

AOP for Everyone - Cracking the Multiple Weavers Problem

Suman Roychoudhury and Jeff Gray

*Department of Computer and Information Sciences, University of Alabama at Birmingham
{roychous, gray} @ cis.uab.edu*

As a result of programming language research over the past fifty years, a veritable “Tower of Babel” exists with multiple billions of lines of legacy code maintained in hundreds of different languages [11]. In fact, legacy languages such as Cobol are estimated to account for a large percentage of existing production software [2]. Yet, the majority of AOP researchers have focused most of their attention on just a few modern languages, such as Java. Although initial interest exists in applying AOP to languages like Cobol [10], a generalized approach that brings aspects to legacy software is still missing.

A naïve proposal would attempt to migrate legacy code into a modern object-oriented language like Java, such that existing tools (e.g., AspectJ) could be applied. Such a proposition is often not possible due to technical, cultural and political concerns within the institution that owns the legacy code. Rather than bringing the code to existing Java-based weavers, a viable alternative is to take AOP principles to the legacy languages and tool environments. Given the large number of languages in use, a solution that mitigates the effort needed to create each new weaver is more desirable than an approach that manually recreates a weaver from scratch for each legacy language.

This article enumerates the challenges in language-parametric weaving and summarizes existing contributions that enable modularization of crosscutting concerns in software artifacts written in various programming languages. By the term language-parametric, we imply a generic solution that can be applied to generate weavers from concrete language descriptions.

Challenges of AOP Legacy Adoption

There are several key challenges to providing an initial methodology that allows experimentation with aspects in languages other than Java. The main obstacles toward adoption of aspects for legacy software are as follows:

Challenge C1 - The Parser Construction Problem: Building a parser for a toy language, or a subset of an existing language, is not difficult. But, designing a parser that is capable of handling millions of lines of production legacy code is an onerous task. As observed in [11], the dominant factor in producing a renovation tool is constructing the parser. Software developers who want to explore modern restructuring capabilities in legacy systems will require industrial-scale parsers to allow them to evaluate the feasibility of adoption within their organization. Incomplete parsers for small research prototypes will not scale and may leave a negative first-impression of AOP.

Challenge C2 – Type Analysis Problem: There are some situations where an aspect language needs to interact with the type system of the base language. For example, in an object-oriented base language the type pattern of a pointcut specification may need to compute all of the subtypes of a given type. In order

to provide rich type patterns, a weaver must be able to build and analyze type information. The static analysis to provide this capability can be complex. A key research challenge is to provide an intermediate analysis capability that can be reused across multiple weaver construction efforts.

Challenge C3 - The Code Transformation Problem: In order to realize the desired remodularization of legacy source, a capability is needed to perform the underlying transformations and rewrites on a syntax-tree or on an abstract source model. This is not an easy task and requires considerable effort if a new weaver is created manually for each language of interest. When a new program restructuring or modularization idea is conceived (e.g., AOP), it is often the case that integration efforts to support a core set of transformations are repeated for each language to which the new idea is applied. Such repetition of effort is unfortunate and strongly suggests the need for further integration of language-parametric transformation techniques into general development practice.

Challenge C4 – Language-Independent Generalization of Transformation Objectives: Although most program transformation engines provide a general toolkit with pre-existing parsers, the transformation rules that actually perform the desired restructuring are encoded to the productions of a specific concrete grammar. Thus, all of the effort that is placed into creating the transformations to enable weaving cannot be reused in other language domains. With mature program transformation systems, the parsers and underlying transformation engine can be reused, but the transformation rules are not reusable due to their dependence on a concrete syntax. A core research issue is an approach that adopts an intermediate syntax to increase the level of reuse of aspect transformation rules across multiple languages.

Challenge C5 – Accidental Complexities of Transformation Specifications: An inherent difficulty associated with using program transformation engines is the low-level of abstraction at which a transformation rule is specified. Due to many accidental complexities, transformations typically are at an improper level of abstraction for general use by programmers. A solution to this challenge can be found in higher-level aspect languages that are layered above and generate down to the lower-level transformation rules. Such convenience layers assist in removing the idiosyncrasies of the underlying transformation technology.

Figure 1 summarizes the challenges of a generic AOP system for legacy software. In addition to these five technical challenges, other obstacles also exist, such as integration into the development environment, and non-technical issues that can thwart AOP adoption (e.g., managerial/organizational/legal issues, or conflict with existing code styles).

In the remainder of this article we introduce the current state-of-the-art to address these challenges and present a selective technique for constructing weavers for multiple languages.

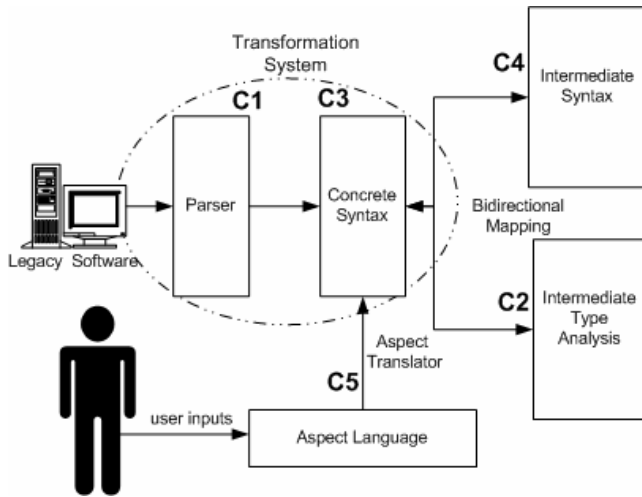


Figure 1. Overview of generic AOP system

Current State-of-the-Art in Legacy AOP

Existing software transformation techniques to enable AOP for different languages can be classified as:

- object-based transforms, such as a visitor object applied to an object model
- intermediate representations that permit primitive transformations to be applied to a set of languages (e.g., .Net CodeDOM [1])
- XML-based transforms that use an XML DOM structure
- term rewriting, such as a transformation rule

This section discusses different state-of-the-art examples of the above techniques within the context of the enumerated challenges.

SourceWeave.Net: The SourceWeave.Net [8] architecture is built on top of CodeDOM, which is the .NET standard for representing source code as abstract syntax trees (ASTs) [1]. Using SourceWeave.NET, a developer can write base and aspect components in standard C#, VB.NET and J#. An XML descriptor file is used to specify the interaction between the aspects and representative components. The technique uses a mapping to identify JoinPointShadows (areas in the source where join points may emerge) and uses a “pointcut-to-join point binding” to isolate parts of the source.

Weave.Net: Weave.NET is a load-time weaver that allows aspects and components to be written in a variety of .Net based languages [12]. It takes an existing .Net binary component as input with crosscutting specifications specified in an XML file. The behavior (implementation specific advice code) of an aspect is provided separately in another .Net assembly. Weave.NET recreates the input assembly, but in this regenerated version, join points are bound to behavior in the aspect assembly as specified in the XML aspect file. Because all transformations are done at the intermediate language (IL), it serves as a language-independent weaver.

AspectCobol: An initial prototype that brings aspects to Cobol was developed through a collaboration of academic and industrial partners [10]. The implementation reuses a pre-existing Cobol front-end to construct an AST that is persisted as XML. The AspectCobol weaver operates on the XML representation using a DOM-based approach. The weaver has similar semantics to AspectJ pointcuts, but uses an imperative language that is closer to Cobol syntax. The weaver provides ad hoc type analysis (e.g., use-to-def site navigation) for more sophisticated data join points.

GenAWeave: The GenAWeave project uses a commercial term rewriting system to construct weavers for legacy languages through program transformation rules [6]. From a large set of available parsers, transformation rules are specified to perform pattern matching on a language-specific AST to select join points. The transformation rules also describe the manner in which advice is to be applied to each selected join point. Although the weaving is performed using low-level transformation rules, a declarative aspect language has been created that generates down to the lower-level rules. This removes the accidental complexities associated with rewrite systems and provides a solution to C5.

General Discussion

From a comparative discussion of these representative approaches, each provides a distinguishing set of strengths and weaknesses. For example, GenAWeave and Weave.Net offer a strong solution to challenge C1 because of the availability of pre-existing industrial scale parsers (however, Weave.Net is limited to applications hosted within .Net). Comparatively, SourceWeave.Net is weak on challenge C1 due to the limited availability of CodeDOM providers beyond a handful of languages.

With respect to challenge C2, the current state-of-the-art is less than satisfactory. Very little work has been performed to make generalized analysis available for weavers of legacy languages. Perhaps the best support is provided by AspectCobol, which performs initial type resolution to resolve identifiers and declarations. However, the analysis is still tied to Cobol and a more generalized approach is missing.

The representation of the underlying abstract source model contributes to several differences affecting the solutions to each challenge. Because of its reliance on CodeDOM, SourceWeave.Net has limitations in terms of expressiveness. C# constructs map reasonably well to CodeDOM, but that is not true of all .Net constructs. It remains to be shown if CodeDOM is applicable to legacy languages like Cobol and Fortran. The strength of SourceWeave.Net is its ability to address challenge C4. Because the source representation is implicitly abstract, any transformation rule that is written may be reused across weavers for multiple programming languages.

GenAWeave’s primary strengths address challenges C1 and C3. The powerful pattern matching and transformation API is a perfect fit for the code transformation challenge. However, this same strength emerges as a weakness for challenge C4. The transformation rules used in GenAWeave must be generalized to be applicable across multiple languages.

A related challenge emerging from the source model concerns the issue of scalability. The verbosity of an XML code representation may hamper the size of an application that can be weaved. It has been reported that an XML representation is up to 50 times larger

than other internal representations and much slower to transform [5]. This may influence the ability of SourceWeave.Net and AspectCobol to handle very large applications. This is not a problem for Weave.Net, which weaves into the IL. It also is not an issue with GenAWeave because the underlying term rewriting engine has been used in practice to transform multiple millions of lines of code [3].

Most of these approaches do not provide a sufficient mechanism to make the effect of weaving transparent to the developer at the conceptual level (i.e., after weaving, the concern is placed inline at all selected join points). This hampers debugging considerably. SourceWeave.Net is a notable exception and attempts to address debugging issues. As seen in the maturation process of AspectJ and AspectC++ [13], further tool support can improve the conceptual transparency of aspects in legacy languages.

The following section provides examples of AOP transformations through term rewriting as part of the ongoing effort in the GenAWeave project. It also justifies the use of term rewriting compared to other approaches.

AOP and Term Rewriting

Term rewriting is a paradigm that is used in fields such as program transformation and theorem proving. In term rewriting, rules define a refinement to a structure by specifying a pattern to be matched and the resulting effect. There is initial experimental evidence to support the benefits of term rewriting for AOP. Our own previous work applied program transformations to modularize several aspects (e.g., updating a progress meter) that emerged from a commercial call center application that was written in Object Pascal [6]. Other researchers have also commented on the feasibility of applying rewriting techniques to AOP [1], [4], [9].

Our choice of term rewriting over object-based or XML-based transforms is supported by the following observations:

- i. A mature program transformation system that performs term rewriting (e.g., the Design Maintenance System (DMS) [3]) can ease the construction effort for weavers of legacy languages by offering a direct solution to challenges C1 and C3. Such systems provide availability of industrial scale parsers for multiple legacy languages, as well as an underlying low-level transformation engine to restructure source code.
- ii. The term rewriting model offers complex JoinPointShadows (e.g., nested conditional statements, as discussed in [14]) and a rich pointcut to join point binding that is informed by join point context information.
- iii. Term rewriting offers powerful pattern matching and efficient tree traversal strategies (e.g., using visitors over AST's) that can scale to several million lines of code.
- iv. With term rewriting, internal API's are available to modify ASTs in an arbitrary manner, thereby allowing more complex and flexible transforms required by legacy applications (e.g., aspects and loops [7]).
- v. In contrast to the verbose AST representation in XML-based approaches, DMS provides internal data structures (e.g., hypergraphs) to represent the underlying AST. This offers an improved level of optimization to support large legacy applications.

An example of term rewriting is shown in Listing 1, which is adapted from our previous work on applying AOP transformation techniques on a commercial call center application that was written in Object Pascal. Listing 1a shows a function in Object Pascal to synchronize a database error handling object. The code (shown in bold) for synchronizing the error handling object was repeated in over 20 different classes for all functions named `Handle` [6].

Listing 1b presents a term rewrite rule that illustrates how this synchronization concern is modularized as an aspect transformation. Such a rule is used as a rewrite specification that maps from a left-hand side (source) syntax tree expression to a right-hand side (target) syntax tree expression (syntactically denoted by “ \rightarrow ” in Listing 1b). The meta-variables used in this example (such as “*t*”, “*fps*”, “*f_mods*”, “*fprt*”, “*id*”) are instances of original grammar symbols (shown in italics) as defined for Object Pascal.

In addition, patterns (such as *probe_func_name*, *LockStmt_OP*, *UnLockStmt_OP*) present in both internal and external forms are used for specifying syntactic elements, analyzing types and performing tree traversals (for brevity, these are not expanded here).

```
function TExDBErr.Handle (ServerType :
  TServerType; E : EDBEngineError): Integer;
begin
  TExHandleCollection(Collection).LockHandle;
  try
    <database error handling code omitted here>
  finally
    TExHandleCollection(Collection).UnLockHandle;
  end;
end;
```

a) Synchronization in Object Pascal

```
rule sync_OP_func (t : stmt_list,
                   f_mods : func_modifiers,
                   id : IDENTIFIER,
                   fps : formal_params,
                   fprt : func_result_type):

qual_func_header_decl -> qual_func_header_decl =

"f_mods function \probe_func_name\(\i id\) \i fps
:\i fprt;
  begin
    \t
  end;"

->
"f_mods function \probe_func_name\(\i id\) \i fps
:\i fprt;
  begin
    \LockStmt_OP\(\i);
    try
      \t
    finally
      \UnLockStmt_OP\(\i);
    end;
  end;"
```

b) A rewrite specification to add synchronization

Listing 1. A synchronization aspect in Object Pascal

In this example, the `try/finally` block that implements the concurrency control is wrapped around the critical section of database error handling code. This is denoted by the statement list “ τ ” meta-variable as shown in this transformation. The pattern `probe_func_name` identifies those functions (i.e., with a signature of the form `function *.Handle(..)`) where the advice needs to be applied. The lock and the unlock patterns are inserted before and after this critical section of the source code.

The rewrite rule presented here is based on the Rule Specification Language defined in DMS [3]. Rewrite rules used by transformation systems could also be specified using a formal notation. For example, the rewrite rule shown in Listing 1b contains a composition of before and after advice; in particular, the lock and the unlock statements are composed before and after the critical section (database error handling code) of the source code and are bound to an execution style pointcut description (pcd). Formally, the composition can be shown as a function of three input parameters (i.e., the set of join points, the pointcut descriptor and the advice):

$$\text{Xform compose}(T_{jp}, pcd_{execute}, Adv_c\{S_b, S_a\}) \Rightarrow \{S_c\}$$

T_{jp} is the join point under context (e.g., in Listing 1, a set of function declarations) that matches the named pointcut descriptor (functions of the form `function *.Handle(..)`) and weaves the advice statements S_b, S_a (i.e., the lock and unlock statements in Listing 1a) as a wrapper over the original function body. In particular, the new composed block S_c can be related by the equation $S_c \Rightarrow \lambda(S_b, T_{jp}, S_a)$, where λ denotes the composition function (in this case it is a try-finally statement that wraps the before and after advice). By substituting the input parameters of the above transform, we obtain:

$$\begin{aligned} T_{jp} &= \text{qual_func_header_decl}; \\ S_b &= \text{LockStmt_OP}() \Rightarrow \text{TExceptionHandler(Collection).LockHandle}; \\ S_a &= \text{UnLockStmt_OP}() \Rightarrow \text{TExceptionHandler(Collection).UnLockHandle}; \\ S_c &= \lambda_{\text{TryFinallyStatements_OP}}(S_b, T_{jp}, S_a) \\ pcd &= \text{execution(probe_func_name(id))}; \end{aligned}$$

Notice that the composition function $\lambda_{\text{TryFinallyStatements_OP}}$ is a wrapper over the original function body declaration. Additional context information like `thisJoinPoint` is exposed by a static semantic analysis phase that accepts T_{jp} as input.

Transformation Strategy

An important requirement for constructing AOP systems is the foundation of a solid join point model. The pointcut description language forms a basis for isolating patterns or structures in the source code where advice is applied. Because rewrite specifications implicitly identify structural patterns in the base code, they are indirectly related to the underlying pointcut specification.

A *transformation strategy* is a set of transformation rules that map to each primitive pointcut based on the arguments passed in the pointcut description. For example, a simple generalization of a method execution pointcut description is shown below:

$$\text{execute}(function_return_type\ objname.method_name(param_types))$$

In this example, variables such as *function-return-type*, *objname*, *method-name*, and *param-types* are passed as arguments to the pointcut specification. There will be multiple rewrite rules that serve as a template for each specific pointcut to join point binding and will be instantiated based on the context of these arguments.

Abstraction Layers for Aspect Languages

A difficulty in using term rewriting systems is the inherent nature of the transforms, which requires knowledge about parsing and the underlying grammar to write complex rewrite functions. This raises unintentional accidental complexities and impedes the adoption of such systems in mainstream software development. Because of these idiosyncrasies, developers may prefer to deal with object-based or XML-based transforms, which are conceptually easier to work with, but do not offer the desired level of scalability.

Our current work offers a technique to build simple convenience layers (i.e., aspect languages layered above a transformation engine) to mitigate these accidental complexities to enable AOP based on term rewriting. For example, the rewrite rule shown in Listing 1b can be declaratively specified by an aspect (as shown below) and then translated to its low-level form as required by the term rewriting system. In this example, the special *compose* functor is used to wrap the try-finally statement before and after the execution of the `Handle` method.

```
aspect synchronizeDBErrorHandler{
    pointcut function_handle() :
        execution(function *.Handle(..));
    before():function_handle() {
        TExceptionHandler(Collection).LockHandle;
    }
    after():function_handle() {
        TExceptionHandler(Collection).UnLockHandle;
    }
    compose():try-finally (thisJoinPoint);
}
```

Genericity of Aspect Transformations

Many transformations can be expressed in terms of an abstract source model. For example, in a large majority of languages, conditional constructs (e.g., an “if” statement) often have the same form. Similarly, the exit and entry execution boundaries of a method (or function) can be generalized across languages. However, there are obvious situations where semantic differences exist between languages, especially across disparate development paradigms (e.g., semantic differences between a structured language and an object-oriented language).

The major obstacle toward generic transformation is the dependency of the rewrite rules on the base grammar, which is the essence of challenge C3. An example is presented in this section to motivate the problem of grammar dependent transformations. The rewrite rule from Listing 1b contains hard-coded reserved words (e.g., “begin”) and grammar productions (e.g., `stmt_list`) from the Object Pascal grammar, which makes it impossible to reuse this transformation in other language domains.

```

rule sync_Java_meths (t: stmt_seq,
                    m mods: meth_modifiers,
                    id: IDENTIFIER,
                    fps: fparams,
                    frt: ret_type) :

method_declaration -> method_declaration =

"\m_mods \frt \method_id\(\id\) \fps
  {
    \t
  } ;"
->
"\m_mods \frt \method_id\(\id\) \fps
  {
    \LockStmt_Java\(\);
    try {
      \s_seq
    }
    finally {
      \UnlockStmt_Java\(\)
    }
  } ;"

```

Listing 2. A rewrite specification for Java synchronization

Listing 2 shows the semantically equivalent synchronization transformation rewritten for Java. A careful observation reveals a marked resemblance between the rule shown previously in Listing 1b and the one shown in Listing 2. For example, the meta-variables (such as “t”, “fps”, “f_mods”, “frt”, “id”) used in the previous rule and specific to the Object Pascal grammar have equivalent representation for the rule shown in the Java grammar. There are several other common features that are beyond the scope of this article, but the fact that each of these rules is tied to a different grammar makes the transformations impossible to reuse.

To improve reuse of rewrite rules, we have investigated an approach that reuses similar structural features within language domains and constructs transformations that are independent of the base language syntax. These generic transformations are not tied to a specific grammar, but serve as a template that can be transformed to a concrete grammar (i.e., generic transformation rules serve as input to a second order transformation that binds the specific grammar).

There are two key issues needed to enable generic transformations to support aspect weaving; namely, structural AST type abstraction, and use of a common intermediate type system. These concepts are explained in the next two sub-sections, followed by an example generic transformation for the synchronization aspect.

Structural AST Types

Transformations that are dependent on the syntactic structure of the base language are known as *structure-dependent transformations*. Such transformations cannot be reused in a different language or platform context. Structural type abstraction in the form of an intermediate abstract syntax tree (*i*-AST) addresses this problem. An *i*-AST is a generic AST specification that is common to a subset of programming languages or dialects.

Examples of the abstract non-terminals that are represented in the *i*-AST grammar for object-oriented languages are Type Declaration, Type Members, Statements, Expressions, Arguments, Sub-Expressions, Attributes, Collections, Identifiers and Primitives. However, it is not intended to capture all forms of structural types for all languages in one *i*-AST grammar, but to

create *i*-AST grammars for each common language paradigm (or dialects of the same language).

The advantage of representing languages in the *i*-AST format is useful for modularizing aspects in a language-parametric manner. This enables reuse of existing parsers and the low-level transformation system. For example, a *statement list* in Object Pascal can be mapped to a *statement sequence* in Java via an intermediate representation of the given non-terminal in *i*-AST. Such mappings from a specific language to *i*-AST enable the parameterization of a language to a generic transformation rule.

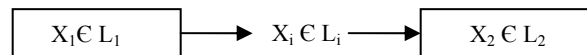
Intermediate Type Analysis

The semantic type information is typically handled through static type analysis. It is often required to create rewrite specifications based on the types used in the base language. For example, a rewrite rule that is to be applied to all classes and their subclasses that match a specific pattern. The `probe_func_name` (Listing 1b) and `method_id` (Listing 2) patterns are actually external functions that perform a static analysis for subtype patterns. Such rules are categorized as *semantic transformation rules*. In special cases, semantic rules may need to operate on values of the data members. However, in the present approach, we restrict to values that are available statically during compile time.

Language-Parametric Transforms

In Listing 1b and Listing 2, transformations are represented in their concrete syntax. Although such a representation is required to bind rewrite rules with the low-level transformation system, it raises an inherent difficulty in reusing transformations to construct AOP systems for new languages. It is often the case that such systems are almost always constructed from scratch with no emphasis on reuse of AOP language design and implementation know-how across base languages. Therefore, it is important to define a language-parametric transformation strategy that is built upon a solid foundation of reusability. Two specific approaches to reusable aspect transformations are *derived transforms* and *synthesized transforms*.

Derived Transforms: A derived transform answers the question, “Given a *transform* X_1 for *language* L_1 , how should a semantically similar *transform* X_2 be derived for *language* L_2 ?” An intuitive solution is to transform X_1 to an intermediate representation X_i and then transform X_i to X_2 (provided there is an established structural and semantic relationship from X_1 to X_i and from X_i to X_2), as shown below:



This also implies a second order transformation; that is, a transformation is itself being transformed to another form. As an example of X_1 , Listing 1b presented a transformation rule for the Object Pascal language. A parametric representation of the intermediate transformation X_i (when translated from its concrete form X_1) is shown in Listing 3. This intermediate form uses *i*-AST to avoid concrete language syntax and forms the basis for deriving concrete transforms for a new language (L_2).

```

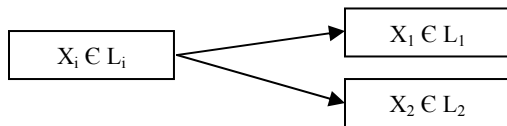
rule sync_intermediate
  (sL:IStatementList, idlist:IIIdList,
   pl:IParamTypeList, rt:IReturnType):
IFunctionDecl -> IFunctionDecl =
  "\meth_signature\(\idlist, pl, rt)
   \Block_Begin \sL \Block_End"
->
  "\meth_signature\(\idlist, pl, rt)
   \Block_begin
   \LockStmt\(\);
   \ITryFinallyStatements \sL
   \UnlockStmt\(\);
   \Block_End"

```

Listing 3. A generalized synchronization transformation

It should be observed that a complete weaver for any language (say L_j) is a set of concrete transforms $\{X_j\}$ for that language. An interesting characteristic of this approach is that once a weaver for a base language has been constructed, weavers for other similar languages can be systematically constructed by stepwise deriving each X_i and then re-constructing each X_j 's $\in L_j$.

Synthesized Transforms: An alternative approach is to construct a transformation directly in the intermediate form and synthesize concrete transforms for each target language, as shown below:



This approach does not rely on the notion of pre-existing weavers, but requires bidirectional mappings between languages and their intermediate forms. In this approach, all of the intermediate transformations are written from scratch and then used to synthesize concrete transformations for each new language.

Conclusion

The majority of development and research in AOP is focused on a few popular programming languages (e.g., Java), neglecting the several billion lines of code written in other languages. Constructing new weavers for other languages from scratch does not take into consideration the power of reusing the existing knowledge in a different language and platform context. The ability to perform generic transformations can have direct results in expanding the applicability of AOP to many languages and domains. A growing body of research has initially applied software transformations to capture crosscutting concerns in order to improve modularity and adaptive maintenance for legacy modernization. The challenges presented in this article represent some of the key requirements for a language-parametric approach to weaver construction an adoption of AOP in legacy systems.

Acknowledgements

We would like to thank Ralf Lämmel for many insightful comments and suggestions on this article. We also benefited from feedback by Paul Klint, Tijs van der Storm, and Jurgen Vinju.

References

- [1] Uwe Abmann and Andreas Ludwig, "Aspect Weaving as Graph Rewriting," *Generative Component-based Software Engineering (GCSE)*, Erfurt, Germany, October 1999, pp. 24-36.
- [2] Ed Arranga, "In Cobol's Defense," *IEEE Software*, March/April 2000, pp. 70-75.
- [3] Ira Baxter, Christopher Pidgeon, and Michael Mehlich, "DMS: Program Transformation for Practical Scalable Software Evolution," *International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004, pp. 350-354.
- [4] 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.
- [5] Roy Germon, "Using XML as an Intermediate Form for Compiler Development," *XML Conference and Exposition*, Orlando, FL, December 2001.
- [6] Jeff Gray and Suman Roychoudhury, "A Technique for Constructing Aspect Weavers Using a Program Transformation System," *International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, March 2004, pp. 36-45.
- [7] Bruno Harbulot and John R. Gurd, "A Join Point for Loops in AspectJ," *4th Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, Chicago, IL, March 2005.
- [8] Andrew Jackson and Siobhán Clarke, "SourceWeave.NET: Cross-Language Aspect-Oriented Programming," *Generative Programming and Component Engineering (GPCE)*, Vancouver, Canada, October 2004, pp. 115-135.
- [9] Paul Klint, Tijs van der Storm, and Jurgen Vinju, "Term rewriting meets aspect oriented programming," Report SEN-E0421 (<http://ftp.cwi.nl/CWIreports/SEN/SEN-E0421.pdf>), December 2004.
- [10] Ralf Lämmel and Kris De Schutter, "What does aspect-oriented programming mean to Cobol?" *International Conference on Aspect-Oriented Software Development (AOSD)*, Chicago, IL, March 2005, pp. 99-110.
- [11] Ralf Lämmel and Chris Verhoef, "Cracking the 500 Language Problem," *IEEE Software*, November/December 2001, pp. 78-88.
- [12] Donal Lafferty and Vinny Cahill, "Language-Independent Aspect-Oriented Programming," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, CA, October 2003, pp. 1-12.
- [13] Olaf Spinczyk, Andreas Gal, Wolfgang Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to C++," *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002, pp. 53-60.
- [14] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, Hridesh Rajan, "On the Criteria to be Used in Decomposing Systems into Aspects," *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Lisbon, Portugal, September 2005.
- [15] Thuan Thai and Hoang Lam, *.NET Framework Essentials*, O'Reilly, 2003.