Dowsing: Domain-based Program Understanding

Richard Clayton, Spencer Rugaber, and Linda Wills[1]
Georgia Institute of Technology

Abstract

Program understanding relates a computer program to the goals and requirements it is designed to accomplish. Understanding techniques that rely only on source code analysis are limited in their ability to derive this relationship. Application-domain analysis is another source of information that can aid program understanding by guiding the source analysis and providing structure to its results. This paper describes the application of a domain-based program understanding process, Synchronized Refinement, to the problem of reverse engineering two intermediate-size software systems: the Mosaic World Wide Web browser and a generic software loader/verifier being developed by the U.S. Army. It discusses the domain analyses undertaken, corresponding source code analyses, and architecture-based techniques for bridging the results of these analyses. It also describes tool support for domain-based navigation and browsing of the program, its associated documents, and the results of the analysis.

Keywords: *reverse engineering, domain analysis, Mosaic, program understanding, software tools*

I. Introduction—Domain Analysis and Reverse Engineering

Program understanding is a crucial part of software development and maintenance, yet it is a largely manual activity that begs for tool support. Some simple tools, such as structure chart generators and cross referencers, do exist, but more can be done. In particular, most existing tools are derivatives of the kinds of language-based analyses that compilers perform. Significant progress in automated program understanding must also use knowledge of the application domain of the program being understood [35][12]. The Domain Analysis and Reverse Engineering (DARE) project at the Georgia Institute of Technology is exploring how best to use domain knowledge to understand programs.

This paper describes a process, Synchronized Refinement [11], for domain-based program understanding (Section II) and a tool suite to support the process (Section III). Tools that have already been built are described (in Section IV) in the context of two software systems—the Mosaic World Wide Web browser [24] and a U.S. Army software loader/verifier (SL/V) for embedded systems in the M1A2 Abrams tank [37]. Mosaic consists of approximately 100KLOC, is written in the C programming language, spans multiple versions, is publicly available, and implements a widely-understood and straight-forward set of requirements. An SL/V is used to download new software into an embedded system and to verify the software's integrity after being downloaded. The SL/V we examined consists of approximately 20KLOC, is written in the Ada programming language, and is part of an existing effort in domain-based software development by the U.S. Army against which our analyses have been compared. Our examination of these two systems raises several issues which we discuss (in Section V), including the role of software architecture in relating code to domain models, the choice of logical representation for the results of the various analyses, and how domain-based

1. Point of contact: linda.wills@ece.gatech.edu.

tools can to be integrated together. Related research in using domain and architectural knowledge to inform program understanding is summarized (in Section VI).

II. Synchronized Refinement—Domain-based Reverse Engineering

Synchronized Refinement (SR) is a technique for reverse engineering software to support documentation, reengineering, or reuse [11]. Reverse engineering produces a high-level representation of a software system from a low-level one, typically the program itself [10]. Synchronized Refinement constructs such a representation of a system from its source code. It proceeds by detecting instances of design decisions in source code and relating the detected decisions to the corresponding element of the application-domain description [34]. Figure 1 describes the flow of data through the various activities that constitute SR.

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 comprises three parts: an elaborated and instantiated domain description (*Application description*), an abstracted program description (*Abstract program description*), and a description of how the code realizes the application (*Mapping/annotations*).

The SR process comprises two parallel activities, roughly corresponding to the top and bottom halves of Figure 1: analysis of the program (on the bottom) and synthesis of the application-domain description (on the top). Analysis consists of detecting the implementation of domain and programming concepts in the code and replacing them by abstracted versions. In this way, the program description becomes smaller and more abstract. Synthesis begins with the domain description expressed as an object-oriented framework [20] (abstract, object-oriented classes). These are instantiated as more is learned from the source code. The two halves of the Figure 1 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 code analysis proceeds, implementation constructs are detected that relate to an expectation. An annotation for each detected construct is recorded together with the relevant expectation, specifying the type of the design decision motivating the choice of construct and the corresponding sections of the program. During the SR process, certain expectations are confirmed and others refuted. A confirmed expectation raises other expectations of the code. Gradually, a hierarchical description of the structure of the program emerges. As the program source code shrinks, the domain description expands.

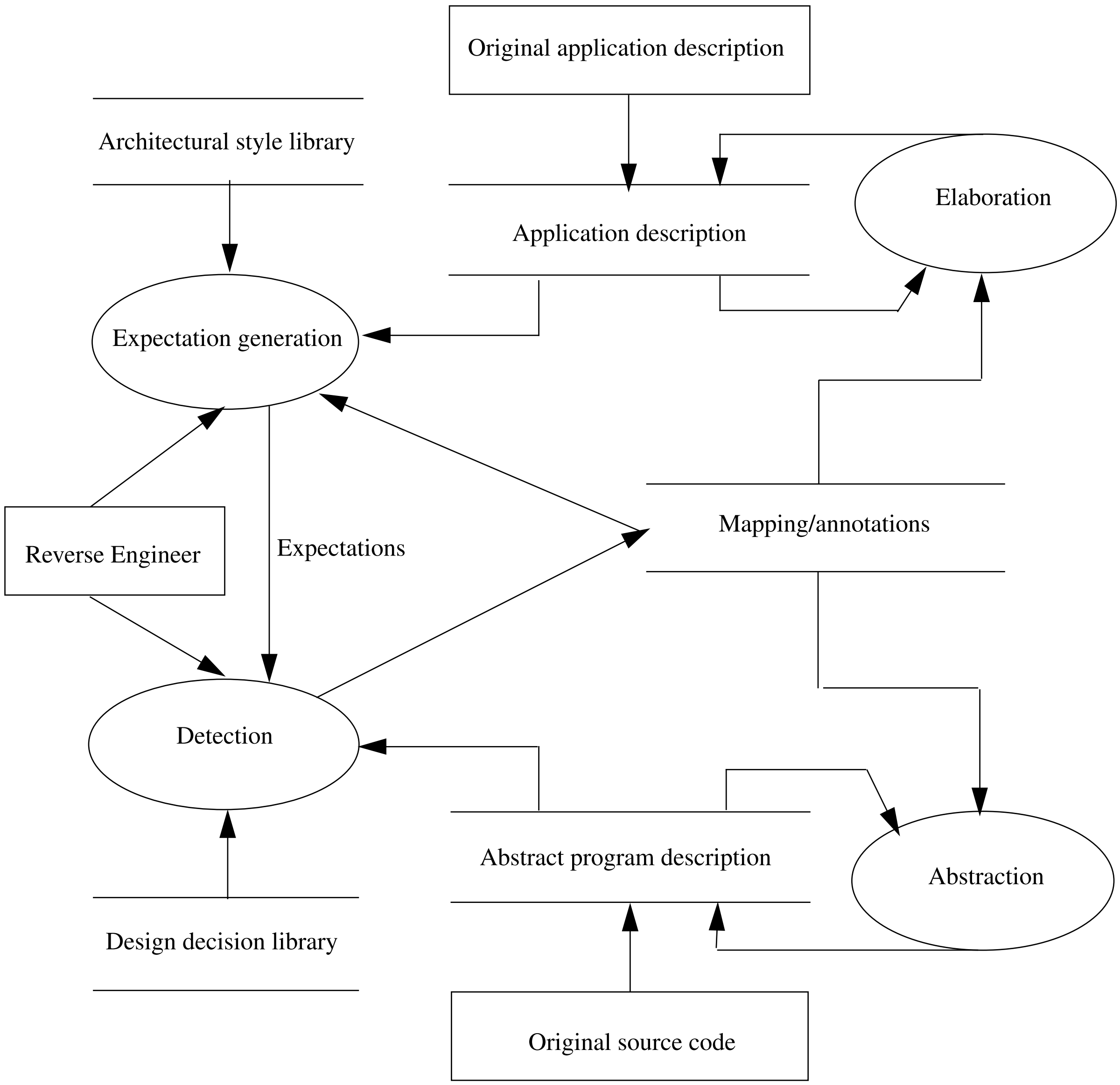

Figure 1: Data Flow Diagram for Synchronized Refinement

III. Domain-based Program Understanding Tools

We call the use of domain-based tools to explore and understand programs "dowsing"[1] to distinguish it from traditional program browsing which either explores the program in terms of its underlying

1. To "dowse" is defined in *Webster's Third New International Dictionary* as "to seek something with meticulous care especially with the aid of a mechanical device".

lines of text or makes use of lexical and syntactic analyses similar to those performed in the early phases of program compilation.

We see a dowser as being the user-visible part of a tool suite whose core is a model of a program's application domain. Figure 2 provides a conceptual architecture for a tool suite to support dowsing. For some of these tools, we have built prototypes that are described in this paper; others are commercially available; and others we have not yet begun to build.

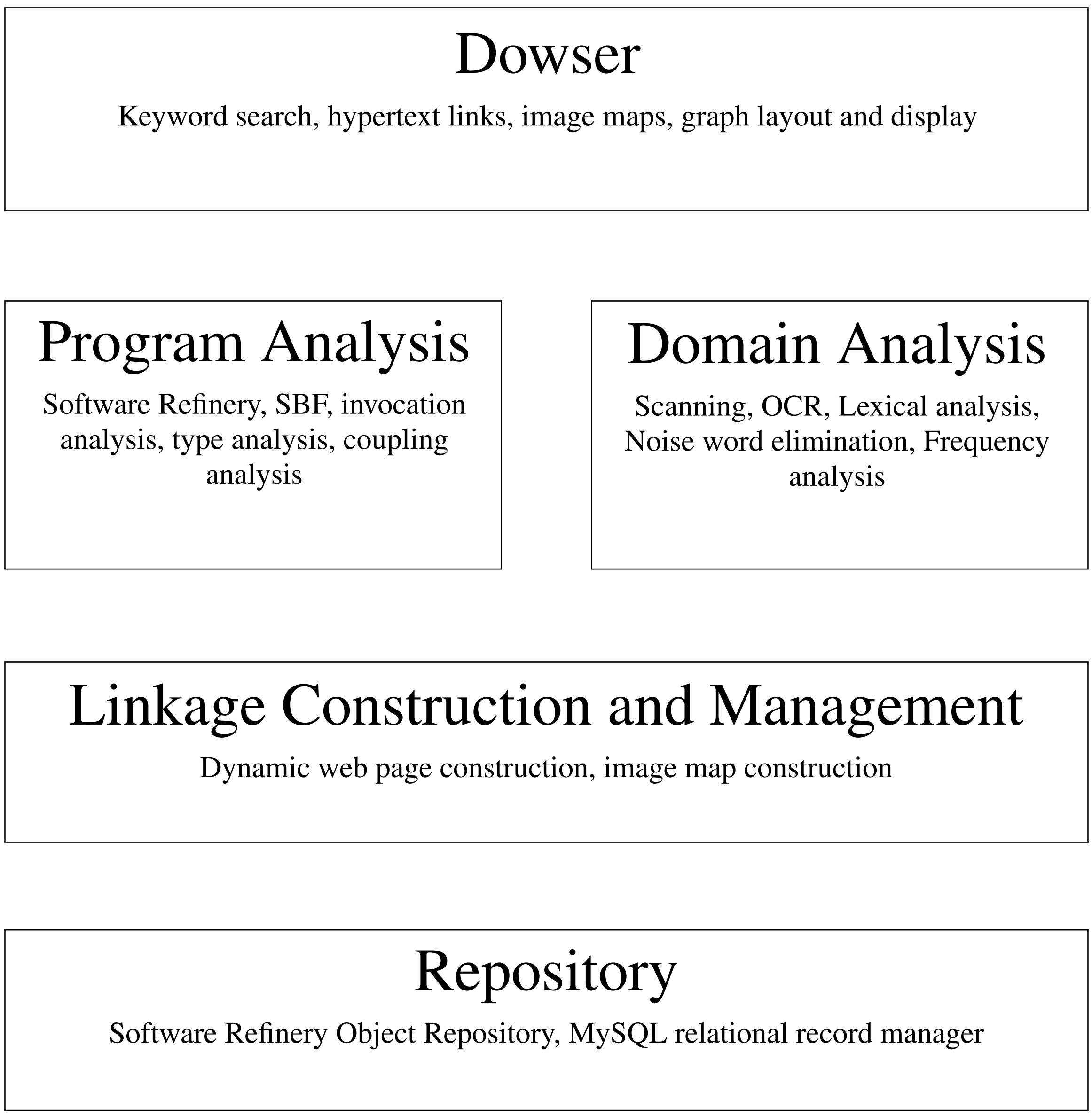

Figure 2: Domain-based Program Understanding
Layered Tools Architecture

There are several components in the tool suite.

- the dowser itself, with which the software maintainer explores a subject system and adds insights to the underlying models;
- code-based tools for analyzing the program;
- domain-based tools for constructing a model of the application domain;
- tools for constructing linkages between the domain and code models and constructing presentations (both graphical and textual) of them;
- tools for maintaining the conceptual and logical models constructed by the previous three types of tools.

A. Dowsing

A software maintainer dowses a software system using a model of the program's application domain. The model gives 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 code and documents via domain concepts. Such access can take the form of keyword searches, Web-page links, or image maps. In addition to giving access to the underlying information, the dowser must provide the programmer with the ability to annotate the corpus of models, code, and documents, to instantiate the models with refined concepts, and to modify the models when difficulties are detected.

B. Program Analysis

Code analysis in DARE consists of two parts: traditional, language-based analyses provided by commercial tools, and value-added research prototypes that relate the program-based information to higher-level concepts, primarily architectural views of how the code realizes non-functional requirements such as performance or security. For the first part, we have used two commercial tools, the Software Refinery tool suite (Refinery) [33] and the Source Browser Facility (SBF) [40] that is provided by SUN Microsystems with its C and C++ compilers. For the second part, three types of relations are constructed: how components interact by invoking each other, how data types are defined in terms of each other, and how modules are coupled to each other. In addition, we have made use of the **dot** graph visualization tool from AT&T [23] to display the resulting analyses graphically.

C. Domain Analysis

For the purposes of this study, we assume that domain knowledge is encoded in 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 completely automatically, we have developed or made use of tools for scanning in text and converting it to ASCII characters [30], breaking it into words and filtering out uninteresting words, and computing word frequencies.

Together, these provide the analyst with an initial approximation of the most important concepts in the domain.

In support of manually constructing a domain model from lexical information, in previous work [28], we have described how a natural language parser can analyze high-level system descriptions to generate a domain model. In that work, we suggested using the analysis to help validate an existing model rather than as a result of its own. Whether or not the natural language parsing is used, the resulting domain model must be encoded in the computer in a form that can be used in subsequent analyses. We have used the Refine language and Dialect tool [14] that are part of the Software Refinery tool suite [33] as the vehicle for performing the encoding.

D. Linkage Tools

Code models and domain models must be linked to enable effective domain-based program understanding. Direct linkages are difficult to make because of the conceptual distance between the code and the domain models. Instead, we propose to use software architecture as a stepping stone between the two.

For example, imagine the following situation related to the Mosaic software. A software maintainer trying to understand the Mosaic source code comes across a significant amount of code related to Mosaic's page caches. But there is no mention of caching in the domain model 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 Mosaic that views Mosaic's modules in terms of how they support performance will feature the cache management software as a major component. The caching code can be related to the cache management component, which exists to satisfy a performance requirement from the application domain.

To construct an architectural stepping stone requires that programming knowledge (such as how to provide increased performance using caching) be encoded and available. For the work we are describing in this paper, 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 need to be constructed are dynamically generated web pages and image maps.

E. Repository Tools

The final type of tools provide a way for entering and managing the conceptual and logical models generated by the other tools. We have explored two such approaches, the Refinery's object-oriented repository [33] and a public-domain relational record-management tool called MySQL [27]. The former has the natural advantages of being congruent to the domain model and integrated with one of the code analysis tools. The latter has the advantages of the relational algebra and a smaller conceptual footprint. Together these tools can provide a powerful lever to the programmer trying to understand a software system. The next section describes the tools in further detail.

IV. Current Tool Suite and Results

We are employing a number of tools in support of the domain analysis and program analysis steps of Synchronized Refinement. We have also developed an initial prototype hypermedia tool to support dowsing and dynamic linkages among documents. This section describes these tools and the results we have obtained so far.

A. Domain analysis tools

Application-domain modeling [31] constructs a model of a problem's application domain. The nature of the model depends on its intended use. Synchronized Refinement uses the application-domain model as a high-level guide for program understanding. For Mosaic, the application-domain model starts 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 source code analyses.

We create an application-domain model for the World Wide Web in three steps. First, various descriptions of the World Wide Web are analyzed to extract relevant information about the domain. Second, the extracted information is organized into a coherent application-domain model by applying the Object Modeling Technique (OMT) [36]. Third, the constructed OMT Object Model is instantiated in a machine-manipulable form using the Software Refinery tool set. This section describes details of this modeling process and the tools we have developed to support it.

The first step in creating the application-domain model is to extract information from descriptions of the World Wide Web. The best descriptions are specifications, which should be authoritative and fairly complete. Unfortunately, the existing specifications for the World Wide Web are fragmented and concentrate on lower level details such as protocols (e.g. HTTP [18]) and fine-grained semantic definitions (e.g. style sheets [39]). Higher-level specifications, when they exist, are usually given in unhelpful forms, such as binary-only reference implementations (e.g. Arena [3]). The alternative to formal specifications are informal descriptions; there are many such informal descriptions of the World Wide Web available both on and off the Web. For Mosaic, we chose Chapter 1 from [6].

With World Wide Web descriptions in hand, object analysis can begin. Object analysis is broken into two parts: word-frequency analysis and object extraction. Frequency analysis is a first pass over the descriptions to determine the important words. After eliminating noise words such as “the”, the remaining words are counted; more frequently occurring words are assumed to be more important than less frequently occurring words. To get a feel for the larger structure of the descriptions, we also looked at digram (e.g.

“server sends”) and trigram counts. Table 1 shows frequency analysis results for Mosaic.

We then performed a noun-and-verb analysis on the World Wide Web description to extract possible object classes and associations. This step was suggested by the original approach to object-oriented analysis proposed by Abbot [1].

Word	Count
server	61
script	36
browser	24
client	23
user	14
information	12
scripts	11
output	11

Table 1: Word frequency analysis results

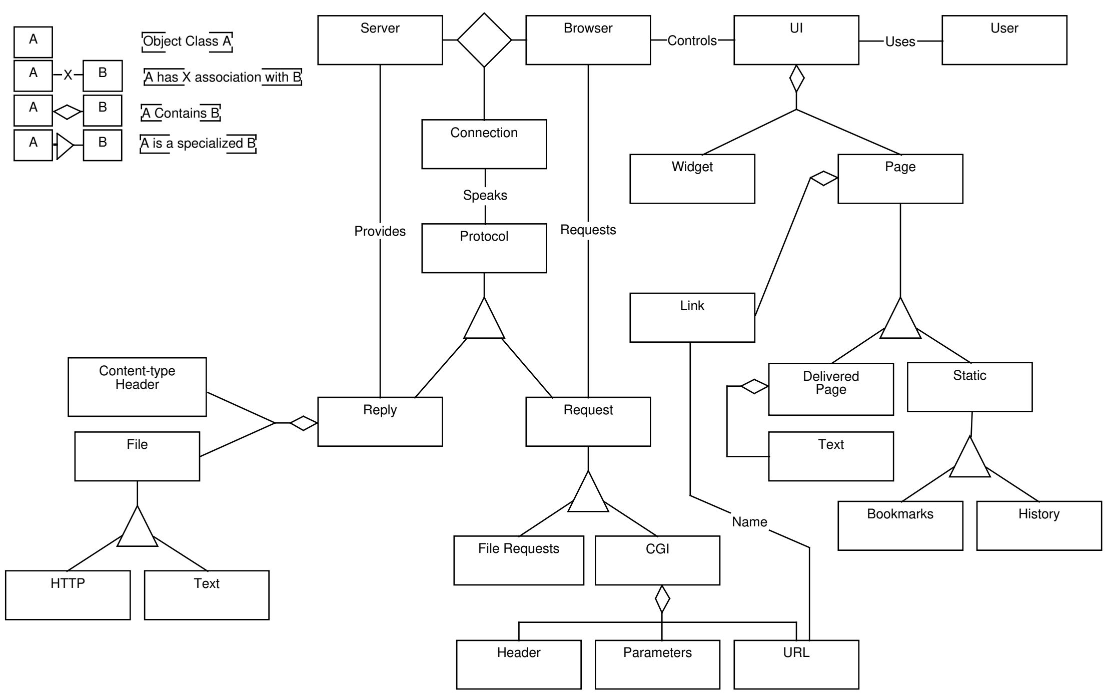

Figure 3: OMT Object Model Diagram for the Web Browser Domain Model

The second step in creating the application-domain model is to take the objects and associations found in the first step and organize them into a coherent model. We chose to organize around the OMT because of its familiarity in the software engineering community. OMT includes several graphical notations for specifying models. Its Object Model expresses the important domain classes and the inter-class associations. We derived 25 objects and 6 associations using the lexical analysis described above. This resulted in the OMT Object Model for the Web-browser domain model shown in Figure 3.

The third step in creating an application-domain model is to realize the OMT Object Model of the World Wide Web in a machine-accessible form. There are a number of motivations for this step. The most important one is that an object-oriented framework is a powerful and useful representation for application-domain information in reverse engineering. It not only provides an organized collection of expectations, but, by instantiating the framework's abstract classes, it serves as a natural place to store the results of the code analyses. Other motivations include the ability to automate various parts of model construction and validation and the flexibility with which the model can be handled once placed into machine-readable form.

We chose the Software Refinery [33] as our base modeling tool. The Refinery is a large, Lisp-based program suite oriented towards program analysis and transformation. The Refinery is well suited to supporting Synchronized Refinement because it can be used to represent both the application domain and the code under analysis. The World Wide Web Object Model shown in Figure 2 is represented using the Object Base, the Software Refinery subsystem responsible for holding the results of program analysis. In addition, we use its Dialect [14] tool to build parsers that translate between Refinery internal and external textual representations of the Object Model.

Our Software Refinery representation has two parts: an internal representation and a parser for translating between an external representation and the internal representation. We consider each part in turn.

An OMT Object Model is represented as a complete binary tree of three nodes. The root represents the whole model; one of the children represents the set of all classes appearing in the model; the other child represents the set of all associations used in the model. Annotations among the two child nodes describe Object Model structure.

The second part of our Software Refinery representation is a parser for translating from an external, textual representation of an OMT Object Model to the internal representation developed in the previous paragraphs. The external representation is textual, using a simple, line-oriented keyword-value structure. For example, a object-model class is the keyword "`class`" followed by a string giving the class name:

```
class "a"
```

The external representation of an object-model association is

```
association "x" ("a" ; "b" ; "c")
```

The string “x” is the association's name; the other strings are the names of the object-model classes that are part of the association. Other features of the OMT Object Model are described similarly.

The Dialect rules for describing grammars are similar to those used in other parser generators; for example, the Dialect rule for recognizing an association is

```
omt-om-association ::=
   [ "association" omt-om-name "(" omt-om-associates + ";" ")" ]
   builds omt-om-association
```

Rules may use extended regular expressions; in this case, one or more `omt-om-associates` may appear separated by a semi-colon. Refine objects may appear as non-terminals (e.g., `omt-om-association` and `omt-om-name`). When Dialect recognizes a non-terminal object, it will construct the object using the components appearing on the right-hand side of the rule.

B. Program analysis tools

High-level models are useful when describing and comprehending complex systems such as programs. A model's utility comes from its abstracting away the details of the system it describes. In contrast, the source code of a program is full of such details. A rough analogy between the high-level models of an application program (in terms of a domain model, design decisions, and a prototypical architectural model) and the architecture of buildings is the difference between the blueprint of the classical Frank Lloyd Wright Prairie House [22] and a particular instance of it. For example, there may be differences in the instantiation that are derived from pragmatic concerns due to the site peculiarities.

Synchronized Refinement matches source code to an application-domain model. Through code analysis, a code abstraction is developed that is resolved with the expectations generated by the models. This is an iterative process reflected by the dataflow cycle in Figure 1 from the *Abstraction* process to *Abstract program description* to the *Detection process* to the *Mappings and annotations* and finally back to the *Abstraction process*. Additionally, the high-level model is also enhanced to fit the source code. In reference to the building architecture analogy, SR also constructs a blueprint of a Prairie House incorporating the peculiarities of this instantiation (i.e., the deviations from the classical).

As part of our study of Mosaic, we performed a variety of code analyses using the Source Browser Facility (SBF) that is provided by SUN Microsystems with its C and C++ compilers. This facility builds a procedural-level model of the code from which we could access raw data necessary for code analysis. After implementing a few analysis programs by hand using the Icon programming language [16], we decided to store our raw data in a relational database and perform our analyses using SQL queries. SQL allowed us to quickly and easily formulate code analyses with a minimum of programming, but with a cost in naturalness in expression. This section presents the results of three primary analyses we performed: invocation analysis, type analysis, and coupling analysis.

Invocation analysis: One way to construct an initial architectural model is to analyze invocations. Invocation (or call graph) analysis determines which subprograms invoke others in a program. For some languages this may prove problematical. In C, for example, a subprogram may be invoked indirectly through a function pointer [26]. In polymorphic languages, determination of the most appropriate version of a method may not be known until runtime. Even with these limitations, invocation analysis can still be useful in generating abstract program representations.

For example, if subprograms are grouped into modules, then the invocation between subprograms in different modules can be used to represent communication between the modules. Also, if a comparison between intra- and inter-module invocations is developed, module coherence [38] can be evaluated. If the modules denote architectural components, intercomponent communication is represented. Finally, from a domain-model perspective, the inter-object communication serves to indicate associations among objects.

A relational analysis of invocation patterns was easy to perform, largely because the SBF provided who-calls-whom information. Controlling the morass of invocation information that can result from analyzing a software system the size of Mosaic is important, so the who-calls-whom information was augmented with information indicating the files in which subprograms were declared and the subdirectories in which files were located. The location information made it easy to specialize invocation analysis to consider, for example, only those files under the **src** subdirectory with "gui" in their name, as illustrated in Figures 4 and 4. In Figure 3, the nodes represent subprograms that are constrained to be in files with "gui" in their name (e.g., **gui.c**); source nodes represent callers and destination nodes represent callees. The graph in Figure 5 aggregates the call-graph information further by coalescing nodes (that is, subprograms) into their defining files. The command that produced each graph is shown in the caption. The **cgq** (call-graph query) command is a shell script which issues SQL queries to the relational database to construct the graph. The graph table is dumped to an external file, which is massaged by an **awk** [2] script and fed to the **dot** graph drawing program [23], producing the graphs shown.

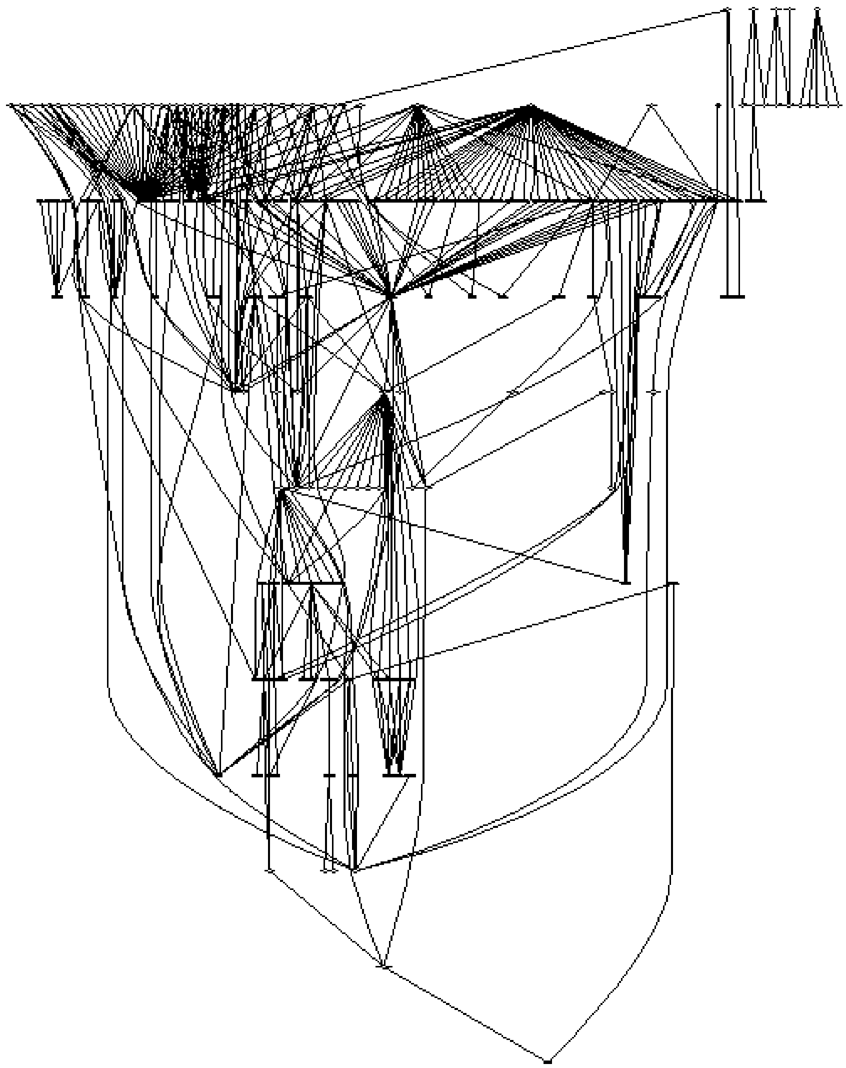

Figure 4: An example call-graph produced by the command: `cgq -nf 'sf=\%gui\% df=\%gui\%'`

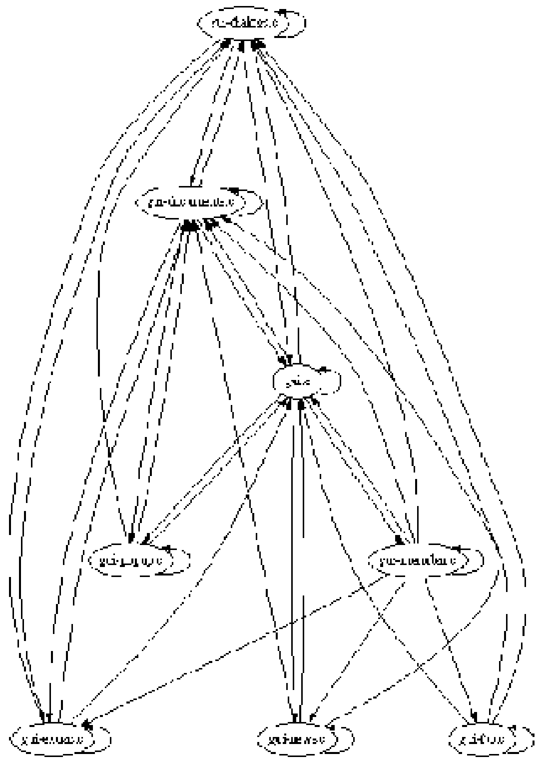

Figure 5: A call-graph produced by: `cgq -nl f -nf 'sf=\%gui\% df=\%gui\%'`

Type analysis: Analysis of source code data types may be used to develop an object-oriented representation of the code. This may be done even if the source language does not natively support object-oriented constructs [7]. *Part-of* associations (aggregation data structures) may be derived from analysis of the hierarchical structure of the types. This may be supplemented by invocation analysis. For example, a datatype's subcomponent may refer to a collection. If a "member" subprogram associated with the datatype retrieves elements from the collection, that would indicate a *part-of* association. This deduction can be made if the selector interface to the object (or abstract datatype) in question returns the elemental type of the collection instead of the collection itself and there are no selectors which do return the collection. In this case the elements of the collection have a relationship to the object. The collection is a manifestation of this relationship's many-to-one property. This analysis involves observing the subprograms' formal arguments and return types and associating subprograms to the abstraction interface of one particular type. Additionally, datatypes serving as collections have to be identified.

Type analysis is the most difficult of the three analyses, largely because the information provided by SBF is fragmented into lexical units and has to be re-assembled into higher-level syntactic and semantic constructs reflecting the program's type structure. This difficulty is not predominantly inherent to the representation analysis types, but the relational implementation of this analysis was less straightforward than was the Icon analysis.

Coupling: The amount of communication between modules can be used as a measure of the strength of the association between them. A measure of the coupling [38] between two modules or objects is indicated by the strength of this association [5]. As noted earlier, invocation analysis can be used to represent intermodule communication. Additionally, access and modification of object-state information is also a measure of the strength of the communication. If a particular program uses abstraction, this access may be represented by a subprogram's invocation. Or it may be represented by a slot or field access into a structure or object.

For Mosaic, we interpreted coupling as common, global-variable access; two subprograms accessing the same global are considered coupled. Coupling analysis via relational modeling was fairly simple; the relational tables contained all the information needed, but inconveniently spread out over several tables. These representational problems, however, were easily fixed by defining a few temporary tables to conveniently coalesce the necessary information.

Determining the degree of coupling within a relational framework is somewhat complicated, particularly when the calculations involve more than simple counting and arithmetic. Fortunately, this problem is easily solved by either adding external subprograms to the SQL intepreter or dumping the data to a file and using an external analysis program.

Our current coupling analysis tool is a shell script that queries the database and produces an external text file; the script does only data gathering, leaving further coupling analysis to external programs. The script, called `global-q`, accepts a list of fields; each row in the output file has values in left-to-right order matching the fields given on the `global-q` command line. For example, the command

```
global-q -d global dir
```

produces an output file, each row of which has two values, the first of which is the name of a global variable and the second of which is the name of the directory in which the global variable is accessed (the `-d` option indicates the output table should be stripped of duplicate rows). Sorting the output file lets us easily determine which global variables are used across directories.

C. Interactive exploration using dynamically generated hypermedia

The terminology established by specification and design documents is often carried over into later portions of the project; these early-occurring documents often set the domain of discourse for the project.

For example, when the specification requires high performance for repeat queries and the design document suggests caching as a technique that avoids duplicating effort, it seems likely that the program text will contain the words “performance” or “efficiency” and “cache” in comments (e.g., “for efficiency, cache the query and its response to avoid resending the same query”) and code (e.g., the routine named `InitQueryCache()`).

Conversely, words appearing in the program text may have come about due to their appearance in preliminary project documents, and understanding the context in which these words occur in preliminary documents may help explain their appearance in the program text. A maintainer examining the program may wonder why there are many routines built on the phrase “QueryCache.” By searching these early project documents, the maintainer may be able to gather insight into why “QueryCache” appears to be important to the program.

As an initial step toward a dowser and the associated linkage tools that connect the domain and code models, we have built a prototype hypermedia tool that dynamically generates web pages linking the evolving domain model, code analyses, and existing documentation, supports keyword querying, and provides annotated image maps relating common terminology across these models and documents. This hypermedia tool performs lexical analyses on the documents associated with software and on the source code itself, creating a concordance of the documents. It then answers keyword queries with visual information on the frequency and relative location of the keywords across the documents. By giving keyword search requests, the maintainer can examine document pages according to the words they contain, giving the maintainer a focused and oriented view into the documents.

Figure 6 shows the keyword searcher’s query page. The page accepts a keyword query. In the example shown, the maintainer is asking to see all documents currently associated with the SL/V software system that contain the words “software” and “download.”

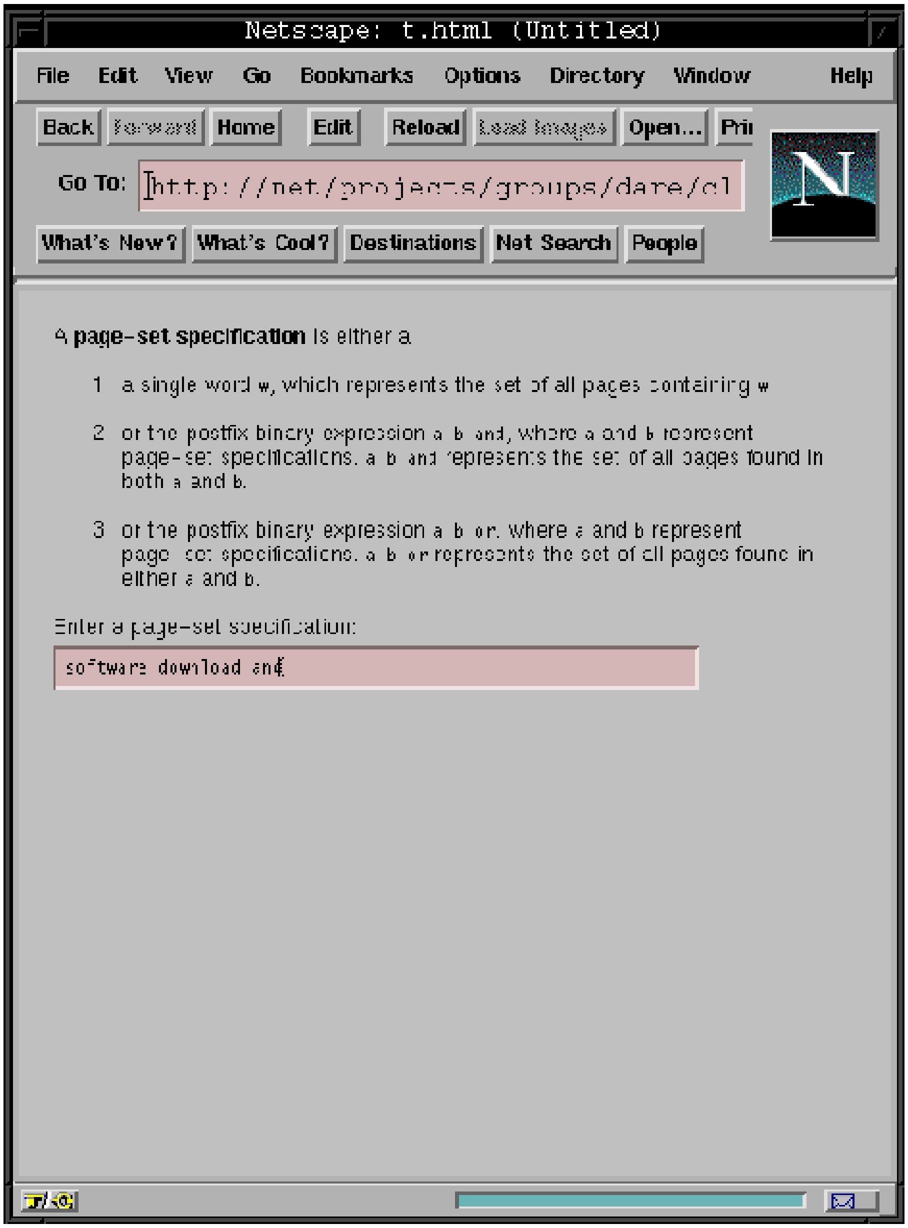

Figure 6: Keyword search query page

The keyword query is parsed by a CGI script and applied to the cross-document concordance. The result is a set of pages containing the words matching the keyword query. These pages are then presented in a response page of “thumbnail” representations of the retrieved pages as shown in Figure 7. Each keyword on the thumbnail pages is color coded (“software” is green and “download” is red in this example) to provide a visual depiction of the pages’ overall match density.

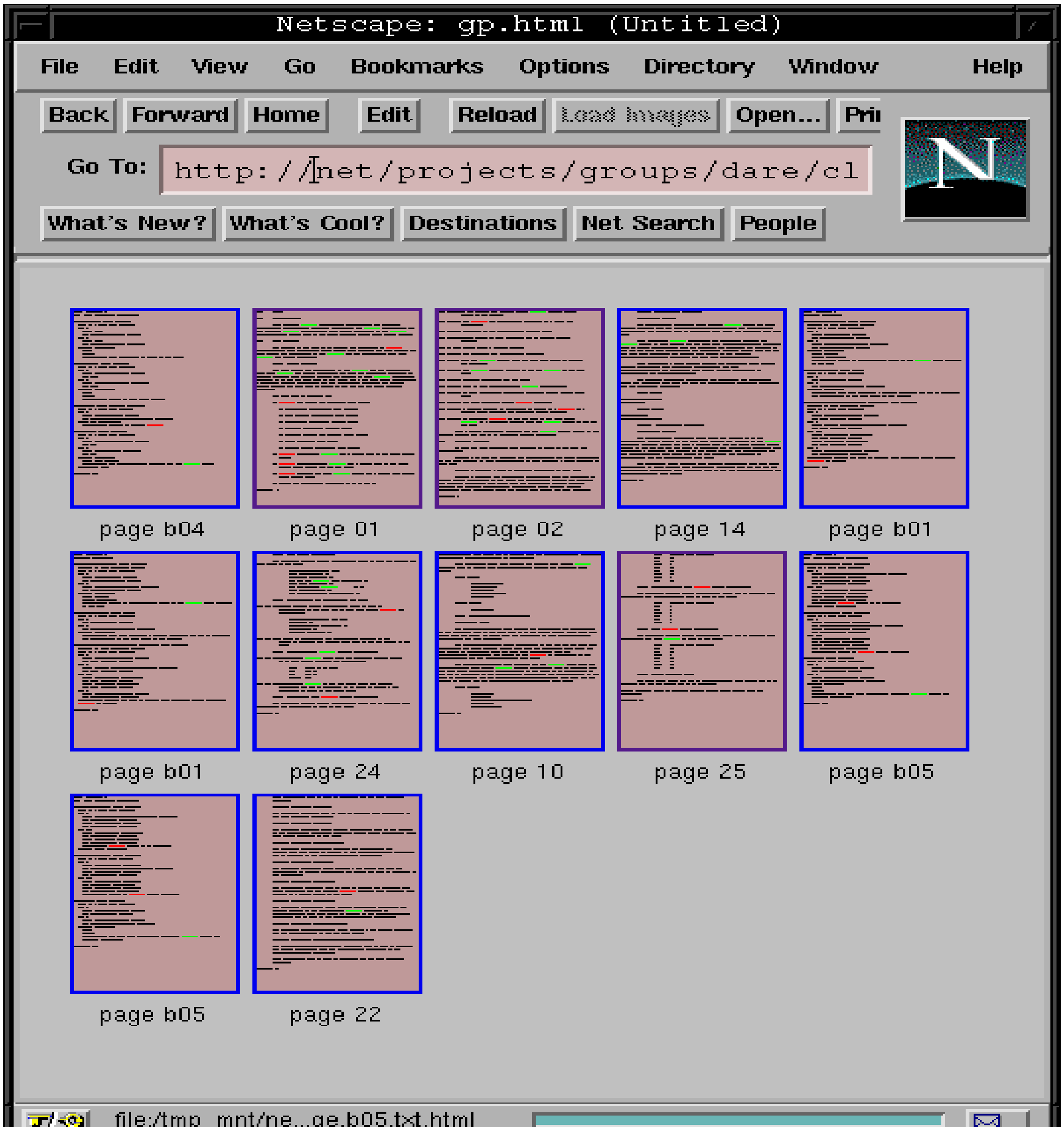

Figure 7: Thumbnail presentation of matching pages

Clicking on a thumbnail page retrieves the associated full-sized pages with matching terms highlighted. By using the browser's forward and backward buttons, as well as by spawning new pages, the maintainer can create a detailed set of connections between the terminology appearing in program text and in preliminary project documents.

V. Open Issues in Linking and Managing Models

An important area of active research in the DARE project is supporting the linking and management of domain and code models as they evolve during Synchronized Refinement. This section discusses the issues under investigation in developing linkage and repository tool support.

A. Architectural Stepping Stone Between Domain and Program Models

The key to domain-driven program understanding is bridging the gap between the conceptual domain model (the application description) and the results of code analysis. We believe an architectural abstraction of the program can bridge this gap. This raises three primary issues:

- What is the relation between the application-domain model and the architecture?
- How can architectural styles be connected to program descriptions resulting from standard code analyses?
- What annotation mechanisms should be used to make these relationships explicit and allow them to evolve as the domain model is refined and program descriptions emerge?

1. Application-domain models and architectures

In mature domains, application programs have a typical set of nonfunctional properties, such as performance, cost, reliability, and security. The primary agents interacting with these programs expect certain sets of behaviors and functions to be provided by the programs. Programs within an application domain make use of architectural mechanisms that implement these properties, behaviors, and functionality; for example, caching, password protocols, and priority schemes. The issue in connecting the application-domain model to the architecture is one of identifying these architectural mechanisms. What are the typical architectural components related to the primary agents of the domain, what are their behaviors, and which protocols do they use in communicating with each other? We need to create a catalog of these architectural models and styles and provide appropriate indexing mechanisms to connect them to the domain concepts. These can be used to generate expectations from the domain model about what architectural structures are likely to be found in the application software.

2. Architecture style and program descriptions

In going from the program to the architectural model, program analyses can provide some structural and behavioral information to produce a rough architectural description of the program. For example, extracting the call graph can provide an initial description of a program's architecture. Object detection techniques [8][15][29][41] can also identify primary components and how they interact. Recognition techniques are needed to detect the use of standard architectural styles and mechanisms that in turn suggest the implementation of some domain concept or domain-specific nonfunctional property.

Some of the key issues in extracting architectural information from program descriptions are the following.

- What representations of programs and architectural styles are appropriate?

- Is there a single representation that spans the continuum from code-level to architectural-level descriptions, or do we need a hybrid, layered representation?
- How do these representations relate to the application-domain model representation?
- What extraction techniques are effective?

3. Annotation

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. It is likely that a truth-maintenance (or reason-maintenance) system will be needed to handle confirmation and refutation of expectations.

B. Repository Issues

The primary goal of the repository is to provide a unifying framework for capturing and managing the evolving domain, architecture, and program descriptions during the SR process. We have been investigating two candidates: the Software Refinery's object-oriented repository and the MySQL relational database. Each provide a unique set of advantages. Refine's repository provides a common representational mechanism for both the OMT domain model and some of the program analyses results.

A relational database, on the other hand, has shown itself capable of representing and managing the myriad details extracted during program analysis [7]. Specifying analyses in terms of relational queries is convenient in terms of the amount of data manipulated and allows for rapid prototyping. However, the relational approach lacks expressiveness in terms of mapping analyses into queries and requires knowledge of relational technology to avoid formulating queries that take inordinate amounts of time or space to evaluate.

VI. Related Work

Recognizing and exploiting structure in domain, architecture, and code is the central problem in supporting program understanding. DARE draws on and benefits from a set of approaches differing at the level of abstraction at which they address the problem.

DARE is similar to approaches taken at the highest level of abstraction, such as LaSSIE [13]. 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 DARE 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 DARE extracts its knowledge from a set of smaller, functionally similar programs; thus, LaSSIE gets domain knowledge through reverse domain engineering, while DARE 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 DARE 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 DARE is interested in promoting reuse at the architectural level.

A high-level approach which greatly influenced our work is the approach taken by DESIRE [4]. DESIRE was the first software understanding systems to make use of a rich domain model. This model contains machine-processable forms of design expectations for a particular domain, as well as informal semantic concepts. It includes typical module organizational structures and the typical terminology associated with programs in a particular application domain. Techniques for recognizing patterns of organization and linguistic idioms in the program are used to generate expectations of the concepts associated with these patterns. Linking the patterns found in the program to conceptual structures in the domain model is based on the correlation (experientially acquired) between the mnemonic names used in the program's procedures, variables, and comments and the terminology recorded in the domain model. While DESIRE uses neural-network-based learning techniques to draw and refine connections between the domain model and the code, DARE supports the reverse engineer in acquiring and refining a model of the domain itself as well as its connection to the code being explored. The Synchronized Refinement process underlying DARE also broadens the domain model to include not only terminological knowledge and typical module decompositions, but architectural structures in general and design-decision rationale.

At a lower level than LaSSIE or DESIRE, Harris, Rubenstein, and Yeh [17] 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.

DARE and the work reported by Harris, Rubenstein, and Yeh are mutually supportive. 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 could fill an important need in DARE. In the other direction, the domain knowledge made available in DARE could serve to direct and sharpen pattern matching. For example, the WWW browser domain establishes positive expectations about task-spawning architectures; such expectations could be used to select candidate patterns for matching and help interpret the resulting matches to distinguish between bag-of-task spawning and pipe-and-filter spawning.

The development of the Synchronized Refinement process is related to research on software *reflexion* models [25]. A software reflexion model compares an architectural model of a program to its actual implementation. It does this by using a map between source code and an architectural model suggested by the analyst. A reflexion model can be used to incrementally refine an architectural model. The process begins with an analyst specifying what the suspected architectural model is for the source code. By incrementally adapting the mappings between major structural elements of the source code (e.g. file names, directories, name fragments), an architectural model is matched to the source code. The differences between the specified and derived models reflect mismatches between the two models. The mappings may be modified to construct a closer convergence between the two. SR incrementally matches source code to domain concepts. This is a large conceptual gap to cross, and an architectural description can serve as an intermediate point. The architectural description must include both the important domain concepts and the major software constructs designed to implement them.

Approaches still lower than those previously described are taken by such systems as DECODE [32], 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.

DECODE compares to the bottom, code-analysis half of DARE. 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 what 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.

Our work on dowsing and linkage tools has been influenced by Johnson's research on interactive software explanation [21] as demonstrated in the **I-Doc** prototype. 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's work supports task-oriented software understanding, our work focuses on supporting the complementary process of domain-oriented exploration.

VII. Conclusions

True program understanding requires relating a program to a description of the computational services it provides. Most program understanding tools concentrate on source-code analysis and ignore the program's application domain. DARE takes advantage of application-domain knowledge to improve the quality of the analysis performed and the effectiveness of the analysis process.

Unfortunately, the vocabulary and concepts used by programmers and application analysts are not identical. To deal with this difficulty, we are exploring linkage and repository support to provide a common representational mechanism using object-oriented frameworks and relational database technology.

Nevertheless, even with a common grounding, differences in abstraction level still complicate understanding. For example, high-level nonfunctional requirements, such as performance constraints, demand specialized solutions, such as caching, to be implemented at an architectural level. Thus the understanding process could benefit from an intermediate architectural description supported by two mappings—from the program source code to architectural components and connectors and from architectural mechanisms to the requirements engendering them. It seems likely that a significant body of experiential knowledge (e.g. which mechanism are useful for improving performance) is required to construct this mapping. Issues arise concerning how to capture, represent, and access this knowledge.

In summary, the goal of the DARE project is to explore the role of domain knowledge in program understanding. To this end, we are concerned with what sorts of tools an analyst can use to support that understanding. This paper has described what such a "dowsing" tool might look like and the underlying information structures it provides. We are exploring these questions in the context of real world software systems.

Acknowledgments

The effort reported on in the paper was sponsored by the Army Research Laboratory under contract DAKF11-91-D-0004-0055. We 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

[1] Russell J. Abbott. "Program Design by Informal English Descriptions." *Communications of the ACM*, 12(11): 882-894, November 1983.

[2] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[3] Arena. `http://www.w3.org/pub/WWW/Arena/`.

[4] Ted J. Biggerstaff, Bharat G. Mitbander, Dallas Webster. "Program Understanding and the Concept Assignment Problem." *Communications of the ACM*, 37(5):72-83, May 1994.

[5] Grady Booch. *Object-Oriented Analysis and Design with Applications, second edition*. Benjamin/Cummings, 1994.

[6] Steven E. Brenner and Edwin Aoki. *Introduction to CGI/PERL*. M & T Books, 1996.

[7] Eric J. Byrne and Gokul V. Subramamian. “Deriving an Object Model from Legacy Fortran Code.” *Proceedings of the International Conference on Software Maintenance*, Monterey, California, November 4-8, 1996, 3-12.

[8] C. Canfora, A. Cimitile, and M. Munro. “A Reverse Engineering Method for Identifying Reusable Abstract Data Types.” *Proceedings of the First Working Conference on Reverse Engineering*, Baltimore, Maryland, pp. 73-82, May, 1993.

[9] Y. F. Chen and C. V. Ramamoorthy. “The C Information Abstractor.” *Proceedings COMPASC 86*, IEEE, 291-298, 1986.

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

[11] Richard Clayton, Spencer Rugaber, Lyman Taylor, Linda Wills, A Case Study of Domain-based Program Understanding, *5th International Workshop on Program Comprehension*, Dearborn, Michigan, May 28-30, 1997

[12] Jean-Marc DeBaud, Bijith M. Moopen, and Spencer Rugaber. “Domain Analysis and Reverse Engineering.” *Proceedings of the 1994 International Conference on Software Maintenance*. Victoria, British Columbia, Canada, September 19-23, 1994, 326-335.

[13] Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge, and Bruce W. Ballard. “LaSSIE: A Knowledge-Based Software Information System,” *Communications of the ACM*, 34(5):35-49, May 1991.

[14] *Dialect User's Guide*. Reasoning Systems, Palo Alto, California, July 1990.

[15] H. Gall and R. Klosch. “Finding Objects in Procedural Programs: An Alternative Approach.” *Proceedings of the 2nd Working Conference on Reverse Engineering*, Toronto, Ontario, pp. 208-216, July 1995.

[16] Ralph E. Griswald and Madge T. Griswald. *The Icon Programming Language*. Prentice Hall, 1983.

[17] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. “Recognizers for Extracting Architectural Features from Source Code.” *Second Working Conference on Reverse Engineering*. Linda Wills, Philip Newcomb, and Elliot Chikofsky, editors, pp. 252--261, IEEE Computer Society Press, July 1995.

[18] HTTP. `http://www.w3.org/pub/WWW/Protocols/`.

[19] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Oriented Approach*. Addison Wesley, 1992.

[20] R. E. Johnson and B. Foote. “Designing Reusable Classes.” *Journal of Object-Oriented Programming*, 1(2):22-35, June/July 1988.

[21] W. Lewis Johnson and Ali Erdem. “Interactive Explanation of Software Systems.” *Automated Software Engineering*, Volume 2, 1996.

[22] H. Koning and J. Eizenberg. “The Language of the Prairie: Frank Lloyd Wright's Prairie Houses.” *Environment and Planning*, 8(B);295--323, 1981.

[23] E. Koutsofios and S. C. North. “Drawing Graphs with dot.” AT&T.

[24] Mosaic. `http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/`.

[25] Gail C. Murphy, David Notkin, Kevin Sullivan. “Software Reflexion Models: Bridging the Gap between Source and High-Level Models.” *ACM SIGSOFT 95 Symposium on the Foundations of Software Engineering*, October 1995.

[26] Gail C. Murphy, David Notkin, and Erica S.-C. Lan. “An Empirical Study of Static Call Graph Extractors.” *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, March 25-29, 1996.

[27] MySQL. `http://www.tcx.se/`.

[28] Sastry Nanduri and Spencer Rugaber. “Requirements Validation via Automated Natural Language Parsing.” *Journal of Management Information Systems,* 12(3):9-19, Winter 1995-96.

[29] P. Newcomb and G. Kotik. “Reengineering Procedural into Object-Oriented Systems.” *Proceeding of the Second Working Conference on Reverse Engineering*, Toronto, Ontario, pp. 237-249, July 1995.

[30] *OmniPage Professional Reference Manual*. Caere Corporation, Los Gatos, California, 1994.

[31] Rubén Prieto-Díaz and Guillermo Arango. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, California, 1991.

[32] Alex Quilici and David N. Chin. “DECODE: A Cooperative Environment for Reverse-Engineering Legacy Software.” *Second Working Conference on Reverse Engineering*, Linda Wills, Philip Newcomb, and Elliot Chikofsky, editors, pp. 156-165, IEEE Computer Society Press, July 1995.

[33] *Refine User's Guide*. Reasoning Systems Incorporated. Palo Alto, California, 1990.

[34] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr. “Recognizing Design Decisions in Programs.” *IEEE Software*, 7(1):46-54, January 1990.

[35] Spencer Rugaber. “Position Paper Domain Analysis and Reverse Engineering.” *Software Engineering Techniques Workshop on Software Reengineering*, Software Engineering Institute, Pittsburgh, Pennsylvania, May 3-5, 1994.

[36] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[37] Software Requirements Specification for the Software Loader/Verifier of the Block Improved Abrams Tanks (M1A2). US Army Armament Munitions and Chemical Command, April 13, 1995.

[38] W. P. Stevens, G. J. Myers, and L. L. Constantine. “Structured Design.” *IBM Systems Journal*, 13(2):115-139, 1974.

[39] Style sheets. `http://www.w3.org/pub/WWW/Style/`.

[40] *Browsing Source Code*. Sun Microsystems. 1994.

[41] A. Yeh, D. Harris, and H. Reubenstein. “Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language.” *Proceedings of the Second Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, pp. 227-236, July 1995.