

# A Language-Independent Approach to Software Maintenance Using Grammar Adapters

Suman Roychoudhury

Department of Computer and Information Sciences

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

1-205-934-5841

[roychous@cis.uab.edu](mailto:roychous@cis.uab.edu)

## ABSTRACT

A long-standing goal of software engineering is to construct software that is easily modified and extended. Recent advances in software design techniques, such as aspect-oriented software development and refactoring, have offered new approaches to address challenges of software evolution. Several tools and language extensions have been developed by others to enable these techniques in a few popular programming languages. However, software exists in a variety of languages. An unfortunate consequence of legacy system adaptation is that new software engineering tools are often developed from scratch without preserving and reusing the knowledge gained from the previous construction of the tool in a different language and platform context. To address this problem, this paper summarizes two core research ideas. First, the concept of extending several software reengineering techniques in disparate 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 for constructing a generic platform using grammar adapters to enable language-independent software maintenance.

**Categories and Subject Descriptors:** D.3.3 [Programming Languages] Language Constructs and Features F.4.2 [Grammars and Other Rewriting Systems] D.2.3 [Software Engineering]: Coding Tools and Techniques

**Keywords:** Grammar Adapters, Program Transformation Systems, AOP, Refactoring

## 1. INTRODUCTION

Across numerous application domains, it has been estimated that several hundred billion lines of legacy code are being maintained in hundreds of disparate languages and paradigms [7]. 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. Considering the benefits of aspect-oriented programming (AOP) [3], software refactoring [6], and other maintenance efforts, it is likely that programmers who use legacy programming languages (e.g., COBOL, FORTRAN) would sense

a need 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. Efforts to provide support for constructing new design tools for legacy languages often face the hurdle of having to construct, from scratch each time, the appropriate parsers and syntax-tree transformation functions. The use of a mature program transformation (PT) system, such as the Design Maintenance System (DMS) [1], or ASF+SDF [4], often eases the construction effort for introducing new features into existing languages. PT systems provide direct availability of scalable parsers and an underlying low-level transformation engine. However, the skill-sets and experience needed to use a PT system effectively make 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.

An objective of the research described in this paper is to abstract the transformation rules into a user friendly high-level aspect language to provide a more appropriate representation to general software developers who may not have experience in directly using transformation systems. A translator can then interpret the high-level constructs and generate the equivalent low-level transformation rules required by the PT system. Another important objective of our research concerns the reuse of generic transformation rules across multiple languages, which can assist in bringing new software development techniques to legacy systems implemented in multiple languages.

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, we define a new term called “grammar adapter” (note: the benefits of a general notion of grammar adaptation and engineering were initially introduced by Lämmel in [4, 5]), which can capture the commonalities across disparate languages in an abstract representation and map the core transformation functions across them. 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. 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.

## 2. TRANSFORMATION FUNCTIONS

This section briefly presents a language-specific transformation rule. More details regarding the syntax of the transformation rule language can be found in [1, 2]. The section summarizes how transformation functions could be applied and subsequently generalized to separate cross-cutting concerns from large legacy code bases. As an example, synchronization is often characterized as a cross-cutting concern. The following example shows the core transformation function required to separate the synchronization aspect in a target application written in ObjectPascal.

```
rule sync_OP_meths (sl: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 \sl end;" ->
  "function \method_id\(\id\) \fps : \frt ;
  begin \LockStmt\(\);
  try \sl
  finally \UnLockStmt\(\);
  end;"
```

Listing 1 – Object Pascal synchronization transformation rule

The above rule inserts the lock and unlock statements around the critical section of source code (represented by “sl”). In Listing 1, the specific parts of the ObjectPascal grammar are italicized to show the dependence of the rule specification on the underlying grammar. To replicate the same transformation in Java, the rule has to be redefined as shown below in Listing 2.

```
rule sync_Java_meths (s_seq:stmt_seq,m_mods:
  meth_modifiers, t:type,id:IDENTIFIER,
  fp:formal_params) :
  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\(\) }
  } ;"
```

Listing 2 – The same synchronization rule redefined for Java

## 3. GRAMMAR ADAPTERS

There is a clear correspondence between the rules defined in Listing 1 and Listing 2, which indirectly reflects the commonalities between the two object-oriented languages and the specific intent of the transformation. However, the low-level rule syntax and its direct dependency on the underlying grammar is not a suitable representation for general purpose software development. Several other examples are described in [2].

To extract the commonalities among transformation rules for different languages, a generic transformation map is specified for each language domain, which acts as a reusable rule template library within that domain. The goal is to perform an engineering effort towards the artifacts built from the notion of Grammarware [4]. Moreover, to raise the rule abstraction level, a high-level language is required to reuse the generic transformations and grammar adapters as a plug-in library. As an illustration, the following aspect language describes the synchronization aspect in a manner that is language-independent. The application of a grammar adapter to a generic transformation rule translates the generic aspect of Listing 3 into the low-level representations needed by the transformation engine (Listing 1 and 2). Although this specific example focused on AOP, we are also applying the

concept of generic transformation rules and grammar adapters to other types of software restructuring techniques, such as generalized approaches to refactoring [6].

```
aspect generic_synchronize {
  before(): execution (//match to method_id) {
    // language specific lock stmt }
  after(): execution (//match to method_id) {
    // language specific unlock stmt }
}
```

Listing 3 – A high-level representation of the previous rules

## 4. EXPERIMENTAL EVALUATION

A prototype aspect weaving tool for ObjectPascal has been constructed on top of DMS and applied to a commercial call-center application [2]. Currently, we are investigating the requirements for a rule template library that can be reused across multiple language domains to assist in the generation of new design maintenance tools. The initial work on grammar adapters is being prototyped 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 industrial collaborators from several domains (e.g., embedded avionics, insurance, and scientific computing) implemented in numerous languages (e.g., C++, COBOL, and FORTRAN). 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] 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
- [3] 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.
- [4] 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/>
- [5] Ralf Lämmel, “Grammar Adaptation,” *Intl. Symposium of Formal Methods Europe (FME)*, Springer-Verlag LNCS 2021, Berlin, Germany, March 2001, pp. 550-570.
- [6] Ralf Lämmel, “Towards Generic Refactoring,” *Third ACM SIGPLAN Workshop on Rule-Based Programming (RULE)*, Pittsburgh, Pennsylvania, October 2002, pp. 15-28.
- [7] William Ulrich, *Legacy Systems: Transformation Strategies*, Prentice-Hall, 2002.