Domain Analysis and Reverse Engineering

Jean-Marc DeBaud, Bijith Moopen, and Spencer Rugaber
Georgia Institute of Technology
Atlanta, GA 30332-0280
{debaud,bmoopen,spencer}@cc.gatech.edu

Abstract

Current reverse engineering technology is typically based on program analysis methods such as parsing and data flow analysis. As such, it is limited in what it can accomplish. Knowledge of the application domain containing a program can help overcome this limit and aid the comprehension process. This paper discusses the relationship of application domain analysis and reverse engineering. Two case studies are presented. The first describes how domain knowledge, expressed as an object-oriented framwork, can aid the reverse engineering process for a well-understood domain. The second studies how reverse engineering can be used to build a domain model. Issues raised by the confluence of domain analysis and reverse engineering are discussed, and implications on future work in the area are suggested.

Keywords: Domain analysis, reverse engineering, report writing

1 Introduction

1.1 The problem

Reverse engineering takes a program and constructs a high level representation useful for documentation, maintenance, or reuse. To accomplish this, most current reverse engineering techniques begin by analyzing a program's structure. The structure is determined by lexical, syntactic, and semantic rules for legal program constructs. Because we know how to do these kinds of analyses quite well, it is natural to try and apply them to understanding a program. But knowledge of program structures alone is insufficient to achieve understanding, just as knowing the rules of grammar for English are not sufficient to understand essays or articles or stories. Imagine trying to understand a program in which all identifiers have been systematically replaced by random names and in which all indentation and comments have been removed [4]. The task would be difficult if not impossible.

The problem is that programs have a purpose; their job is to compute something. And for the computation to be of value, the program must model or approximate some aspect of the real world. To the extent that the model is accurate, the program will succeed in accomplishing its purpose. To the extent that the model is comprehended by the reverse engineer, the process of understanding the program will be eased.

In order to understand a program, therefore, it makes sense to try and understand its context: that part of the world it is modeling. Given that the source code by itself is not sufficient to understand the program, the question arises whether there is an alternate approach better suited to the needs of reverse engineering. This paper argues that application domain modeling provides such an approach.

1.2 What is domain analysis?

A *domain* is a problem area. Typically, many application programs exist to solve the problems in a single domain. Arango and Prieto-Diaz [3] 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.

Domain analysis. According to Neighbors [9], *domain analysis* "is an attempt to identify the objects, operators, and relationships between what domain 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 in order to support initial software development and reuse, but its artifacts and approaches will prove useful in support of reverse engineering as well.

Domain representation. 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. Among the aspects that might be included in such a representation are domain objects and their definitions, including both real world objects like "tax rate tables" and concepts like "long term capital gains"; solution strategies/plans/architectures like "partial order of computation"; and a description of the boundary and other limits to the domain like "federal, personal income tax return." 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.

Relationship to reverse engineering. 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.

1.3 Approach taken

In order to better understand the relationship between domain analysis and reverse engineering, we have undertaken two case studies. The first explores how reverse engineering can be aided by the existence of a domain model. We have used a domain model, expressed as an object-oriented framework, to guide the reverse engineering of an application program. Section 2 presents our observations on this process. The second case study investigates how a domain model can be developed by reverse engineering a program. In this case, the evolving domain model is expressed using an Entity Relationship diagram and a data dictionary. This case study is presented in Section 3. The final section of the paper discusses the issues that are raised by integrating domain analysis and reverse engineering.

1.4 Research context

The domain that we have chosen to use is Reporting Writing. This is a stable, 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 the data from one or more input files.

The software system that we analyzed in these case studies is the Installation Materiel Condition Status Reporting System (IMCSRS) [1].

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 Case study I: using a domain model to guide the reverse engineering effort

In this case study, we use a domain model as a reference to guide the reverse engineering effort. We assume that the program functionality under scrutiny falls mostly within the chosen domain. By matching program fragments with domain concepts, we expect to gain a quicker and better understanding of the program. We observe how the domain model creates expectations and a purpose hierarchy, in turn guiding the reverse engineering. Throughout the case study, we express the result of the reverse engineering effort using the domain model notation. In this section, we first present our domain model technology: object-oriented frameworks. We then show domain model fragments from the Report Writing domain and proceed by describing an exercise that uses the domain model to guide reverse engineering. We conclude with some observations on the case study approach and results.

2.1 Object-oriented frameworks

.The construction of a domain model entails performing a domain analysis. We have used Arango's Common Process of domain analysis [2] to construct the Report Writing domain model. Generically, a domain analysis has four major steps: selecting a domain, bounding the domain, eliciting and articulating the domain concepts and operations, and expressing the domain in a representation. The domain representation technology is the crucial step for later use of the domain model.

In the case of this research, we have chosen to use object-oriented frameworks as the domain model representation mechanism. It is a novel use of the technology. An object oriented framework is a carefully crafted set of abstract classes that collaborate to carry out responsibilities in order to 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 a subclass. Each abstract class is a reusable design of a component [8]. We found the object model hierarchy very useful in representing the generalization specialization relationships of the domain concepts. The Report Writing framework provides a clear and normative structure to guide the reverse engineering effort through feature expectations. We also found that frameworks are very easily extended when new domain features are integrated in the model; so long as no dramatic conceptual changes are made to the domain.

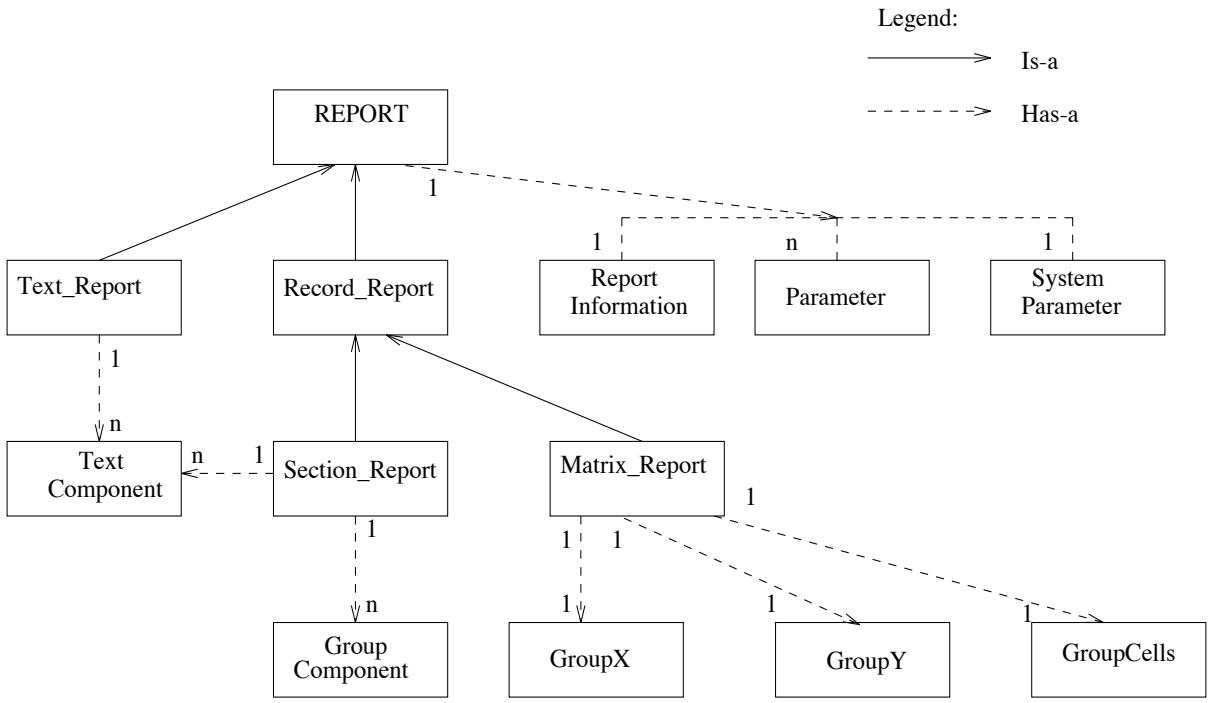

Figure 1 Report Framework components and relationships

2.2 Domain model Fragments

The transition from domain concepts to object-oriented frameworks decomposes a domain in terms of components, mechanisms, synchronizations and a mapping to the object model. At first, we found this transition to be non-trivial. We decided to design a domain specific language (DSL) to capture the domain concepts and help in the transition. Later, we found the DSL to be very useful in recording the changes to the domain model as it evolved. We have structured the DSL as a non-procedural, declarative specification languages. A number of rules and heuristics to map the language to class hierarchy shells have been found [6]. They proved very helpful in designing and manipulating the Report Writing framework. In Figure 1, we present the overall class organization of the Report Writing framework. In this model, reports are either text-based--a set of text strings or calculated fields--or record-based--a set of rows fetched by queries. Record-based reports are structured around groups that describe and organize the rows fetched. There are two types of record reports. A Section report is a sequence of groups; a Matrix report is a two dimensional array whose cell values are a function of the axis values.

In Figure 2, we refine the concept of group. In record-based reports, groups define sections and/or subsections of reports. A section corresponds to the definition and handling specification of one batch of data. It is also possible to nest one group inside another for one level. This forms the parent-child relationship where one value (the parent) serves as selection criterion for a set of others (the children).

To illustrate what is involved in order to model and describe a component in the framework, we choose a simple component to understand. Yet, it is complex enough to represent in details the domain concept representation via object-oriented frameworks. In Figure 3, we present the Summary component. Summaries are operations performed on fields fetched and printed in a report. For example after every sale of the day is printed, one may want to get a total sales indicator. So while the report is printed, a sum must be kept of all the sales. It is also possible to imagine that a head office would print all sales, store by store, printing a total after each store and a grand total at the end of the report. Here we have introduced another concept, the one of sums reset to zero at different times. In fact, the model articulates most numeric summary functions around periodic and running functions. *Periodic* denotes summaries reset at the group level, and *running* denotes those set to zero only once, when the report starts.

Figure 1 Summary Abstract Class Hierarchy

2.3 The case study

For the sake of brevity, we describe only the part of the reverse engineering process related to the summary component. At our disposal are two documents: the Report Writing framework, i.e., the domain model, and a program whose functionality was thought to conform to the Report Writing model. Thorough familiarity with the domain model and the framework, including its protocol interfaces, was a precondition to starting the reverse engineering.

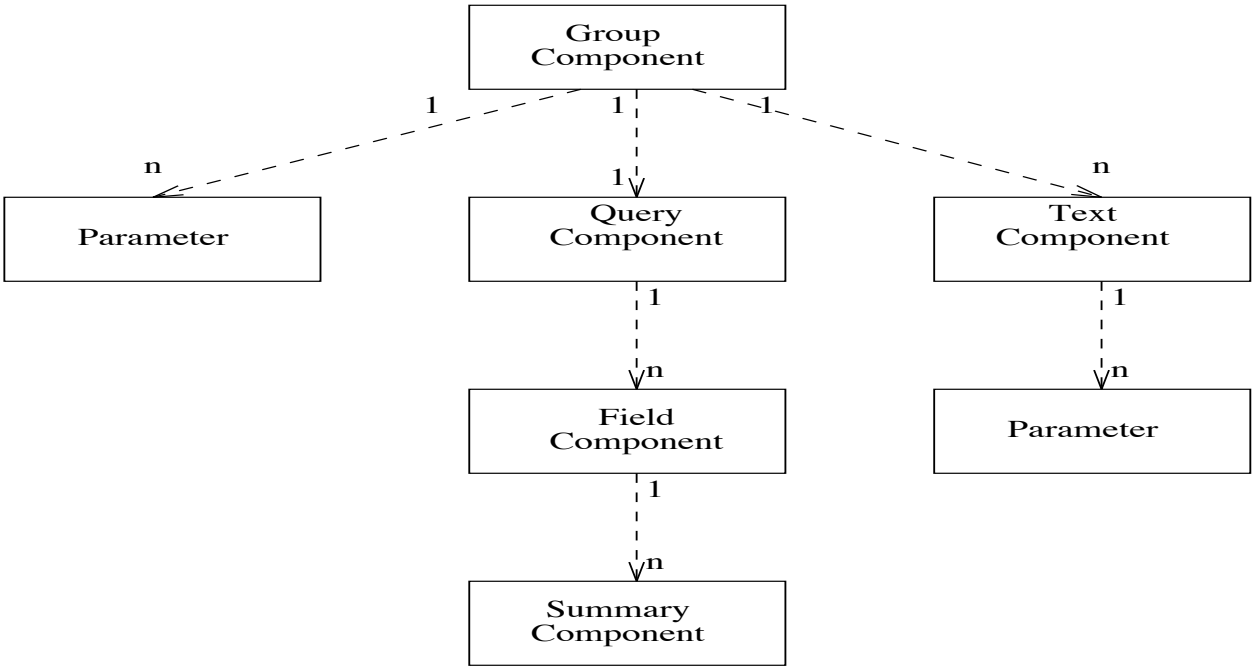

Figure 2 Refined view of a Group component

The first important observation made was that the program analysis process was naturally structured by the expectations engendered by the framework. It became evident that the reverse engineer had built an internal model, not unlike a purpose hierarchy of concepts, how they related to each other and what their purposes were. But it also became apparent that each *purpose token* (a code fragment implementing an expectation), such as summaries, was itself decomposed in a purpose sequence made of patterns. Further, each pattern had a level of confidence attached to it that reflected a measure of the likelihood of having identified a purpose token in the source code. Focusing on a particular feature in source code quickly meant scanning the code for patterns that would suggest the existence of that feature. In Table 1 below we present the purpose token for summaries.

This process worked both ways. If we were before hand looking for summaries, we would specifically look for the patterns listed in Table 1. Reciprocally, stumbling on such patterns while attempting to identify another component would automatically elicit the summary component hypothesis. We found these expectations, derived from the domain model, to be normative in guiding the reverse engineering process.

The second important observation was that the domain model cannot predefine all the possible summary functions that potentially exist. Yet, the process inherent in constructing all the custom summary functions we found is the one in the domain model. Hence, the framework design for that component and its associated model was deemed robust. We found that many reports define their own summaries using a wide range of calculation, sometimes distributed among a number of layers of subcomputations. But the mechanism to make summaries remains the same. It is a stable concept.

A derived observation we also found significant was the ease with which we could add new concrete summary classes to the framework if the generality of the summary made it worth doing so. This was enabled by the design of the summary component abstract class--a single interface to worry about--and its specification enforcement mechanism.

The next important observation is that this method does not deal very well with code not related to the domain. The more unrelated code there is in a program, the more the reverse engineering effort is distracted. The worse case appears when unrelated code manifests itself in patterns that can be matched in purpose tokens.

Recording the reverse engineering effort was singularly simplified with a domain model. After some attempts to use the DSL, it became clear that a more effective method was to instantiate the object-oriented framework. One would select the relevant component, such as summaries, and instantiate it. All that must be recorded are the actual parameters used with the interface of the component. Hence, the recording of the reverse engineering effort becomes the creation of the same program in a new technology, not withstanding code unrelated to the domain.

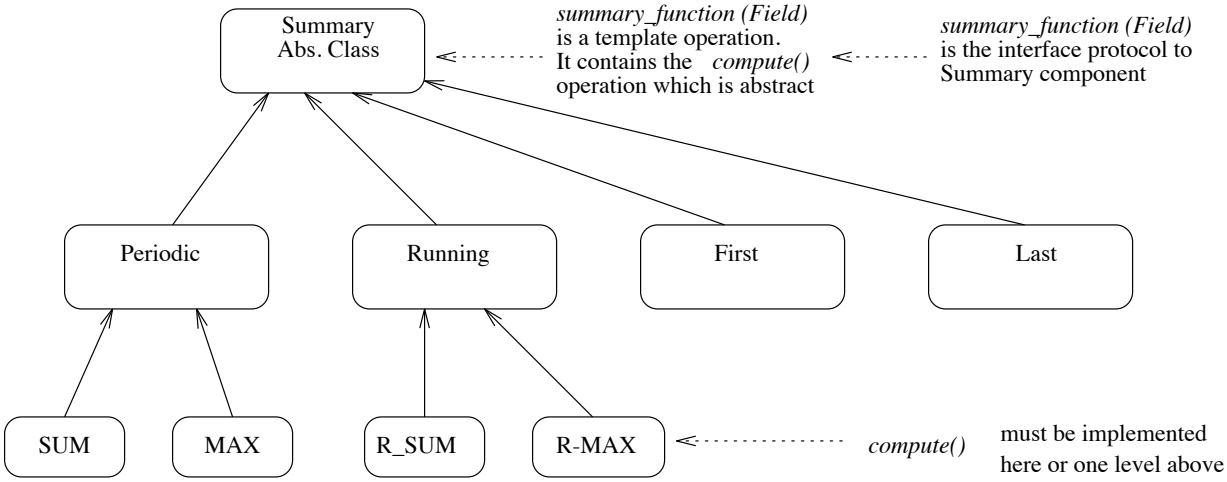

Figure 3 Summary Abstract Class Hierarchy

2.4 Conclusions from the case study

Object-oriented frameworks provide a clear and normative structure to guide the reverse engineering effort through feature expectations and purpose patterns. To achieve this, domain analysis is crucial in understanding the domain. Frameworks are also easily extended with new domain features (i.e., concrete classes), so long as no dramatic conceptual changes are made to the domain. Frameworks, like every model representation technology, are prone to difficulties when used to model fluid domains. Domain analysis must produce a model that is robust to changes in the domain; a non trivial task. Frameworks are based on the object model, making their overall understandability to users somewhat easier than other representations. Frameworks are themselves implementations. In that respect, they can help the discovery of purpose patterns by providing source code constructs that can be matched with the code to reverse engineer, given that the dissimilarities of the two languages are not too drastic.

Having first developed the Report Writing framework for forward engineering purposes, we were impressed by the capabilities of the technique to record the process of reverse engineering via the framework partial instantiation. It was an unexpected benefit. We were also impressed by the power of domain models to build expectations and their related prescription abilities.

We must also caution the reader that the Report Writing domain is a well understood one, though it is nontrivial. This effort was successful both because the domain is fairly stable and because it is not extremely complex. Developing a robust domain model for flight controls, cockpit design or operating systems would be a better validation of the technology.

3 Case study II: growing a domain model by reverse engineering

In this case study, we reverse engineered a program named P14AGU from the domain of report writing and studied the domain model that resulted. This program prints a report called AGU017. It is part of a group of COBOL programs used in IMCSRS--a material status reporting system used by the U. S. Army. The program consists of 636 lines of code of which 243 are PROCEDURE DIVISION statements. This, in COBOL, is the actual executable code. The rest of the statements in the program are data declarations and definitions found in the WORKING STORAGE and FILE sections.

3.1 Process

Synchronized Refinement [10] was used to reverse engineer the program. This technique analyzes the program text from the bottom up, looking for stereotypical cues that signal the implementation of design decisions. At the same time, it synthesizes an application description from the top down, using expectations derived from the various domains relevant to the program. For example, the report writing domain suggests that somewhere in the program should exist code for counting lines, columns, and pages and for printing header and footer information on each page. When suggestive variable names are encountered in the code, an effort is made to confirm the expected use and to annotate the derived description. As expectations are met or refuted, the description of the system grows. As domain artifacts are identified, they are abstracted from the code, causing it to shrink in size. In this way, a complete application description grows synchronously with a refined and abstracted program description.

The results of Synchronized Refinement are recorded as a sequence of annotations. Each annotation indicates

Patterns	Level of Confidence
Variable name suggests a summary.	Generates a strong expectation. But it must be confirmed later.
Declaration of a numeric variable.	Generates some expectation. Could be a summary.
(*) Assignment of a variable to 0 early in the program.	Generates some expectation. Could be a summary.
Assignment of a variable to 0 in the middle of the program.	Generates some expectation. Could be a summary. Reinforced if (*) was observed.
(+) Use of a variable to summarize a field value.	Generates a good expectation.
Same as (+) but inside a loop fetching records one at a time.	Generates a strong expectation.
Use of another identified summary in formula to operate on a variable.	Generates a very strong expectation.
Statement printing a variable using, even indirectly, a field.	Generates a strong expectation.
Statement printing a variable using, even indirectly, a field. And, thevariable already had generated a good or strong expectation.	Certifies the expectation.

TABLE 1 Summary Purpose Token

that a particular domain concept has been implemented with a specific collection of programming language constructs. Moreover, the implementation reflects a specific decision made by the designer of how best to represent the concept in the code.

3.2 Initial expectations

The reverse engineer knew initially that IMCSRS was a management information system and that the particular program was responsible for producing a report. These facts set up some initial expectations that were further elaborated on by reading the program's introductory commentary.

```
*REMARKS. THIS PGM SPOOLS OUT AN EQUIP
AVAIL DENSITY RPT FROM
*    DATA ON DISK CREATED IN PGM P07AGU
- DATA IS ALREADY
*    SORTED ON STA-DIV, ECCLIN AND UIC -
PAGE EJECT FOR STA-DIV,
*   PGM COMPUTES % OR AND 3 TOTALS LINES
WHEN ECCLIN CHANGES.
```

Thus, P14AGU produces an equipment availability density report from the input data. Also, the data is sorted on the fields, STA-DIV, ECCLIN and UIC. The program gives a page eject for any change in the value of the STA-DIV field. It also computes the percentage OR and three totals whenever ECCLIN changes.

Some of the initial expectations engendered by this information includes the existence of code related to the following concepts: an input file; page management, including page headers and footers; overall report header and summary; and group management code, particularly some means by which totals and averages are computed. Note that the expectations are more than a list, some concepts, such as page footer, are clearly subordinate to others, such as page management. Note also, that concepts corresponding to entities (objects) were more apparent than those expressing relationships.

Domain-generated expectations together with programming experience suggested an organization to the code that looks like the following.

```
begin
    print Report Header;
    for each input record
      print out a line of the report;
      process_grouping;
      process_pagination;
    endfor;
    print Report Footer;
end;
```

Of course, other program architectures are possible, and as more applications from this domain are analyzed, it is to be expected that a more general description of the architecture for report writing programs will evolve.

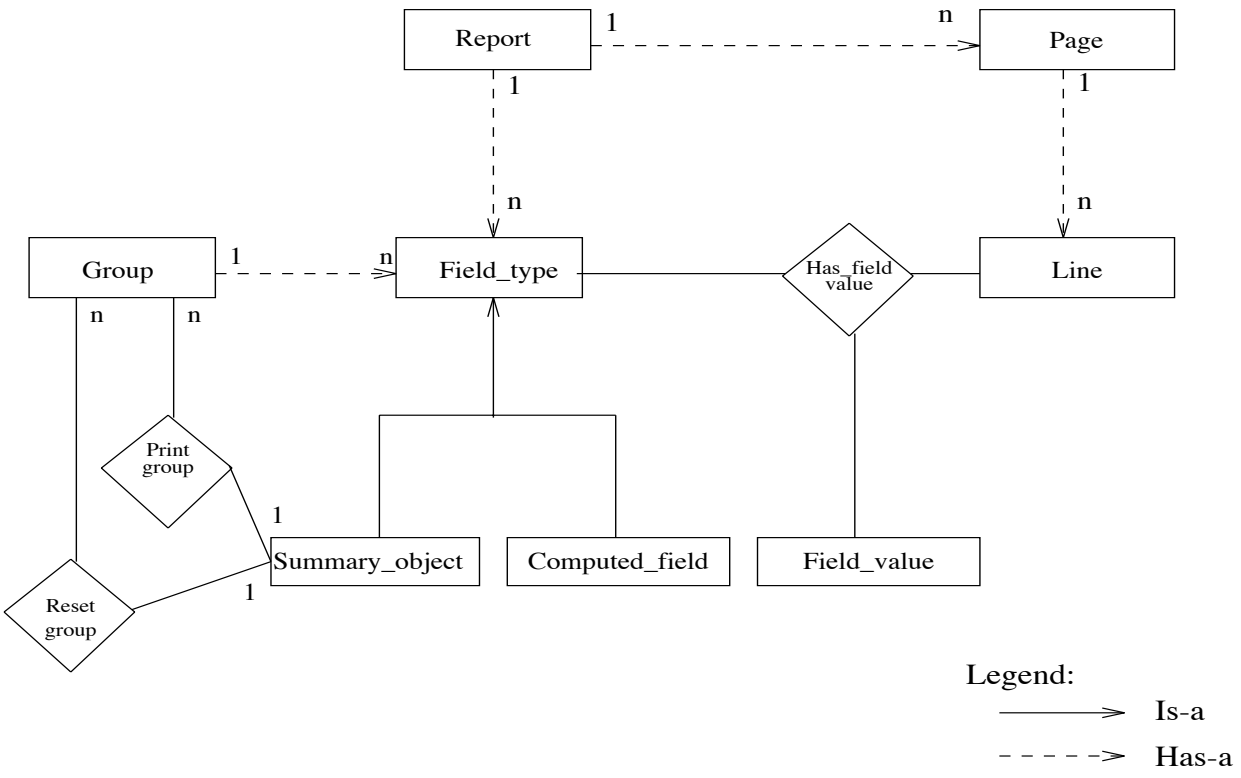

Figure 4 Domain Represented by an Entity Relationship Diagram

The architecture provided enough guidance to begin the program analysis process. In fact, a textual scan of the program, looking at paragraph labels and variable names resulted in tentative recognition of code for printing headers, recognizing group boundaries, pagination, computing group totals, and final report processing.

3.3 Example

Our initial expectations were, by and large, confirmed during the course of program analysis. However, the model had to be refined as we learned more about the program. For example, the concept of a summary field was added to the model as follows.

The three sorted fields mentioned in the REMARKS were taken to imply the existence of three levels of grouping based on sorted input records. The REMARKS also mentioned three total lines that were printed when ECCLIN, which is a field in the input record, changes its value. This suggested that the grouping is hierarchical, with the uppermost level based on ECCLIN. Thus what could be expected for the grouping implementation was:

```
for each input record
    print a report line
    case grouping-1
      print group-1, group-2 and group-
3 totals
    case grouping-2
      print group-2 and group-3 totals
    case grouping-3
      print group-3 totals
    update the groupings appropriately
```

Grouping-1 was expected to be based on ECCLIN as three total lines were said to be associated with it. Groupings 2 and 3 were supposed to be based on STA-DIV and UIC although the exact relationship was not initially clear. These expectations were corroborated by the existence of three totaling paragraphs:

`0080-1-TOTAL-UP, 0090-2-TOTALS, 0100-3-TOTALS.`

The WORKING STORAGE section entries corresponding to these totaling paragraphs were recognized and added to the domain model as *summary objects*.

Totaling code from the program consisted of a sequence of four IF statements, each of which checked for a field value change, updated a total, and stored the new field value for later comparison. An example of one of these is the following.

```
IF POS-1 NOT = WS-POS-1
    PERFORM 0080-1-TOTAL-UP THRU 0085-
EXIT
    PERFORM 0090-2-TOTALS
```

```
PERFORM 0100-3-TOTALS
MOVE P-OR TO WS-OR-STD
MOVE P-ECCLIN TO WS-ECCLIN.
```

The IF statements implements the case statements in the expectations. Also, the field that was expected in this piece of code, ECCLIN, is found here. But the presence of four IF statements instead of three was unexpected. Also, the ECCLIN field IF statement seemed to do only one level of totaling. This eventually lead us to understand that the initial REMARKS were incorrect and that four levels were present in the sorted input data instead of the indicated three.

3.4 Representation

The reverse engineer had an initial conception of what the Report Writing domain meant. As the study proceeded, this model was refined. Important missing concepts were incorporated. Relationships were refined based on the knowledge gained from the process. The final version took the form of an Entity Relationship (ER) diagram and an associated data dictionary. A portion of the ER diagram is shown in figure 4.

3.5 Observations

Although we were able to successfully construct an initial domain model starting from P14AGU, the process was not as straightforward as we thought. In particular, we noted the following phenomena in the course of our efforts.

- In addition to the domain of report writing, it was essential for the reverse engineer to also understand the domain of programming. In particular, knowledge of how abstract constructs can be represented in a particular programming language was required.
- A domain model can help detect bugs. In the case study, the domain model pointed out a discrepancy between the REMARKS and the actually code. This implies that automated tools must be able to deal with improperly implemented domains.
- Sometimes the program structure suggested by the domain model has been distorted in its course to implementation. This phenomenon can take the form of displaced code. In the case of P14AGU, code for constraint checking was expected it to be in the grouping code, whereas it was found with the PAGE-END paragraph. The lead to a confused model until the phenomenon was recognized and understood.
- The level of the programming language in which the program is written can play an important role. If it is too low, the reverse engineer must "decompile" before matching expectations to code. Obviously tools can help with this. For example, in P14AGU, IF statements were used in situations where a CASE statement would have been clearer, if the language supported it.
- Programs will typically implement concepts from more than one domain. In our case study, in addition to the report writing and programming domains, there is an equipment domain, without knowledge of which, true understanding of the program is impossible. In addition, many programs make use of mathematical knowledge that must be understood by the reverse engineering. Often, too, knowledge of a specific machine is required.
- An automated mechanism for abstracting code is essential. Some code exists only to provide an optimization. It is not essential to the underlying program functionality. Being able to replace it with a clearer (albeit less efficient) abstraction promotes further understanding
- Programmers write code in stylized ways, sometimes called cliches or idioms. Understanding a program may involve unraveling an idiom to discover what the code is actually accomplishing. Wills has made some progress in recognizing these patterns [11].

In conclusion, we were able to construct a formal domain model from the code. However, clearly one instance of a program from the domain is not sufficient validation. Moreover, the value of doing so is severely modulated by the presence of multiple domains.

4 Discussion

4.1 Issues raised by the case studies

The case studies described above have raised a variety of issues. They can be broken into the categories of representation, methods, and tools.

Representation

- The fundamental question concerning representation is what is the best form for a domain description to take in order to support reverse engineering, or whether, in fact, a single, "best" representation can be devised [5].In our case studies, we used object oriented frameworks and Entity Relationship diagrams, but a plethora of alternatives exist. Although domain theorists do not yet agree on how to represent domain information, a consistent representation is a prerequisite to broadly applicable tools.
- Related to this question is the issue of how much formality a domain representation should entail. Many of the domain models in the literature use sophisticated mathematical techniques. Not only does this present a barrier to some potential users, but it raises the question of how best to deal with informal information. Of course, some degree of formality is a prerequisite for tool support.
- Another issue concerns the relation of the domain representation to the program description that emerges as a result of the reverse engineering process. If a domain has a natural structure or if programs solving domain problems tend to have a favored architecture, then the program description should somehow mirror this. But what if the program includes several domains, each with their own preferred structures?
- Several technical questions also exist concerning domain representations. How much detail should they include? How should they deal with optional information? How should they express abstractions such as might arise with a parameterized domain?

Methodology

- Perhaps the overriding question of this research is whether domain analysis can help in the reverse engineering process at all. Our case studies indicate that this is so, but more work needs to be done.
- Corollary to this is the question of how best to make use of the domain knowledge obtained. For example, even if we imagine existing, complete, well-organized descriptions for each of the domains related to an application program, it is not clear how best to combine them to understand a program. Which one should we start with? How do we coordinate a search for multiple expected constructs derived from several domains?
- A subsidiary methodological issue concerns knowledge of the domain learned while examining a program. We would like domain descriptions to grow and become more complete over time, but domain descriptions need to be definitive, and the reverse engineer need not be a knowledge engineer nor have sufficient expertise to judge the accuracy, relevance, and placement of the new information in the domain description.

Tools

- They not only include a lot of information, but the information is highly interrelated. The question then arises of how best to access this information? Are program browser-like tools sufficient? CASE tools? Or is a new approach required?
- Tools that access domain information may have to do a lot of specialized inferencing, for example, to confirm that a given program contains a valid implementation of some domain concept. What are the implications of this? A variety of inferencing tools exist that can be categorized as trading off power for efficiency. Where on this curve is the right place for domain-based reverse engineering tools?
- An intriguing question pertains to tool generation. Mature domains enable application generation technology, such as report writers. How about the inverse? Can we build application analyzer generators? In fact, at least one such tool exists, GENOA, a language-independent analyzer generator [7].
- Finally, what should be done with all the existing reverse engineering tools that do not take advantage of domain knowledge? Can they be adapted or integrated? Need they be?

4.2 Implications

The variety of approaches to domain analysis discussed above suggest themes that bear upon the use of domain analysis for reverse engineering.

- First, domain analysis, as it exists today, is primarily intended to support reuse. As such, concerns for information modularization and retrieval are paramount.
- There is a role for both formal models and informal information; the former supporting precise mappings to solutions, and the latter aiding in problem expression, as well as design rationale capture.
- Domain analysis, as currently practiced, is concerned both with problem analysis and solution design. This merging of concerns flies in the face of traditional software engineering advice to avoid prematurely confounding problem analysis with consideration of solutions.
- There is a strong concern in domain analysis with structural issues. Delineation of basic objects, operations, and associations is disciplined by the use of classification and aggregation abstractions.
- Finally, because domains are inherently more general than the problems they subsume and because domain models are intended to foster specialized solutions, inferencing and program generation technology are strongly indicated.

4.3 Conclusions

The argument for the use of domain analysis in software development is compelling: we need to improve productivity, and to do this, we should reuse as much existing software and its associated documentation as possible. We obtain maximum leverage in reuse by using the highest possible level of abstraction--domain knowledge.

The argument for relating domain analysis to reverse engineering is equally convincing: reverse engineering involves understanding a program and expressing that understanding via a high level representation; understanding concerns both what a program does (the problem it solves) and how it does it (the programming language constructs that express the solution); and the more knowledge we have about the problem, the easier it will be to interpret manifestations of problem concepts in the source code. Based on this logic, We fully expect that any major breakthrough in the automated program understanding and reverse engineering area to take significant advantage of domain information.

Acknowledgement

The authors gratefully acknowledge the support of the Army Research Laboratory through contract DAKF 11-91-D-004-0019.

References

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

2. Guillermo Arango, Domain Analysis Methods, in *Software Reusability*, ed. W. Schaeffer, R. Prieto-Diaz, and M. Matsumoto, pp. 17-49, Ellis Horwood, New York, 1993.

3. Guillermo Arango and Ruben Prieto-Diaz, 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.

4. Ted J. Biggerstaff, Design Recovery for Maintenance and Reuse, *IEEE Computer*, vol. 22, no. 7, July 1989.

5. Richard Clayton and Spencer Rugaber, The Representation Problem in Reverse Engineering, *Proceedings of the First*

Working Conference on Reverse Engineering, Baltimore, Maryland, May 21-23, 1993.

6. J.-M. DeBaud, From Domain Analysis to Object-Oriented Frameworks, A Reuse Oriented Software Engineering Methodology, Thesis Proposal, Georgia Institute of Technology, January 1994.

7. Premkumar T. Devanbu, GENOA - A Customizable, Language- and Front-End Independent Code Analyzer, *Proceedings of the Fourteenth International Conference on Software Engineering*, pp. 307-319, Melbourne, Australia, May 1992.

8. R. E. Johnson and B. Foote, Designing Reusable Classes, *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22-35, June/July 1988.

9. James M. Neighbors, *Software Construction from Components*, Ph.D. thesis, TR-160, ICS Department, University of California at Irvine, 1980.

10. Stephen B. Ornburn and Spencer Rugaber, Reverse Engineering: Resolving Conflicts between Expected and Actual Software Designs, *Proceedings of the Conference on Software Maintenance*, pp. 32-40, Orlando, Florida, November 1992.

11. Linda Mary Wills, *Automated Program Recognition by Graph Parsing*, 1358 (Ph.D. Thesis), MIT Artificial Intelligence Laboratory, July 1992.