A Software Re-Engineering Method using Domain Models

Jean-Marc DeBaud and Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Abstract

Current software re-engineering technology is typically based on program analysis methods such as parsing and data flow analysis. This is inadequate for two reasons. First, such methods inherently fail to capture the context or purpose of the program. Second, the results of the program comprehension are not directly usable in program evolution. In this paper, we introduce a method that addresses both of these problems. We use a domain model to understand the context of a program and an object-oriented framework to record that understanding. The main step of this method consists of the construction of an executable domain model whose scope covers a family of target programs. A program is then reverse engineered using the domain model both as a guide and as a recording medium. In the last step, developers re-engineer the target artifact using its abstract domain-driven representation. We present a thorough example to illustrate this approach. Issues raised by the confluence of domain analysis and representation, reverse engineering, and artifact evolution are discussed. Implications on future work in the area are suggested.

Keywords: Program re-engineering, domain analysis, reverse engineering, program evolution, program understanding, reuse infrastructure, software architecture, object-oriented framework.

1. INTRODUCTION

1.1. The Problem

The process of software re-engineering is multiphased. In the first phase, the re-engineering of a software artifact entails the comprehension of both *what* the artifact does, i.e., its context, and *how* it accomplishes its purpose, i.e., its structure and control flow. This phase traditionally corresponds to reverse engineering. In the second phase, the artifact is evolved using both the information gained in the first phase and its new specifications. Because of the difficulties of each of these phases, program re-engineering is a complex endeavor.

An artifact's structure and control flow can be determined by lexical, syntactic, and semantic rules for legal program constructs. We know how to do these kinds of analyses well. Tools such as Reasoning Systems' Software Refinery [18] alleviate most practical problems in discovering a program structure and control flow. But knowledge of program structures is alone insufficient to achieve program understanding.

Programs have a purpose. They exist because of some computational needs. But a computation has value only if it models or approximates some aspects of the real world. Insofar as the model is accurate, the program will succeed in performing as expected. And, to the extend that the model is comprehended by the reverse engineer, the process of understanding the program will be eased. Hence, to understand what a program does, one must understand the context in which it evolves; that is, the part of the world it is modeling or its *application domain*.

To complete the re-engineering process, a software artifact must be evolved once it is fully comprehended. The form and representation used to record the comprehension of an artifact are critical to its utilization in the evolution step. Should the new evolution specifications concern the artifact's application domain, then it may be necessary for the programmer to use the domain model to implement them or to acquire some domain knowledge. Hence, the evolution step also benefits from the existence of a domain model.

1.2. Approach Taken and Results

In a previous paper [7] we have argued that application domain modeling provides key concepts to facilitates program context comprehension. In this paper we argue that an *executable* application domain model provides a key technique to perform the two phases of software artifact re-engineering.

The gist of our re-engineering method consists of the construction of an executable, domain-specific reuse infrastructure and its use to drive, record and evolve software artifacts. The main steps of the method are as follows. First, an application domain must be chosen. Second, a domain analysis is performed upon that realm and a domain model is created. Third, that model is expressed in executable form. The particular technology we use is object-oriented frameworks [12]. At that stage, what is indeed a domain-specific reuse infrastructure is in place. Fourth, the application domain model is used to guide artifact comprehension, i.e., the reverse engineering. By a process of instantiating the object-oriented framework, the results of the artifact comprehension are recorded. This process in fact amounts to specializing the reuse infrastructure according to the recovered artifact specifications. At the end of the artifact understanding process, its functionality is replicated by the framework. Now, as the fifth and last step, the evolution of the artifact can begin. This is done by augmenting and/or modifying the previous set of instantiations.

While the process of domain analysis and framework construction is difficult and time consuming, we found our efforts rewarded in many respects. First, we experienced a substantial improvement in the time it took us to comprehend existing programs. We estimate, conservatively, to have speeded the understanding step by a factor of two. Second, recording the artifact specifications generated from the comprehension process was vastly simplified by simply having to parametrize the framework. Third, artifact evolution was also greatly simplified because that process meant evolving the artifact specification as opposed to the source-code. Our experience shows that complex artifact evolution became a matter of minutes for someone familiar with the domain.

This paper is organized as follows. In Section 2, we introduce the domain model used throughout this paper. Section 3 describes the concurrent process of artifact reverse engineering and framework instantiation. Section 4 illustrates artifact evolution. The final section of this paper discusses issues raised by the integration of domain analysis, domain evolution, reverse engineering and validation.

1.3. Related Work

The software engineering community has lately taken a great interest in domain centered approaches. The seminal work of DRACO [16] has been followed by a substantial number of research endeavors [13] [14] [17] [19]. Arango and Prieto-Diaz present current directions on domain analysis and software system modeling in [2]. Yet domain based re-engineering is relatively new.

To develop a domain model of programming language tools, Devambu [9] reverse-engineers it from an existing meta model, GENOA/GENII [8], that he had previously developed. This domain model is then used to construct analysis tools such as CFLOW, CSCOPE and CIA [10]. To produce a domain of acceptable quality this approach pre-supposes input programs embodying a model of the target domain. Another approach, illustrated by Hildreth [11], proposes to use the existence of an ad-hoc, i.e., non-formal domain model to help recover program requirements. In successfully recovering TCAS requirements from specification, Hildreth exposes the power domain-centered reverse engineering, even using a non-formal domain. This method is somewhat similar to ours but lacks the second phase.

1.4. Research Context

The domain that we have chosen to use is Report-Writing. This is a mature, well-understood domain that has been successfully modeled by database management system vendors in the form of report-writing tools. By *report writing* we mean that a program's responsibility is to generate an output report whose contents represent and/or summarize data taken from one or more databases or input files.

The software system that we analyzed for this research is the Installation Material Condition Status reporting System (IMCSRS) [3]. This standard U.S. Army management information system consists of approximately 10,000 lines of COBOL code, broken into 15 programs. IMCSRS is responsible for using input transactions to update a master file and then producing a variety of reports describing the status of Army materiel.

2. THE REPORT-WRITING DOMAIN MODEL

This section covers the first three steps of our method. We choose a domain (step one), Report-Writing, we construct a model (step two, section 2.2) and we introduce its executable form (step three, section 2.3).

But first, we say a word about domain modeling and reverse engineering.

2.1. Domain Modeling and Reverse Engineering

A *domain* is a problem area. Typically, many applications programs exist to solve the problem in a single domain. Arango and Prieto-Diaz [1] give the following prerequisites for the presence of a domain: the existence of comprehensive relationships among objects in the domain, a community interested in solutions to the problems in the domain, a recognition that software solutions are appropriate to the problems in the domain, and a store of knowledge or collected wisdom to address the problems in the domain.

According to Neighbors [15], *domain analysis* "is an attempt to identify the objects, operators, and relationships between what experts perceive to be important about the domain." As such, it bears a close resemblance to traditional systems analysis, but at the level of a collection of problems rather than a single one. Domain engineering/modeling/analysis is an emerging research area in software engineering. It is primarily concerned with understanding domains to support initial software development and reuse, but its artifacts and approaches prove useful in support of reverse engineering as well.

In order for domain analysis to be useful for software development, reuse, or reverse engineering, the results of the analysis must be captured and expressed, preferably, in a systematic fashion, hence the need for a *representation* method [2]. Among the aspects that might be included in such a representation are domain objects and their definitions, including both real world objects and concepts; solution strategies/plans/architectures; and a description of the boundary and other limits to the domain. An unresolved issue, of importance both to software developers and reverse engineers, is the exact form of the representation and the extent of its formality.

What role might a domain description play in reverse engineering a program? In general, a domain description can give the reverse engineer a set of expected constructs to look for in the code. These might be computer representations of real world objects or algorithms or overall architectural schemes. Because a domain is broader than any single problem in it, there may be expectations engendered by the domain representation that are not found in a specific program. Because a program is not always accurate or up-to-date, there may be things missing or incorrectly expressed in the program, despite contraindications in the domain representation. And, because a program is often used for more than one purpose, it may include components that do not appear at all in the domain representation.

Nevertheless, a domain representation can establish expectations to be confirmed in a program. Furthermore, the objects in the domain representation are related to each other and organized in prototypical ways that may likewise be recognized in the program. Hence, a domain representation can act as a schema for controlling the reverse engineering process and a template for organizing its results.

2.2. Report-Writing Domain Model

Figure 1 shows a structural (surface) view of the main components of a report. Four logically independent components, ReportTitlePage, ReportHeader, ReportFooter, ReportTrailerPage, are used for general presentation purpose but they are of little importance in the construction of a report. Although there are three main types of reports: Text, Record and Matrix, we concentrate here only on the report of type Record for the sake of simplicity. Another component, Report_Init, supplies initialization information such as the page length.

In general terms, a Record report is constructed from the sequential output of Pascal-like records or SQL-like rows taken from a master file. The selection of the records can be constrained according to many criteria. A report can also be partitioned and summaries can be accumulated over these partitions or over the entire report. Further, text chunks can be added and parametrized to add diverse information to enhance report readability. Examples of such additions are report headers and footers. A *group* is a conceptual segmentation of a report where the unit of computation concerns one main input file over which a *pivot* is defined. Conceptually, a pivot is an artifact that partitions a sequence of records originating from an input file according to values in a set of fields. Each group has a pivot structure.

To gain a better sense of the domain model, one must describe the dynamic behavior of the domain-specific software architecture (DSSA) components. We use a path expression notation [4] to do this. Path expressions present a clear and unambiguous description of the composability of components. It allows a top level description of patterns of interactions. In this notation, a semicolon indicates a sequence of processes and a word (token) surrounded by brackets is a component. A stated, bracketed component denotes one that can be omitted. A

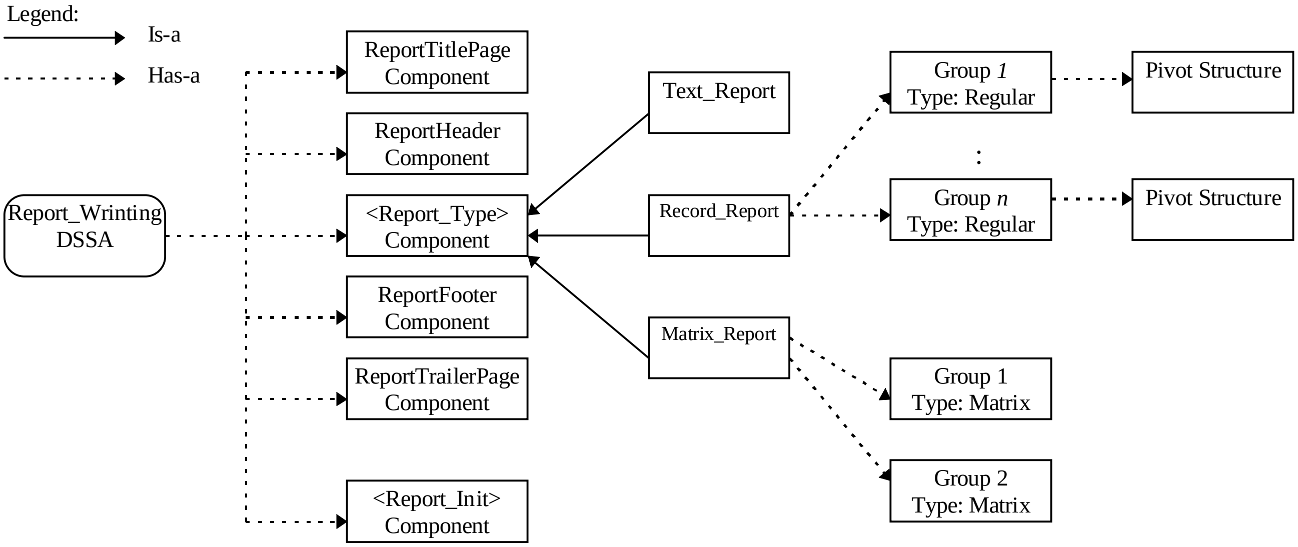

Figure 1: Structural View and Components of the *Report* Domain specific software architecture.

sequence of names separated by commas and surrounded by a pair of brackets denotes a choice where any, but only one, of the component can be chosen. Using this notation, the following prescribes how the main DSSA components are to interact:

General Report-Writing:

Path Report $DSSA_{PE}$ ={(Configure);
(ReportTitlePage)*;(ReportHeader)*;
(Text_Report, Report_Type, Matrix_Report);
(ReportFooter)*; (ReportTrailerPage)*}* **end**

Instantiated for a record report:

Path Report $DSSA_{PE}$ ={(Configure);
(ReportTitlePage)*;(ReportHeader)*;
(Process_Group$_1$(Process_PivotStructure) ;...;
Process_Group$_n$(Process_PivotStructure))
(ReportFooter)*; (ReportTrailerPage)*}* **end**

The *pivot* structure is at the heart of a *report writing*. Figure 2 presents the control flow overview of the algorithm represented by the structure. The concept of pivot was first introduced in the Report-Writing DSSA structural view. A Report-Writing pivot (later referred to as the main pivot) is used by the control structure that opens, read and sequentially processes records from a master file. In the course of that operation, peripheral processing can be done to handle headers, footers and summaries. These peripheral processes happen deterministically during the report writing operation. Pivots serve as partition definition. A new partition is generated each time a new value appears in any of the fields of the set. There can be multiple sets of fields, each defining multiple levels of partition. This results in multiple, embedded sub-pivot structures. Embedded pivot structures are linked together simply by creating the sub-pivot new value test in the body of the parent pivot. Sub-pivot structures are simplified version of the main pivot structure: no reading of the master file occurs. Partitioning is done using a set of fields.

Some comments are necessary to clarify the functionality of the pivot structure (Figure 2). We briefly go through the structure's main steps. At the beginning, a master file is opened. The structure attempts to read a first record and an EOF test is performed.

If the EOF test is negative, the control flow proceeds to another test that determines whether or not a new pivot value has appeared. If it is not the case then the control flow moves to the pivot body. There, the row is processed, summaries are accumulated both at the group and pivot levels. If there are child pivots they are recursively reached in the parent-child direction. But, if a new pivot value has appeared, there are two cases. If it is the very first time that a new pivot value appeared, that is when the report program first starts; a group and pivot header must be printed for the first pivot. For the second and later pivot values, only the pivot header must appear. If it is not the first time, then the old pivot footer and summaries should be processed, the pivot-dependent summaries should be reset to 0 and a new pivot header printed.

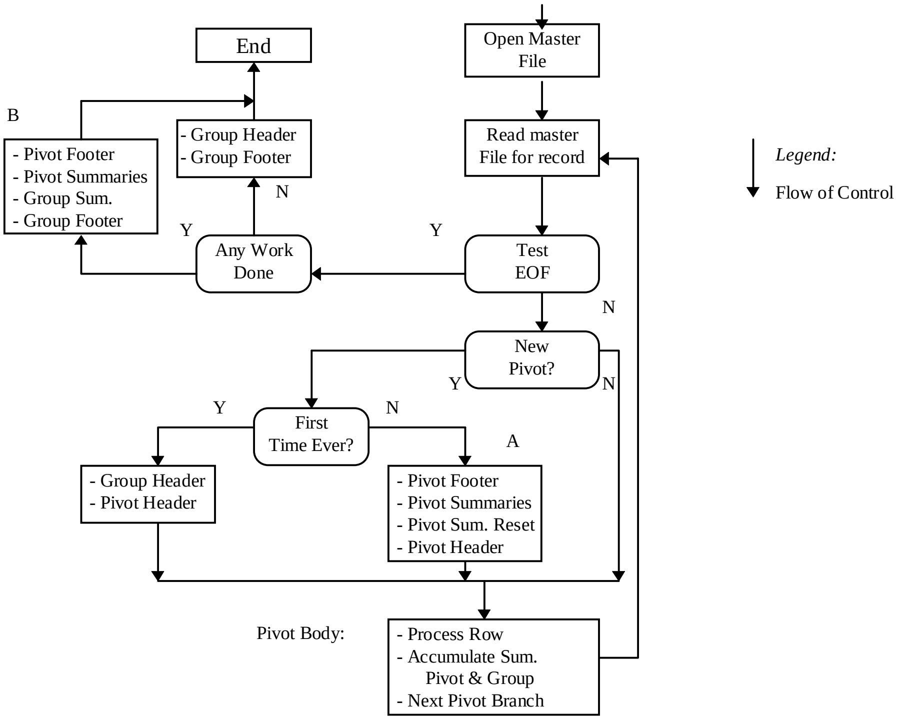

Figure 2: Report-Writting Pivot Structure

If the EOF test is positive, then the structure asks if any work must be done. That is if the master file is empty or not. If no operation is performed, then the group header and footer are printed and the structure is exited. If some operation is performed then the pivot footer and summaries are printed. This is followed by processing the group summaries and footer.

In the pivot structure described above, the order of some operation can be interchanged. In boxes A and B, the decision to process the pivot footer before the pivot summaries is an arbitrary one. Their sequential order could be reversed depending on the particular situation. The same can be said about the group footer and summaries. We should also mention that not all the different elements of a pivot structure are necessary for its existence. For instance, report authors can decide not to have any summaries, pivot header or footers. It is entirely up to the specifics of the program requirements to decide which of these roles are used.

2.3. An Executable Domain Model

To engineer our executable domain model, we use the technique of object-oriented frameworks. An object-oriented framework is a carefully crafted set of abstract classes that collaborate to carry out responsibilities and together embody a reusable design for an entire class of applications or subsystems [6]. An abstract class is an incompletely specified class that is designed to be a template for one or many subclasses. As such, a class is a reusable design of a component [12]. The Report-Writing domain-specific software architecture serves as a guide to understand and document the framework functionality.

Documenting a framework is a difficult endeavor. It is important to convey both structural and dynamic characteristics of the system. In this research, we have used the work of Campbell [5] to present these characteristics. To document a framework, Campbell uses a list of the main abstract class's interface with parameter specification together with a presentation of how these abstract classes function with one another. In this paper, we present only the Report-Writing DSSA and the Pivot control flow structure already introduced to provide the reader with an understanding of the framework functionality.

Using the executable domain.

With the object-oriented framework technology, programming the executable domain model is reduced to instantiating a necessary number of parameters. That is, the framework can work only if a minimal set of parameters has been specified so that a complete, although perhaps rather skeletal, report can be generated. Once this is

Parameter ADT	Explanation
<Report_Type>	Denotes the type of report.
<Report_Groups>	Contains a sequence of Groups.
<Groups>	An ordered sequence of ADTs with two slots: <File_Description> and <Main_Pivot> (plus others, unimportant in this context.)
<File_Description>	Contains a name and an ordered sequence of <File_field> ADTs. Each File_field ADT is composed of a string of characters denoting the name of the field, a type and a length.
<Main_Pivot>	ADT with six slots, of which only Pivot is required in the minimal set. All other slots are optional. Pivot, ;an ordered sequence of fields defining the pivot constraints Printable ;an ordered sequence of Format_Fields to gather information about the location and format of each File_Field to be printed at that pivot level. Header ;an ordered sequence of text objects Footer ;an ordered sequence of text objects Summaries ;an ordered sequence of summaries Sub_Pivot ;a sub pivot ADT that is similar to this one.
<Report_Initialization>	A sequence made of a number of initialization parameters used to set default values regarding to operation of the framework. Included in these are the units used; in most COBOL programs, the unit gradient is the character, the page height, the width, the left, right, top and bottom margin.

Table 1: The Report Writing minimal parameter set (Report type: record)

done, other parameters can be used to specify more complex reports. It is important to understand the origin of such parameters. They are come into existence during the construction of the framework. For each framework component, there are necessary parameters, and for each report there are necessary components. Together, a set of component requires a number of parameters, some necessary, some optional. The Report-Writing minimal parameter set for record reports is reported in Table 1.

3. Reverse Engineering And Domain Model Instantiation

This is the fourth step in our methodology, and it embodies the active process of reverse engineering. We illustrate this step by concurrently instantiating the framework while recovering the target program specifications. For the purpose of this experiment, we have chosen PGUA13, a report program of type record taken from the IMCSRS suite, that contains 648 lines of COBOL. We use the Report-Writing architecture in general and the pivot structure in particular to drive the search for key features in the report.

The first key feature to recover in the minimal set of parameters is the file whose records are processed and its structure. There could be more than one such file but if this is the case, the first processed file (or one conceptually independent of others as shown using program slicing techniques) should be our target. A simple search for a file reading statement followed logically by some instruction looping back to the file reading statement suffices. We obtain from PGUA13 the chunk of code indicated in Figure 3.

This loop structure indicates that `IG09AGU` is the main file whose records are processed. We search P13AGU to find in the DATA DIVISION section the record structure WS-D-REC as indicated in line 427. We return:

```
0097  01  WS-D-REC.
0098      03 WS-D-UIC                     PIC X(6).
0099      03 FILLER                       PIC X.
0100      03 WS-D-SEQ                     PIC XXX.
0101      03 WS-D-NOMEN                   PIC X(8).
0102      03 FILLER                       PIC XX.
0103      03 WS-D-MODEL                   PIC X(10).
0104      03 WS-D-ECC-LIN                 PIC X(8).
0105      03 WS-D-AUTH                    PIC XXX.
0106      03 WS-D-OH                      PIC XXX.
0107      03 WS-D-POSS                    PIC X(5).
0108      03 WS-D-AVAL                    PIC X(5).
0109      03 WS-D-O-SUP                   PIC X(5).
0110      03 WS-D-O-MAINT                 PIC X(5).
0111      03 WS-D-S-SUP                   PIC X(5).
0112      03 WS-D-S-MAINT                 PIC X(5).
0113      03 FILLER                       PIC XXXX.
0114      03 WS-D-UTIL                    PIC X.
0115      03 WS-D-CARD-CODE               PIC X.
0116      03 WS-D-NOMEN-P                 PIC X(10).
0117      03 WS-D-OR-P                    PIC X(2).
0118      03 WS-D-ORG-NAME-P              PIC X(20).
0119      03 WS-D-ALO-P                   PIC X.
0120      03 WS-D-STATION-P               PIC X(5).
0121      03 FILLER                       PIC X(12).
0122      03 WS-SORT-KEY                  PIC X(21).
```

```
0026  0020-READ-INPUT
0427      READ IG09AGU INTO WS-D-REC
0428          AT END
0429          GO TO 0290-FINAL-PROCESSING.
0430      ADD 1 TO WS-CDI.
0431      IF WS-STATION-HOLD EQUAL TO 'ZZZZZ'
0432          GO TO 0040-MOVE-FIELDS.
0433  0030-CK-STATION.
0434      IF WS-D-STATION-P EQUAL TO WS-STATION-HOLD
0435          GO TO 0050-CK-ECCLIN.
0436      PERFORM 0190-MAJ-RTN THRU 0210-MAJ-EXIT.
0437  0040-MOVE-FIELDS.
0438      MOVE WS-D-STATION-P TO WS-STATION-HOLD WS-HD-STATION.
0439      PERFORM 0240-WRITE-HD THRU 0260-EXIT.
0440      MOVE WS-D-UIC TO WS-UIC-HOLD.
0441      MOVE WS-D-ECC-LIN TO WS-ECC WS-P-ECC-LIN.
0442      MOVE WS-D-NOMEN-P TO WS-P-NOMEN.
0443      MOVE WS-D-OR-P TO WS-P-DA-OR-AVG WS-ORP.
0444      MOVE 1 TO WS-PG-CNT.
0445  0050-CK-ECCLIN.
0446      IF WS-D-ECC-LIN EQUAL TO WS-ECC
0447          GO TO 0070-CK-UIC.
0448      PERFORM 0190-MAJ-RTN THRU 0210-MAJ-EXIT.
0449      MOVE WS-D-ECC-LIN TO WS-ECC WS-P-ECC-LIN.
0450      MOVE WS-D-NOMEN-P TO WS-P-NOMEN.
0451      MOVE WS-D-OR-P TO WS-P-DA-OR-AVG WS-ORP.
0452      MOVE WS-D-UIC TO WS-UIC-HOLD.
0453  0070-CK-UIC.
0454      IF WS-D-UIC EQUAL TO WS-UIC-HOLD
0455          GO TO 0080-ADD-TOTALS.
0456      PERFORM 0090-MIN-RTN THRU 0110-MIN-EXIT.
0457      MOVE WS-D-UIC TO WS-UIC-HOLD.
0458  0080-ADD-TOTALS.
0459      MOVE WS-D-POSS TO WS-TOTAL-HOLD.
0460      MOVE WS-D-AVAL TO WS-D-AVAL-HOLD.
0461      MOVE WS-D-O-SUP TO WS-D-O-SUP-HOLD.
0462      MOVE WS-D-O-MAINT TO WS-D-O-MAINT-HOLD.
0463      MOVE WS-D-S-SUP TO WS-D-S-SUP-HOLD.
0464      MOVE WS-D-S-MAINT TO WS-D-S-MAINT-HOLD.
0465      ADD WS-TOTAL-HOLD TO WS-D-TOTAL (1) WS-A-TOTAL (1).
0466      ADD WS-D-AVAL-HOLD TO WS-D-TOTAL (2) WS-A-TOTAL (2).
0467      ADD WS-D-O-SUP-HOLD TO WS-D-TOTAL (3) WS-A-TOTAL (3).
0468      ADD WS-D-O-MAINT-HOLD TO WS-D-TOTAL (4) WS-A-TOTAL (4).
0469      ADD WS-D-S-SUP-HOLD TO WS-D-TOTAL (5) WS-A-TOTAL (5).
0470      ADD WS-D-S-MAINT-HOLD TO WS-D-TOTAL (6) WS-A-TOTAL (6).
0471      GO TO 0020-READ-INPUT.
```

Figure 3: PGUA13 Main pivot structure

Using the above information, one can now start to instantiate the executable domain model:

<Report_Type> is *Record*.
<Report_Groups> is [*GroupOne*] and it contains only the skeleton of the above loop structure specifications. It is automatically created for any new record report, yet it does not contain any specific information now.
<File_Description> for *GroupOne* can be fully specified now because we have found the file structure: [{*WS-D-UIC*,*char*,*6*},{*VOID1*,*char*,*1*},{*WS-D-SEQ*,*char*,3},{*WS-D-NOMEN*,*char*,*8*},...,{*WS-SORT-KEY*,char,21}]
<Main_Pivot> is empty at this stage.
<Report_Initialization> contains default values that will be altered only if the reversed specifications mandate it.

At that stage, we endeavor to locate the main pivot, using the pivot structure shown in Figure 2. A pivot value will be characterized by one or more data file fields comprised in the condition of a test statement. Each one of these will be compared to a value holder that represents the field invariant. For the main pivot, this test will always be located logically below the file read. Sub-pivot conditions, if any, will come logically, one after the other, after the main pivot. Searching through the source code using these guidelines, we find the main pivot at line 434. The field is WS-D-STATION-P and the value holder WS-STATION-HOLD. We notice that a trick was used in lines 431-432 to test for the first value holder. This coding practice renders the code fragile, what would happen if WS-D-STATION-P took the value ‘ZZZZZ’ during run time? Its type does not preclude it.

Now we can begin to write the specification of the main pivot. Note that we have no information yet on the other parameter slots.

<Main_Pivot> is {[*WS-STATION-HOLD*], , , , , }

Using the same reasoning used for the main pivot, we find the first sub-pivot WS-D-ECC-LIN with value holder WS-ECC at line 446. Likewise, we find the second sub-pivot WS-D-UIC with value holder WS-UIC-HOLD at line 454. We can now parametrize the framework with:

<Main_Pivot> is {*[WS-STATION-HOLD], , , , , SubPivot_1*}
<SubPivot_1> is {*[WS-D-ECC-LIN], , , , , SubPivot_2*}
<SubPivot_2> is {*[WS-D-UIC], , , , ,* }

The minimal set of parameter is now structurally complete. Together, they represent the partitioning algorithm used in the report. This is the most important step. Once the partitioning scheme is understood, the rest is only a matter of ‘dressing up’ each partition with summaries, headers, footer and data fields according to the original source-code. Overall, P13AGU has three levels of partition. Each is mainly concerned with collecting summary numbers.

To complete the process, all the source-code must be accounted for by a parameter in the framework. For instance, in the second sub-pivot, lines 459 to 470 must be checked one by one. This leads to the creation of an array of six summaries, accumulated every time the control passes through this part of the program. To complete the summary specifications, formatting and location information must be also be recovered and specified in the framework. Format recovery is done by searching through the slice defined by the summary names and locating the definition in the WORKING STORAGE SECTION. Location recovery is done in a similar fashion.

4. RE-ENGINEERING P13AGU

This is the fifth step of our re-engineering method. In this step, the software artifact is evolved according to new specifications. To get a broader understanding of the types of changes that our method can handle, we ask ourselves the following question: What type of changes would one want to make to a report program? For each, we ponder how successful we would be handling the artifact evolution.

Additional specification. This kind of evolution enhances a report with additional functionality without altering what already exists. Typical examples of such changes are the addition of a summary, header, footers and sub-pivots (additional partition); though sub-pivots can only be added to the last pivot/sub-pivot in the partitioning sequence. Additional specifications are handled simply by additional framework parametrization. For instance, another sub-pivot could be added to P13AGU to logically partition it further so as to gather more refined summaries. For instance, we would have to modify <SubPivot_2> to {*[WS-D-UIC], , , , , SubPivot_3*} and create <SubPivot_3> to be {*[WS-D-O-SUP], , , , ,*}.

Transparent specification. This special type of change denotes the redefinition of variables without modifying the report definition. In maintenance, a situation often arises where data files are evolved. New data fields appear or, most often, some field type or length is changed to accommodate unforeseen input cases. Here, our methodology would also functions well. For a given file, the <File_Description> parameter would be modified. For instance, the field WS-D-NOMEN in line 101 could be extended to 20 characters. <File_Description> would become: [{*WS-D-UIC,char,6*}, {*VOID1,char,1*}, {*WS-D-SEQ,char*,3}, {*WS-D-NOMEN,char,20*},...,{*WS-SORT-KEY,*char,21}]. Dependent variables could then be sliced and modified automatically.

Destructive specification. This is the hardest kind of change. It concerns mainly a change in the main pivot/sub-pivot partitioning structure or in the scope of summary definition. Yet, we have experienced strong success in performing changes of that type. Again, simple changes in the order or in the nature of the pivot structure, done via the framework parametrization, allowed us to dramatically restructure and evolve programs. For instance, we can inverse the order of the sub-pivots in P13AGU. <SubPivot_2> would become <SubPivot_1> and inversely. This dramatically changes the logical partitioning of AG09AGU. Other summary calculations that subscribe to some new artifact evolution specifications can be accumulated.

We note that a combination of these types of changes can occur. Our experience suggests that transparent, additional and destructive specification changes should be done in that order.

5. CONCLUSIONS

In this paper we presented a reuse-based re-engineering method. We demonstrated that, as in a straight forward engineering situation, the use of a domain model in reverse engineering provides high leverage. We maximize this leverage by developing an executable domain model that not only guides the reverse engineering process, but also helps record that process while it is ultimately executable. The latter allows for direct artifact re-engineering.

Domain analysis is time consuming, but it provides high level abstractions that are very powerful. When these are articulated around a domain-specific software architecture, leverage is maximized. This is an advantage because it enables the restriction of the framework parametrization to a limited number of values. The maturity of the Report-Writing domain is crucial to our approach. Constructing an executable domain over an unstable realm is very difficult and we think ultimately fruitless. To further our understanding of representation issues, we are now exploring the possibility of joining related domains (i.e., domains sharing some abstractions) together in a larger reuse infrastructure. This will provide us with a better insight as for the validity of our approach. Domain representation and tool questions as well as related implications are treated in [7].

Current domain analysis methods do not provide a precise and complete framework to ensure total coverage of analyzed domain. In such, there can be no guarantee that the resulting model is complete. We have tried to alleviate this problem by using the technique of frameworks and, in particular, of abstract classes. We use an abstract class's abstract operations both to fix the principle functionality of the class and to serve as a template for potential specialization cases that were not envisioned during domain analysis. This way, the domain model can be easily extended. Yet, when fundamentally new abstractions arise, they often lead to 'semantic earthquakes' and force a major redefinition of the framework structure. In mature domains, the latter case is less likely to happen.

One limitation of our method stems from the lack of the ability to automatically evolve programs in their native source-code language. The executable domain parametrization does not help in accomplishing this task. On the other hand, the comprehension gained of the program during step 4 provides a solid ground to evolve the artifact by hand. Another limitation stems from the nature of the framework development. This method is founded on the belief that the framework performs as specified. This is only a conjecture as it is very difficult, and probably not economically viable, to formally prove its correctness. Yet, it is conceivable to imagine that one could use a commercially available product in place of the framework. It would suffice to understand such a tool well enough to substitute framework parameters for this tool programming or manipulation language.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the original support of the Army Research Laboratory through contract DAKF 11-91-D-0004-0019.

REFERENCES:

[1] Arango, Guillermo and Prieto-Diaz, Ruben. Domain Analysis Concepts and Research Directions, in *Domain Analysis and Software Systems Modeling*, ed. Ruben Prieto-Diaz and Guillermo Arango ,IEEE Computer Society Press, 1991.

[2] Arango, Guillermo. Domain Analysis Methods. In *Software Reusability*. (Eds.) W. Schaeffer, R. Prieto-Diaz, and M. Matsumoto. Ellis Horwood, New York, 1993, pp. 17-49.

[3] *Automated Data Systems Manual, Installation Material Condition Status Reporting System (IMCSRS), Functional User's Manual*, Commander FORSCOM, AFLG-RO, Ft. McPherson, Georgia, April 1, 1984.

[4] Campbell R. H. The Specification of Process Sychronization by Path-Expressions. *In Lecture Notes in Computer Science*, pages 89-102, 1974.

[5] Campbell R. H. and Islam, N. A Technique for Documenting the Framework of an Object-Oriented System. *Technical report UIUCDCS*, University of Illinois at Urbana-Champain.

[6] DeBaud, Jean-Marc. From Domain Analysis to Object-Oriented Frameworks, A Reuse Oriented Software Engineering Methodology. Thesis Proposal, CIMR TR# 94-04, Georgia Institute of Technology, January 1994.

[7] DeBaud, Jean-Marc, Moopen, Bijith, and Rugaber, Spencer. Domain Analysis and Reverse Engineering, *Proceedings of the Conference on Software Maintenance*, pp. 326-335, Victoria, British Columbia, September 1994.

[8] Devambu, Prem. "GENOA/GENII - A customizable, language - and front-end - independent code analyzer", *Fourteenth International Conference on Software Engineering*, Melbourne, Australia, 1992.

[9] Devambu, Prem and Frakes, Bill, Extracting Formal Domain Models from Existing Code for Generative Reuse. Unpublished Technical Report.

[10] Frakes, W.B., C.J. Fox, B.A. Nejmeh. *Software Engineering in the UNIX/C Environment*, Prentice-Hall, 1991.

[11] Hildreth, Holly. Reverse Engineering Requirements for Process-Control Software, *Proceedings of the Conference on Software Maintenance*, pp. 316-325, Victoria, British Columbia, September 1994.

[12] Johnson, Ralph E. and Foote, Brian. Designing Reusable Classes. *Journal of Object-Oriented Programming*, June/July 1988, Volume 1, Number 2, pp 22-35.

[13] Katz, S., Richter, C.A., The, K. PARIS: A system for reusing partially interpreted schemas. In *9th International Conference on Software Engineering* (Monterey, Calif. Mar. 1987). IEEE Computer Society Press, Los Alamitos, Calif., pp. 377-385.

[14] Lubars, M. Domain analysis and domain engineering in IDeA. In *Domain Analysis and Software System Modeling*. IEEE Computer Society Press, Los Alamitos, Calif. 1991. pp 163-178.

[15] Neighbors, James. “Software Contruction from Components”, PhD thesis, TR-160, ICS Department, University of California at Irvine, 1980.

[16] Neighbors, James. DRACO: A Method for Engineering Reusable Software Systems. 1989 ACM, Inc. Addison-Wesley Publishing Co., Reading MA.

[17] Peterson, S., Kang, K., Cohen, S., and Hess, J. Feature-Oriented Domain Analysis (FODA). Feasibility Study. *Technical Report CMU/SEI-90-TR-21*, Software Engineering Institute, Pittsburgh, PA 15213, Nov. 1990

[18] Reasoning Systems, Inc., Palo Alto, CA. REFINE User’s Guide, 1990. For REFINE (TM) version 3.0

[19] Simos, M. The growing of an Organon: A hybrid knowledge-based technology and methodology for software reuse. In *Domain Analysis and Software System Modeling*. IEEE Computer Society Press, Los Alamitos, Calif. 1991. pp 204-221.