Short Running Title:	The use of domain knowledge in program understanding

Contact Author:	Spencer Rugaber
Address:	College of Computing Georgia Institute of Technology Atlanta GA 30332-0280 USA
Phone:	(404) 894-8450
Fax:	(404) 894-9442
email:	spencer@cc.gatech.edu

The use of domain knowledge in program understanding

Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Abstract

Program understanding is an essential part of all software maintenance and enhancement activities. As currently practiced, program understanding consists mainly of code reading. The few automated understanding tools that are actually used in industry provide helpful but relatively shallow information, such as the line numbers on which variable names occur or the calling structure possible among system components. These tools rely on analyses driven by the nature of the programming language used. As such, they are adequate to answer questions concerning implementation details, so called *what* questions. They are severely limited, however, when trying to relate a system to its purpose or requirements, the *why* questions.

Application programs solve real-world problems. The part of the world with which a particular application is concerned is that application's *domain*. A model of an application's domain can serve as a supplement to programming-language-based analysis methods and tools. A domain model carries knowledge of domain boundaries, terminology, and possible architectures. This knowledge can help an analyst set expectations for program content. Moreover, a domain model can provide information on how domain concepts are related.

This article discusses the role of domain knowledge in program understanding. It presents a method by which domain models, together with the results of programming-language-based analyses, can be used to answers both *what* and *why* questions. Representing the results of domain-based program understanding is also important, and a variety of representation techniques are discussed. Although domain-based understanding can be performed manually, automated tool support can guide discovery, reduce effort, improve consistency, and provide a repository of knowledge useful for downstream activities such as documentation, reengineering, and reuse. A tools framework for domain-based program understanding, a *dowser*, is presented in which a variety of tools work together to make use of domain information to facilitate understanding. Experience with domain-based program understanding methods and tools is presented in the form of a collection of case studies. After the case studies are described, our work on domain-based program understanding is compared with that of other researchers working in this area. The paper concludes with a discussion of the issues raised by domain-based understanding and directions for future work.

1. Introduction

1.1. Motivation

Program understanding comprises all activities by which knowledge is gained about a program. *Reverse engineering* is a systematic form of program understanding that 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 construction. Because we know how to do these kinds of analyses quite well, it is natural to try and apply them to understanding programs.

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*. The task would be difficult if not impossible.

The problem is that a program has a purpose; its 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 succeeds in accomplishing its purpose. To the extent that the model is comprehended by the program reader, the process of understanding the program is 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 accordingly [Overton 1971]. 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.

* This thought experiment was developed by Biggerstaff [1989].

Another 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 legal and financial knowledge. 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 lead to its own maintenance headaches.

A 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 computing in a programming language can get in the way of seeing the forest of the problem the program solves. That is, source code, concerned as it is with managing the execution of a computer, is at too low a level of abstraction to effectively describe a program's context.

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 program understanding. This paper argues that application domain modeling provides such an approach.

1.2. Domains

A *domain* is a problem area. Typically, many application programs exist to solve the problems in a single domain. Arango and Prieto-Díaz [1991] 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 problems in the domain, a recognition that software solutions are appropriate to problems in the domain, and a store of knowledge or collected wisdom to address 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*. In the income tax example, a given computation likely obtains its operands from the results of other computations. The set of computations and the dependencies among them form a partial-order relation, and programs in this domain are likely to be structured in a way to maintain and efficiently implement this relation.
- 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.

1.3. Domain analysis

According to Neighbors [1980], *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.

---

* One possible explanation for this is that the domains we currently understand best are those that have been around the longest and for which solution strategies have converged. The consensus solution strategies or architectures then become a factor in defining the domain and deciding whether a problem is in the domain at all.

Domain engineering, modeling, and analysis is an active 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 have proven useful in support of program understanding as well.

1.4. Domain models

In order for domain analysis to be useful for software development, reuse, or program understanding, 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 model are domain objects and their definitions, including both real-world objects like tax rate tables and concepts like long-term capital gains; solution strategies like the use of a partial-order relation; and a description of the boundary and other limits of the domain like federal, personal, income tax return.

1.5. Relationship to program understanding

What role can a domain model play in understanding a program? In general, a domain model can give the reader a set of expected constructs to look for in a program. These can 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 a partial-order relation.

Because a domain is broader than any single problem in it, there may be expectations engendered by the domain model that are not found in a specific program (the inventory algorithm may not appear in a program to compute personal income taxes but might occur 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 model. 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 model, such as a checkbook-balancing feature in an income tax package.

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

1.6. A scenario for domain-based program understanding

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 requires a deep understanding of the program. Hence, the program will first be reverse engineered before the new version is designed and implemented.

1.6.1. Domains

The system being migrated is responsible for managing status information about equipment. It accepts descriptions of status changes, updates a master status database, 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 parsing experience has accumulated that Ada library routines exist which should be used instead of writings, debugging, testing, and documenting new parsing code.

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 an 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 most commercial database products include programs that automatically produce reports given a report format and the constituent data.

Finally, there are two other domains that play a role in most 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 or 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. Together, these six domains comprise the external knowledge using which the status reporting program was developed. Hence it is natural to try to use this knowledge in understanding it.

1.6.2. Method and representation

In this scenario, the reverse engineering of the equipment program proceeds using the technique of Synchronized Refinement [Rugaber et al. 1990]. 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.

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 model does not indicate that this ordering is required, neither does is exclude the possibility. In general, the emerging application description will contain knowledge that goes beyond what the domain model specifies. Alternatively, it may be desirable to update an incomplete EQUIPMENT domain model to include this possibility.

The results of Synchronized Refinement are recorded as a sequence of annotations. Each annotation indicates that a particular domain concept has been implemented with a specific collection of programming-language constructs. Moreover, the annotation records the category of decision made by the designer. As domain-engendered expectations are met or refuted, the description of the application grows. As design decisions are identified, their implementations are abstracted from the code, causing it to shrink in size. In this way, a refined application description grows synchronously with an abstracted program representation.

Section 2 of this paper describes the Synchronized Refinement method for domain-based program understanding. *Section 3* discusses issues and alternative notations for expressing domain models.

1.6.3. Tools

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

Of course, programming-language-based 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 to be applied, 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 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 program-understanding process.

Confirmation and modification tools are also required to support domain-based program understanding, and they should be relatively sophisticated in their inferencing capabilities. It is likely that domain knowledge is organized using a number of relationships, and testing whether the code satisfies a relationship generally requires sophisticated, programmable tools. This holds true as well for tools to support the program abstraction process that is a part of Synchronized Refinement. We call domain-based program understanding tools *dowsers*. And *Section 4* describes our experiences with building and using them.

1.7. Overview

This paper describes our experience with domain-based program understanding. In addition to the sections on methods, representation, and tools mentioned above, *Section 5* summarizes the many case studies we have performed to explore this area. *Section 6* describes efforts by other researchers on the use of domain knowledge, in support of both software development and reverse engineering. The paper concludes in *Section 7* with a discussion of the issues raised by this research and directions for future work.

2. A domain-based program understanding method

2.1. Overview of Synchronized Refinement

Synchronized Refinement (SR) is a method for reverse engineering software for documentation, reengineering, or reuse. Reverse engineering produces a high-level representation of a software system from a low-level one, typically the program itself. Synchronized Refinement constructs such a representation of a system from its source code and from a model of its application domain. It proceeds by detecting design decisions in source code and relating the detected decisions to the corresponding element of the application-domain description. *Figure 1* describes the flow of data through the various activities that constitute SR. In the figure, rectangles denote

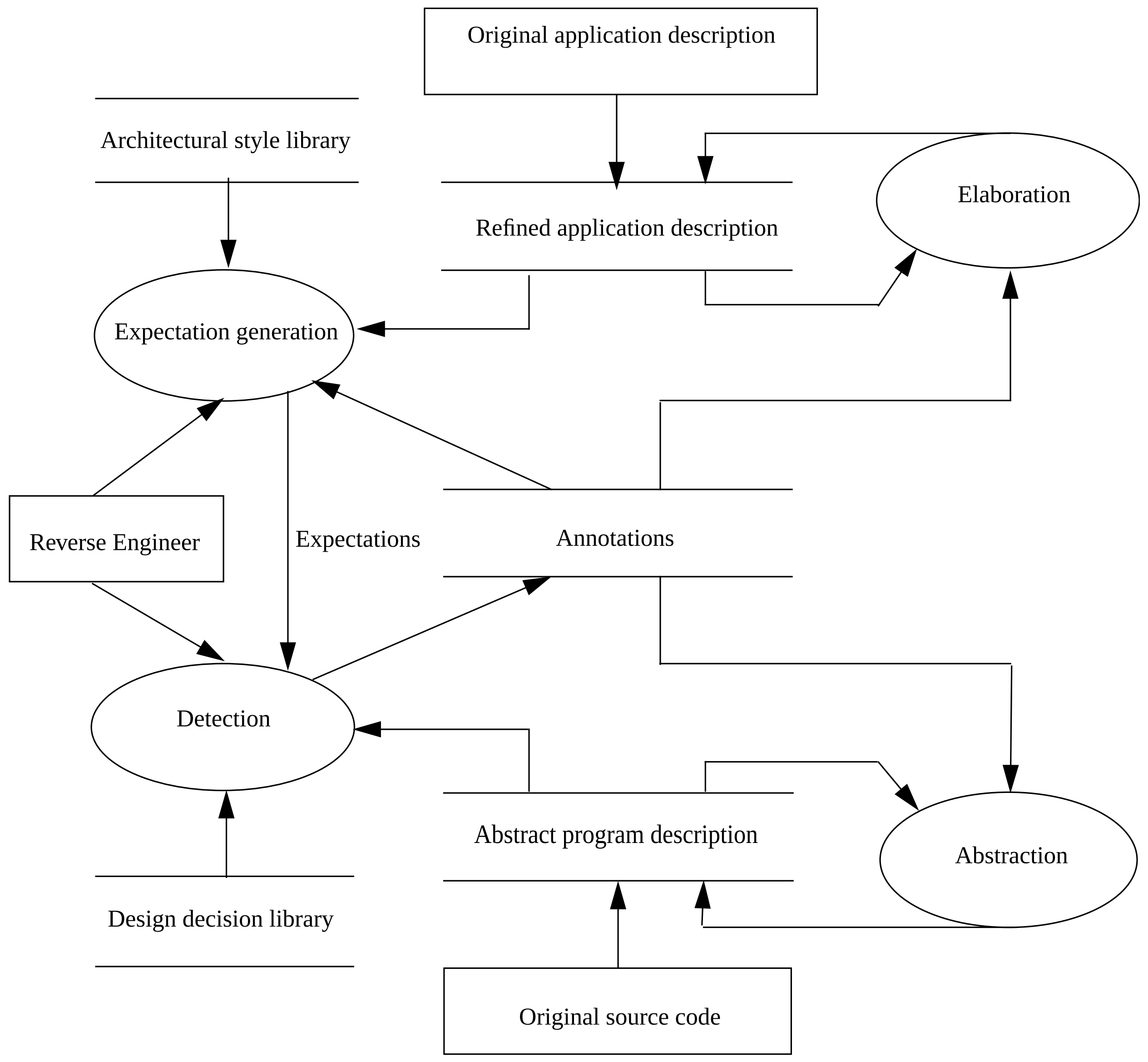

Figure 1. Data Flow Diagram for Synchronized Refinement.

sources of data, ovals indicate activities, and parallel horizontal lines enclose the names of information repositories.

Synchronized Refinement takes as input the source code of a software system (labelled *Original source code* in *Figure 1*) and a description of the application domain that the system supports (*Original application description*). In addition, various sources of programming knowledge may be available to the analyst to aid the understanding process. (These are labelled *Architectural style* *library* and *Design decision library* in *Figure 1*.) The output consists of three parts: an elaborated and instantiated domain description (*Refined application description*), an abstracted program description (*Abstract program description*), and a description of how the code realizes the application (*Annotations*). The SR process consists of two parallel activities, roughly corresponding to the top and bottom halves of the diagram: analysis of the program (on the bottom) and synthesis of the application description (on the top). The analysis process consists of detecting implementations of design decisions and replacing the detected code with an abbreviated version. In this way, the program description becomes smaller and more abstract. The synthesis process begins with the application description, expressed in terms of domain models, which is then elaborated as more is learned from the source code. The two parts of the diagram communicate in terms of expectations. That is, domain concepts must be ultimately realized in terms of code constructs, and by navigating the domain description, expectations are established for what should be found in the source code. Likewise, the process of code analysis confirms or denies that certain expectations are satisfied.

As the code analysis process proceeds, decisions are detected that relate to an expectation. The annotation for a detected decision is recorded together with the relevant expectation, specifying the type of the decision and the corresponding sections of the program (in the repository labelled *Annotations*). During the process, certain expectations are confirmed and others may be refuted. A confirmed expectation may engender others. Gradually, a hierarchical description of the structure of the program emerges. As the program source code shrinks, the application description expands.

2.2. Input formats

The Synchronized Refinement process takes its input from four sources: the source code, an application domain description, and programming knowledge in the form of an architectural-style library and a design-decisions library.

Program source code. The program to be analyzed is written in some programming language. It is assumed by SR that the analyst has a working knowledge of the language and that the program successfully compiles. In most cases, SR is not concerned with the vagaries of compiling to specific platforms.

Domain description. An SR domain description takes the form of one or more domain models. Although a variety of representations are described in *Section 3*, this section assumes that object-oriented frameworks [Johnson and Foote 1988] (OOFs) are used to model application domains. An OOF comprises a set of cooperating abstract classes as well as the protocols by which they communicate. A class corresponds to a domain concept and its associated features. A feature is either a public method that objects of the class can provide or a component of the object's state (an instance variable). The specific OO notation is not of concern to SR because the details of method implementation are not present. Instead, a specification of functionality is expressed, either formally or informally, as a set of expectation. The classes in an application domain collaborate to accomplish the application's goals. The collaboration can be thought of in terms of associations among the classes in the framework or as protocols for interobject messaging by which the classes cooperate to accomplish their ends. Associations are notated using both standard constraints (e.g., cardinality constraints) and invariants specified in the same notation used to indicate method functionality. In the case of protocols, path notation is used [Campbell 1974]. This is a variant of regular expressions denoting allowable sequences of method invocations among the constituent classes. The protocol together with the component classes can be thought of as providing an architecture for solutions to problems residing in the application domain.

Architecture-style library. An *architectural style* is "an idiomatic pattern of system organization." [Garlan and Shaw 1995] The SR style library is a mapping between style names and the high-level architectural description expressed in terms of the design decisions used to compose them. The library is populated with commonly occurring styles such as *client-server* and *pipe-and-filter*. The library serves as a repository for generating expectations and for quickly elaborating a domain description once an expectation is ratified.

Design-decision library. Programmers elaborate a design into a program by using programming-language constructs to implement various kinds of design decisions. Reverse engineers using SR try to detect instances of such decisions by applying both their knowledge of the application domain and their experience in programming. The design decision library is a repository of such knowledge. Specifically, the repository is organized as a hierarchical classification of design decision prototypes. Prototypes are described more completely in *Section 2.5*.

2.3. Output formats

Output from SR consists of three artifacts: an application description, an abstract program description, and a mapping between the two expressed as a series of annotations. In addition, it may occasionally be the case that the design-decision library or the architectural-style library are updated if examples of missing generic programming knowledge are detected in the code.

Abstract program description. As SR proceeds, examples of particular categories of design decisions are detected in the source code. Once such an example is recognized and confirmed, it can be replaced in the code by an abstract description of its role. This results in two versions of the program: the original and the abstracted version. Two things should be noted about this representation. First, the detected example may not occur contiguously in the program text. This implies that what is actually extracted from the original version may not correspond to a single syntactic construct. SR treats these constructs as *plans* [Soloway et al. 1988] each of which can be thought of merely as a collection of related syntactic elements. The second thing to note is that SR does not require that both the original and abstracted version of the code be retained for all detected decision as long as such versions can be reconstructed if required.

Application description. As understanding of the code grows, it needs to be represented so that it can be used for its intended purpose such as reengineering, reuse, or documentation. The form that SR uses for this is to instantiate the OO classes used for the domain description.

Annotations. The artifacts described in the previous two sections must be synchronized. That is, it must both be possible for an analyst to locate the specific code that implements a domain concept and to determine for any code construct what role it fills in the application domain. This two-way mapping is provided in the form of a set of annotations. An annotation consists of three pieces of information. The first is the type of design decision that the reverse engineer detected. This corresponds to an entry in the design decision library. If no such site exists, then the library needs to be updated appropriately. The second piece of information indicates the specific place in the source code at which the design was implemented. This corresponds to the set of syntactic constructs described above. The third piece of information is the place(s) in the OO framework where the code segment has been instantiated. Together the three pieces of information allow the reconstruction of the understanding process and provide the constituents of the high-level representation required of a reverse-engineering technique.

2.4. Steps

The two key concepts underlying Synchronized Refinement are abstraction-based design-decision detection in source code and domain-driven application-model synthesis. The two ideas are manifested synchronously, and an annotated representation is generated linking the results. Source code analysis proceeds by detecting program constructs and abstracting them from the code, thereby compressing it. The application-description synthesis process explores the application-domain model in order to generate target concepts for detection in the code and to build up the application description. In this way, the source code continually shrinks while the application description grows more complete.

2.5. Design decisions

Software design is the process of expressing structure, functionality, and behavior of a problem solution in the medium of a particular programming language. If we want to effectively understand software we must appreciate not only its source-code statements, but also its architecture and other design choices. To do so, it is important to take advantage of as much of the reasoning that went into designing the existing version as possible. This reasoning is collectively called *design rationale,* and Synchronized Refinement structures the rationale that it detects based on the abstraction mechanisms used in the program’s source code. The abstraction mechanisms are implementations of design decisions made by the original programmer or designer. For example, choosing a dynamic data structure versus one with fixed bounds reflects the decision that an original requirement allows an unlimited number of items rather than arbitrarily constraining the implementation to a fixed length. This is an example of an explicit decision implemented by using a programming language’s dynamic-memory-allocation features rather than by statically allocating a fixed-length structure.

Design decisions can be divided into categories based on the type of abstraction they provide. For example, many languages have a feature for bundling together or packaging a group of named constructs to better control naming conflicts and enforce information hiding. This is an example of the *encapsulation* abstraction. Among the types of design decisions that are useful to detect during the process of reverse engineering are the following.

- *Composition/decomposition:* Programs are built up from parts, and problems are broken down into smaller, more easily solvable sub-problems. This type of decision is manifested in the programming-language code by such constructs as packages, record structures, and subprograms.
- *Encapsulation/interleaving:* Subcomponents interact with each other. If the interactions are limited and occur through explicit interfaces, the component is said to be encapsulated. If, usually for reasons of efficiency, two or more design ideas are realized in the same section of code or by the same data structure, then the components corresponding to those ideas are said to be interleaved.
- *Generalization/specialization:* Often one component is similar to another. It may then be possible to construct a more general and higher-level, parameterized component capable of realizing both as special cases. In object-oriented programming, the process is sometimes reversed, with the more general component being constructed first, and the special cases added later.
- *Representation*[*]*:* In translating from the problem domain to the solution program, decisions are made that result in program components serving as implementations for application domain entities. If efficiency is a concern, high-level programming constructs can be further represented by other constructs closer to the machine, such as using an array to represent a stack. Languages such as Ada support explicit descriptions of representation, but the reverse engineering of programs from older languages requires the explicit recognition of these decisions.

---

* This use of the term *representation* should not be confused with its use as a notation for expressing a high-level understanding of a program.

- *Data/Procedure:* Programs are sequences of computations organized by control structures. Variables are ways of saving intermediate results for later use, either to avoid recomputation or to simplify the expression of the computation. The introduction of a variable is an important design decision that is, unfortunately, too easy to make without appropriate thought and annotation.
- *Non-determinism removal:* As the design process moves from abstract specifications to concrete machine implementations, the designer is often forced to limit the set of possible computations. For example, a design specification requiring the use of a potentially infinite stack may in fact be implemented using a fixed-length array together with an index variable. The choice to use an array and index is a representation decision; the choice to limit the size of the array reduces the non-determinism in the specification.

There is no strict order for detecting design decisions. Complex programs are typically designed as layers of abstractions, and, as one layer is detected during program understanding, it opens the door for the detection of other decisions in higher layers. Nevertheless, it is possible to give the following guidelines for the detection process.

- Particularly where older programming languages are used, begin by looking for the occurrence of modern control structures implemented by more primitive constructs. An example of this is the use of nested `IF` statements and `GOTO`s to implement a `SWITCH` statement. (representation)
- Likewise, look for the use of primitive data structures to represent unavailable ones, such as the use of specific integer or character constants to encode one of a set of values. (representation)
- It may sometimes be necessary to reconfigure the control flow of the program. This is essentially what a restructuring tool provides. The resulting code should be functionally equivalent to the original. (representation)

- Look for special cases; that is, look for similar sections of code that differ only in a small number of ways. Replace these by the parameterized use of a more abstract construct. (generalization)
- If the language does not support modularization, look for code that should be grouped together and separated from the rest of the program by an abstract interface. (encapsulation)
- In contrast, sometimes a programmer intentionally interleaves the accomplishment of two objectives within the same section of code. This is often done for reasons of efficiency. To understand the code, it becomes necessary to segregate the two functions and annotate them separately. (interleaving)
- If several program variables are used together to implement an unavailable construct, annotate that fact. (composition)
- Make sure that variables are only used for one purpose. Replace dual-use variables by two separate variables. (interleaving)

2.6. Summary

Synchronized Refinement is a technique for gaining understanding of software. It uses not only the source code itself, but also a description of the software's application domain. It produces an abstracted program description, an elaborated domain description, and annotations describing the connections between the two.

3. Representation

Reverse engineering is all about representations [Clayton and Rugaber 1993]. In traditional reverse engineering, which is driven by program analysis, there are three important kinds of representations. The first is concerned with presenting the results of a specific program analysis. For example, a call graph provides a graphical display of a relation derived from elements in the source code. Such representations can be displayed graphically, can be stored in a database for later access, or can take the form of a textual report intended for human consumption.

The second kind of representation of concern to traditional reverse engineering is called an intermediate representation. An *intermediate representation* is produced by a compiler or other program analysis tool and saved for later use by other tools. The most common of such intermediate representations used for reverse engineering is the abstract syntax tree (AST). An AST is a digested parse tree on which various transformations have been performed. The transformations typically remove punctuation, coalesce lists, and collapse intermediate nodes with only one descendant. The result is not only more compact than the original parse tree but also at a higher level of abstraction. Using the AST, other analysis tools do not need to reparse a program, thereby saving considerable effort. Other traditional intermediate representations include control-flow graphs, data-flow graphs, and program dependency graphs [Rugaber 1996].

The third form of representation used by traditional reverse engineering is provided by a database management system or other repository to hold the results of program analysis. Two typical mechanisms are the relational database, such as used by the CIA [Chen and Ramamoorthy 1986] and CIA++ tools [Grass and Chen 1990]; and the object-oriented repository, such as provided in the Software Refinery tool set [Reasoning 1990].

For domain-based reverse engineering, another form of representation is important—a mechanism for representing domain knowledge possibly including a way to express the relationship between what is learned about a program and the application domain. We have conducted a variety of case studies using different representations for domain knowledge, including predicate logic, algebraic specifications, frame-based knowledge representation, entity–relationship models, static object models, and object-oriented frameworks. This section describes the representations used and summarizes the results. Details of the case studies are provided in *Section 5*

(Appended to the title of each subsection is the name of the corresponding subsection of *Section 5* in which the case study that used the representation is described.)

3.1. Predicate logic (Section 5.4)

Predicate logic is ubiquitous in computer science. When used to model an application domain, it takes the form of axioms describing the important domain entities and the relationships among them. The application domain for which we used predicate logic was SOLAR-SYSTEM KINEMATICS, the position and movement of objects in the solar system, including both naturally occurring elements (satellites and planets) and space vehicles engaged in exploratory missions. The software system being reverse engineered comprised a mature library of components written in Fortran and provided by the Jet Propulsion Library.

Axioms took two forms. The more abstract form modeled the solar systems, the laws of geometry, and various transformations describing the measurement of time in the presence of relativistic effects. The second and parallel set of axioms described the components of the library, primarily indicating how typed data might legally flow among them. The two sets of axioms were developed by scientists at the NASA Ames Research Center as part of the Amphion automated programming effort [Lowry et al. 1994]. Space scientists can automatically generate programs using Amphion by graphically specifying the geometric relationships among the elements of the problem. Amphion converts the graphical relationships into a theorem and attempts to prove the theorem using the application-domain axioms. If the proof is successful, the parallel axioms describing the program components are used to actually generate a Fortran program to solve the original program.

The original set of axioms describing the Fortran library were constructed manually. Our job as reverse engineers was to automatically analyze the library in order to augment the domain description. In particular, we were concerned with issues such as how the library dealt with errors, what should be done with routines that computed more than one result, and how to restrict the expression of the axioms to more accurately mirror Fortran subroutines in cases where they did extensive argument checking before allowing computation to proceed. That is, the reverse engineering task was to analyze the software, based upon the existing axioms, in order to augment them to more completely specify the library.

3.2. Algebraic specification (Section 5.7)

Another formal representation technique that we are currently exploring is algebraic specification. Whereas predicate logic describes the state of a computation as it proceeds, algebraic specification uses equations to denote the relationships between the operations that a domain provides. Additionally, the equations can be interpreted as rewrite rules for expressing how a computation progresses.

We have used algebraic specifications to model a class of numeric algorithms called ROOT FINDERS. A root finder is a function that takes as input an interval on the real line and a real-valued function on that interval and tries to determine a root of the function within the interval. There are a wide class of root-finding algorithms with robustness–performance trade-offs. We have constructed an algebraic specification encompassing these and are using it to drive the analysis of a Fortran root finder called `ZEROIN` [Forsythe 1977]. `ZEROIN` is a short but complex subroutine which adaptively uses three different root finding methods on a single invocation in order to guarantee convergence while providing performance.

The particular algebraic specification tool that we are using is called Specware [Jullig et al. 1995]. Specware not only supports the algebraic specification of software but is also able to automatically generate provably correct C++ code satisfying the specification. To do this, it requires complex formal methods for combining specifications while maintaining correctness properties. It is this ability to combine specifications that attracted us to it for modeling the interleaved algorithms used by `ZEROIN`.

3.3. Knowledge representation language (Section 5.3)

Knowledge representation languages are commonly used in the Artificial Intelligence research community as a flexible way of modeling knowledge. A knowledge representation language allows an analyst to describe an entity as a set of attribute-value pairs. Attributes provide a set of properties describing a class of entities; values for the attributes characterize individual elements of the class. In some cases, a knowledge representation language also supports the ability to describe one class of entity as a subclass of another, thereby reducing the total amount of effort required to describe a domain.

We used a knowledge representation language to model the domain of software USER-INTERFACE WIDGETS. We were particularly interested in the problem of migrating an interface from a traditional textual, command-based interface into a graphical user interface. Migration has three parts: detecting user-interface code in legacy software, deciding how to replace it, and performing the actual transformation.

We used a knowledge representation in support of the first two tasks—detecting user-interface code and determining suitable replacements. The former task, in turn had two parts: locating sections of code that performed user-interface operations and determining the nature of the interaction. It was the second part for which a model of the domain was required. In particular, we built a domain model for USER-INTERFACE WIDGETS. The user interface code was matched against abstract elements of the domain model to determine the nature of the interaction. The results of this determination was then used to suggest replacement widgets from the target graphical user-interface toolkit.

This project used the Classic knowledge representation language from AT&T [Borgida et al. 1989; Brachman et al. 1990; Resnick et al. 1993]. Classic comprises both a language and a truth-maintenance system capable of automatically inferring implications when new knowledge is added to a model. In particular, as we learned more about the software being analyzed, Classic automatically determined candidate replacement widgets which provided required user-interface functionality.

3.4. Entity–Relationship diagrams (Section 5.1)

Virtually any representation technique used by software analysts and designers for originally developing a software system can be used by reverse engineers to express the system's application domain. One of our early studies made use of Entity–Relationship (ER) models [Chen 1976]. ER models are normally used as part of conceptual database design in order to better understand the nature of a problem before constructing a logical data model. An ER diagram consists of nodes denoting entities and decorated arcs denoting associations or relationships among the connected entities.

We used an Entity–Relationship model as part of a study of migrating Cobol information systems to a fourth-generation language. The particular domain of interest was REPORT-WRITING. A report writer is a computer program that takes as input a description of the format and contents of a report and produces the report as output. Report writers can usually be found as bundled software provided with database management systems, but many older information systems included the data extraction and formatting directly in the program source code. In this case study, we used the ER model to express the expected features of the software, as required by Synchronized Refinement.

3.5. Static object models (Section 4.2.1, Section 5.5, and Section 5.6)

A similar, but more up-to-date representation for modeling the structure of software is a static object model. Static object models are part of the Object Modeling Technique (OMT) developed by Rumbaugh and his colleagues at General Electric [1991]. OMT uses three forms of representations: state machines to describe behavior, dataflow diagrams to convey functionality, and static object models to express structural properties. A static object model is similar to an ER model in that it consists of nodes denoting entities and arcs indicating associations. Although the symbols differ somewhat from ER diagrams, at a gross level, the contents are similar. One refinement, however, is that the nodes in a static object models can be thought of as classes in the sense of object-oriented analysis. That is, a node is annotated with information about attributes and about the functionality provided by instances of its particular class.

We have used static object models to provide a graphical overview of an application domain as part of our dowser tool effort described in *Section 4*. The case studies associated with this research concern two domains. The first is internet WEB-BROWSERS, and the second is SOFTWARE LOADER/VERIFIERS (tools for downloading software into embedded systems and checking its integrity upon arrival). Our particular interest was in providing a graphical representation of an application domain from which an analyst could navigate to particular sections within system documents or into the source code itself.

3.6. Object-oriented frameworks and path expressions (Section 5.2)

One limitation of static object model diagrams is their inability to express how a collection of object-oriented classes collaborate to accomplish some larger purpose. Although the diagrams include arcs denoting associations, these associations are often too abstract to drive the search for their implementations. Object-oriented frameworks (OOFs) [Johnson and Foote 1988] extend object models to express collaborations. An OOF is a collection of abstract classes which, when instantiated appropriately, can express the complex collaboration of a set of classes. An example of an OOF is a recursive-descent parser in which classes correspond to non-terminals in the grammar of the language being parsed. These classes collaborate by invoking each other recursively.

When considering collaboration, an object-oriented frameworks is best thought of as a design technique rather than a representation. To more completely specify the nature of a collaboration, we have augmented them with path expressions [Campbell 1974]. A *path expression* is a form of regular expression useful for representing the history of a complex computation. We use them to indicate which classes are responsible for providing functionality at specific points within a computation.

We used OOFs with path expressions to model the REPORT-WRITING domain mentioned in *Section 3.4* and *Section 5.2*. With the object-oriented framework technology, representing a domain model is reduced to instantiating class parameters. That is, the framework can work only if a minimal set of parameters has been specified so that a complete, albeit skeletal, report can be generated. Once this is done, other parameters can be used to specify more complex reports.

One extra advantage of using object-oriented frameworks derives from their origins as abstract classes. As information is gleaned from a program, the abstract classes can be instantiated with the specifics of the implementation. In this way, the OOF serves both as the domain model and as a representation for the information learned during analysis.

3.7. Comparison

The representations described in this section can be compared along various dimensions.

- **Formality:** the extent to which the representation has a formal semantics and requires mathematical sophistication to use;

- **Tool support:** the extent to which the representation is supported by tools;
- **Support for annotation:** the extent to which the representation is capable of supporting annotations describing how program constructs are related to domain entities;
- **Support for reengineering:** the extent to which the reverse engineering representation can be used to drive program reengineering or evolution.

Table 1 summarizes the various representation techniques described in this section along these dimensions.

Table 1. Summary of Representations.

Representation	Domain	Formality	Tool Support	Annotation Mechanism	Support for Reengineering
Predicate Calculus	SOLAR SYSTEM KINEMATICS	High	Automatic reasoning systems	Two-tier as with Amphion	Theorem prover
Algebraic Specification	ROOT FINDING	High	Specware and others	Code generators	Code generators
Knowledge Representation Language	USER INTERFACE WIDGETS	Moderate	Classic	Hooks to code	Suggested replacements
Entity–Relationship Diagrams	REPORT WRITERS	Moderate	CASE tools	Code generators in CASE tools	Code generators in CASE tools
Static Object Models	WEB BROWSERS; SOFTWARE LOADER/ VERIFIERS	Moderate	CASE tools	Code generators in CASE tools	Code generators in CASE Tools
Object-Oriented Frameworks	REPORT WRITERS	Moderate	None	Instantiation	Inheritance and refinement

3.8. Conclusion

Among the aspects that might comprise a domain representation are domain terms and their definitions, including both real world objects and required functionality; solution strategies and 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 representation 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 implementation 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 a single 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.

4. Tools

Software maintenance and enhancement activities are the largest part of total lifecycle costs [Boehm 1981]. Moreover, understanding source code and change requirements dominate maintenance and enhancement effort [Fjeldstad and Hamlen 1983]. Consequently, tools for analyzing and browsing software artifacts that improve their understandability can significantly aid the overall software development process. Yet most artifacts, whether textual documents or source code, are accessed either sequentially (in the case of documents) or hierarchically, based on programming language syntax rules (in the case of source code). It is one of the theses of our work that effective access to software artifacts is enabled when that access is organized around the structure of the problem that the software is solving, its application domain.

The term *browsing* is often used when a software engineer visually examines existing artifacts. Browsing can be supported by software tools such as text editors, screen paginators, or text searching tools. Recently, hypertext has become a popular way to peruse documents by providing non-sequential, cross-referenced access, both within and between documents. In the case of source code, more sophisticated tools are available, including those which perform extensive analyses and store the results in a database to support complex queries. Even in the case of advanced source code browsers, however, the browsed material is organized around its syntactic structure; that is, how the programming language statements are nested and parsed. This may not make obvious to the reader the details of how the program actually solves an application-domain problem.

We have introduced the term *dowsing* for the process of exploring software artifacts based on the structure of the software's application domain [Clayton et al. 1997, 1998]. Webster's Third New International Dictionary defines *dowsing* as "to seek something with meticulous care especially with the aid of a mechanical device." We are also interested in mechanically supported search, in this case, within and among software artifacts. In particular, we are investigating how to use application-domain information to generate, organize, and present artifacts to software maintainers.

We have developed a domain-oriented tools framework (a *dowser*) for dowsing software artifacts. Artifacts include informal textual documents, structured documents, graphical depictions, and source code. Tools include those for analyzing and organizing textual documents and generating domain models, for analyzing source code for domain concepts, for constructing various visualizations of application domains and source code, and for automatically generating domain-based hypertexts.

4.1. Scenario

Imagine a situation in which a software engineer is maintaining and enhancing a software simulator for hardware systems such as massively parallel processors, caches, pipelined systems, or superscalar architectures. Simulations of these systems are built by researchers and engineers as an integral part of designing and understanding new architectural innovations. Suppose an engineer plans to extend an existing sequential simulator for a parallel processor. The following are some of the activities and questions that might arise.

- **Identifying types of actors:** The engineer is interested in domain-oriented types, such as whether the system clock is discrete or continuous and what types of parallel instructions are simulated, rather than focusing solely on low-level data types (e.g., integers or strings) or common abstract data types (e.g., linked lists or hash tables).
- **Understanding the domain-specific software architecture:** A common problem in simulating parallel systems is correctly modeling and coordinating concurrent events using a sequential process [MacDougall 1987; Zeigler 1976]. There are standard domain-specific software architectures used to solve this problem and to ensure that events are not generated or processed out of order. Two of the most common are *event-driven* (in which a centralized event agenda keeps track of pending events and orders the event processing) and *synchronous* (in which event generators, such as processing nodes, are simulated in lock-step with a global clock). The engineer is interested in what type of architecture is used in the simulation at hand, because this affects how new events are added to the simulation.
- **Checking domain-oriented constraints:** In hardware simulation, there are constraints on the chronology of simulated events which are satisfied using standard mechanisms. It is helpful to answer a question about how a given domain-oriented constraint is satisfied by connecting it to the program mechanism that maintains the constraint. For example, there is a constraint in

simulating parallel systems that events are processed in chronological order (that if several events are pending, no event with a later time-stamp is processed before one with an earlier time-stamp). If an event-driven simulation is being used, this constraint is often satisfied by using a priority-queue implementation of the event agenda, with arrival time as the priority. Understanding how the priority queue implementation relates to event handling is a key pre-requisite to successfully modifying the simulator.

The scenario suggests various forms of automated support such as the following.

- Queries posed in terms of the domain vocabulary and concepts;
- Queries about relationships between concepts or actors in the domain;
- Results of queries portrayed graphically to convey the relationship among the concepts of interest;
- Identification of typical programming solutions used in the domain, such as reference architectures, patterns, and programming cliches;
- Data-type analysis combined with concept assignment [Biggerstaff et al. 1994] to connect programmer-defined types with domain concepts.

4.2. A tools framework

Software artifacts are normally organized using the directory structure of a computer's file system and the syntactic structure mandated by the software's programming language. Of course, non-source-code documentation can provide domain-centric pointers to source-code constructs. Unfortunately, such pointers typically suffer from several difficulties: being out-of-date with respect to the source code, indicating specific code without specifying the nature of the connection between the documentation and the code, and not supporting delocalized (non-contiguous) mappings between documents and code [Soloway et al. 1988]. Our approach is to provide semi-automated support, organized around the domain model, for deriving domain and code models from the actual software artifacts and for exploring the software.

To explore the efficacy of dowsing, we have designed a tools framework, called a dowser, and populated it with a variety of commercial and research tools. The tools can accept requests in the form of keyword searches, SQL queries, hypertext link-following, and graphical selections. Output consists of a variety of diagrams or textual accesses into the source code and documentation.

The software maintainer uses the dowser primarily as an exploration environment which provides access to source code, textual documents, and tool-specific structured reports and diagrams. Besides providing access to existing artifacts, the dowser is also capable of controlling a variety of analysis tools and generating further artifacts in the form of reports and graphical depictions. Of course, these derived documents should also promote further exploration and must therefore support the same kinds of access and queries as the original artifacts. As illustrated in *Figure 2*, there

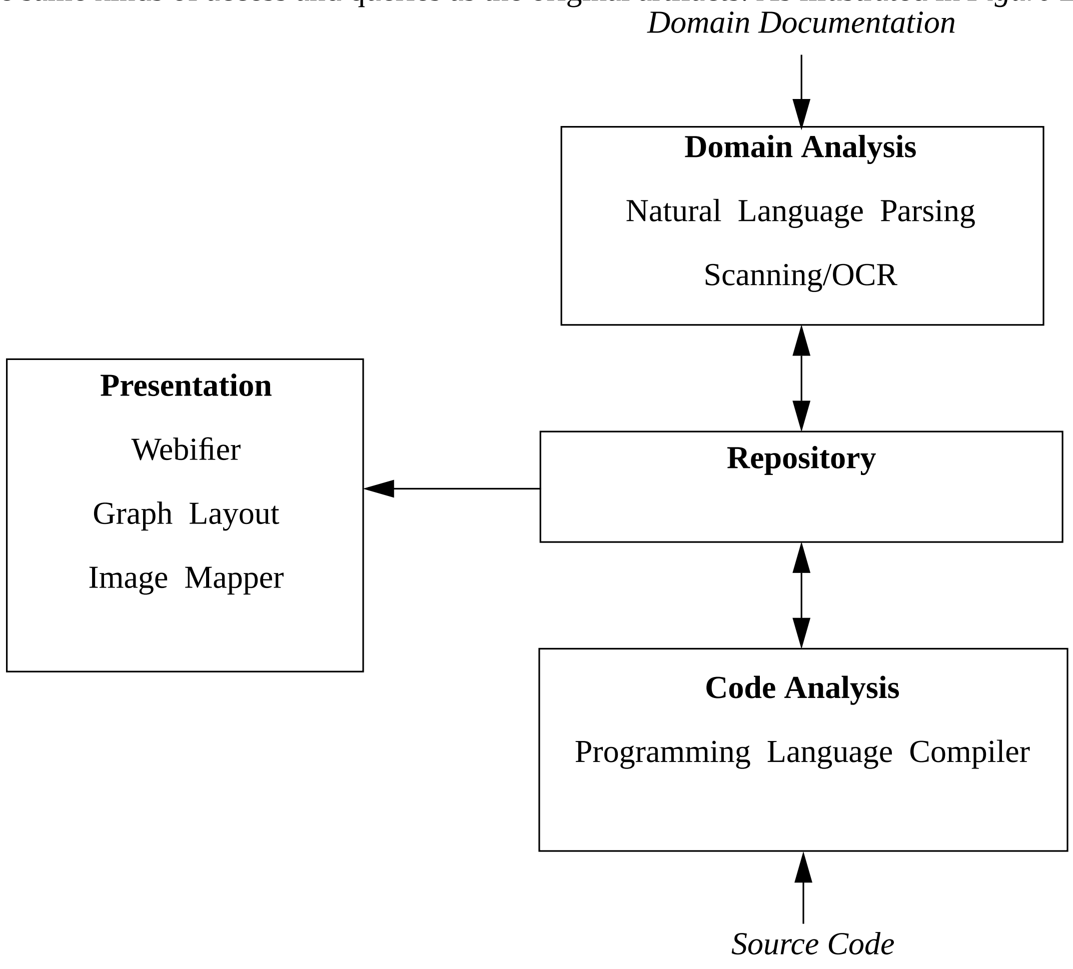

Figure 2. Dowser tools framework.

are several components in the dowser tools framework.

- Tools for constructing a model of the application domain from textual and diagrammatic documents;

- Source-code-based tools for analyzing programs;
- Repository tools for maintaining the conceptual and logical models constructed by the domain-modeling and program-analysis tools;
- Hypermedia tools for constructing linkages between the domain and program models and generating presentations, both graphical and textual, of them;
- A user interface (not shown) with which the software maintainer explores the software artifacts.

In general, each of these components is represented by multiple tools. Hence, we use the phrase *tools framework* rather than merely tool suite.

4.2.1. Domain analysis tools

To support dowsing, a software maintenance tool must provide access to software artifacts in terms of domain concepts. Two ways of conveying such concepts to an analyst are textually and graphically. Textual expression of domain concepts makes use of domain vocabulary. Graphical depictions convey relationships among domain concepts.

The key difference between the dowser and traditional browsers is that the user can explore the artifact base in terms of application-domain concepts as well as traditional textual and source-code methods. Domain access is provided in terms of an explicit domain model comprising two parts: a vocabulary and an static object model. The vocabulary consists of key domain concepts from which keyword searches can be composed. It also serves as the starting point for the construction of the static object model. This model consists of domain actors and objects and the associations among them. It can naturally be conveyed in graphical form and consequently supports visual exploration.

We assume that domain knowledge is obtained from various textual descriptions and diagrams such as might be found in a requirements document. Consequently, such descriptions must be converted into a formal model of the domain. Although this cannot be done entirely automatically, we have developed or made use of tools for scanning in text and converting it into ASCII characters [Caere 1994], for formatting the resulting text stream into structured HTML (webifier), and for performing several types of lexical analyses. These lexical analyses include breaking text into words and filtering out uninteresting words, parsing it to determine parts of speech, detecting candidate objects, actions, and states, and computing word frequencies (N-Gram).

4.2.2. Program analyzers

The dowser includes two types of program analysis tools: traditional, commercially available, language-based analyzers, and research prototypes that relate the program-based information to higher-level concepts, primarily architectural views of how the program realizes non-functional requirements such as performance or security. For the first type, we have used two commercial tools, the Software Refinery tool suite (Refinery) [Reasoning 1990] and the Source Browser Facility (SBF) [SUN 1994] that is provided by SUN Microsystems with its Solaris C, C++, and Fortran compilers. For the second type, we have built tools to compute three relations: how components interact by invoking each other, how user-defined data types are defined in terms of each other, and how modules are coupled to each other.

4.2.3. Repository tools

At the core of the dowser is a repository for entering and managing the conceptual and logical models generated by the other tools. We have explored two such tools, the Refinery's object-oriented repository and a public-domain, relational, record-management tool called `MySQL` [MySQL 1999]. The former has the natural advantages of being congruent to the domain representation and integrated with one of the code analysis tools. The latter approach has the advantages of the SQL relational-algebra query language and a smaller conceptual footprint.

4.2.4. Hypermedia-based linkage tools

Code models and domain models must be interlinked to enable effective domain-based program understanding. Direct linkages are difficult to make because of the conceptual distance between the program source code and domain models. Consequently, we use software architecture as a stepping stone between the two. As an example, imagine the following situation related to an internet browser. A software maintainer trying to understand the browser’s source code comes across a significant amount of code related to the browser's page cache. But there is no mention of caching in any of the requirements documents for web browsing. If anything is mentioned at all, it is that page retrieval should be rapid. This is a non-functional requirement, and caches are one way of satisfying it. An architectural description of the browser that views its modules in terms of how they support performance will feature the cache management software as a major component. The caching code can then be related to the cache management component, which exists to satisfy a performance requirement from the application domain.

Constructing an architectural stepping stone requires that programming knowledge (such as how to provide increased performance using caching) be available. Currently, we assume that the knowledge exists in the mind of the software maintainer. Nevertheless, we need to provide some passive mechanisms for entering and maintaining the linkages within the computer. And we need a way of providing intelligent presentations of the information to the maintainer. By *intelligent* we mean that the presentations enable the user to follow the links. Examples of such presentations that we construct are dynamically generated web pages, image maps, and thumbnail sketches of the results of keyword-search requests.

4.2.5. User interface

A software maintainer dowses a software system using a model of the program's application domain. The model provides both a textual dictionary of domain vocabulary and a graphical depiction of the major entities and relationships in the domain. Not all of the entities and relationships are realized in any single program in the domain, so it is important for the dowser to indicate how the particular program instantiates the domain model. This is done by giving the software maintainer access to the underlying program source code and documents via domain concepts. Such access can take the form of keyword searches, SQL queries, web-page links, or image maps selections. We have used a front end consisting of HTML forms and CGI scripts to provide this interface. We are currently building a more robust version using tcl/tk [Ousterhout 1994].

4.3. Conclusions

One of our research hypotheses is that dowsing is a more effective way to access software artifacts for the purpose of gaining understanding than is traditional, source-code browsing. To test this hypothesis we have designed a tools framework, a dowser, and populated it with a variety of commercial and research tools. We then used the dowser to look at two applications (described in *Section 5.5* and *Section 5.6*). And we have informally found that the dowsing framework is a comprehensive and integrated way to deal with the disparate forms of loosely related software artifacts. Nevertheless, our underlying thesis that domain-oriented browsing is a more effective approach to program understanding than traditional browsing needs to be validated through actual use in performing maintenance tasks, which we can now use the dowser to test. To this end, we are currently constructing a version of the dowser for public release.

5. Case studies

During the course of investigating domain-based program understanding, the primary research method has been the case study. This section summarizes the case studies performed. For each case study, we give a table containing the details of the study and then a brief discussion of the research question explored and the results obtained. The order of presentation roughly corresponds to the order in which the case studies were performed.

5.1. Growing a domain model while reverse engineering

Table 2. TRANSOPEN Case Study—Part 1.

Time period	1992-1993
Sponsor	Army Research Laboratory
Program name	IMCSRS
Program type	Management information system
Program description	Materiel status maintenance, update, and reporting
Primary domain	REPORT-WRITING
Programming language	Cobol
Lines of code	636 LOC (part of a system consisting of 10 KLOC)
Representation used	Entity–relationship diagram
Tool support	Software through Pictures CASE tool
References	[DeBaud et al. 1994; Eidbo et al. 1993]

Background. The Army Research Laboratory sponsored the TRANSOPEN project 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 project comprised work on communications protocols, database integration, and business process re-engineering. Our part in this project explored the actual transition of the software, including issues of strategy selection, reverse engineering process, and tool support.

Objective. The specific objective of this case study was to explore how to grow a domain model during the course of reverse engineering a program. An initial model of the report-writing domain was used to generate expectations for what to look for in the code. As the code was read, expectations were confirmed or refuted. Occasionally, observations were made that required the domain model to be reorganized or enhanced.

Results. A domain model was successfully constructed. Moreover, using the domain model helped achieve an enriched understanding of the program. In addition, during the course of program understanding, documentation bugs were detected.

Observations. This was our first case study of domain-based program understanding. Consequently, we learned many things which were confirmed by later studies, including the following.

- Programs typically implement concepts from more than one domain. 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 Cobol was required.
- A domain model can help detect bugs in the situation where the code did not meet the domain expectations.
- Sometimes the program structure suggested by the domain model has been distorted in its course to implementation. This might mean that the code should be restructured. Alternatively, in some cases, other domains may have influenced program structure.
- The level of the programming language in which the program is written can play an important role. If it is too low, the reverse engineering will experience difficulty in matching expectations to code.
- REPORT-WRITING is a mature domain; our success on this case study must be tempered with the possibility that less well-defined domains may prove more problematic.

5.2. Understanding programs using a domain model

Table 3. TRANSOPEN Case Study—Part 2.

Time period	1993-1995
Sponsor	Army Research Laboratory
Program name	IMCSRS
Program type	Management information system
Program description	Material status maintenance, update, and reporting
Primary domain	REPORT-WRITING
Programming language	Cobol
Lines of code	636 LOC (part of a system consisting of 10 KLOC)
Representation used	Object-oriented framework
Tool support	None
References	[DeBaud 1994; DeBaud et al. 1994; DeBaud and Rugaber 1996; DeBaud 1996]

Background. This case study was also performed as part of the TRANSOPEN project and used the same program as the study described in *Section 5.1*.

Objective. The primary objective of this study was to explore the use of a domain model in driving the program-understanding process. A secondary objective was to explore the use of object-oriented frameworks (OOFs) as a representation vehicle for the domain model and to hold the results of the annotation process.

Results. An object-oriented domain model was constructed and used to guide the program understanding process. Moreover, documentation of the program understanding was easily realized by instantiating the classes in the framework. Furthermore, additions to the domain model were easily accommodated by specializing (subclassing) the classes in the framework.

Observations. This case study was done in conjunction with the one in *Section 5.1*. Consequently, redundant observations will not be repeated. However, we did observe the following additional points.

- Object-oriented frameworks enforce a natural decomposition of objects into their constituents. This structure supports the process of looking for high-level (domain) concepts in terms of low-level (program) constructs. That is, useful expectations were easy to generate using the OOF.

- On the negative side, the OOF does not make readily apparent a program's functional goals. Because object-orientation presumes a viewpoint in which entities (nouns) are primary, functions (verbs) and how they together accomplish the program's purpose becomes difficult to discern.

- Domain models are inherently incomplete. However, when a deficiency is detected, an OOF's specialization hierarchy easily supports extending the domain model to include new information.

- Annotation is readily accomplished by instantiating the OOF's abstract classes.

- Because an OOF is an operational model (includes behavioral information), an opportunity exists to easily translate the OOF representation of a program into a different (object-oriented) programming language.

- The method does not deal well with code from a domain other than the one being used to drive the understanding process. That is, success is dependent on the extent to which the totality of a program's domains have been modeled.

5.3. User interface migration

Table 4. MORPH Case Studies.

Time period	1993-1997
Sponsors	Army Corps of Engineers Research Laboratory, Defense Advanced Research Projects Agency
Program names	Knowledge Worker System, Weltab, Various public domain C programs
Program type	Information systems, Games
Program description	Various programs with textual (command-based) user interfaces
Primary domain	USER-INTERFACE WIDGETS
Programming language	C
Lines of code	From 100 LOC to 30 KLOC
Representation used	Knowledge representation language
Tool support	Software Refinery, Classic knowledge-representation and truth-maintenance system
References	Moore et al. 1993, 1994; Moore 1996; Moore and Rugaber 1997a, b]

Background. User-interface software technology is changing rapidly. Consequently, there is a great need for automating the migration of legacy systems to use modern graphical user interfaces (GUIs). Fortunately, there are two factors that make this problem somewhat easier than the general problem of program reengineering. First, the user interface part of a program's source code is relatively independent of the functional part. Hence, understanding the entire program may not be required. Secondly, even with legacy software, the implementation of a user interface is often done using a library of subprograms. This makes the detection of the user-interface part of the program even easier by merely looking for calls to members of the library.

The domain model represented in Classic was used to perform three different tasks: to provide structure (expectations) to the process of detecting user-interface portions of the legacy software, to provide a repository for information detected as the program was being analyzed, and to suggest replacement widgets.

Objective. The objective of the study was to model the domain of user interface software technology and use the model to support the process of migrating programs from textual, command-based user interfaces to graphical user interfaces. As a secondary goal, we explored the use of a knowledge representation language for expressing the domain model.

Results. We actually conducted a series of case studies performed over an extended period of time. A migration method was devised and validated, and a tool was built that supports the process of detecting user-interface software written in the C language and suggesting alternatives written in various GUI toolkits. Both the method and the tool are called MORPH (Model-Oriented Reengineering Process for Human–computer interfaces).

Observations.

- The user-interface software technology domain is not nearly as standardized as we expected. Although different toolkits all provide much the same functionality, the various widget classes are not identical, with overlaps and inconsistencies preventing direct translation. The more abstract classes in the domain model are required for successful modeling and migration.
- MORPH has been able to model a variety of GUI toolkits including tcl/tk, Java Abstract Windowing Toolkit (AWT), Motif, and MS-Windows.
- MORPH was also robust enough to deal with a wide variety of legacy C programs including information systems, an e-mail reader, and various computer games.
- Replacing calls to one subprogram library with calls to another, even when done in an intelligent way, is only part of the problem of migrating textual, command-based user interfaces to support GUIs. Programs using GUIs are typically organized architecturally in the style of reactive (event-loop) programs. Conversely, legacy programs are most often organized procedurally, with a main program driving subprograms hierarchically down to the level where calls are made to an I/O library. This means that to do a thorough migration requires restructuring the overall program architecture, a potentially much more difficult problem than merely substituting library calls.

5.4. Domain-model extension via code analysis

Table 5. NAIF Library Case Study.

Time period	1994-1995
Sponsors	NASA, DARPA
Program name	Navigation Ancillary Information Facility Library - SPICELIB
Program type	Scientific subroutine library
Program description	Software for scientist to analyze data from space missions
Primary domain	SOLAR SYSTEM KINEMATICS
Programming language	Fortran
Lines of code	600 routine, 234 KLOC
Representation used	Predicate logic expressed as Lisp rules
Tool support	Software Refinery
References	[Rugaber et al. 1996, 1995a, b]

Background. This domain-based reverse-engineering project was sponsored by the National Aeronautics and Space Administration (NASA) Ames Research Laboratory and the Defense Advanced Research Projects Agency (DARPA). NASA researchers supported efforts at the Jet Propulsion Laboratory (JPL) to build scientific/engineering applications involving 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. An extensive library, called the Navigation Ancillary Information Facility Library (NAIF) SPICELIB library, exists that can be used by scientists to help construct such applications. The library is written in Fortran and contains 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 NASA wanted to understand it better. In particular, they wanted to have a formal specification to support automatic application generation as described in *Section 3.1*, and to do so requires reverse engineering the library.

Objective. Our primary objective was to reverse engineer the library, looking for particular constructs bearing on each subprogram's specification as expressed in its domain model.

Results. We were able to successfully extend the domain model in several interesting ways. For example, SPICELIB includes a sophisticated error handling capability, accesses to which are written in such a way as to provide two benefits. The first was that the style of the accesses is conventional enough that detection is eased. Secondly, many detected error situations correspond directly to preconditions for successful subprogram execution. These preconditions could then be added to the subprogram's specification. Another example concerns subprograms that returned more than one result. There are two reasons for this practice. One is that the Fortran language did not provide a direct representation for compound data types such as record structures. The subprograms actually return a single domain entity, but Fortran requires multiple results to accomplish this. The second reason was efficiency—it is faster and smaller to compute the multiple results in one subprogram than to break it into partially redundant pieces. Even in this situation however, the domain model has to be extended to report that both results are computed.

Observations.

- Using the predicate calculus as a domain representation worked well when trying to detect preconditions for program execution. Detected preconditions could be easily expressed as predicates extending the model.
- One reason for this facility is that the subprograms considered were largely self-contained and functionally organized. That is, the subprograms were primarily concerned with computations rather than with manipulating data structures.
- Moreover, the kinds of understanding that we were trying to obtain was mostly *black-box* in nature. That is, we were concerned with the circumstances under which a subprogram would legally execute and the effect of the execution on the results achieved. We were not concerned with how the subprogram achieved the results.

5.5. Domain architectural analysis

Table 6. DARE Case Study—Part 1.

Time period	1997-1998
Sponsor	Army Research Laboratory
Program name	Mosaic
Program type	System software
Program description	Version 2.4
Primary domain	WEB BROWSERS
Programming language	C
Lines of code	100 KLOC
Representation used	Static object model
Tool support	Dowser
References	[Clayton et al. 1997, 1998a]

Background. Programs and domain descriptions are at vastly different levels of abstraction. To be able to use domain descriptions as a way of accessing programs requires that bridges be built mapping domain concepts to corresponding program constructs. As part of the Domain Analysis and Reverse Engineering (DARE) project, we looked at the Mosaic web browser, a publicly available, intermediate size, C program. We performed this analysis in the context of our development of dowsing tools, as described in *Section 4*.

Objective. The primary objective of this case study was to explore the relationship between domain-based information and programming-language-based information. In particular, how visible domain concepts are in the program source code? We applied ideas from software architecture to this problem.

Results. The primary result of the study was that a software architectural description can play a useful role in relating domain information to program information. Many program constructs occur to address specific non-functional requirements, such as performance or reliability. And the highest design level at which such concerns are manifest is the software architecture. Consequently, it is easier to map from domain concepts via architectural units to code constructs than it is to go directly. In the DARE case study, the application-domain model started as an abstract model derived from existing textual descriptions of the World Wide Web and evolves toward more specialized models by incorporating detailed information extracted from architectural analysis of source code and written documents.

Observations.

- It is difficult to find the implementations of non-functional requirements in source code without the help provided by an architectural description. Conversely, there are many implementation details for which an architectural description can be used to find the appropriate correspondent in the domain description.
- Mature domains have preferred solutions to high-level implementation problems such as satisfying non-functional requirements. Such solutions are manifested at the architectural level. Knowledge of what sort of (architectural) solutions to expect can ease the understanding process.
- There are some relatively straightforward source code analyses which can provide useful architectural information. Three that we used are the calling (who-calls-whom) relation, the definition by the programmer of complex data types from simpler ones, and knowledge of how global data is accessed by the various subprograms.
- The above analyses may be used in various ways to cluster and abstract a high-level representation of a program. However, these analyses may be subverted by artifacts inserted during program implementation. For instance, a designer may have interleaved two logical modules into one or used one function with an interleaved design to construct functionality for two different modules. This confounds the analyses unless detection of the design decisions can also be incorporated into the abstraction process.
- Understanding how architecture is used to solve implementation problems is knowledge-intensive. It is likely that further research into software architecture will be required before effective automation will be enabled.

5.6. Validating a domain model with code analysis

Table 7. DARE Case Study—Part 2.

Time period	1997-1998
Sponsor	Army Research Laboratory
Program name	M1A2
Program type	System utility
Program description	Downloads executables into embedded systems
Primary domain	SOFTWARE LOADER/VERIFIER
Programming language	Ada
Lines of code	43 KLOC
Representation used	Static object model
Tool support	Optical character recognition, dowser, RMT [Murphy et al. 1995]
References	[Clayton et al. 1997, 1998a]

Background. This case study (also part of the DARE project) examined the domain of SOFTWARE LOADER/VERIFIERS (SL/Vs). An SL/V is a tool for downloading binary executables into mobile, embedded systems such as might be found in tanks or airplanes. We obtained an example SL/V and its related documents from the U.S. Army Tank and Automotive Command (TACOM) where it was part of their SPAIDS effort (Software Program for Affordability Improvements in Depot Systems). The SL/V is written in the Ada language and comprises 43,000 lines of code. The documentation is available only in hardcopy form, so we made use of scanning and optical character recognition tools for converting it into electronic form.

Objective. The primary objective of the study was the same as that in the Mosaic study described in *Section 5.5*. A secondary objective was to use a domain model that had been generated via reverse engineering as the validation of a separate domain model generated by an independent development team working for TACOM to construct a generic SL/V reusable across a variety of embedded systems.

Results. Different software systems require different tools. In this case study, we were dealing with an Ada program. Consequently, we made use of the Software Refinery. We were also given hardcopy documents for which no corresponding electronic versions were available. To accommodate this, we scanned in the documents and used commercial optical-character-recognition software to construct a soft copy. We also wrote our own software to convert the soft copy into HTML so it would be more accessible within the dowser framework. Thus, one primary result of this effort was to validate the robustness of the dowser framework by forcing it to deal with a wider variety of tools.

Synchronized Refinement requires a domain model. In this case study, we generated one by lexically analyzing frequently occurring words in the SL/V documents to determine important concepts. We then used the Object Modeling Technique [Rumbaugh et al. 1991] to construct a static object model suitable for Synchronized Refinement. As a side benefit, our object model was used to help validate an independent object model being generated in the course of developing a generic SL/V for the U.S. Army.

Observations.

- Domain information generated by reverse engineering should be consistent with that arising from conventional domain analysis. There are important differences in granularity and mapping, but it is possible to identify domain concepts in code.
- Conversely, reverse engineering, by analyzing an existing program, can support the evolution of that program by identifying manifestations of requirements that may be affected by a proposed change.

5.7. Representation of complex interleaved domains

Table 8. Algebraic Specification Case Study.

Time period	1997 - current
Sponsor	National Science Foundation
Program name	ZEROIN
Program type	Numerical
Program description	Subprogram from mature numerical library
Primary domain	ROOT FINDING
Programming language	Fortran
Lines of code	102 LOC
Representation used	Algebraic specification
Tool support	Specware
References	None yet

Background. Even small domains can be complex when manifested in software. And domain concepts can be widely dispersed within a program. We have over the course of several years examined a small program (ZEROIN) that does numeric computations [Clayton et al. 1998b; Rugaber et al. 1990; Rugaber 1997]. The program is only 102 lines of Fortran and does not make use of complex data structures. However, it is difficult to understand, and serves as a good test of reverse engineering methods, representations and tools.

Objective. The primary objective of the study is to explore the use of algebraic specification as a representation for domain information. In particular, the mathematical nature of ZEROIN maps well to the formal nature of the equations used to express semantics in this representation. In addition, we have available to us the Specware tool which supports not only the specification of software algebraically, but also includes code generators capable of producing a provably correct implementation of the specification [Jullig 1995]. Consequently, by trying to express root finding in Specware and by using it to generate ZEROIN, we can explore the effectiveness of algebraic specifications for representing real code.

Results. This work is still in progress. We have produced a partial specification of the root finding domain and used it to produce a simple root finder. As we become more familiar with Specware, we intend to recreate the whole program.

Observations.

- The device used by Specware to automatically generate code from specifications uses functional composition. That is, in situations where two domain concepts are closely related, the generated code uses function calls to implement the relation. By contrast, ZEROIN uses tightly interleaved code with no function calls other than to the function whose root is being found. Hence, this should be a good test of the flexibility of algebraic specifications.
- Specware is a newly released tool. We are still learning how to use it and to overcome its limitations. For example, it does not currently include built-in specifications for real numbers, an essential ingredient in root finding.

5.8. Tool support for domain-based browsing

Table 9. Dowser Case Study.

Time period	1998 - current
Sponsor	Spectra Research
Program name	BTTSIM
Program type	Simulator
Program description	Missile guidance simulator
Primary domain	GUIDANCE AND CONTROL
Programming language	Fortran
Lines of code	1773 LOC
Representation used	Static object model, call graph,
Tool support	Dowser
References	None yet

Background. The dowser described in *Section 4* is a research prototype. We are in the process of scaling it up to deal with production software. Our sponsor, Spectra Research, is concerned with the effort to migrate simulation software to conform to the new High Level Architecture (HLA) proposed by the Defense Modeling and Simulation Office (DMSO) [DMSO 1999]. HLA supports the composition of simulations into federations. To enable composition, participating simulations must provide various pieces of information, including a domain object model.

Objective. Our main objective is to stress the dowser framework with a new domain. A secondary objective is to compare our approach to domain modeling with that of DMSO.

Results. This work is in progress. The particular simulation we are looking at concerns missile guidance. We have constructed a domain model, analyzed available documentation, and performed initial code analysis. Simultaneously, we are migrating the dowser user interface to tcl/tk to improve its portability.

Observations. None yet.

5.9. Summary

The wide variety of case studies that we have looked at provide evidence for robustness of our approach. In particular, programs are written in Cobol, Fortran, C, and Ada; and representations cover predicate logic, algebraic specification, a knowledge representation language, object-oriented frameworks, and traditional software engineering notations including entity–relationship diagrams and static object models. Sizes of programs have ranged from 100 LOC to 100 KLOC, and domains have included information systems, simulations, games, and numeric computations. Finally, we have made use of a wide variety of tools, including those which are commercially available, research prototypes, and those we have built ourselves. We will discuss the issues raised by these studies and what we intend to look at next in *Section 7*.

6. Related work

Although domain analysis has been an active area of research for some years, the application of domain analysis to program understanding has a much smaller literature. This section gives an introduction to the general literature on domain analysis and then goes on to survey relevant work relating it to reverse engineering and program understanding.

6.1. Approaches to domain modeling, analysis, and representation

The best overall introduction to the literature of domain analysis is the book by Prieto-Díaz and Arango [1991]. Here we describe several approaches that have influenced our work on the application of domain information to program understanding.

6.1.1. 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. An advocate of this approach is Srinivas [1991a]. 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 [1991b] 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.

6.1.2. Denotational

Another formal approach to modeling is derived from denotational semantics. Several denotational-specification languages, such as VDM [Jones 1990] and Z [Spivey 1987], 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 directly 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 reduced abstractness.

6.1.3. Draco

Another way of looking at a domain is as a programming language and its associated tools. Neighbors [1989, 1980] 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 language-based 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.

6.1.4. 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 [1988], 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 [1988], a “framework becomes more reusable as the relationship between its parts is derived in terms of a protocol instead of using inheritance.”

6.1.5. 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 [1989], 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 which 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.

6.1.6. 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 Prieto-Díaz [1989]. Although it was designed to facilitate retrieval of reusable components, faceted classification 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.

6.1.7. Technology books

Technology Books are a “low tech” approach to domain analysis and reuse, developed by Arango and his colleagues at Schlumberger [1993]. 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, stored in an on-line, object-oriented repository. 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 32 KLOC 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.

6.1.8. 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 [1988]. 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 translator-building tools.

6.1.9. 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 [1992]. In Batory's work, the major role of domain analysis is in defining the interfaces to the layers. A useful and 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" designs, 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: "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."

6.1.10. Conclusions

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 emphasized.

6.2. Domains and reverse engineering

6.2.1. 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 [1989]. 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. DESIRE itself is a prototype 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. A concept is distinguished by its features—various pieces of evidence that indicate the presence of the concept. Instances are actual occurrences of a concept in the code, with features bound to appropriate program entities. 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 entity-relationship 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.

6.2.2. LaSSIE

Our work is similar to approaches taken at the highest level of abstraction, such as LaSSIE [Devanbu et al. 1991]. LaSSIE is a knowledge-based information system designed to support maintenance and enhancements on a large software system. The LaSSIE knowledge base uses frames to represent information about the software’s domain, architecture, and code. A formally defined inheritance relation between frames provides a semantics exploited by the knowledge base to produce useful answers to imprecise questions. LaSSIE and our work have similar means, while they differ as to their ends. Both projects recognize the importance of explicitly embodying domain, architecture, and code knowledge about a system. Differences in means are largely a result of process. For example, LaSSIE extracts its knowledge from a single, well-documented system, while our work extracts its knowledge from a set of smaller, functionally similar programs; thus, LaSSIE gets domain knowledge through reverse domain engineering, while our work gets its domain knowledge through domain analysis. The ends of the two projects are also different. LaSSIE supports maintenance and enhancements of a single, existing system, while our work develops a framework which can be instantiated to create several different but related systems. In one sense, LaSSIE is interested in promoting reuse at the sub-architectural (e.g. module or subprogram) level, while our work is interested in promoting reuse at the architectural level.

6.2.3. Harris, Rubenstein, and Yeh

At a lower level than LaSSIE or DESIRE, Harris, Rubenstein, and Yeh [1995] [Yeh 1995] describe their pattern-matching techniques for extracting architectural-level features from source code. Patterns describe architectural entities and their interconnections; pattern matching is suitably fuzzy to allow for matches in the presence of obscure or missing features. The resulting set of matched entities and interconnections can then be combined, again under fuzzy pattern matching, to form architectural features. An architectural description of the software provides an important midpoint between the domain description and the actual code. To the extent that this kind of pattern matching can construct such architectural descriptions, it fills an important role. In the other direction, the domain knowledge made available in our work could serve to direct and sharpen pattern matching.

6.2.4. DECODE

Approaches still lower than those previously described are taken by such systems as DECODE [Quilici and Chin 1995], which uses cooperative, bottom-up code analysis to create object-oriented descriptions (that is, data plus operations) of existing code. The code is analyzed using pattern-matching techniques scalable over large bodies of code. DECODE automatically generates some objects from the concepts matched in the code, but the reverse engineer cooperates with DECODE to create and organize further objects. What is missing from DECODE is the explicit connection between a set of objects derived from code and concepts in the problem domain. This lack can be seen in the nature of the questions that DECODE can answer; it can do well on code-up questions (e.g., to which object does this code belong?) but less well on concept-down questions (e.g., does any input validation get done?). To a certain extent domain concepts are embodied in the patterns that can be recognized, but when these reach their limits, the reverse engineer is thrown back on whatever domain resources are available outside of DECODE.

6.2.5. I-DOC

Our work on dowsing and linkage tools has also been influenced by Johnson and Erdem's research on interactive software explanation [1997] as demonstrated in the I-DOC tool. We share the goals of supporting interactive querying to explore results of code analyses and of using hypermedia to support browsing annotated software artifacts. Johnson's focus has been on supporting queries that are typically made in the context of performing specific maintenance tasks. While Johnson and Erdem's work supports task-oriented software understanding, our work focuses on supporting the complementary process of domain-oriented exploration.

6.2.6. Hildreth

One final approach, illustrated by Hildreth [1994], 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 of domain-centered reverse engineering, even using a non-formal domain.

6.2.7. Conclusions

Based upon our own experience and the work described in this section, several tentative conclusions can be suggested. *Domain knowledge* is a generic term, characterizing the need of virtually every program understanding effort to relate a program's source code to some model of a program's intent. Stated in this fashion, the need for domain knowledge is tautological to program understanding. To truly leverage domain knowledge, several other characteristics are required.

- **Expectations:** the process of understanding the source code should be driven by the domain knowledge, not be merely a passive target. This is similar to Brooks' early model of program understanding as an attempt to map the program "domain" to the application domain by detecting *beacons* in the source code [1983].

- **Role of informal and formal knowledge:** programs are inherently more formal than domains. Hence, there will always be a need to understand both formally and informally defined information. Which is not to say that effort should not be placed in formalizing domains. Formal domains enable application generation thereby easing not only the programming process but also reducing maintenance costs.
- **Understanding tasks**: program understanding is usually driven by the need to accomplish a specific task, such as correcting or enhancing a program. To the extent that corrections reflect a mismatch between application-domain requirements and program source code, understanding the domain can help correct the program. To the extent that enhancements reflect a program elaboration to more completely implement domain possibilities, a domain model can leverage the process.
- **Role of architecture:** architecture is a stepping stone between application requirements and program structure. Many domain models include architectural information (e.g. reference architectures and domain-specific software architectures). As such, progress in architectural analysis will play a key role in domain-based program understanding.

7. Issues and research directions

We have now been studying domain-based program understanding for over five years. We believe more strongly than ever our initial thesis concerning the essential role of domain knowledge in understanding programs. Nevertheless, there are still a variety of questions that must be addressed.

7.1. Method

- The overriding question is how to manage domain knowledge. For example, how do we coordinate a search for multiple expected constructs derived from several domains?
- How should we deal with other kinds of knowledge such architectural styles [Garlan and Shaw 1995] and categories of design decisions? How should we access them, and how should they be connected to program descriptions resulting from standard code analyses?
- We would like domain descriptions to grow and become more complete over time, but domain descriptions need to be definitive, and the reverse engineer may not be a knowledge engineer nor have sufficient expertise to judge the accuracy, relevance, and placement of the new information in the domain description.
- Inherent in the discussion of domain engineering is 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, Prieto-Díaz has described domain analysis as the extension of systems analysis to collections of problems [1991]. 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 program understanding.

7.2. 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? Certainly, domain theorists do not yet agree on how to represent domain information, but 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 formality present a barrier to some potential users, but it raises the question of how best to deal with informal information, such as the heuristics, indentation rules, and naming conventions. 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 program understanding. 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?
- As the domain model, architectural description, and program description emerge and evolve, how are their relationships captured in annotations? What representations are useful for annotations and what types of inferencing on annotations are needed? During the refinement and elaboration of the domain, architecture, and program models, expectations are generated about how they are connected. How should the expectations be represented and managed?

7.3. Tools

- All tools must confront the issue of how prescriptive to be. At one extreme are tools that dictate the exact process that users must go through to accomplish some task. At the other extreme are tools that are entirely reactive to user requests. In our work with dowsers, we have taken more of the latter approach. But the issue remains open to further exploration.
- 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 cat-

egorized as trading off power for efficiency. Where on this curve is the right place for domain-based reverse-engineering tools?

- We have performed various kinds of architectural linkage analyses. All of these generated large amounts of data that we have viewed with a graph-layout tool. While the tools we have used are robust enough to deal with large graphs, the results may be too “busy” to be useful to the analyst. We would like to look at several software visualization techniques that might further the analyst's understanding.
- The program understanding tools which currently comprise the dowser perform static analyses. It is natural to ask whether we can take advantage of dynamic information. For example, can we make use of information from actual executions to color an otherwise confusing graph, indicating “hot spots,” similar to an MRI scan of a human brain taken while the subject is performing some task.
- An intriguing question pertains to tool generation. Mature domains, such as report writers, enable application-generation technology. How about the inverse? Can we build application-analyzer generators? In fact, at least one such tool exists, GENOA, a language-independent analyzer generator [Devanbu 1992].
- 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?

7.4. 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 is to interpret manifestations of problem concepts in the source code. Based on this logic, we fully expect that any major break-through in the automated program understanding and reverse engineering area to take significant advantage of domain knowledge.

Glossary

1. *Application generator:* a tool for automatically generating application descriptions from domain models.
2. *Architectural style:* “An idiomatic pattern of system organization.” [Garlan and Shaw 1995]
3. *Browsing:* the visual examination of an existing artifact such as a program or a document.
4. *Composition/decomposition:* A category of design decision describing how programs are built up from parts and problems are broken down into smaller, more easily solvable sub-problems. [Rugaber et al. 1990]
5. *Data/procedure:* A category of design decision describing the extent to which a computation has been captured as part of a program state or the extent to which it has been expressed as a procedural program unit. [Rugaber et al. 1990]
6. *Delocalization:* The phenomenon where “programming plans [are] realized by lines scattered in different parts of the program.” [Soloway et al. 1988]
7. *Design decision:* An explicit refinement of a program design. Each decision reflects the elaboration of a design or programming plan into a more concrete realization. [Rugaber et al. 1990]
8. *Design rationale:* A record of the reasoning used to produce a software artifact from a design specification.
9. *Domain:* A problem area. Typically, many application programs exist to solve the problems in a single domain. The following prerequisites indicate 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. [Arango and Prieto-Diaz 1991]
10. *Domain analysis:* “is an attempt to identify the objects, operators, and relationships between what domain experts perceive to be important about the domain.” [Neighbors 1980]
11. *Domain model:* “The domain model should serve as a unified, definitive source of reference when ambiguities arise in the analysis of problems or later during the implementation of reusable components, a repository of the shared knowledge for teaching and communications, and

a specification to the implementer of reusable components. ... A model of a domain should include information on at least three aspects of a problem domain: concepts to enable the specification of systems in the domain; plans describing how to map specifications into code; and rationales for the specification concepts, their relations, and their relation to the implementation plans." [Arango and Prieto-Diaz 1991]

12. *Domain-specific software architecture (DSSA):* "A domain specific software architecture (DSSA) is an assemblage of software components, specialized for a particular type of task (domain), generalized for effective use across that domain, and composed in a standardized structure (topology) effective for building successful applications." [Loral 1999]
13. *Dowser:* A domain-based program understanding tool. [Clayton et al. 1998]
14. *Dowsing:* The process of exploring software artifacts based on the structure of the software's application domain. [Clayton et al. 1998]
15. *Encapsulation/interleaving:* A category of design decision describing the extent to which program components are isolated from each other or are merged together. [Rugaber et al. 1990, 1996]
16. *Generalization/specialization:* A category of design decision describing a situation where similar program constructs have been generalized into a named program unit or the extent to which the program is capable of partitioning a computation into special cases. [Rugaber et al. 1990]
17. *Interleaving:* "The merging of two or more distinct plans within some contiguous textual area of a program." [Rugaber et al. 1990, 1996]
18. *Intermediate representation:* the result of partially analyzing a program. Typically, other analyzers access an intermediate representation in order to reduce redundant effort. [ACM 1995]
19. *Non-determinism removal:* A category of design decision describing a situation where the designer has chosen to limit available implementation possibilities. [Rugaber et al. 1990]
20. *Object oriented framework:* "An object-oriented abstract design." [Johnson and Foote 1988]
21. P*ath expression:* A form of regular expression useful for representing the history of a complex computation implemented via collaborating object-oriented classes. [Campbell 1974]

22. *Program plan:* A description or representation of a computational structure that the designers have proposed as a way of achieving some purpose or goal in a program. [Soloway et al. 1988]
23. *Program understanding:* activities by which knowledge is gained about a program. [Rugaber 1996]
24. *Representation:* (1) A notation used to express an abstracted program model; (2) A category of design decision in which the designer chooses to express a design idea in terms of a separate abstraction. Usually, the separate abstraction is, in fact, more concrete (closer to the computer hardware or programming language abstract machine) than the original design idea. [Rugaber et al. 1990]
25. *Reverse engineering:* “The process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.” [Chikofsky and Cross 1990]
26. *Software design:* The process of expressing the structure, functionality, and behavior of a problem solution in the context of a particular programming language.
27. *Software evolution:* “A continuous change from a lesser, simpler, or worse state to a higher or better state [for a software system].” [Arthur 1988]
28. *Synchronized Refinement:* a method for reverse engineering software system comprising both bottom-up program analysis and top-down domain-model synthesis. [Rugaber et al. 1990]

Acknowledgment

The author gratefully acknowledges the sponsors who supported this work including the Army Research Laboratory, the Defense Advanced Research Projects Agency, the National Science Foundation, the National Aeronautics and Space Agency, the Army Corps of Engineers Research Laboratory, and Spectra Research, Inc..

The work described in this paper could not have been done without the energy, enthusiasm and contributions of my colleagues and student research assistants including, in alphabetical order, Hernan Astudillo, Richard Clayton, Srinivas Doddapaneni, Jean-Marc DeBaud, Bijith Moopen, Melody Moore, Phil Seaver, Kurt Stirewalt, and Linda Wills.

I would also like to thank the U. S. Army Tank and Automotive Command for supplying the SL/V software to us, and Lyman Taylor for helping with the code analysis.

References

ACM (1995), *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, ACM.

Arango, G. and R. Prieto-Díaz (1991), "Domain Analysis Concepts and Research Directions," In *Domain Analysis and Software Systems Modeling,* R. Prieto-Díaz and G. Arango, Eds., IEEE Computer Society Press, pp. 9-32.

Arango, G., E. Schoen, and R. Pettengill (1993), "A Process for Consolidating and Reusing Design Knowledge," *15th International Conference on Software Engineering,* IEEE Computer Society Press, Baltimore, Maryland, pp. 231-242.

Arthur, L. J. (1988), *Software Evolution*, John Wiley & Sons.

Batory, D. and S. O'Malley (1992), "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Transactions on Software Engineering and Methodology,* 1, 4, pp. 355-398.

Biggerstaff, T. J. (1989), "Design Recovery for Maintenance and Reuse," *IEEE Computer,* 7 22, 36-49.

Biggerstaff, T. J., B. G. Mitbander, and D. Webster (1994), "Program Understanding and the Concept Assignment Problem," *Communications of the ACM,* 37, 5, 72-83.

Boehm, B. (1981), *Software Engineering Economics,* Prentice Hall.

Borgida, A., R. J. Brachman, D. L. McGuinness and L. A. Resnick (1989), "CLASSIC: A Structural Data Model for Objects," In *Proceedings ACM SIGMOD International Conference on Management of Data.*

Brachman, R., D. McGuinness, P. Patel-Schneider, L. Resnick, and A. Borgida (1990), "Living with CLASSIC: When and How to Use a KL-ONE-Like Language," In *Principles of Semantic Networks,* J. Sowa, Ed., Morgan Kaufmann.

Brooks, R. (1983), "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies,* 18, 543-554.

Caere Corporation (1994), *OmniPage Professional Reference Manual*, Los Gatos, California.

Campbell, R. H. (1974), “The Specification of Process Synchronization by Path-Expressions,” In *Lecture Notes in Computer Science,* Springer-Verlag, 16, 89-102.

Chen, P. P. (1976), “The Entity-Relationship Model–Toward a Unified View of Data,” *ACM Transactions on Database Systems,* 1, 1, pp. 9-36.

Chen, Y. F. and C. V. Ramamoorthy (1986), “The C Information Abstractor,” In *Proceedings COMPASC 86,* IEEE, pp. 291-298.

Chikofsky, E. J. and J. H. Cross II (1990), “Reverse Engineering and Design Recovery: A Taxonomy,” *IEEE Software,* 7, 1, 13-17.

Clayton, R. and S. Rugaber (1993), “The Representation Problem in Reverse Engineering,” In *Proceedings of the First Working Conference on Reverse Engineering*, pp. 8-16.

Clayton, R., S. Rugaber, L. Taylor, and L. Wills (1997), “A Case Study of Domain-based Program Understanding,” In *5th International Workshop on Program Comprehension*, pp. 102-110.

Clayton, R., S. Rugaber, and L. Wills (1998), “Dowsing: A Tools Framework for Domain-Oriented Browsing of Software Artifacts,” In *Proceedings ASE 99,* pp. 204-208.

Clayton, R., S. Rugaber, and L. Wills (1998), “On the Knowledge Required to Understand a Program,” In *The Fifth IEEE Working Conference on Reverse Engineering,* pp. 69-78.

Clayton, R., S. Rugaber, and L. Wills (1997), “Domain Based Design Documentation and Component Reuse and their Application to a System Evolution Record; Final Report,” College of Computing, Georgia Institute of Technology, `http://www.cc.gatech.edu/reverse/dare/final_report/index.html`

Cleaveland, J. C. (1988), “Building Application Generators,” *IEEE Software,* 5, 4, 25-33.

DeBaud, J.-M. (1994), “From Domain Analysis to Object-Oriented Frameworks, A Reuse Oriented Software Engineering Methodology,” Technical Report CIMR TR# 94-04, Center for Information Management Research, Georgia Institute of Technology.

Debaud, J.-M (1996), “Lessons From a Domain-based Reengineering Effort,” In *Proceedings of the Third Working Conference on Reverse Engineering*, pp. 217-226.

DeBaud, J.-M., B. Moopen, and S. Rugaber (1994), “Domain Analysis and Reverse Engineering,” In *Proceedings of the Conference on Software Maintenance,* pp. 326-335.

DeBaud, J.-M. and S. Rugaber (1995), “A Software Re-engineering Method Using Domain Models,” In *International Conference on Software Maintenance*, pp. 204-213.

Defense Modeling and Simulation Office (1999), “High Level Architecture (HLA),” `http://hla.dmso.mil/.`

Devambu, P. T. (1992), “GENOA/GENII - A customizable, language - and front-end - independent code analyzer.” In *Fourteenth International Conference on Software Engineering,* pp. 307-319.

Devanbu, P., R. J. Brachman, P. G. Selfridge, and B. W. Ballard (1991), “LaSSIE: A Knowledge-Based Software Information System,” *Communications of the ACM,* 34, 5, 35-49.

Eidbo, M., M. Ammar, R. Clark, R. Clayton, S. Doddapaneni, R. Dodge, M. McCracken, B. Nguyen, W. Roberts, S. Rogers, and S. Rugaber (1993), “Transitioning to the Open Systems Environment (TRANSOPEN) Final Report,” Technical Report CIMR - 93-01, Center for Information Management Research, Georgia Institute of Technology.

Fjeldstad, R. K. and W. T. Hamlen (1983), “Application Program Maintenance Study: Report to Our Respondents.” In *Proceedings GUIDE 48,* Philadelphia, Pennsylvania, *Tutorial on Software Maintenance,* G. Parikh and N. Zvegintozov, Eds., IEEE Computer Society.

Forsythe, G., M. Malcolm, and M. Moler (1977), *Computer Methods for Mathematical Computations,* Prentice Hall, 161-166.

Garlan, D. and M. Shaw (1995), *Software Architecture: Perspectives on an Emerging Discipline,* Prentice Hall.

Grass, J. E. and Y.-F. Chen (1990), “The C++ Information Abstractor,” In *1990 USENIX Conference,* pp. 265- 277.

Harris, D., H. B. Reubenstein, and A. S. Yeh (1995), “Recognizers for Extracting Architectural Features from Source Code,” In *Second Working Conference on Reverse Engineering,* L. Wills, P. Newcomb, and E. Chikofsky, Eds, IEEE Computer Society Press, pp. 252-261.

Hildreth, H. (1994), “Reverse Engineering Requirements for Process-Control Software,” In *Proceedings of the Conference on Software Maintenance,* pp. 316-325.

Johnson, R. E. and B. Foote (1988), “Designing Reusable Classes,” *Journal of Object-Oriented Programming,* 1, 2, 22-35.

Johnson, W. L. and A. Erdem (1997), “Interactive Explanation of Software Systems,” In *Automated Software Engineering,* 2, 1, 53-75.

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

Jullig, R., Y. V. Srinivas, L. Blaine, L.-M. Gilham, A. Goldberg, C. Green, J. McDonald, and R. Waldinger (1995), *Specware Languages Manual, Version 1.1,* Kestrel Institute.

Loral Federal Systems - Owego (1999), “DSSA - Domain-Specific Software Architectures (DSSA),” Owego, New York, `http://www.owego.com/dssa/foils/dssa_foils.ps.`

Lowry, M., A. Philpot, T. Pressburger, and I. Underwood (1994), “Amphion: Specification-based Programming for Scientific Subroutine Libraries,” In *SAIRAS’94*.

MacDougall, M.H. (1987), *Simulating Computer Systems: Techniques and Tools.* The MIT Press.

Moore, M. (1996), “Rule-Based Detection for Reverse Engineering User Interfaces,” In *Proceedings of the Third Working Conference on Reverse Engineering,* IEEE Computer Society Press, pp. 42-48.

Moore, M. and S. Rugaber (1997), “Using a Knowledge Representation for Understanding Interactive Systems,” In *Proceedings of the International Workshop on Program Comprehension*, pp. 60-67.

Moore, M., and S. Rugaber (1997), “Domain Analysis for Transformational Reuse,” In *Proceedings of the Fourth Working Conference on Reverse Engineering*, IEEE Computer Society pp. 156-163.

Moore, M., S. Rugaber, and H. Astudillo (1993), “Knowledge Worker Platform Analysis Final Report,” Technical Report CIMR - 93-02, Center for Information Management Research, College of Computing, Georgia Institute of Technology.

Moore, M., S. Rugaber, and P. Seaver (1994), “Knowledge-based User Interface Migration,” In *Proceedings of the 1994 International Conference on Software Maintenance,* pp. 72-79.

Murphy, G. C., D. Notkin, and K. Sullivan (1995), “Software Reflexion Models: Bridging the Gap Between Source and High-Level Models,” In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering,* ACM, pp. 18-28.

MySQL (1999), `http://www.tcs.se.`

Neighbors, J. (1980), *Software Construction from Components,* PhD Dissertation, ICS Department, University of California at Irvine.

Neighbors, J. M. (1989), “Draco: A Method for Engineering Reusable Software Components,” In *Software Reusability / Concepts and Models, volume 1*, T. J. Biggerstaff and A. J. Perlis, Eds., Addison Wesley.

Ousterhout, J. K. (1994), *Tcl and Tk Toolkit,* Addison Wesley.

Overton, R. K. et al. (1971), “A Study of the Fundamental Factors Underlying Software Maintenance Problems: Final Report,” Corporation for Information Systems Research and Development.

Prieto-Díaz, R. (1989), “Classification of Reusable Modules,” In *Software Reusability / Concepts and Models, volume 1,* T. J. Biggerstaff and A. J. Perlis, Eds., Addison Wesley, pp. 99-123.

Prieto-Díaz, R. (1991), “Domain Analysis for Reusability,” In *Domain Analysis and Software Systems Modeling,* R. Prieto-Díaz and G. Arango, Eds., IEEE Computer Society Press, pp. 63-69.

Prieto-Díaz, R. and G. Arango (1991), *Domain Analysis and Software Systems Modeling,* IEEE Computer Society Press, Los Alamitos, California.

Quilici, A. and D. N. Chin (1995). “DECODE: A Cooperative Environment for Reverse-Engineering Legacy Software,” In *Second Working Conference on Reverse Engineering,* L. Wills, P. Newcomb, and E. Chikofsky, Eds., IEEE Computer Society Press, pp. 156-165.

Reasoning Systems Incorporated (1990), S*oftware Refinery Toolkit*, Palo Alto, California.

Resnick, L. A. et al. (1993), *CLASSIC Description and Reference Manual for the Common LISP Implementation Version 2.1,* AT&T Bell Labs, Murray Hill, N.J.

Rugaber, S. (1996), “Program Understanding,” In *Encyclopedia of Computer Science and Technology, Supplement 20,* 35, A. Kent and J. G. Williams, Eds., Marcel Dekker, pp. 341-368.

Rugaber, S. (1997), “An Example of Program Understanding,” Technical Report GIT-CC-98-14, College of Computing, Georgia Institute of Technology.

Rugaber, S., S. B. Ornburn, and R. J. LeBlanc, Jr. (1990), “Recognizing Design Decisions in Programs,” *IEEE Software*, 7, 1, 46-54.

Rugaber, S., K. Stirewalt, and L. Wills (1995), “Detecting Interleaving,” In *International Conference on Software Maintenance,* pp. 265-274.

Rugaber, S., K. Stirewalt, and L. Wills (1995), “The Detection and Extraction of Interleaving Code Segments,” Technical Report GIT-CC-95-49, College of Computing, Georgia Institute of Technology.

Rugaber, S., K. Stirewalt and L. Wills (1996), “Understanding Interleaved Code,” *Automated Software Engineering*, 1-2, 3, 47-76.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen (1991), *Object-Oriented Modeling and Design,* Prentice-Hall.

Soloway, E., J. Pinto, S. Letovsky, D. Littman, and R. Lampert (1988), “Designing Documentation to Compensate for Delocalized Plans.” *Communications of the ACM,* 31, 11, 1259-1267.

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

Srinivas, Y. V. (1991), “Algebraic Specification of Domains,” In *Domain Analysis and Software Systems Modeling*, R. Prieto-Díaz and G. Arango, Eds., IEEE Computer Society Press, pp. 90-124.

Srinivas, Y. V. (1991), *Pattern Matching: A Sheaf-Theoretic Approach*, PhD Dissertation, Department of Information and Computer Science, University of California at Irvine.

Sun Microsystems (1994), *Browsing Source Code*.

Yeh, A., D. Harris, and H. Reubenstein (1995), “Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language,” In *Proceedings of the Second Working Conference on Reverse Engineering*, pp. 227-236.

Zeigler, B.P. (1976), *Theory of Modeling and Simulation*, John Wiley and Sons.