A Technique for Constructing Aspect Weavers Using a Program Transformation Engine

Jeff Gray and Suman Roychoudhury
Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Boulevard
Birmingham AL 35294
{gray, roychous} (at) cis.uab.edu

ABSTRACT

As aspect-orientation grows in influence, the scope of applicability also will need to expand. The new approaches for improved modularization offered by aspect-orientation can provide benefits not only to new development efforts, but to legacy systems as well. A difficulty with legacy system adoption of aspect-orientation, however, is in the construction of new weavers for the disparate programming languages in which the legacy software is coded. In this paper, we describe our experience with using a program transformation system as the underlying engine for weaver construction. In particular, the capability for weaving aspects into Object Pascal (Delphi) is demonstrated using the Design Maintenance System (DMS). From this Object Pascal weaving environment, the improved modularization of several crosscutting concerns in a commercial software application is shown. The paper also describes an initial approach for construction of language-independent aspect weavers.

1. INTRODUCTION

Software development occurs in a polyglot world. Presently, there are literally multiple billions of lines of legacy code maintained in hundreds of disparate languages and paradigms. In fact, a recent Gartner report, as cited in [34], estimates that there are 180-200 billion lines of existing COBOL source code in production use (of course, the *total* amount of legacy code is much higher when other languages are considered). Yet, the majority of language researchers and tool vendors have focused their attention on just a few popular languages, such as C++ and Java. Although the wide-spread acceptance of these languages ensures that a large pool of developers is available for experimentation, it is important to investigate how new research ideas can be retrofitted into legacy systems that were constructed from many different languages.

AOSD 04, March 2004, Lancaster UK.

It is a common occurrence to push new programming paradigms into legacy languages. For instance, efforts to add objects to languages have resulted in Ada 95, the FORTRAN 2000 standardization process, as well as object-oriented COBOL variants. Given this tendency of paradigm adoption through language evolution, it is likely that programmers who use legacy languages will also come to understand the benefits of Aspect-Oriented Software Development (AOSD). There are several problems, however, to providing an initial set of tools to allow experimentation with aspects in languages other than Java. The two main obstacles to providing language-independent *and* platform independent aspect tools are:

The Grammar Recovery and Parser Construction Problem

Building a parser for a toy language is not difficult. But, uncovering the grammar and designing a parser that is capable of handling millions of lines of production legacy code is an onerous task. As observed in [23], "Measuring this and other projects, it became clear to us that the total effort of writing a grammar by hand is orders of magnitude larger than constructing the renovation tools themselves. So the dominant factor in producing a renovation tool is constructing the parser." For assessment purposes, software developers who want to explore the capabilities of aspects in various languages 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 aspects.

The Weaving Engine Problem

Furthermore, providing a facility to perform the underlying weaving transformations and graph rewrites on a syntax-tree [1] is not easy. When a new program restructuring or modularization idea is conceived, 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-independent transformation techniques into general development practice.

Other researchers have also identified the need to elevate grammars and transformations as first-class citizens in the software design and evolution process (e.g., the notion of "Grammarware," as presented in [20]). Fortunately, the problems of grammar/parser recovery and syntax-tree transformation have an initial solution in the form of program transformation (PT) systems. However, there are large variations

in the capabilities offered among PT systems. Many transformation environments work well as exploratory research tools for investigating foundational principles of language construction and evolution. However, very few PT systems provide the industrial-scale maturity needed to trust their insertion into a large development project [4]. Even among the class of commercially available and industrially proven PT systems, it is naïve to assume that such systems offer a panacea to both the grammar recovery and transformation problems for all cases. For example, in the case where a new language (not initially supported by the PT system) is to be introduced and combined with other languages, it is still a significant engineering effort to construct the parser for a complex language from within a PT system. In this paper, we assume the existence of a PT system that already has a large collection of pre-defined parsers for languages in many different paradigms (e.g., object, functional, and declarative languages).

Over the past two years, we have been using a commercial program transformation system to parse and transform a large C++ avionics application that is over one million lines of code and consists of several thousand components (for video demos of this work, see http://www.gray-area.org/Research/C-SAW/). However, we have found that most program transformation systems operate at a level that is not appropriate for general development. Our long-term goal is a framework for language and platform-independent weaving. In this paper, we describe our initial results for constructing a weaver for a specific programming language using the lower-level constructs of a transformation engine. From this initial experience, several observations are made in the paper to establish the feasibility of generic weaver construction using a mature transformation system.

In the next section, we introduce a commercial application that was used as our case study. In Section 3, a set of rules for a specific program transformation engine is used to weave in the crosscutting concerns identified in Section 2. A generalized aspect language built from the transformation rules is defined in Section 4. The paper contains a brief section on related research, and concludes with a summary and description of future work.

2. BACKGROUND CASE STUDY

This section describes the crosscutting concerns that were identified in a commercial distributed application implemented in Delphi, Borland's Object Pascal IDE. The case study application is currently the market-leading solution for resource management software for call-centers and multi-channel contact centers. Three different utility applications within this suite each had their share of problems with respect to scattered and tangled code. The utilities that serve as the case study for this paper were implemented in 42K source lines of Delphi code (the actual size of the main host application is unavailable, but is estimated to be at least an order of magnitude larger – the case study presented here focuses only on the utility applications that offered support to the main application).

We provide a general discussion of several crosscutting concerns that were identified. Other crosscutting concerns exist in these utilities (e.g., database access control logic that is spread over a dozen classes), but we focus on a subset of all identified aspects due to lack of space. The crosscutting concerns described in this section are those that occurred most frequently (i.e., in 20 or more locations). There were no automated aspect mining tools available to help discover these Delphi aspects; they were identified from the intuition and experience of one of the authors who wrote the code. For each concern, the number of times that the implementation redundantly appears is provided, which implies the amount of code that can be removed when modularized as aspects (e.g., Listing 1 appears 62 times and contains about 5 lines of code per case). Section 3 describes our experimentation into weaver construction to ameliorate the modularity problems exhibited by each crosscutting concern.

2.1 Processing Dialog Meter

The Schema Manager is a utility that assists customers in upgrading to a new database schema after installing an update to the application software. It manages the schema evolution problem by converting a database instance to a new schema. Utilities like the Schema Manager often provide feedback to the user in the form of a processing dialog, or meter, which indicates the progress of the overall task. The updating of the progress meter represents a crosscutting concern because the code to increment the meter is spread across the methods that perform much of the functionality (e.g., deleting database triggers, compiling new stored procedures, and other evolution tasks). Listing 1 contains a redundant code fragment that appears in 62 different places of the Schema Manager. This code is necessary to update the processing dialog after each database evolution task is completed.

```
Inc(TotalInserts);
if not ProcDlg1.Process(TotalInserts/TotalCalc) then
  begin
    ProcDlg1.Canceled := True;
    Result := True;
    exit;
  end;    // if not Process
```

Listing 1. Progress Meter Updating

Replication of exception handling code can have negative consequences [28]. With respect to error handling in the Schema Manager, the code fragment in Listing 2 appears 33 times in various methods in order to stop the processing dialog after an exception. It would be desirable to have a way to create a single separate module that describes all of the functionality of updating the progress meter.

```
on E : Exception do
  begin
    dmSERVERS.HandleException(E);
    dmSERVERS.ProcessingDialog.Canceled := True;
  end;
```

Listing 2. Exception Handling Code for Processing Dialog

2.2 Logging of SQL Query Statements

Another crosscutting concern that is scattered throughout the Schema Manager is the logging of SQL code. As the Schema Manager utility upgrades the customer's database to a new schema, all of the SQL commands that are generated to perform the upgrade are logged to a file so that they can be examined later in the event of a problem (please see the fragment in Listing 3). Although a special logging object was created, the numerous places and contexts where the object is called may vary. In fact, the methods of the logging object are invoked in over 50 different places in the Schema Manager. Please notice that the logging call is also context-dependent and parameterized by the name of the query object. The ability to collect the logging actions in a single module would aid in better separation

of this canonical logging concern. Unfortunately, for Delphi and most other programming languages, there are no language constructs to provide these desired capabilities.

```
with dmSERVERS.qryCreateTriggers do
begin

   <statements that build a SQL Create Trigger>

  LogSQL.AddSQL(dmSERVERS.qryCreateTriggers, True);
  ExecSQL;

end;
```

Listing 3. Logging of SQL Query DDL Statements

2.3 Language Internationalization Utility

There are several tasks involved in internationalizing software. One technique is to represent all translations of each text string in a resource Dynamic Link Library (DLL). The creation of this library, however, requires a tool that assists in the management of all of the different strings for all of the supported written languages. The LangMan utility was created to support such a task.

The implementation of LangMan resulted in 24 classes. Several of the classes interact with all of the controls within a Graphical User Interface (GUI) and update a database during any modification to GUI widgets. Among all of the events that are processed in the application, a "dirty bit" is used to keep track of whether a modification is made to a widget. There are 29 unique places in the LangMan source code where access to the Boolean variable `EditMadeDirtyBit` is made.

There were only two different contexts in which the `EditMadeDirtyBit` was accessed. One context simply dealt with setting the value to true or false, based upon a particular situation, and involved lazy-writing of the edit. This was spread across several diverse classes and represented 9 of the places where this concern occurred. The other context in which access to the dirty bit appeared dealt with performing a specific action based upon the value of `EditMadeDirtyBit`. The code for deciding the next action, based on the value of the bit, was identical in each source code location and always occurred as the first statement in a widget click event handler (see Listing 4). Thus, this other redundant code was found in 20 different places in the LangMan application. Any modification or change to the way in which a text string was stored often required a change to the way in which this concern was implemented. This required adaptations to many locations in the code in order to make the change. Forgetting to update the change in any one of these places could result in a loss, or corruption, of data during the modification of a language string.

```
// The user wants to perform another search
// using the same search criteria
procedure TLangMan.SearchAgainClick(Sender: TObject);
begin

  // Perform an update if an edit occurred that might
  // change the focus of the listview
  if EditMadeDirtyBit then
     SaveDBControls;

…
end;
```

Listing 4. Preamble for Widget Button Clicks

2.4 Database Error Handler Synchronization

Often, a commercial application must work with databases from several different vendors (e.g., Oracle, Interbase, and SQL Server). In such a situation, exception handling of database errors is a major difficulty because each database has its own way of raising exceptions. The same conceptual error (e.g., a null in a required field) may be raised in completely different ways with dissimilar error codes. The application, however, must make this transparent while interpreting the exception to provide a meaningful message back to the end-user. To accomplish this transparency, a database error handling DLL was created and integrated into the main call-center application. This library contained 23 classes. The majority of these classes were responsible for handling specific types of exceptions using the "Chain of Responsibility" pattern [13]. After the code was created for the error handlers, a new requirement was added. It was determined that the exception handling code must be thread-safe because numerous clients would be accessing the database at the same time. The addition of this concurrency concern resulted in a manual invasive change to over 20 classes.

An example error handler is shown in Listing 5. In that listing, lines 4-5 and 7-9 represent this single synchronization concern. Furthermore, this exact code is replicated in all of the entry and exit points of each type of error handler.

```
1 function TExNullField.Handle(ServerType: TServerType;
2                       E : EDBEngineError) : Integer;
3  begin
4    TExHandleCollection(Collection).LockHandle;
5    try
6      <database error handling code omitted here>
7    finally
8      TExHandleCollection(Collection).UnLockHandle;
9    end;
10 end;
```

Listing 5. Synchronization in a Database Error Handler

3. WEAVER TRANSFORMATION RULES

This section provides an introduction to the program transformation system that was used in our experimentation. Following the introduction, each of the crosscutting concerns described in Section 2 is modularized as an aspect that is weaved into the target code using low-level transformation rules.

3.1 The Design Maintenance System (DMS)

The Design Maintenance System (DMS) is a program transformation system and re-engineering toolkit developed by Semantic Designs [4]. The core component of DMS is an Abstract Syntax Tree (AST) term rewriting engine that provides powerful pattern matching and source translation capabilities. In DMS terminology, a language domain represents all of the tools (e.g., lexer, parser, pretty printer) for performing translation within a specific programming language. DMS provides pre-constructed domains for several dozen languages (32 languages were supported at the time of our experimentation). Moreover, these domains are very mature and have been used to parse several million lines of code in various domains, including all of the Delphi code of our case-study application. From our survey of the available transformation tools, DMS was the only tool to supply a Delphi domain that was ready for immediate use. Regarding the scalability of the approach, if transformation is to be performed on a language not currently supported as a pre-

constructed domain, significant effort may be required to develop a new domain. Of course, the amount of effort to create a new domain is proportional to the complexity of the new language. Lexer and parser specification in DMS is performed using a process similar to traditional approaches used by compiler generators (e.g., regular expressions and grammar rule definitions). In addition to the available domains, the underlying DMS transformation engine provides the machinery needed to perform invasive software transformations on legacy code (please see [2] for a description of several other foundational approaches to invasive transformation). The DMS rule engine can be generically applied to the parse tree of any defined domain.

To summarize, we chose DMS for this project because of the maturity of the tool, as compared to other transformation engines, and the immediate availability of a large collection of pre-constructed domains. Furthermore, DMS is the only commercial program transformation engine that has been comprehensively assessed in a positive review by a leading IT analyst (please see http://www.butlergroup.com).

The DMS Rule Specification Language (RSL) provides basic primitives for describing the numerous transformations that are to be performed across the entire code base of an application. The RSL consists of declarations of patterns, rules, conditions, and rule sets using the external form (surface syntax) defined by a language domain. 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 tree.

Example DMS transformation rules are given in this section to emphasize the preliminary steps for constructing a Delphi weaver. Please note that we do not expect software developers to actually write transformations at this level. The purpose of this section is to introduce the lower-level transformations that will drive a generalized aspect language. Each of the concerns identified in Section 2 will be revisited to demonstrate the use of RSL to weave in each concern. In each case, there are two key parts to the weaving process: 1) the identification of the join points in the source AST that match a given pattern; and, 2) specifying rewrite rules to operate on those points to derive a new representation (i.e., adding advice).

3.2 Weaving the Processing Dialog Meter

Listing 6 presents the complete RSL transformation rule for weaving the processing meter concern described in Listing 1. On the first line of this transformation rule, the domain to which the rule can be applied is identified (in this case Object Pascal, or Delphi). Patterns describe the form of a syntax tree. Often, they are used for matching purposes to find a syntax tree having a specified structure (as such, they provide a type of quantification across a code base [11]). Additionally, patterns can appear on the right-hand side (target) of a rule to describe the resulting syntax tree after the rule is applied. Patterns can be combined to form larger patterns. The `advice` pattern in Listing 6 specifies the statement associated with the advice of the processing dialog concern (here, `advice` is just a user-defined name for a pattern – the word "advice" has no formal semantics within RSL, but is so named because it conceptually represents the concern that is to be weaved). The code that is associated with the `advice` pattern is the same conditional statement from Listing 1.

The Object Pascal grammar defines the *if_statement* and *statement_list* production rules that are evident in the pattern and rule specifications. Throughout the paper, parts of the Object Pascal grammar are italicized and DMS/RSL reserved words are boldfaced in order to highlight the differences. No visual adornments are given to the regular Delphi source code.

The RSL rules describe a directed pair of corresponding syntax trees. 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. The rule `probe_progress_meter` isolates each node (call to function `Inc`) that increments the database insertion counter. At each join point, the `advice` pattern is weaved into the progress meter. In this case, the former *statement_list* associated in the increment statement is rewritten (syntactically denoted by "**->**" in RSL) to a new *statement_list* that appends the advice to the increment. Rules can be combined into rule sets that form a transformation strategy by defining a collection of transformations that can be applied to a syntax tree.

Meta-variables are used as placeholders for sub-trees, and specified using an escape syntax (i.e., "*identifier*"). An RSL meta-variable can represent the tree defined by a pattern or a parameter to a rule. In Listing 6, the meta-variable reference "`\advice\(\)`" names the tree that is associated with the `advice` pattern and appended to the increment statement. In the next subsection, Lisitng 7 contains parameters (e.g., `\id1`, `\id2`, and `\slist`) to the `probe_logging` rule that serve as placeholders to holes that are filled during the term rewrite process.

```
default base domain ObjectPascal.

pattern advice(): if_statement =

 "if not ProcDlg1.Process (TotalInserts/TotalCalc) then
    begin
      ProcDlg1.Canceled := True;
      Result := True;
      exit;
    end;".

rule probe_progress_meter():
     statement_list -> statement_list =

  "Inc(TotalInserts);"
->
  "Inc(TotalInserts); \advice\(\);".

public ruleset applyrules =  { probe_progress_meter }.
```

Listing 6. Transformation Rule for Updating Progress Meter

3.3 SQL Logging Transformations

Surprisingly, separating the logging of the SQL data definition commands, as shown in Listing 3, was the most difficult aspect to represent in the RSL. The difficulty stemmed from the "`with`" construct in Delphi, which is a shorthand notation for referencing fields within an object by setting a context block. The "`with`" statement of Listing 3 provides a context for accessing the fields of the query object (e.g., `dmSERVERS.qryCreateTriggers`) without having to prefix each reference in the block with the object name. Yet, the logging call that was embedded in this context required the name of the bounded query object.

The trick for the RSL logging rules, as shown in Listing 7, is to trace back to the "with" statement that contains the query object. This is accomplished with an external pattern called add_log_stmt. There are certain things that cannot be specified in the RSL, such as tree-walking strategies. In such cases, it is possible to write external functions that escape from the RSL and return a value. In DMS, exit functions are written in a functional language called PARLANSE, which is a parallel language for symbolic expression that provides an enriched set of interfaces for performing operations on ASTs. Although not described in this paper, the special parallel constructs provided by PARLANSE can offer performance improvements while traversing the hierarchical tree structure during pattern search [4]. Within the AOSD community, there has been extensive research in adaptive and strategic programming to address traversal strategies [26, 27], but there was no mechanism to apply these ideas directly to RSL.

```
default base domain ObjectPascal.

external pattern add_log_stmt (slist1:statement_list,
                      slist2:statement_list,
                      id1:IDENTIFIER,
                      id2:IDENTIFIER): statement_list =
       'add_log_statement' in domain ObjectPascal.

pattern advice(id1:IDENTIFIER, id2:IDENTIFIER):
  statement_list = "LogSQL.AddSQL (\id1.\id2 , True);".

pattern func_call_sig(): "ExecSQL".

rule probe_logging(id1:IDENTIFIER,id2:IDENTIFIER,
                   slist:statement_list):
   with_statement -> with_statement =

  "with \id1 . \id2 do
   begin
     \slist
   end"
->
  "with \id1 . \id2 do
   begin
     \add_log_stmt\(\slist \, \advice\(\id1 \,\id2\) \,
                    \id2 \,\func_call_sig\(\)\)
   end".

public ruleset applyrules =  { probe_logging }.
```

Listing 7. Transformation Rule for SQL Logging

```
(lambda (function boolean AST:Node )function
   (value (local (;; );;
      (;;
          (ifthen(== ~t (AST:ContainsString ?))
            (;;
              (= search_string (AST:GetString ?))
              (ifthen (== (@ search_string)
               arguments:4)  (return ~t)) ifthen
            );;
           )ifthen
           (return ~f)
         );;
       )local
       ~f
    )value
 )lambda
…
```

Listing 8. PARLANSE Visitor Function

The objective of the add_log_stmt external pattern is to insert a new log statement before every call to the ExecSQL statement. However, the parameters to be logged come from the variable access definition that is attached to the "with" statement. Due to space limitation we cannot show the full source for this external pattern, however the visitor function used to find child nodes that match the pattern func_call_sig can be found in Listing 8. Note that the fourth argument that is passed to the external pattern is the function call identifier ExecSQL. The visitor returns true whenever it finds a match to this call statement in the syntax tree. From the external pattern, all matching placeholders are returned and the RHS of the rule weaves in the advice to transform the original syntax tree.

3.4 Transforming Dirty Bits

Recall from Section 2.3 that a dirty bit was used to determine if a lazy-write was needed to update the state of a database as a result of an edit to a language string. That concern required a simple conditional statement to be attached to the beginning of all widget "Click" event handlers. Listing 9 is an RSL transformation that provides support for weaving this aspect into the LangMan code.

```
default base domain ObjectPascal.

external condition func_sig_has_click(id1:IDENTIFIER,
                                      id2:IDENTIFIER)
                = 'func_sig_has_click'.

pattern advice(slist:statement_list) : statement_list =

  "if EditMadeDirtyBit then
      SaveDBControls;
   \slist".

pattern isClick(id:IDENTIFIER): IDENTIFIER
             = id if func_sig_has_click(click(), id).

pattern click (): IDENTIFIER = "Click".

rule probe_dirty_bit (id1:IDENTIFIER, id2:IDENTIFIER,
                      fps:formal_parameters,
                      slist:statement_list):
   implementation_decl -> implementation_decl =

  "procedure \id1 . \isClick\(\id2\)  \fps ;
   begin
     \slist
   end;"
->
  "procedure \id1 . \id2   \fps ;
   begin
     \advice\(\slist\)
   end;".

if ~[modslist:statement_list .slist matches
          "\:statement_list \advice\(\modslist\)"].

public ruleset applyrules =  { probe_dirty_bit }.
```

Listing 9. Listing of RSL Rule for Weaving DirtyBits

The advice pattern in Listing 9 represents the simple conditional statement that is to be prefixed to the widget Click methods. The patterns isClick and click are used to identify the placeholder in the source AST. The left-hand side (LHS) of the rule probe_dirty_bit transforms the source syntax tree to its new representation depending on the external condition func_sig_has_click, which is invoked from the isClick pattern. Note that external conditional functions are coded in PARLANSE. This external condition is needed to match the wildcard "*Click" specification. It is not possible within RSL to look into the contents of a syntax-tree node, but this can be accomplished in an external condition.

The function `func_sig_has_click` takes two arguments. The first argument is the identifier node that denotes places of interest in the search process. The second argument is a constant identifier string that is used to match the place holders in the source tree. Listing 10 shows the PARLANSE function that is used to perform the wildcard pattern matching. It utilizes the pre-defined DMS StringScan and ASTInterface libraries to perform the scanning operation over the placeholders. The function returns true if it finds a slot as specified by the pattern `click`. The `advice` is applied to all placeholders that match this given pattern.

The DMS re-write engine will continue to apply all sets of rules until no rules can be fired. It is possible to have an infinite set of rewrites if the transformations are not monotonically decreasing (i.e., when one stage of transformation continuously introduces new trees that can also be the source of further pattern matches). Notice that there is a condition specified at the bottom of Listing 9. This condition describes a constraint stating that the set of rules should be applied only to those join points where a transformation has not occurred already. Specifically, it states that the rules should be applied when it is not the case that the advice is already prefixed to a statement list. The transformation rule will be applied only once to each Click event handler. Without this condition, the rules would be applied iteratively and fall into an infinite loop.

```
(define func_sig_has_click
 (lambda Registry:MatchingCondition
   (let (;;(= [const_string (reference string)]
                          (AST:GetString arguments:1))
           (= [search_string (reference string)]
                          (AST:GetString arguments:2))
           (= [scanner StringScan:Scan ]
           (StringScan:MakeScan search_string))
        );;

      (value
         (while (== (StringScan:End? (. scanner)) ~f)
              (ifthenelse
                 (StringScan:MatchString? (. scanner)
                    const_string)
                    (return ~t)
                 (StringScan:Advance (. scanner))
               )ifthenelse
          )while
        ~f
      )value

    )let
  )lambda
)define
```

Listing 10. PARLANSE External Conditional Function func_sig_has_click

3.5 Error Handler Transformation

The concurrency control concern from Listing 5 can be weaved using RSL in a style similar to those transformations already shown. In the transformation of Listing 11, the `try/finally` block that implements the concurrency control is wrapped around the critical section. The original critical section is denoted by the *statement_list* that is represented by the `slist` meta-variable in the transformation. The pattern `probe_handle` identifies the slot from the function signature where the advice needs to be applied. The `lock` and the `unlock` patterns are inserted before and after the critical section of the source tree.

```
default base domain ObjectPascal.

pattern probe_handle(id:IDENTIFIER):
        qualified_identifier = "\id.Handle".

pattern unlock():statement =
       "TExHandleCollection(Collection).UnLockHandle".

pattern lock():  statement =
       "TExHandleCollection(Collection).LockHandle".

pattern advice(slist:statement_list): statement_list =
  "\lock\(\);
    try
       \slist
    finally
       \unlock\(\);
    end;".

rule probe_synchronize(slist:statement_list,
                       id:IDENTIFIER,
                       fps:formal_parameters,
                       frt:function_result_type):

   implementation_decl -> implementation_decl =

  "function \probe_handle\(\id\) \fps : \frt ;
   begin
     \slist
   end;"
->
  "function \probe_handle\(\id\) \fps : \frt ;
   begin
     \advice\(\slist\)
   end;".

if ~[modsList:statement_list .slist matches
           "\:statement_list  \advice\(\modsList\)"].

public ruleset applyrules =  { probe_synchronize }.
```

Listing 11. RSL Rules for Modularizing Synchronization

4. AN ASPECT DOMAIN

The transformation rules provided in the previous section describe the low-level details that need to be written in DMS in order to affect the desired weavings. We do not expect software developers to actually write such transformations – the representation is not at an appropriate level for general use. This section introduces our initial work on an aspect domain in DMS that is layered on top of the lower-level rules in order to provide a more appropriate representation. With this domain, users can write their aspects in a traditional aspect-oriented language that resembles AspectJ, rather than the low-level DMS transformation rules. However, the translator of the aspect domain has complete access to RSL, PARLANSE, and the various interfaces provided by the DMS Reengineering Toolkit. Before discussing the specific technique for constructing the aspect weaver, it may be instructive to re-visit the previous section and take a closer look at the DirtyBit example (Listings 4, 9, and 10).

4.1 Dirty-Bit Revisited

The main points of interest in Listing 9 are the definitions of two patterns: `advice` and `isClick`. The pattern `isClick` is similar to a pointcut expression because it is tied to an external condition that matches a named point in the execution flow (i.e., all methods ending in "Click"). The `advice` pattern in Listing 9 is similar to a before advice in AspectJ – notice the prefix of the conditional statement occurs before the original statement list (i.e., "`\slist`"). However, the only thing in our current

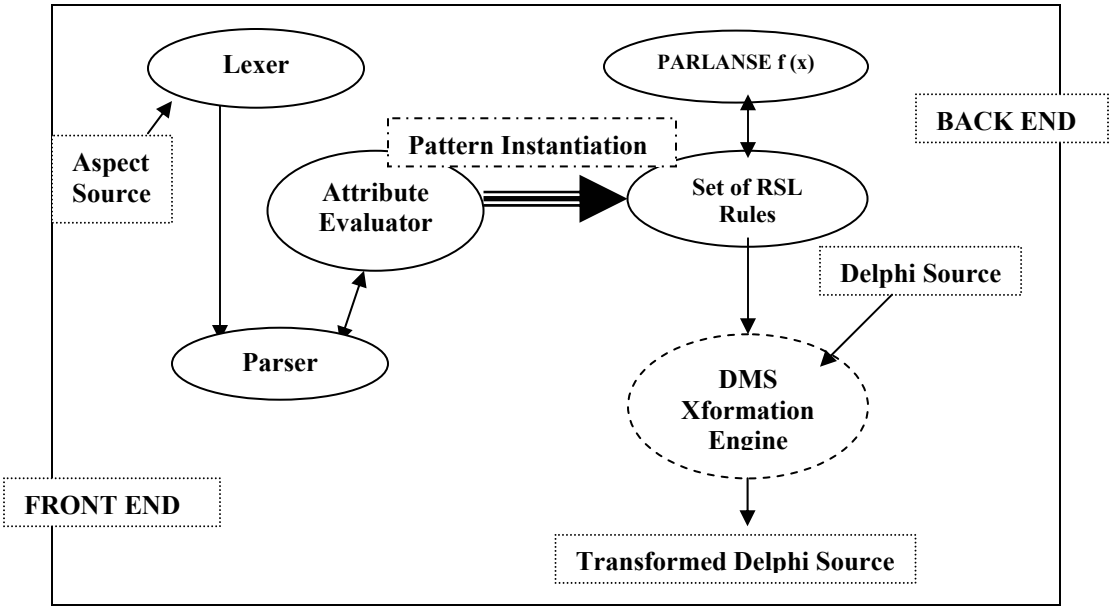

Figure 1. Overview of Weaver Construction Process

definition that prevents this `advice` pattern from being reused is its concrete binding to the conditional statement (i.e., "`if EditMadeDirtyBit then SaveDBControls`"). Similarly, the `isClick` and `click` patterns are not reusable because they are bound to a constant identifier string (i.e., "Click"). The strategy to overcome this limitation is to parameterize these two patterns and instantiate them from a higher-level of language abstraction (i.e., an aspect domain). By parameterizing these two patterns, the `probe_dirty_bit` rule can be renamed to a general rule that captures the essence of before advice. Such a generalization provides one of the many transformation rules that are needed to support the generic transformations of an aspect language.

```
aspect ProbeDirtybit {

  pointcut method_click() :
                    execution(procedure *.*Click(..));

  before() : method_click() {
     if EditMadeDirtyBit then
       SaveDBControls;
   }
}
```

Listing 12. DirtyBit Example in Aspect Domain

Although the Delphi domain was pre-existing in DMS, the initial tools (e.g., lexer, parser, pretty printer) for a general aspect domain were constructed from scratch. This effort will be amortized over each new programming language to which it is applied (note that much of the advice body is pure Delphi code that can be parsed by the pre-existing Delphi parser). Listing 12 shows the DirtyBit example re-written in the aspect domain.

The initial aspect domain is similar to the style of AspectJ [19], but our initial effort is very much a modest subset of AspectJ capabilities. Our first effort supports primitive pointcuts such as call, execution, set, and get, and advice such as before and after. However, this does not include support for cflow or around advice, although these additions are planned for the near future. Our initial effort does not properly address dynamic situations that require a run-time check to determine whether a match is made to a particular join point – the initial effort is purely static at transformation time. Currently, there is no support for reflective constructs like thisJoinPoint.

The `ProbeDirtyBit` aspect produces the same result as the transformation rule of Listing 9. However, instead of hard coding the identifier string "Click" in the RSL specification (Listing 9), the pattern associated with the execution join point (Listing 12) is mapped to a parameterized pattern that is based on the previous `isClick` and `click` patterns (Listing 9). Similarly, the before advice of Listing 12 is instantiated to a parameterized version of the `advice` of Listing 9.

Observe from Listing 12 how the RSL and its associated PARLANSE definitions have been abstracted from the programmer such that he/she is oblivious to the underlying DMS machinery. In a similar way, the other examples presented in Section 3 can be specified in the new aspect domain. For instance, the rule shown in the progress meter example is simply a primitive pointcut to the call on method "`Inc`."

4.2 Overview of Weaver Construction

The principles observed above form the basis for constructing an AspectDelphi weaver. In this sub-section, the front-end and back-end of the weaver are briefly described. An illustrative overview of the weaver construction process is shown in Figure 1. The front-end construction begins with the creation of a lexer and parser for the aspect domain. The lexical creation process is relatively simple and makes use of the DMS Lexical domain. The parser creation process is automated by the DMS StringGrammar domain that generates a LR parser from a grammar definition. Similar to AspectJ, much of the pattern matching is based on type patterns, where a type pattern defines a non-terminal representing the syntax for regular expressions. Much of the lexer and parser can be reused to build aspect weavers for other programming languages. In addition to the

lexer and parser, an attribute evaluator is needed to navigate the syntax tree and evaluate expressions at various nodes of the AST. The lexer, parser, and attribute evaluator form the building blocks for the front-end of the weaver. As mentioned before, mature parsers for many languages already exist within DMS such that base language tools do not need to be re-created.

The back-end consists of the parameterized RSL specifications, which are concretized from the primitives (e.g., call, execution, set , get) specified by the front-end of the system. Note that the RSL specifications are independent of any concrete expressions of the base language and are generic for each primitive class. For example, changing the pointcut expression of Listing 12 to "`execution(*.*Foo(..))`" would not change the RSL specification, but would provide a different parameterized argument. As shown in Figure 1, the generalized set of RSL transformations interact with customized PARLANSE functions that are used for pattern matching.

The bridge between the front-end and back-end is fulfilled by the pattern instantiation process, which occurs during attribute evaluation. The instantiation binds the pattern expressions written in the Aspect domain to its corresponding RSL specifications. The pattern parser retrieves the AST representation of the type patterns and passes them as arguments to the parameterized RSL patterns and rules. Finally, the concrete patterns, rules and original source files are processed by the DMS transformation engine, which weaves the aspects to the base code.

The construction of aspect weavers using this approach has many open questions that we plan to explore further. For example, the generic approach suggested in this section is directed toward an object-oriented (OO) language. The pointcut definitions in this section are defined in terms of specific OO constructs. For languages that are not object-oriented, the pointcut declaration rules may not be applicable. This suggests that the generic approach to weavers may be paradigm-specific. Our future work will investigate the use of DMS to construct weavers for non-OO languages.

5. RELATED WORK

It is commonly known within the AOSD community that aspect weaving can be performed using a general transformation framework for a specific programming language. This observation was first made by Fradet and Südholt as an early position paper [12]. In similar work, a detailed description of a weaver for a declarative language was provided by Lämmel [22], which used functional meta-programs to weave aspects. Our work described in this paper is not focused on foundations of transformation systems, but rather the scalability to which legacy languages can be supported by existing transformation engines.

Several researchers have contributed valuable results in the area of language extension frameworks. As an example, the Jakarta Tool Suite (JTS) contains the basic tools to support the addition of new programming features to Java [3]. JTS assists in the construction of new pre-processors for domain-specific languages that are transformed into Java. Another tool, called JastAdd, is a weaver and compiler construction system for Java based on AST transformation using JavaCC [18]. Although all of these tools have advanced technologies for extension and analysis, these efforts are still bound to a specific language (i.e., Java). In the GENOA system, Devanbu has observed that many program analysis tools offer a fixed-point solution such that their internal structure is unusable in other similar contexts. For example, the parser, type checker, and parse-tree analysis algorithms for a C++ metrics tool are often not reused in other C++ static analysis tools [10]. GENOA claims to have support for re-targetable front-ends, but it is not evident that GENOA provides a diverse set of commercial-grade parsers to realize this claim.

In addition to DMS, there are other popular program transformation systems, such as ASF+SDF [35], Stratego [31, 36], and TXL [8, 33]. DMS, as well as other transformation systems, aim to make it easier to define languages and transformations over those languages. We chose DMS for our research because of our past experience using it on a different project while in collaboration with the vendor of DMS (Semantic Designs). We have confidence that many of the pre-existing parsers that have been defined in DMS are capable of parsing large-scale industrial legacy software. There are many well-defined language definitions provided within DMS that have been used to parse multiple-millions of lines of commercial code. It is possible that the technique described in this paper could also apply to other transformation systems like ASF+SDF. For the technique described in this paper to have a real impact, the ability to parse large code bases in multiple languages is paramount toward providing a framework for injecting aspects into legacy systems.

At the AOSD web site (http://aosd.net/technology), several weaver research prototypes are described for various languages. Aside from AspectJ [19], perhaps the most mature of these is AspectC++ [30]. Our previous experience in trying to apply AspectC++ to a large avionics system revealed that the C++ parser for AspectC++ is not of the quality needed for large-scale industrial projects. In fact, the current download site for AspectC++ (http://www.aspectc.org) indicates a pre-alpha release. A promising alternative to AspectC++ is described in [14], which uses policy classes to weave in temporal invariants along class boundaries. In addition to AspectJ and AspectC++, the following weaver prototypes have been implemented and available for download:

- Apostle for Smalltalk (http://www.cs.ubc.ca/labs/spl/projects/apostle/)
- AspectS for Squeak (http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/)
- AspectR for Ruby (http://aspectr.sourceforge.net/)

There are numerous efforts that support construction of aspect-specific weavers for a specific programming language. The capabilities offered by these tools and frameworks permit new aspect languages to be developed to extend a specific base programming language. An aspect-specific framework is described in [7], which is concerned primarily with issues of concurrent programming (e.g., synchronization, scheduling). Associated with the goals of the concurrency framework, the concept of composing multiple aspect-specific languages is explored in [6]. Related to this aspect-specific language area, the XAspects effort provides a capability for exploration of new domain-specific aspect languages [29]. The XAspects work, however, also is limited to Java development.

Other researchers have also identified the benefits of language-independent aspect weaving. In [21], a language independent approach for .Net is presented. The technique weaves concerns into the Common Language Infrastructure (CLI) of .Net. Although similar in intent to our goals, the fact that this approach is tied to a specific platform and virtual machine makes it less applicable to large legacy systems that were developed prior to the existence of .Net.

Our own previous research was focused on the idea of a weaver construction framework for meta-modeling tools [15]. These weavers are at a higher level of abstraction and permit the separation of crosscutting concerns at the modeling level. That research used techniques from generative programming [9] to instantiate new weavers for each domain that was modeled. This provided domain-independence for each model weaver [16]. This previous work on domain-independent model weavers led us to the idea of applying a similar approach to programming languages. The general technique described in this paper, however, is focused on lower-level issues of source code transformation.

6. CONCLUSION

The crosscutting concerns introduced in the case study of Section 2 highlight the benefits that aspect-orientation can bring to legacy systems. Given the historical tendency of languages to evolve by adopting new paradigms, it is reasonable to assume that aspect-oriented concepts will be integrated into many more programming languages. To expedite this adoption, tools that provide assistance for program restructuring are needed [17]. This will help early adopters assess the feasibility of AOSD within their own organization.

It is our contention that initial efforts to bring aspect-orientation to legacy systems should be robust and mature to the degree that they can be applied readily to large pre-existing applications. The scalability of such a requirement demands the availability of parsers that have been proven capable of handling large collections of source code. Toy parsers will only frustrate users to the point of potential abandonment of adoption.

We believe that development of each new weaver should minimize the duplication of effort from previous weaver construction. A mature program transformation system (like DMS) offers a repository of mature parsers and a general rewrite engine for manipulating syntax trees. These two features help to reduce significantly the effort required to construct new weavers when coupled with a generalized framework that abstracts the details of the accidental complexities of using the transformation system.

The work described in this paper was performed after the official release of the case study application and never used in actual development. However, the case study provides a real context for experimenting with weaver construction for legacy systems. The aspect transformations that were described in Section 3 and generalized in Section 4 can be applied to other commercial Delphi applications. Even though our Delphi weaver environment can parse large amounts of code, there are still many future improvements that can be made to this modest first effort.

The long-term goal of our future work is to provide a language-independent framework for constructing weavers using DMS. This future work will involve the generalization of transformation rules that capture the essence of aspect semantics, as briefly discussed in Section 4. Advances toward this future goal have already been made. Although support currently exists for a subset of AspectJ ideas, it is necessary to look deeper into expanding the purview of this subset by specifying other generalizations for patterns and rules (e.g., patterns for control flow pointcut declarations, and rules for around advice are but a few of the obvious targets for immediate investigation). After the generalized patterns and transformations rules are stabilized, true language-independence will require that these generalizations also be grammar-independent. That is, the generalizations should not refer to productions in a specific grammar. We have an initial idea that applies grammar adapters to each new language that is to be instantiated within the weaver transformation framework (note: the benefits of a general notion of grammar adaptation were initially introduced by Lämmel in [24]). The goal is to make the generalized aspect transformation rules oblivious to the base language on which it operates. The concept of grammar adapters may offer benefits in other types of transformations, such as generalized approaches to refactoring [25] that are language-independent. We plan to explore this idea as future work.

The approach taken in this paper is heavily influenced by the AspectJ notion of AOSD. We are interested in broadening the applicability of the idea to other popular AOSD techniques. That is, a long-term goal is to explore the possibilities of bringing programming language-independence to hyperslices and multi-dimensional separation of concerns [32], adaptive programming and the Demeter method [27], and composition filters [5].

Several videos are available at the project web site (http://www.gray-area.org/Research/GenAWeave). These videos demonstrate the transformations performed on the cases studies described in the paper.

7. ACKNOWLEDGMENTS

We thank Ira Baxter of Semantic Designs for his many insightful comments as we trajected the DMS learning curve to realize the ideas presented in Sections 3 and 4. We extend our appreciation to the anonymous reviewers who offered several suggestions toward the improvement of this paper.

8. REFERENCES

1. Uwe Aßmann and Andreas Ludwig, “Aspect Weaving as Graph Rewriting,” *Generative Component-based Software Engineering (GCSE),* Springer-Verlag LNCS 1799*,* Erfurt, Germany, October 1999, pp. 24-36.
2. Uwe Aßmann, *Invasive Software Composition*, Springer-Verlag, 2003.
3. Don Batory, Bernie Lofaso, and Yannis Smaragdakis, “JTS: Tools for Implementing Domain-Specific Languages,” *Fifth International Conference on Software Reuse*, Victoria, Canada, June 1998, pp. 143-153.
4. 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.
5. Lodewijk Bergmans and Mehmet Aksit, “Composing Crosscutting Concerns using Composition Filters,” *Communications of the ACM*, October 2001, pp. 51-57.

6. Johan Brichau, "Composable Aspect-Specific Languages," *GPCE Young Researchers Workshop*, Pittsburgh, Pennsylvania, October 2002.
7. Constantinos Constantinides, Tzilla Elrad, and Mohamed Fayad, "Extending the Object Model to Provide Explicit Support for Crosscutting Concerns," *Software - Practice and Experience*, June 2002, pp. 703-734.
8. James Cordy, Thomas Dean, Andrew Malton, and Kevin Schneider, "Source Transformation in Software Engineering using the TXL Transformation System," Special Issue on Source Code Analysis and Manipulation, *Journal of Information and Software Technology (*44, 13) October 2002, pp. 827-837.
9. Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.
10. Prem Devanbu, "GENOA—A Customizable, Front-end-retargetable Source Code Analysis Framework," *ACM Transactions on Software Engineering and Methodology*, April 1999, pp. 177-212.
11. Robert Filman and Daniel Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota, October 2000.
12. 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.
13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
14. Tanton Gibbs and Brian Malloy, "Weaving Aspects into C++ Applications for Validation of Temporal Invariants," *7th European Conference on Software Maintenance and Reengineering*, Benevento, Italy, March 2003, pp. 249-258.
15. Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM*, October 2001, pp. 87-93.
16. Jeff Gray, Ted Bapty, Sandeep Neema, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan, "An Approach for Supporting Aspect-Oriented Domain Modeling," *Generative Programming and Component Engineering (GPCE)*, Springer-Verlag LNCS 2830, Erfurt, Germany, September 22-25, 2003, pp. 151-168.
17. William G. Griswold, D. Notkin, "Automated Assistance for Program Restructuring," *Transactions on Software Engineering and Methodology*, July 1993, pp. 228-269.
18. Görel Hedin and Eva Magnusson, "JastAdd-an Aspect-Oriented Compiler Construction System," *Science of Computer Programming*, April 2003, pp. 37-58.
19. 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.
20. 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/
21. Donal Lafferty and Vinny Cahill, "Language-Independent Aspect-Oriented Programming," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, California, October 2003, pp. 1-12.
22. Ralf Lämmel, "Declarative Aspect-Oriented Programming," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Antonio, Texas, January 1999, pp. 131-146.
23. Ralf Lämmel and Chris Verhoef, "Cracking the 500 Language Problem," *IEEE Software*, November/December 2001, pp. 78-88.
24. Ralf Lämmel, "Grammar Adaptation," *International Symposium of Formal Methods Europe (FME)*, Springer-Verlag LNCS 2021, Berlin, Germany, March 2001, pp. 550-570.
25. Ralf Lämmel, "Towards Generic Refactoring," *Third ACM SIGPLAN Workshop on Rule-Based Programming (RULE)*, Pittsburgh, Pennsylvania, October 2002, pp. 15-28.
26. Ralf Lämmel, Eelco Visser, and Joost Visser, "Strategic Programming Meets Adaptive Programming," *2nd International Conference on Aspect-oriented Software Development (AOSD)*, Boston, Massachusetts, March 2003, pp. 168-177.
27. Karl Lieberherr, Doug Orleans, and Johan Ovlinger, "Aspect-Oriented Programming with Adaptive Methods," *Communications of the ACM*, October 2001, pp. 39-41.
28. Martin Lippert and Cristina Lopes, "A Study on Exception Detection and Handling Using Aspect-Oriented Programming," *International Conference of Software Engineering (ICSE)*, Limmerick, Ireland, June 2000, pp. 418-427.
29. Macneil Shonle, Karl Lieberherr, and Ankit Shah, "XAspects: An Extensible System for Domain Specific Aspect Languages," *Domain-Driven Track at Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA Companion)*, Anaheim, California, October 2003, pp. 28-37.
30. 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.
31. *Stratego – Strategies for Program Transformation*, http://www.stratego-language.org.
32. Peri Tarr, Harold Ossher, William Harrison, and Stanley Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *International Conference on Software Engineering (ICSE)*, Los Angeles, California, May 1999, pp. 107-119.
33. *The TXL Programming Language*, http://www.txl.ca
34. William Ulrich, *Legacy Systems: Transformation Strategies*, Prentice-Hall, 2002.
35. Mark van den Brand, Jan Heering, Paul Klint, and Pieter Olivier, "Compiling Rewrite Systems: The ASF+SDF Compiler," *ACM Transactions on Programming Languages and Systems*, July 2002, pp. 334-368.
36. Eelco Visser, "Stratego: A Language for Program Transformation Based on Rewriting Strategies. System Description of Stratego 0.5," *12th International Conference on Rewriting Techniques and Applications (RTA),* Springer-Verlag LNCS 2051, Utrecht, The Netherlands, May 2001, pp. 357-361.