintenance System (DMS) [2], or ASF+SDF [3], provide direct availability of scalable parsers and an underlying low-level transformation framework to modify source programs. This low-level transformation framework (that generally uses term-rewriting or graph-rewriting techniques to modify base code) can form the basis of constructing aspect weavers [4]. In our previous research, we demonstrated how a term-rewriting PTE like DMS can be used to construct an aspect weaver for a general-purpose programming language (GPL) like ObjectPascal [1]. However, a PTE-based weaver construction process raises new challenges and faces inherent accidental complexities; i.e., the rewrite rules used to modify base programs are difficult to compose, which makes it accessible to only language researchers and is generally hard to comprehend by average software developers. Moreover, the rewrite rules are often tied to the grammar of the source language (e.g., ObjectPascal), which impedes reusability when this language changes. In addition, the entire weaver is rendered unusable if one switches to a new transformation engine during weaver evolution.

1.1 Motivation and Approach

To eliminate some of the accidental complexities associated with PTEs, but still leverage the power of such systems, this research investigates a relatively new approach to constructing aspect weavers for GPLs. Specifically, the approach uses a layered architecture and combines model engineering with program transformation techniques to construct aspect weavers. Figure 1 presents an overview of the model-based weaver construction process.

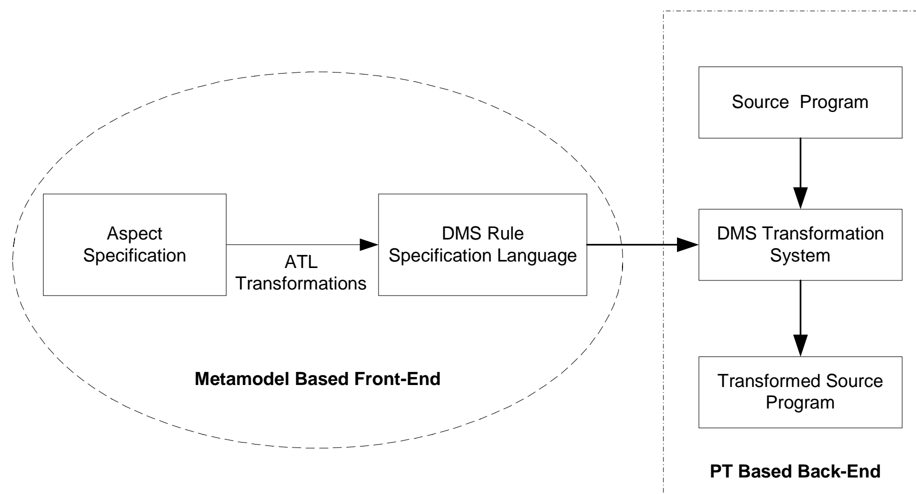

Figure 1. Overview of Model-Based Weaver Construction

The front-end or top layer of the weaver uses a high-level aspect language similar to AspectJ [5], which is then translated to low-level transformation rules (e.g., the DMS Rule Specification Language - RSL) used by the back-end of the system. The translation from the source aspect specification to the target RSL specification is performed using ATL (ATLAS Transformation Language), which is a model transformation language [6] and constitutes the heart of the constructed weaver. The translated rewrite rules are then processed by DMS to weave aspects into source programs.

Both the source aspect language (front-end) and target transformation rules (back-end) are captured as metamodels within the AMMA (ATLAS Model Management Architecture) toolsuite, which is an open source model engineering toolkit available on Eclipse.org (http://wiki.eclipse.org/index.php/AMMA). AMMA offers KM3 (Kernel Meta Meta Model) to describe the abstract syntax of the source and target metamodels [7]. The concrete syntax is specified in another notation, namely, TCS (Textual Concrete Syntax) [8]. The following section discusses the source aspect language model and the target RSL model in more detail.

2 The Source and Target metamodel

The core elements of a metamodel for an aspect language are: join points, pointcuts, and advice that form the basis of a Join Point Model (JPM) – please note that this definition is influenced by AspectJ [5]. Join points are execution points of a program, such as a method invocation or a particular exception being thrown. Pointcuts are means of identifying a set of join points through a predicate expression. Advice defines actions to be performed when a particular join point is reached. A fragment of the source metamodel to describe the high-level aspect language is shown in Figure 2. The bottom part of this figure gives an excerpt of the KM3 specification, whereas the top part shows how the corresponding Ecore metamodel is rendered by the EMF (Eclipse Modeling Framework) metamodel editor.

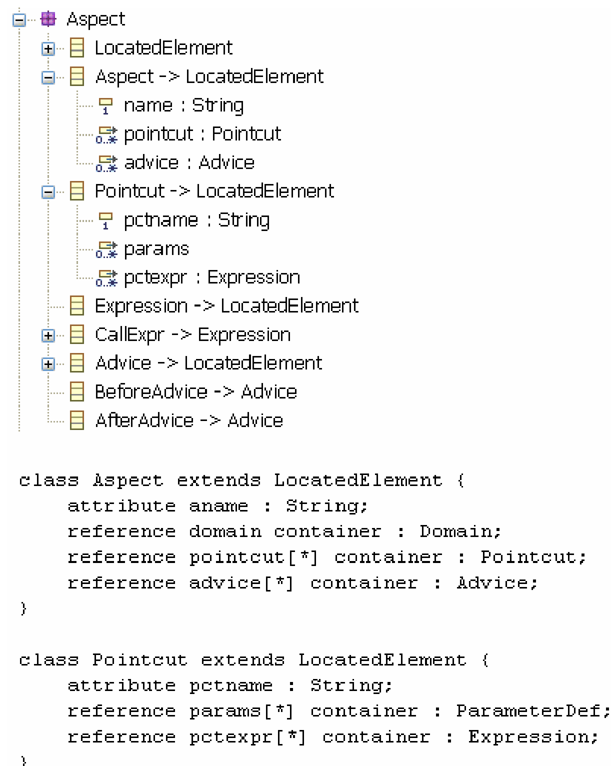

Figure 2. A metamodel excerpt of the source aspect language

In accordance to the definition of JPM, aspects defined in this metamodel contain multiple pointcuts or advice, such that every pointcut is comprised of several pointcut parameters (params) and pointcut expressions (pctexpr). The "*" symbol is used to denote unbounded multiplicity (sometimes noted 0..n). A pointcut expression can be further expressed as a new type (i.e., a call expression, an execution expression, or a cflow expression). Similarly, using the inheritance property, an advice can be defined as a *BeforeAdvice*, *AfterAdvice* or *AroundAdvice* and may contain advice parameters and advice statements. The keyword "container" is used to define the hierarchical

relationship between parent and child members. The keyword “reference” denotes that a property is defined as a reference to another type. As an example, notice that the property “pointcut” is defined as a reference to a new type (*Pointcut*), which in turn contains references to other types (*ParameterDef* and *Expression*).

In contrast to KM3, which specifies the hierarchy or relationship (class structure, or abstract syntax) between different model elements, TCS is used to specify the concrete syntax for the same metamodel. The corresponding snippet of a TCS specification to describe the concrete syntax of the aspect language is shown in Listing 1. TCS uses a grammar-like notation and describes the syntax of the source language. Listing 1 shows how different model elements like *Aspects*, *Pointcuts* and *Advice* are organized in terms of a syntax tree for the source aspect language.

```
template Aspect main
     :      "aspect" name "{" pointcut advice "}"
     ;
template Pointcut
     :      "pointcut" "(" params {separator = ","} ")"
             pctname ":" pctexpr {separator = "&&"} ";"
     ;
```

Listing 1. An excerpt of the TCS specification for source aspect language

The target metamodel in our architecture is the DMS Rule Specification Language (RSL) [2]. RSL provides basic primitives for describing numerous transformations that are to be performed across the entire code base of an application. It consists of declarations like patterns, rules, conditions, and rule sets. Patterns describe the form of a syntax tree. They are used for matching purposes to find a syntax tree having a specified structure. 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. Rules can be combined into rule sets. The patterns and rules can have associated conditions that describe restrictions on when a pattern legally matches a syntax tree, or when a rule is applicable on a syntax tree. Figure 3 gives a snippet of the RSL metamodel under consideration. The right-hand part of this figure presents a KM3 excerpt, whereas the left-hand part shows how the corresponding Ecore metamodel is presented by the EMF metamodel editor.

Primary Benefit: An advantage of decoupling the source aspect model from the target RSL model is that even by replacing the existing RSL model with a different one (e.g., the rewrite engine model used by ASF+SDF), the source aspect model need not be altered. Thus, the weaver construction process is itself modularized and can reuse the same source aspect model for different target platforms (i.e., rewrite specifications used by different PTEs). In this sense, the concept of platform-independence promoted by Model-Driven Engineering (MDE) is realized through PTE-independence.

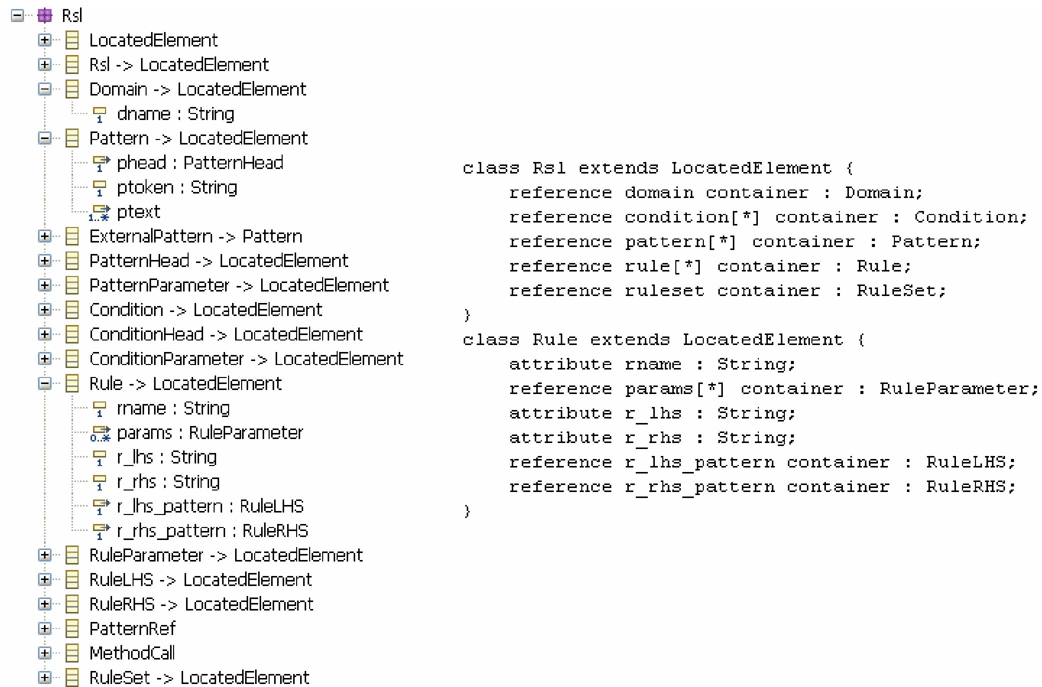

Figure 3. A metamodel excerpt for the Rule Specification Language (RSL)

3 Translation Scheme

As stated in Section 1.1, the core of the approach consists of ATL transformations that appropriately translate a given aspect specification to its corresponding rule specification. Figure 4 describes the high-level overview of the metamodel-based transformation approach.

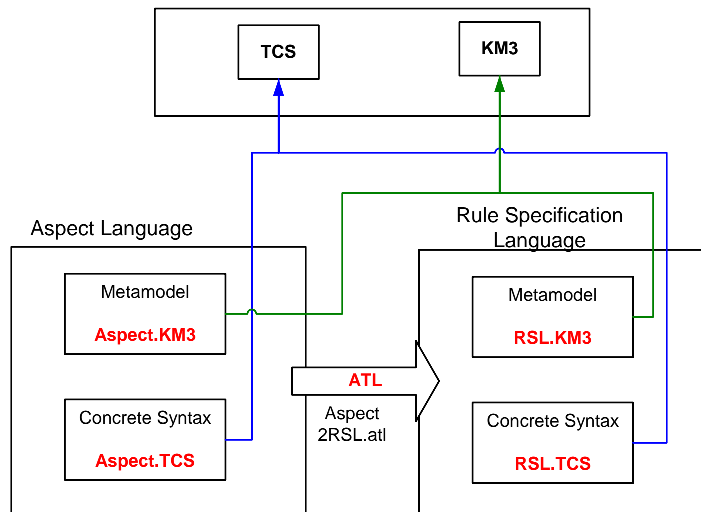

Figure 4. Overview of metamodel-based transformation

In Figure 4, the source aspect specification that conforms to a source aspect metamodel is transformed into an RSL specification that conforms to a target RSL metamodel. Both the source and target languages are defined using KM3 (metamodel or abstract syntax) and TCS (concrete syntax) as shown in Section 2 of the paper. The translated RSL rules are then processed by DMS to perform the actual weaving for a source program written in a specified GPL.

3.1 Case Study – A Real Language

For experimental discussion, we choose Delphi (ObjectPascal) as our source GPL. Delphi was chosen based on our past experience in building a Delphi aspect weaver using DMS [1]. To build an aspect weaver for Delphi using the approach presented in Figure 4, the architecture needs to be slightly adapted. It may be noted that the aspect metamodel may somewhat differ from one GPL to another. For example, new pointcuts can be defined for new language constructs that exist in a particular GPL (e.g., "with" construct in Delphi), which may not be present in another GPL. Although the majority of the join points or pointcut declarations are similar for a particular programming paradigm (e.g., objects), there may be slight variations among the languages concerned. Hence, the first design decision in defining a source aspect model is to make the metamodel as general as possible for the particular programming paradigm. When deciding to construct a weaver for a particular GPL, one can inherit from the base aspect metamodel, and add new model elements that belong to that GPL, or use the existing features already available in the base metamodel. Doing this relies on metamodel extension [9]. For our Delphi weaver, new pointcut definitions that only belong to Delphi were added as extensions to the base aspect metamodel. Aspect weavers for other languages can similarly inherit from the base aspect metamodel and add their respective extensions if necessary. However, the target metamodel need not be altered as every source specification belonging to any GPL may be translated to a rewrite specification that conforms to the same target metamodel (i.e., the PTE metamodel). Conversely, changing the target PTE does not require changing the source metamodel.

In this case study several aspects as identified in a Delphi (Object Pascal) application were selected for experimentation, which are borrowed from [1]. One such aspect represents the updating of a progress-meter in a database (DB) schema manager application. As several evolution tasks (e.g., deleting database triggers, compiling new stored procedures) were executed in the DB schema manager utility, the processing dialog meter was required to be updated accordingly. Listing 2 shows the Delphi code that was necessary to update the progress meter – this code appeared in 62 different places in the schema manager utility [1]. Lines 2-7 show the redundant code that was executed after each call to the *Increment* (Inc) function.

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

Listing 2. Progress Meter Updating

Listing 3 shows the corresponding aspect specification to capture the update of the progress meter. The statements in the *after* advice indicate the code to be weaved into the base program after each call to the *Increment* function. The aspect represents the input to the weaver, which is then translated to corresponding rewrite rules using model transformation techniques. These rewrite rules that are then fed into the PTE to perform the actual weaving.

```
domain ObjectPascal;
aspect progress_meter {
  pointcut() IncCall : call(Inc(TotalInserts));
  after() :  IncCall
  {
    if not ProcDlg1.Process(TotalInserts) then
        begin
          ProcDlg1.Canceled := True;
          Result := True;
        exit;
    end;
  }
}
```

Listing 3. Source Aspect Specification for Progress Meter Dialog

The ATL transformations that perform the translations from the source aspect specification into the target RSL specification follow a general translation scheme. An excerpt of the ATL specification used to translate the progress meter aspect to its corresponding RSL specification is shown in Listing 4. The first ATL rule (*AspectDomain2RSLDomain*) maps the domain name defined in the source aspect metamodel to the matching domain name in the target RSL metamodel. The source and target patterns are identified using the keywords *from* and *to*, respectively. The expression `Aspect!Domain` is used to reference the *Domain* metamodel element in the *Aspect* metamodel. The second rule (*Aspect2Rsl*) is the root-level rule and is fired before any other rules are executed. The translation scheme is typically defined in this rule and subsequent rules are fired according to the root-level translation scheme. For example, *advice* (source) gets translated to patterns (target), which in turn fire the corresponding ATL rule, namely *AfterAdvice2Pattern.* This rule further translates the elements of an *AfterAdvice* to elements of a RSL *Pattern* (i.e., it binds a source *advice* statement to a target RSL pattern text element). Similar pointcuts are mapped to rules that subsequently call the *Pointcut2Rule* ATL rule.

```
rule AspectDomain2RSLDomain {
   from
       s : Aspect!Domain
   to
       t : Rsl!Domain (
           dname <- s.dname
       )
}

rule Aspect2Rsl {
   from
       s:  Aspect!Aspect
   to
       t:  Rsl!Rsl (
           domain  <-  s.domain,
           pattern <-  s.advice,
           rule    <-  s.pointcut,
           ruleset <-  rs
        ),
       rs: Rsl!RuleSet (
           rsname <- s.aname,
           rname  <-  s.pointcut->
              collect(e|e).pctname
        ),
  ...
 }
```

```
rule AfterAdvice2Pattern {
   from
      s   : Aspect!AfterAdvice
   to
      t   :  Rsl!Pattern (
              ptoken <- statement_list',
              ptext  <- pt,
              ...
              ),
      pt :  Rsl!AdviceStmt (
              name <- s.advStmt.stmt
              ),
   ...
}

rule Pointcut2Rule {
   from
      s :  Aspect!Pointcut
   to
        t : Rsl!Rule (
             rname <- s.pctname,
             r_lhs <- 'statement_list',
             r_rhs <- 'statement_list',
             ...
             ),
     ...
}
```

(1) (2) (3) (4)

Listing 4. ATL specification for transforming Aspect to RSL (i.e., a model transformation generating program transformation rules)

The ATL specification essentially establishes the mappings between the various components of the source aspect metamodel to the matching elements in the target PTE metamodel. Additionally, for every set of pointcut specification (e.g., call, execution, set, or cflow) there exists a different translation scheme. The above translation scheme is defined for a *function call* join point and will work not only for the progress meter aspect but also for other aspects that use a *function call* pointcut specification. Listing 5 shows the generated rewrite rules (target RSL specification) after applying the ATL transformations (Listing 4) on the source aspect specification (Listing 3).

```
default base domain ObjectPascal.
pattern after_advice() : statement_list =
   "if not ProcDlg1.Process(TotalInserts) then
       begin
           ProcDlg1.Canceled := True;
           Result := True;
       exit;
    end".
rule IncCall() : statement_list -> statement_list =
  $Inc(TotalInserts)
->
  $Inc(TotalInserts)  advice().
public ruleset progress_meter = {IncCall}.
```

Listing 5. Generated RSL rules used for actual weaving

The translated rules are internal to the system and are further processed by the DMS transformation engine to weave in the aspects to the base Delphi (ObjectPascal) program. For more information and complete source code, please refer to http://www.cis.uab.edu/softcom/GenAWeave.

4 Related Work

Fradet and Südholt first observed that aspect weaving can be performed using a general transformation framework for a specific programming language [10]. Similarly, a detailed description of a weaver for a declarative language was provided by Lämmel [11], which used functional meta-programs to weave aspects. 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 [12]. In one of our previous work, we demonstrated how a PTE can be used to construct a weaver for an arbitrary language like ObjectPascal [1]. However, an inherent difficulty in using a PTE is the associated accidental complexity that makes such a tool hard to use in mainstream software development. Our current work particularly alleviates this problem but still leverages the power of PTEs by automatically generating most of the rewrite rules used by PTEs to weave aspects into source programs.

5 Conclusion

The majority of research in this area mainly focuses on how aspects can be applied to MDE [13]. This research illustrates how MDE can also assist in building aspect weavers. The metamodel-based approach to aspect weaver construction is suitably applicable to both the source aspect language and the target rules language, and can be realized with relative ease as each have limited constructs similar to DSLs. The main advantage of this approach is the ability to modularize the weaver construction process by decoupling the source aspect language metamodel from the target PTE metamodel. The source aspect metamodel need not be altered even if one chooses to opt for a different target PTE, only a new PTE metamodel needs to be developed. Conversely, for every new language, one needs to add the appropriate metamodel extensions to the base aspect metamodel, but no change to the target metamodel is needed. Another advantage is that both the aspect language (source) and rules language (target) can evolve independent of each other. This leads to new features being added to the weaver with minimum cost on maintenance (i.e., only new ATL mappings are added). As part of future work, we are considering to reuse part of the core ATL transformations from one source language to another. This can be achieved by slight modification to the design in which a section of the core transformations would be defined as abstract. Concrete ATL rules belonging to a particular source language weaver can be subsequently inherited from these abstract rules based on the source language's join point model. We believe that this can considerably speed up the weaver construction process as the heart of the weaver depends on ATL transformations. Thus, by combining model engineering techniques with existing PTEs, one may avoid the associated accidental complexities, but still leverage their power to construct aspect weavers that are easier to modularize, evolve and maintain.

Acknowledgement

This work is supported in part by an NSF CAREER award (CCF-0643725) and by the OpenEmbeDD project.

References

1. Gray J, and Roychoudhury S, "A Technique for Constructing Aspect Weavers using a Program Transformation Engine," *AOSD '04, International Conference on Aspect-Oriented Software Development*, Lancaster, UK, March 2004, pp. 36-45.
2. 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.
3. Brand M.G.J. van den, J. Heering, P. Klint, and P.A. Olivier "Compiling Rewrite Systems: The ASF+SDF Compiler.," *ACM Transactions on Programming Languages and Systems*, July 2002, pp. 334-368.
4. Aßmann U, and Ludwig A, "Aspect Weaving with Graph Rewriting," *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, Erfurt, Germany, September 1999, pp. 24-36.
5. 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.
6. Jouault F, and Kurtev I, "Transforming Models with ATL," *Model Transformations in Practice Workshop at MoDELS*, Montego Bay, Jamaica, September 2005.
7. Jouault F, and Bézivin J, "KM3: a DSL for Metamodel Specification," *8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, LNCS, Bologna, Italy, pp. 171-185.
8. Jouault F, Bézivin J, and Kurtev I, "TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering," *International Conference on Generative Programming and Component Engineering*, Portland, OR, October 2006, pp. 249-254.
9. Barbero M, Jouault F, Gray J, and Bézivin J, "A Practical Approach to Model Extension," *Third European Conference on Model Driven Architecture Foundations and Applications (ECMDA 2007)*, Haifa, Israel, June 2007.
10. Pascal Fradet and Mario Südholt, "Towards a Generic Framework for Aspect-Oriented Programming," *Third AOP Workshop, ECOOP '98 Workshop Reader*, Springer-Verlag LNCS 1543, Brussels, Belgium, July 1998, pp. 394-397.
11. Ralf Lämmel, "Declarative Aspect-Oriented Programming," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Antonio, TX, January 1999, pp. 131-146.
12. Florian Heidenreich, Jendrik Johannes, and Steffen Zschaler, "Aspect Orientation for Your Language of Choice," *11th International Workshop on Aspect-Oriented Modeling*, Nashville, TN, September 2007.
13. Janos Sztipanovits, John Bay, Larry Rohrbough, Shankar Sastry, Douglas C. Schmidt, Don Wilson, and Don Winters, Escher, "A New Technology Transitioning Model," *IEEE Computer*, Vol. 40, No. 3, March 2007.