DOMAIN ANALYSIS AND REVERSE ENGINEERING

Spencer Rugaber

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
(404) 894-8450
spencer@cc.gatech.edu

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[1]. 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. But should not the context be described in the program documentation or in comments in the program text? In principle this is true, but in practice there are several reasons why this may not be the case.

First, programs change, and often documentation does not change synchronously [23]. A successful program evolves to meet new requirements, to improve efficiency, to fix problems, or because the original approximation no longer provides an accurate model of the real world context of the program.

The second reason is that a program typically solves a specific problem, but the model it assumes is much broader. Think, for example, of a program that computes income taxes owed. Computationally, such a program performs simple arithmetic on a few input values, but understanding the program well enough to modify it to reflect a change in the tax laws requires extensive knowledge of our laws and our economy. Looking only at the documentation for this single program may not provide a broad enough view of the program's context.

A related reason is that programs often do not exist in isolation. That is, a set of programs may jointly solve a collection of related problems. For example, a suite of programs that exist to manage a business (payroll, accounting, taxes, inventory, billing, order processing, etc.) share a great deal of knowledge about the company, and describing this information in the documentation of each program would either be redundant or fragmentary.

The final reason why documentation of a single program is often not sufficient to understand the program's context is that the trees and underbrush of computations in a programming language can get in the way of seeing the forest of the problem it solves. That is, source code is often not the best way to think about

[1] This thought experiment was developed by Biggerstaff [7].

or describe a problem solution.

Given that the source code by itself is not sufficient to understand the program and given that traditional forms of documentation are not likely to provide the information needed, the question arises whether there is an alternate approach better suited to the needs of reverse engineering. This essay argues that application domain modeling provides such an approach.

2. DOMAIN ANALYSIS

Domains

A *domain* is a problem area. Typically, many application programs exist to solve the problems in a single domain. Arango and Priéto-Diaz [2] 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.

Once recognized, a domain can be characterized by its vocabulary, common assumptions, architectural approach, and literature.

- The problems in a domain share a common vocabulary. In the income tax domain, terms like "adjusted gross income," "dependent," and "personal exemption" are commonly used.
- The programs that solve problems in a domain may also share common assumptions or tactics. For example, in the income tax domain it is understood that there are multiple places where the same information, such as adjusted gross income, must be supplied and that all of these sites must be altered when any one of them is changed.
- It may be the case that a common architectural approach is used to solve problems in a domain[2]. In the income tax example, a given computation will likely obtain its operands from the results of other computations. The set of computations and the dependencies among them form a partial order, and programs in this domain are likely to be structured in a way to maintain and take advantage of this ordering.
- A domain exists independently of any programs to solve its problems. It likely has its own literature and experts. For example, there are many "how-to" books in the income tax domain, and experts seem to pop up like weeds every spring.

Domain Analysis

According to Neighbors [20], *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.

---

[2] One possible explanation for this is that the domains we currently understand best are those that have been around for a while and for which satisficing solution strategies have evolved. The solution strategies/architectures then become a factor in defining the domain and deciding whether a problem is in the domain at all.

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 like tax rate tables or deductions. Or they may be algorithms, such as the LIFO method of appraising inventories. Or they might be overall architectural schemes, such as a data flow architecture for implementing the computational partial order described above.

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 (the inventory algorithm may not appear in a program to compute personal income taxes but might in a business tax 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, such as a checkbook balancing feature in an income tax package.

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.

3. APPROACHES TO DOMAIN MODELING

Domain analysis is a new area of research in software engineering. It has proceeded by case study and theoretical analysis. There is not yet any consensus on approaches, but several interesting efforts are described here to give an idea of the diversity of the field.

Algebraic

One approach that been used successfully is algebraic specification. The idea is an extension of the work on abstract data types where the semantics of a type are given by equations relating combinations of operations performed on data items of the type.

One advocate of this approach is Srinivas [31]. In his research, the equations for a type engender a theory of that type, and there are a variety of ways to combine theories to describe more elaborate situations. His thesis [32] describes an application of these ideas to the domain of pattern matching. The resulting domain model is quite general and has been specialized to describe a variety of algorithms such as Boyer and Moore's algorithm for string searching and Earley's parsing algorithm.

Denotational

Another formal approach to modeling is derived from the denotational semantics. Several denotational specification languages, such as VDM [17] and Z [30], have been developed and successfully applied to the specification of programming languages, databases, and other domains. The denotational approach differs from the algebraic approach because a variety of mathematical theories (sets, lists, tuples, and maps) can be used by the analyst. In fact, VDM and Z include operations from these theories directly in their syntax. This provides the analyst with added modeling power at the potential cost of prematurally considering representation details.

Draco

Another way of looking at a domain is as a programming language and its associated tools. Neighbors [21] has taken this approach with his work on the Draco system. Besides the grammar for the programming language, a domain description includes a parser, a pretty printer, a collection of source-to-source (intradomain) optimizations, a collection of components to describe mappings into lower level domains, algorithmic interdomain generators, and domain-specific analysis tools.

In Draco, there are three kinds of domains. The highest level consists of *application* domains from which actual applications are built. *Modeling* domains are intermediate and encapsulate engineering knowledge. At the lowest level are *execution* domains generating programs in some base language.

Using a linguistic approach enables automation and permits Draco to capture the history of refinements made between application specification and code. On the other hand, interdomain architectural knowledge is

distributed among the various domains comprising a system and is therefore harder to comprehend and to work with as a whole.

Object Oriented Frameworks

Another language-derived approach is the use of an object oriented framework as a way to specify a domain. A *framework*, as defined by Johnson and Foote [16], is "an object-oriented abstract design." Object orientation implies that the concepts comprising a framework are organized hierarchically with higher concepts being more abstract and general versions of lower ones.

The authors describe both white-box and black-box frameworks, with the latter being more desirable. A white-box framework is one in which application-specific behavior comes from adding methods to subclasses. This, of course, requires knowledge of how the super classes work. Black-box frameworks, on the other hand, allow new components to supply application-specific behavior without requiring any understanding of the implementation of existing classes. For Johnson and Foote [16], a "framework becomes more reusable as the relationship between its parts is derived in terms of a protocol instead of using inheritance."

Knowledge Representation

Another "object-oriented" approach has evolved from expert system technology. An expert system is an organized collection of knowledge together with an inferencing mechanism for reasoning about the knowledge. Typically, an expert system is used to advise or to diagnose rather than to structure an application. But the use of an inference engine adds considerable power.

Borgida and his colleagues have developed a knowledge representation language, called CLASSIC [9], and its associated processor. CLASSIC is capable of specifying concept hierarchies and inference rules about a domain. The CLASSIC inference engine can automatically classify new objects and determine what existing concepts subsume a new one. The appeal of this approach is the combination of automatic inferencing with triggered rules to provide a powerful mechanism for manipulating a domain model.

Facets

One of the primary issues with domains is how their descriptions should be organized in order to best access the information they contain. Object oriented models use inheritance (either single or multiple) and aggregation as the main organizing principles. An alternative, called *faceted classification*, has been investigated by Priéto-Diaz [26]. Although it was designed to facilitate retrieval of reusable components, facets may prove valuable for specifying domain information as well.

The essence of faceted classification is a controlled and structured index vocabulary. A *facet* is a group of related terms that comprise a perspective or viewpoint or dimension of the space being modeled. Manipulation of the order of the facets in a domain and of the terms within a facet can customize a model to a specific task. Moreover, weights can be used to specify similarity of concepts. In another paper [25], Priéto-Diaz has presented a process model for applying these ideas to domain analysis.

Technology book

Technology Books are a "low tech" approach to domain analysis and reuse, developed by Arango and his colleagues at Schlumberger [3]. The idea of a Technology Book is to capture information concerning an engineering problem space during the development of a product for use during development of later products. Information includes problem specific language definitions, formal models, demonstrations, design issues, assumptions, constraints, dependencies, modules, implementations, formal explanations, and other rationale.

The early Technology Books were informal collections of materials, but later evolved to be structured documents defined by templates. Now they are stored in an on-line object oriented repository, and automated modeling and editing tools are being developed.

Technology Books have been successfully used in support of reuse, with a predicted 70% saving as subsequent products are developed in a domain. However, the effort to construct a Technology Book is significant. A 32KLOC assembler program required over ten separate books. Interestingly, rather than avoiding the burden of writing documentation, analysts enjoyed the opportunity to consolidate the knowledge developed during systems analysis.

Application Generators

When a domain is fairly well understood, it becomes possible to express its problem space in a systematic fashion. This, in turn, enables automation of the construction of solution programs. For example, the problem of generating reports for data processing applications is understood well enough that virtually every database vendor offers a report generation capability so that users can describe desired reports at a high level of abstraction without resorting to programming.

The process of automatically producing solution programs is called *application generation*, and tools that perform this function are called *application generators*. Cleaveland describes application generators as translators from specifications into application programs and lists domains where application generation has been successfully applied: data processing, databases, user interfaces, and parsers [11]. He goes on to define a process for building generators that includes domain recognition and bounding, model specification, determination of variant and invariant parts, definition of a language to describe the variant parts and the format of generated products, and implementation.

Cleaveland's paper also describes an *application generator generator* tool that he developed, called Stage. Stage takes two forms of input, a grammar that can be used to specify application problems and an annotated description of the ultimate application program solutions. The annotations describe how the user input specification details relate to and affect the generated products. He lists a variety of successful uses of Stage including user interfaces, finite state machines, testing tools, hardware design translators, structured assemblers, and translation building tools.

Domain Analysis and Layered Software Architectures

One approach to software design that has proven particularly popular over time is the layered architecture. That is, a program is thought of as a sequence of abstract machines, each calling upon the resources of its inferior neighbors and providing services to its superiors. The major advantage of the approach is modularity but at the cost of decreased efficiency due to the number of interlayer function calls.

Batory and his colleagues have explored how this architectural approach can be exploited for a specific, well understood domain, database management systems[3] [5, 6]. In Batory's work, the major role of domain analysis is in defining the interfaces to the layers. A useful/reusable layer describes a data type and a small collection of services, each described with a type signature. The services provided by a layer generalize across the set of typical algorithms for providing such services. For example, a layer in a database management system may provide data compression services, and the job of the domain analyst is to describe an interface for which most typical data compression algorithms are special cases. A layered architecture enables "mix and match" design, where components can be plugged into a layer to satisfy non-functional constraints such as efficiency or robustness, and where layers themselves can be added or removed as user requirements dictate. The use of a formal type model enables an algebraic notation for specifying compositions of layers and the development of an application generator in the form of a database system compiler for quickly constructing systems, for example, generating university INGRESS in thirty minutes.

Batory is one of the few authors to explicitly relate domain analysis to reverse engineering. But his emphasis is on the use of reverse engineering to build the domain model rather that the other way around [4]: "Using existing systems as a guide to standardize the decomposition of systems and designing generic interfaces for components/realms is the essence of domain modeling."

DESIRE

The research project that most directly addresses the issues of domain analysis and reverse engineering is the DESIRE project at MCC, undertaken by Biggerstaff and his colleagues [7, 8]. Biggerstaff is concerned with *design recovery* as distinct from reverse engineering, and it is "the domain model [that] differentiates design recovery research from such superficially similar efforts as reverse engineering." Moreover, design recovery takes advantage of informal information obtained from variable names and program comments in addition to the results of formal program analysis that is typical of traditional reverse engineering.

---

[3] A recent paper generalizes the layered approach to network software systems/protocol suites [4].

DESIRE itself is a working prototype of a design recovery system. It is organized around the ideas of concepts, features, and instances. A *concept* is a part of a domain model, typically a term from the application domain vocabulary such as "adjusted gross income". A concept is distinguished by its *features*—various pieces of evidence that indicate the presence of the concept. For example, the abbreviation "AGI" is often associated with the concept of adjusted gross income. *Instances* are actual occurrences of a concept in the code, with features bound to appropriate program loci, for example, a variable named **agi**.

The process of design recovery that DESIRE promotes consists of three steps. The first step is a broad area search for *linguistic idioms*, informal cues such as variable names or textual comments. Then a local structured search binds features to code. The user is then asked to confirm the overall concept binding.

The DESIRE prototype is constructed from a variety of pieces including a hypertext system, an ER browser, the Common Lisp Object System, and a Prolog interpreter. Furthermore, there is an experimental component that combines a neural network and a semantic net to facilitate feature detection.

Observations

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 prematurally 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. A SCENARIO FOR DOMAIN BASED REVERSE ENGINEERING

To my knowledge, there has been no previous research explicitly relating domain analysis to reverse engineering other than Biggerstaff's work on the DESIRE system [7]. However, it is appealing to look at the benefits that might accrue and any issues that may arise from such an association. This investigation begins with an imaginary scenario that describes a methodology, representation, and tools for reverse engineering that take advantage of domain knowledge. Section 5 summarizes the issues raised by the scenario.

Background

Suppose it is desired to migrate a management information system from Cobol to Ada. Because the distance between the languages is large, both technically and philosophically, the migration will be used as an opportunity to restructure the program into a more object oriented architecture and to otherwise sand down any rough edges that have arisen during the maintenance history of the program. This process will require a deep understanding of the program. Hence, the program will be first reverse engineered before the new version is designed and implemented.

Program Domains

The system being migrated is responsible for managing status information about equipment. It accepts descriptions of status changes, updates the master status data base, and prints a variety of reports. Several domains can be identified pertaining to this program. First there is the domain of parsing. In this case, the parsing is simple—the input data consists of records with fixed length fields. Nevertheless, enough experience has accumulated that Ada library routines exist for parsing that should be used instead of writings, debugging, testing, and documenting new code. And exactly how to map this program's parsing requirements onto the

library is a question of domain analysis.

In some sense, a "status" domain also exists. The ideas that items of equipment can be in a variety of states and that the states can change based either on new information or on the passage of time ought certainly to be located and documented in the program text.

There is also the equipment domain, including information on how items of equipment are named, what types exist, and how collections of equipment are organized. It is easy to imagine other programs that would share this knowledge, for example, to keep track of financial data about the equipment or to do resource allocation.

Perhaps the most substantial and well understood domain pertaining to this program is the domain of report writing. In fact, this domain has matured to the extent that application generators exist to automatically produce reports given a format and the constituent data.

Finally, there are two domains that play a role in many programs. The first is the domain of mathematics. In the case of the equipment program, only a little knowledge of basic arithmetic is required. But in the case of an engineering/scientific application, for example, sophisticated mathematical techniques, such as how to iteratively solve partial differential equations, must be understood in order to comprehend how the application works.

The second ubiquitous domain is that which comprises programming knowledge. Some of this information is generic. For example, the equipment program has a sorting component, the algorithm for which might be translated directly from Cobol into Ada. But some of the knowledge is language dependent. For example, the equipment program makes extensive use of Cobol's ability to alias and overlay areas of working storage. Much of existing reverse engineering technology is focussed on exploiting program domain knowledge in understanding programs [1].

Methodology and Representation

In this scenario, the reverse engineering of the equipment program proceeds using the technique of *Synchronized Refinement* [22]. This technique analyzes the program text from the bottom up, looking for stereotypical cues, also called *cliches* [27], or *plans* [29], or *idioms* [24], 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.

Meanwhile, in the process of confirming the existence and use of the page management code, it is noticed that a generated report includes summary information that presupposes that the input data is ordered in a specific way. While the equipment domain does not indicate that this ordering is required, neither does is exclude the possibility. In fact, the emerging program description must contain knowledge that goes beyond what the domain description specifies. Furthermore, it may be desirable to update the equipment domain model to include this possibility.

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.

Tools

The reverse engineering process in this scenario is not entirely manual. In fact, several automated tools can be imagined to support the process. First is a tool for browsing domain descriptions. Because these descriptions are complex and highly interrelated, the tool could prove useful to support navigation. Moreover, as the understanding of the application grows and as annotations are made to the emerging program description, the domain browser can serve as a code browser as well.

Specialized code representation tools are also required. Many of these already exist, such as those for generating cross reference information, call trees, and data dependency diagrams. But there will always be new ones, such as one to graphically display Cobol memory overlays and aliases. Moreover, these tools should be incremental in the sense that as understanding of the application program grows and as the code is abstracted, the tools must accurately reflect the new knowledge. Of course, the ensemble of tools should also appear to be

cohesive in order to better support the overall reverse engineering process.

Confirmation and modification tools are also required to support domain based reverse engineering, and they should be relatively sophisticated in their inferencing capabilities. It is likely that domain knowledge will be organized using a number of relationships, and testing whether the code satisfies a relationship will, in general, require programmable tools like the Software Refinery toolkit [14]. This holds true as well for tools to support the program abstraction process that is a part of Synchronized Refinement.

5. ISSUES

The scenario raises many questions concerning how best to make use of domain analysis in support of reverse engineering. The questions can be partitioned into the areas of methodology, representation and tools.

Methodology

1. Perhaps the overriding question of this research is whether domain analysis can help in the reverse engineering process at all. Clearly, this essay assumes so, but the assumption needs to be validated. The projects described in the next section are intended to explore this question.
2. 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 the income tax program, it is not clear how best to use 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?
3. 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.

Representation

1. 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 [10]. Certainly, domain theorists do not yet agree on how to represent domain information, but a consistent representation is a prerequisite to broadly applicable tools.
2. 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, such as the heuristic that indicates not to investigate deducting medical expenses until they form a significant fraction of income. Of course, some degree of formality is a prerequisite for tool support.
3. 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?
4. 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?

Tools

1. Domains are complex. 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?

2. 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?
3. 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 [12].
4. 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?

An Overarching Issue

At a recent International Conference on Software Engineering, there was much discussion of domains [15]. Inherent in this discussion was the assumption that a well-understood domain often coexists with a preferred architecture for solving problems in the domain. But does this not unnecessarily confuse problem with solution? For example, Priéto-Diaz has described domain analysis as the extension of systems analysis to collections of problems [25]. Systems analysis, of course, is a prerequisite for design and to the establishment of an architectural approach to problem solution. So, how can domain analysis, which happens early in the life cycle, coexists with architectural design, which occurs much later? A related issue also arises of how to represent and take advantage of this architectural knowledge when performing reverse engineering. Perhaps the preferred architecture can act as a coordinating principle for the reverse engineering process.

6. DOMAIN ANALYSIS AND REVERSE ENGINEERING PROJECTS AT GEORGIA TECH

Researchers in the College of Computing at the Georgia Institute of Technology are actively investigating the relationship of domain analysis and reverse engineering through a variety of projects. The projects involve domains ranging from the report writing and equipment domains (comprising the management information system described in the scenario), to user interface toolkit components, to the kinematics of natural and artificial objects in our solar system.

The TRANSOPEN Project[4]

The Army Research Laboratory has for several years sponsored research at Georgia Tech to investigate the transition of existing Army management information systems from their traditional batch, mainframe environment to an interactive, distributed, workstation, open systems environment. The work includes researchers concerned with communications protocols, database integration, and business process re-engineering. Our part in this project has been concerned with the transition of the actual software, including issues of strategy selection, reverse engineering process, and tool support [13, 28].

Our current work with the project directly involves domain analysis. We have taken a specific domain, report writing, and performed a domain analysis on it. We are experimenting with ways to represent the results of the analysis with a knowledge representation language so that we can use the representation to drive the reverse engineering of several applications in the domain. We hope to learn several things from this work directly addressing the issues raised in Section 5: To what extent has the domain knowledge aided (or hindered) the reverse engineering process? How should the domain model be most effectively represented? And what tools would facilitate the process? In the future, we hope to look at less well-defined domains and at the issues arising from the use of more than one domain in the same application.

The Knowledge Worker System (KWS) Project[5]

The Army Corps of Engineers Research Laboratory (CERL) has sponsored the development of a personal information system called Knowledge Worker. The original version was developed in the C language for IBM-compatible personal computers running MS-Windows and interfacing to the Oracle Relational Database Management System on a remote server. Now CERL is interested in an Ada version for a POSIX workstation running

[4] Students involved include Bijith Moopen, Richard Clayton, Jean-Marc DeBaud, Sri Doddapaneni, and Gary Pardun.
[5] Participants include my colleague, Melody Moore, and students Hernan Astudillo, Yee Huat Gan, and Phil Seaver.

Motif. We have been advising them on the transition process [18]. Of particular interest to us is the issue of re-engineering the MS-Windows user interface to work with Motif [19]. On the surface it might appear that one has merely to textually replace MS-Windows library routine names with Motif ones. Unfortunately, however, things are not so simple. Not only is there not a direct match between the libraries, but there are subtle architectural differences between the two toolkits.

For example, in MS-Windows an option exists whether the application program or the user interface runtime library is responsible for visually indicating that a button has been pressed. In Motif, only the latter option is available. Conversely, in MS-Windows a button exists that can be in one of three different states; in Motif, all buttons are limited to at most two states. Finally, and most troublesome, Motif widgets[6] are arranged hierarchically—specialized widgets inherit features and functions from more general widgets. In MS-Windows, each widget must supply all of its own features and functions. Difficulties like these significantly complicate the problem of selecting appropriate surrogates when adapting an application to a new windowing system.

Commercial vendors have tried to solve these problems, but the users we spoke with were dissatisfied with their products, complaining that either they handled only the superficial translation aspects or they required the engineer to describe the interface in a proprietary language—a non-trivial effort that approximates the effort required to do the translation directly.

Our approach involves a deeper understanding of user interface toolkits and widgets. In fact, we found ourselves modeling such devices as part of a domain. We began by using CLASSIC to describe a part of the Motif widget set and then presented it with a description of an MS-Windows widget taken from KWS. CLASSIC was able to automatically suggest appropriate Motif widgets to use. We have since grown the model to include generic end-user interface requirements resulting in a comprehensive toolkit-independent representation of the domain. Because CLASSIC allows arbitrary LISP procedures to be invoked when an inference is made, automatic code translation is enabled.

Our current work on this project involves dealing with the architectural issues mentioned above and extending our model to deal with other classes of widgets. Also, we intend to try this approach on applications with no graphical interface at all, such as those using character-oriented window libraries, like CURSES, or textual, command language interfaces.

NAIF Library Project[7]

Our newest project is sponsored by the NASA Ames Research Laboratory. NASA researchers are supporting efforts at the Jet Propulsion Laboratory (JPL) to build scientific/engineering applications involved with satellites and other space missions. An example application concerns the sending of messages from a ground station on Earth, via a relay satellite, to a space vehicle in orbit around Mars.

Currently, an extensive library, called the Navigation Ancillary Information Facility (NAIF) SPICELIB library, exists that can be used by scientists at JPL to help construct such applications. The library is written in Fortran and consists of 600 routines dealing with issues such as frame of reference translation, speed of light delays, and ephemeris data. However, the library is not as useful as it should be, and JPL would like to improve retrieval and composition of appropriate subroutines. In particular, they would like to have a formal library specification, and to do so requires reverse engineering the library.

What makes the project particularly interesting to us is that the NASA researchers have constructed a formal, comprehensive, and validated domain model for this class of applications [33]. It is our intent on this project to use the domain model in support of the reverse engineering process; in this case, to incorporate the extensive informal information provided in the in-line program comments into the domain model.

Other Related Projects at Georgia Tech

We have recently submitted a proposal to an industrial sponsor to support the development of a domain-based program browser (or *dowser*[8]). The point of the work is that providing the reverse engineer with a view

[6] A widget is a low-level user interface construct such as a button or a scroll bar.
[7] I am working with Dr. Linda Wills on this project.
[8] Dowsing is the ancient art of finding water, minerals, and other useful resources underground by use of a divining rod.

of the application domain via a navigational aid will prove more helpful for some maintenance tasks than would a traditional program browser.

We are also looking at the use of formal program semantics as a representational vehicle for reverse engineering information[9]. Although this work is not directly domain related, as mentioned above, reverse engineering representation issues are tightly coupled with those of domain representation.

7. CONCLUSION

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, I fully expect that any major breakthrough in the automated program understanding and reverse engineering area to take significant advantage of domain information.

Acknowledgement

I would like to thank Linda Wills for her thoughtful comments on this paper.

References

1. *Proceedings Working Conference on Reverse Engineering*, IEEE Computer Society Press, Baltimore, Maryland, May 21-23 1993.
2. Guillermo Arango and Ruben Priéto-Diaz, ‘‘Domain Analysis Concepts and Research Directions,’’ in *Domain Analysis and Software Systems Modeling*, ed. Ruben Priéto-Diaz and Guillermo Arango, IEEE Computer Society Press, 1991.
3. Guillermo Arango, Eric Schoen, and Robert Pettengill, ‘‘A Process for Consolidating and Reusing Design Knowledge,’’ *15th International Conference on Software Engineering*, pp. 231-242, IEEE Computer Society Press, Baltimore, Maryland, May 17-21, 1993.
4. Don Batory and Sean O’Malley, ‘‘The Design and Implementation of Hierarchical Software Systems with Reusable Components,’’ *Transactions on Software Engineering and Methodology*, vol. 1, no. 4, pp. 355-398, ACM, October, 1992.
5. Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas, ‘‘Scalable Software Libraries,’’ *Sigsoft*, pp. 191-199, ACM, December, 1993.
6. D. S. Batory, ‘‘Concepts for a Database System Compiler,’’ *Proceedings of the ACM Principles of Database Systems Conference*, 1988.
7. Ted J. Biggerstaff, ‘‘Design Recovery for Maintenance and Reuse,’’ *IEEE Computer*, vol. 22, no. 7, July 1989.
8. Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster, ‘‘The Concept Assignment Problem in Program Understanding,’’ *Proceedings of the First Working Conference on Reverse Engineering*, pp. 27-43, Baltimore, Maryland, May 21-23, 1993.
9. Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick, ‘‘CLASSIC: A Structural Data Model for Objects,’’ *Proceedings ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, May-June 1989.

[9] This work is being done with student Kurt Stirewalt.

10. 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.

11. J. Craig Cleaveland, ‘‘Building Application Generators,’’ *IEEE Software*, vol. 5, no. 4, pp. 25-33, July 1988.

12. 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.

13. Melody Eidbo, Mostafa Ammar, Russ Clark, Rich Clayton, Srinivas Doddapaneni, Rob Dodge, Mike McCracken, Binh Nguyen, Webb Roberts, Steve Rogers, and Spencer Rugaber, ‘‘Transitioning to the Open Systems Environment (TRANSOPEN) Final Report,’’ CIMR - 93-01, Center for Information Management Research, College of Computing, Georgia Institute of Technology, April 14, 1993.

14. Reasoning Systems Incorporated, *Software Refinery Toolkit*, Palo Alto, California.

15. N. Iscoe, ‘‘Panel on Domain Modeling,’’ *15th International Conference on Software Engineering*, IEEE Computer Society Press, Baltimore, Maryland, May 17-21, 1993.

16. R. E. Johnson and B. Foote, ‘‘Designing Reusable Classes,’’ in *Domain Analysis and Software Systems Modeling*, ed. Ruben Priéto-Diaz and Guillermo Arango, pp. 138-147, IEEE Computer Society Press, 1991.

17. C. B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall, 1990.

18. Melody Moore, Spencer Rugaber, and Hernan Astudillo, ‘‘Knowledge Worker Platform Analysis Final Report,’’ CIMR - 93-02, Center for Information Management Research, College of Computing, Georgia Institute of Technology.

19. M. Moore and S. Rugaber, ‘‘Issues in User Interface Migration,’’ *Proceedings of the Third Software Engineering Research Forum*, Orlando Florida, November 1993.

20. James M. Neighbors, ‘‘Software Construction from Components,’’ PhD thesis, TR-160, ICS Department, University of California at Irvine, 1980.

21. James M. Neighbors, ‘‘Draco: A Method for Engineering Reusable Software Components,’’ in *Software Reusability / Concepts and Models*, ed. Ted J. Biggerstaff and Alan J. Perlis, vol. 1, Addison Wesley, 1989.

22. 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.

23. R. K. Overton and et al., ‘‘A Study of the Fundamental Factors Underlying Software Maintenance Problems: Final Report,’’ (NTIS: AD 739479 and AD 739872), Corporation for Information Systems Research and Development, December 1971.

24. A. J. Perlis and S. Rugaber, ‘‘Programming with Idioms in APL,’’ *APL79 Conference Proceedings Part 1*, Association for Computing Machinery, Rochester, New York, May 30 - June 1, 1979. Also in *APL Quote Quad*, vol. 9, no. 4, June 1979.

25. Ruben Priéto-Diaz, ‘‘Domain Analysis for Reusability,’’ *Proceedings of IEEE COMPSAC-87*, Tokyo, Japan, October 1987.

26. Ruben Priéto-Diaz, ‘‘Classification of Reusable Modules,’’ in *Software Reusability / Concepts and Models*, ed. Ted J. Biggerstaff and Alan J. Perlis, vol. 1, pp. 99-123, Addison Wesley, 1989.

27. Charles Rich and Linda M. Wills, ‘‘Recognizing a Program’s Design: A Graph-Parsing Approach,’’ *IEEE Software*, vol. 7, no. 1, pp. 82-89, January 1990.

28. Spencer Rugaber and Srinivas Doddapaneni, ‘‘The Transition of Application Programs From COBOL to a Fourth Generation Language,’’ *Conference on Software Maintenance - 93*, Montreal, Canada, September 27-30, 1993.

29. Eliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert, ‘‘Designing Documentation to Compensate for Delocalized Plans,’’ *Communications of the ACM*, vol. 31, no. 11, November 1988.

30. J. M. Spivey, *Understanding Z: A Specification Language and Its Formal Semantics*, Cambridge University Press, 1987.

31. Yellamraju V. Srinivas, ''Algebraic Specification of Domains,'' in *Domain Analysis and Software Systems Modeling*, ed. Ruben Priéto-Diaz and Guillermo Arango, pp. 90-124, IEEE Computer Society Press, 1991.

32. Yellamraju V. Srinivas, ''Pattern Matching: A Sheaf-Theoretic Approach,'' PhD Thesis, TR 91-41, Department of Information and Computer Science, University of California at Irvine, May 1991.

33. Mark Stickel, Richard Waldinger, Michael Lowry, Thomas Pressburger, and Ian Underwood, *Deductive Composition of Astronomical Software from Subroutine Libraries*. Submitted for publication.