

Towards Language-Independent Weaving Using Grammar Adapters

Suman Roychoudhury and Jeff Gray

Department of Computer and Information Sciences

University of Alabama at Birmingham, Birmingham, AL, 35294, USA

{roychous, gray}@cis.uab.edu

RESEARCH AREA

Legacy Software Evolution and Restructuring using Program Transformation Systems and Aspect-Oriented Programming - the paper presents the concept of abstract transformation rules that enable program restructuring in legacy-based systems. A generic platform is described that uses grammar adapters in conjunction with transformation systems to construct aspect weavers for legacy languages.

ABSTRACT

Current practices to support software evolution and restructuring, such as aspect-oriented software development and refactoring, have offered new capabilities for addressing the challenges of adapting software. Aspect weavers and refactoring engines enable the evolution of software in a modular and extensible manner, thereby improving the structure of the program and rendering new behavior. However, the majority of research in these areas has been focused on a few popular programming languages (e.g., Java), neglecting the several billion lines of code written in other languages. Moreover, many software maintenance tools are constructed from scratch, which does not take into consideration the power of reusing the existing knowledge in a different language and platform context. To address this problem, two key research ideas are introduced in this brief position paper. First, the concept of extending software restructuring techniques (e.g., aspect weaving) to several different programming languages is explored. A core focus of this objective is the abstraction of transformation functions to enable design maintenance in legacy based systems. The second research objective extends the first goal to understand the fundamental science of constructing a generic platform using grammar adapters to enable language-independent software evolution.

1. INTRODUCTION

Across numerous application domains, it has been estimated that several hundred billion lines of legacy code are maintained in hundreds of disparate languages and paradigms [12]. As demands for adaptation to existing software increase, future requirements will necessitate new strategies for improved modularization and design preservation in order to support the requisite adaptations. To adapt and evolve to requirements changes, new approaches (e.g., Aspect-Oriented Software Development

(AOSD) [2]) have shown initial promise in assisting a developer in isolating points of variation and configurability. Considering the benefits of aspect-oriented programming (AOP) [7], software refactoring [3], and other maintenance efforts, it is likely that programmers who use legacy programming languages would desire to adopt these new techniques into their own programming environment. Unfortunately, there is little support for extending new software development concepts into existing legacy code, but initial investigation is underway even for languages like COBOL [9].

As pointed out in [4], a major obstacle faced during the construction of aspect weavers or refactoring engines for legacy based languages is the unavailability of mature parsers and appropriate transformation functions that can restructure a large code base. The use of a mature program transformation system, such as the Design Maintenance System (DMS) [1], or ASF+SDF [8], often eases the construction effort for introducing new features into existing languages. Program transformation systems provide direct availability of scalable parsers and an underlying low-level transformation engine to restructure source code. However, the skill-sets and experience needed to use a program transformation system makes such tools accessible only to language researchers. Moreover, the low-level transformation rules are often bound to a specific programming language and are not generally reusable across different language domains. Two key research objectives can be identified in order to bring about source-level aspect weaving across multiple languages:

- A primary objective of the research described in this paper is to abstract the accidental complexities associated with a transformation system (e.g., parsing algorithms, abstract syntax tree representations, underlying language grammar) and generate the transformation rules from a high-level language representation. This enables software developers, who may not have direct experience in using transformation systems, to indirectly take advantage of the low-level rule specification language for restructuring their source code. A translator can then interpret the high-level constructs and generate the equivalent low-level transformation rules required by the program transformation system.

- The core of this research concerns the specification and reuse of generic transformation rules across multiple languages, which can assist in bringing new software development techniques to legacy systems implemented in multiple languages. To support this objective, recent research in programming languages indicates that there may indeed be deep structures that are common among programming languages within a specific programming paradigm. To understand the commonalities among programming languages, a new term is introduced: a *grammar adapter* can capture the commonalities across disparate languages in an abstract representation and map the core transformation functions across them. However, certain restrictions need to be defined to assert the boundaries of a common class of programming languages.

A *generic transformation rule* can be specified using an *abstract grammar*, and a *grammar adapter* can transform the generic transformation to its language-specific form. The boundaries of the generic transformations need to be grouped for each common class of programming languages (e.g., object-oriented, procedural, functional) due to their inherent similarity. Conceptually, the commonalities of each programming language domain will be extracted into the abstract grammar and generic transformation rules will be specified to accomplish the required transformation. The following section illustrates these concepts within the context of support for aspect weavers.

2. LANGUAGE-INDEPENDENT ASPECT WEAVING

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

public Integer HandleDBError(TServerType
                             ServerType,
                             EDBEngineError E)
{
  ErrHandlerCollection.LockHandle(Collection);
  try {
    <database error handling code omitted here>
  }
  finally {
    ErrHandlerCollection.UnLockHandle(Collection);
  }
  ...
}
```

Figure 1. Synchronization code in Java and Delphi

This section summarizes the application and generalization of transformation functions to separate crosscutting concerns from large legacy code bases. As an example, synchronization is often characterized as a crosscutting concern. The top of Figure 1 shows a sample piece of code written in Object Pascal that synchronizes access to a database error handling routine. An equivalent Java representation of the same concept is given in the bottom of the figure. The underlined parts of each code listing represent the synchronization crosscutting concern as implemented in each language. The synchronization code in this figure may crosscut many related methods.

To enable separation of crosscutting concerns (underlined parts of Figure 1) from the base implementation, DMS transformation rules are introduced in Figure 2. These rules describe a directed pair of corresponding syntax trees and are 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. In this example, the transformations represent the concrete low-level transforms that would be needed to weave the synchronization concern into specified methods of Object Pascal and Java.

```
rule sync_OP_meths (s1:stmt_list, id:IDENTIFIER,
                  fps: formal_params,
                  frt:func_result_type):
qual_func_header_decl -> qual_func_header_decl =
"function \method_id\(\id\) \fps : \frt ;
  begin \s1 end;" ->
"function \method_id\(\id\) \fps : \frt ;
  begin \LockStmt\(\);
    try \s1
      finally \UnLockStmt\(\);
    end;"
...

rule sync_Java_meths (s_seq:stmt_seq,m_mods:
                    meth_modifiers,
                    t:type,id:IDENTIFIER,
                    fp:fparams) :
method_declaration -> method_declaration =
"\m_mods \type \method_id\(\id\) \fp
 { \s_seq } ;" ->
"\m_mods \type \method_id\(\id\) \fp
 { \LockStmt\(\);
   try { \s_seq }
   finally { \UnLockStmt\(\) }
 };"
...
```

Figure 2. Low-level Transformation Rules

The `sync_OP_meths` (see Figure 2) rule works on the Object Pascal grammar and transforms the qualified function header declaration node in the Abstract Syntax Tree (AST). The statement meta-variable (`s1:stmt_list`) characterizes the critical section of the source code (i.e., the actual error handling code). The pattern `method_id` identifies the methods where the advice needs to be applied. The `LockStmt` and `UnLockStmt` patterns are inserted before and after the critical section of the source code. Note that a concrete transformation can implicitly encode the knowledge of the

underlying domain semantics. For more information on the syntax and semantics of the rule specification language, please refer to [1, 4].

2.1 Extracting the Commonalities – The Abstract Grammar

By comparing the Object Pascal transformation rule at the top of Figure 2 with the Java transformation rule in the bottom (see `sync_Java_meths` in Figure 2), an obvious similarity can be observed with the different meta-variables that represent the nodes in the two ASTs. For example, the meta-variables `stmt_list`, `formal_params`, `func_result_type`, `qual_func_header` decl in the ObjectPascal domain have commonalities with the corresponding meta-variables in the Java domain. The similarities between the syntax of the base languages motivate the concept of a *grammar adapter*. The reason it is called a *grammar adapter* is because it works on an abstract representation which makes it independent of the syntax of the concrete base language grammar. However, transformations applied to this abstract representation can be reflected or adapted to bring changes in multiple base languages. The general idea is presented below that depicts an initial mapping between the concrete and abstract (meta-syntax) parts of Object Pascal and Java (we believe that other OO languages would have similar mappings).

```

st_list@ag → stml_list@OP
ID_list@ag → id@OP
param_list@ag → paramformal_params@OP
ret_type@ag → func_result_type@OP
meth_signature@ag → qual_func_header_decl@OP
.....
st_list@ag → stmt_seq@Java
ID_list@ag → id@Java
param_list@ag → fparams@Java
ret_type@ag → type@Java
meth_signature@ag → method_declaration@Java
.....

```

Figure 3. Abstract Grammar representation

Figure 3 describes the intention of a transformation map that links the abstract grammar to a concrete grammar. The abstract grammar fundamentally captures the abstract form of a language (i.e., generic nodes within a common subset of programming languages), which can then be mapped to the concrete notations of the base language. By raising the grammar abstraction level, the synchronization rule as presented in Figure 2 can be re-written in a more generic style (see the `sync_generic` rule in Figure 4, which is written in terms of an abstract grammar common to languages such as Object Pascal and Java).

The generic transformation rule (`sync_generic` of Figure 4) captures the essence of the various non-terminals as observed in both the Object Pascal and Java grammar. An obvious choice of the Object Pascal and Java was made due to their distinct similarities. An assessment of other

object-oriented languages and their dialects may reveal other common mappings.

```

rule sync_generic (sL:st_list, i:ID_list, pl:
                    param_list,
                    rt:ret_type):
std_func_decl -> std_fun_decl =
"\meth_signature\(\id\,\pl\,\rt)
 \std_start \sL \std_end" ->
"\meth_signature\(\id\,\pl\,\rt)
 \std_start
  \LockStmt\(\);
  \try_finally_decl \sL
  \UnLockStmt\(\);
 \std_end"

```

Figure 4. Generic Transformation Rule

Complete language independence is difficult to achieve (if not impossible to address), but defining the paradigm boundaries (structural, object-oriented, functional) and reusing the commonalities even with certain restrictions can proportionately reduce the amount of rework required to construct aspect weavers and refactoring tools. This is even more likely for various dialects of the same language. It is not proposed to map different language paradigms into a single abstract grammar, but separate grammars from each paradigm. Also note that the boundaries of program transformations are often limited to specific nodes in the AST (e.g., declarations and statements), which indicates that the abstract grammar may not be a superset, but acts on a set of non-terminals from the base domains.

2.2 Raising the Level of Abstraction and Bindings to a Grammar Adapter

Our subjective opinion is that low-level program transformations are beyond the grasp of typical programmers. The benefit of the grammar adapter and abstract grammars could only be perceived if the low-level transformations required to support program restructuring could be instantiated from a language notation that is similar to an aspect language at a higher-level. The proposed initial syntax and semantics of the aspect language (see Figure 5) is comparable to that of AspectJ [7]. However, a future investigation may require looking beyond the AspectJ constructs to accommodate a richer extension to the existing pointcut / advice specification that can map complex transformation routines. In Figure 5, the pointcut mechanism declares those locations in the execution of the program that are affected by the aspect, with before and after advice specifying the behavior. In this case, the advice behavior is abstract and actually language independent, achieving the initial idea of cross-language aspect reuse. The grammar adapter is the mechanism by which the bindings are made between the abstract grammar (i.e., Figure 3), generic transformations (i.e., Figure 4) and the abstract aspect (i.e., Figure 5) to generate the low level concrete transformations (i.e., Figure 2) that can implicitly capture the semantics of the underlying domain. We have currently implemented an aspect language domain that

maps down to the low-level transformations needed to perform the weaving.

```
aspect generic_synchronize {
  before(): execution (mname@meth_signature) {
    //insert base language specific lock statement
  }
  after(): execution (mname@meth_signature) {
    //insert base language specific unlock
    statement
  }
}
```

Figure 5. High level Aspect Representation

3. EXPERIMENTAL EVALUATION

A prototype aspect weaving tool for ObjectPascal has been constructed on top of DMS and applied to a commercial call-center application [4]. The initial work on grammar adapters is being implemented as an Eclipse plugin and will be experimentally validated against several criteria, including: the level of reusability of the grammar adapters across multiple language domains, the subjective ease of use for software developers, and the quantifiable effort spent in constructing new maintenance tools using the approach. The experimental evaluation will be based on legacy systems obtained from collaboration with industry and academic researchers. The planned evaluation will be performed on legacy software from several domains (e.g., real-time embedded systems, financial applications, and scientific computing) that are implemented with different languages (C++, COBOL, and FORTRAN). A summary of the experimental platforms follows:

Embedded Avionics Software: Our initial effort on using a program transformation system was based on Bold Stroke, which is a product-line architecture written in C++ (over 3 million lines of code in 6000 components) that was developed by Boeing in 1996 to support families of mission computing avionics applications for a variety of military aircraft. Our initial investigation demonstrated evolution of Bold Stroke from aspects captured in higher-level modeling languages [5].

Business Enterprise Planning Software: Given the abundance of legacy software written in COBOL, the potential impact for any new modularization technology should be evaluated on existing COBOL code bases. Through collaboration with a large American insurance company, the research is being applied to financial systems written in COBOL (running on IBM AS-400). The importance of applying aspect-oriented ideas to COBOL is well documented in [9].

High Performance Scientific Software: As observed in [11], a reinvented methodology for parallel programming, as applied to high-performance computing (HPC), should embrace recent concepts from software engineering. HPC applications are rarely designed and implemented “from scratch.” Rather, they are developed most often by reengineering existing sequential FORTRAN applications

using ad-hoc approaches, resulting in a non-scientific approach of trial and error. We have initiated discussions with the HPC laboratory at UAB to apply AOSD and refactoring techniques to address the challenges of legacy scientific software. In addition, we have also applied our techniques to a popular scientific library (e.g., Blitz++) that uses large scale C++ templates libraries [10].

4. CONCLUSION

The primary contribution of the research described in this position paper represents an exploration into the science for supporting language-independent software transformation in order to improve modularity and adaptive maintenance for legacy modernization. The research is investigating the underlying science to support abstract grammars and grammar adapters as a foundation for language-independent software transformation and analysis. The ability to perform generic transformations can have direct results in expanding the applicability of AOP and refactoring to many languages and domains. Furthermore, a generative methodology to permit synthesis of aspect weavers and refactoring engines for multiple languages using grammar adapters provides a language experimentation environment for investigating ideas in new paradigms (e.g., aspects combined with parametric polymorphism) without constructing all of the underlying parsing and transformation mechanisms from scratch. Another contribution is the application of grammar adapters to generate language-independent analysis and reverse engineering capabilities (e.g., clone detection and aspect mining).

The software and video demonstrations related to this research project can be found at:

<http://www.cis.uab.edu/gray/Research/GenAWeave>

5. REFERENCES

- [1] 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. 625-634.
- [2] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, Mehmet Aksit, “Aspect-Oriented Software Development,” Addison-Wesley 2004.
- [3] Martin Fowler, *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.
- [4] Jeff Gray, and Suman Roychoudhury, “A Technique for Constructing Aspect Weavers using a Program Transformation Engine,” *AOSD '04, International Conference on Aspect-Oriented Software Development*, Lancaster, UK, March 22-26, 2004, pp. 36-45.

- [5] Jeff Gray, Jing Zhang, Yuehua Lin, Suman Roychoudhury, Hui Wu, Rajesh Sudarsan, Aniruddha Gokhale, Sandeep Neema, Feng Shi, and Ted Bapty, "Model-Driven Program Transformation of a Large Avionics Framework," Generative Programming and Component Engineering (GPCE), Vancouver, Canada, October 2004, pp. 361-378.
- [6] Steve Holzner, *Eclipse: A Java Developers Guide*, O'Reilly & Associates, 2004.
- [7] Gregor Kiczales, Eric Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, "Getting Started with AspectJ," *Communications of the ACM*, October 2001, pp. 59-65.
- [8] Paul Klint, Ralf Lämmel, and Chris Verhoef, "Towards an Engineering Discipline for Grammarware," submitted for journal publication, August 17, 2003, 32 pages. <http://www.cs.vu.nl/grammarware/agenda/>
- [9] Ralf Lämmel and Kris De Schutter, "What does aspect-oriented programming mean to Cobol?" *Proceedings of AOSD 2005*, Chicago, IL, March 2005.
- [10] Suman Roychoudhury, Jing Zhang, and Jeff Gray, "Weaving into Template Libraries," Submitted for review, available at: <http://www.cis.uab.edu/gray/Pubs/template-weave.pdf>.
- [11] Anthony Skjellum, Purushotham Bangalore, Jeff Gray, and Barrett Bryant, "Reinventing Explicit Parallel Programming for Improved Engineering of High Performance Computing Software," *ICSE 2004 Workshop on Software Engineering for High Performance Computing System Applications*, Edinburgh, Scotland, May 2004.
- [12] William Ulrich, *Legacy Systems: Transformation Strategies*, Prentice-Hall, 2000.