A Case Study of Domain-based Program Understanding

Richard Clayton, Spencer Rugaber[1], Lyman Taylor
College of Computing
and
Linda Wills
School of Electrical and Computer Engineering
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 the Mosaic World Wide Web browser software. It discusses the domain analysis undertaken, the corresponding source code analysis we plan to perform, and the strengths and limitations of available automated tools.

I. Introduction

A. Goal

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 automating program understanding must also use knowledge of the goals of the program being understood [15] [6]. This knowledge takes the form of an application-domain description. The Domain Analysis and Reverse Engineering (DARE) project[2 3] at the Georgia Institute of Technology is exploring how best to use domain knowledge to understand programs.

B. Approach

DARE is using the Synchronized Refinement (SR) program understanding technique [14], the Software Refinery tool set [13], and a domain representation based on object oriented frameworks [9] to explore an intermediate sized software system, the Mosaic World Wide Web browser [10]. SR coordinates the simultaneous activities of bottom-up program analysis and top-down application-domain model elaboration. The Software Refinery is an integrated set of program analysis and transformation tools that includes an object oriented repository to hold the results of analysis. The domain representation we are using takes advantage of the repository and the underlying

1. Point of contact: spencer@cc.gatech.edu.
2. Effort sponsored by the Army Research Laboratory under contract DAKF11-91-D-0004-0055.
3. Effort sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-2-0229. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

Refine language to describe the application domain as an object oriented framework (a collection of cooperating abstract classes). As program exploration proceeds, the framework is instantiated with concrete information concerning how the abstract domain ideas are actually implemented.

C. Case Study

The case study described in this paper concerns the analysis of the Mosaic internet web browser. The source code for Mosaic consists of approximately 100,000 lines of code written in the C programming language. The code is readily available, easily testable, and documentation is available. The research described in this paper explores the specific issue of how to relate the implementation-independent domain description to various architectural-level analyses performed on the source code itself.

II. Synchronized Refinement

Synchronized Refinement is a technique 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 [5]. Synchronized Refinement constructs such a representation of a system from its source code. 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 comprise 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 consists of 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 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 abstract object oriented classes. These are instantiated 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 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.

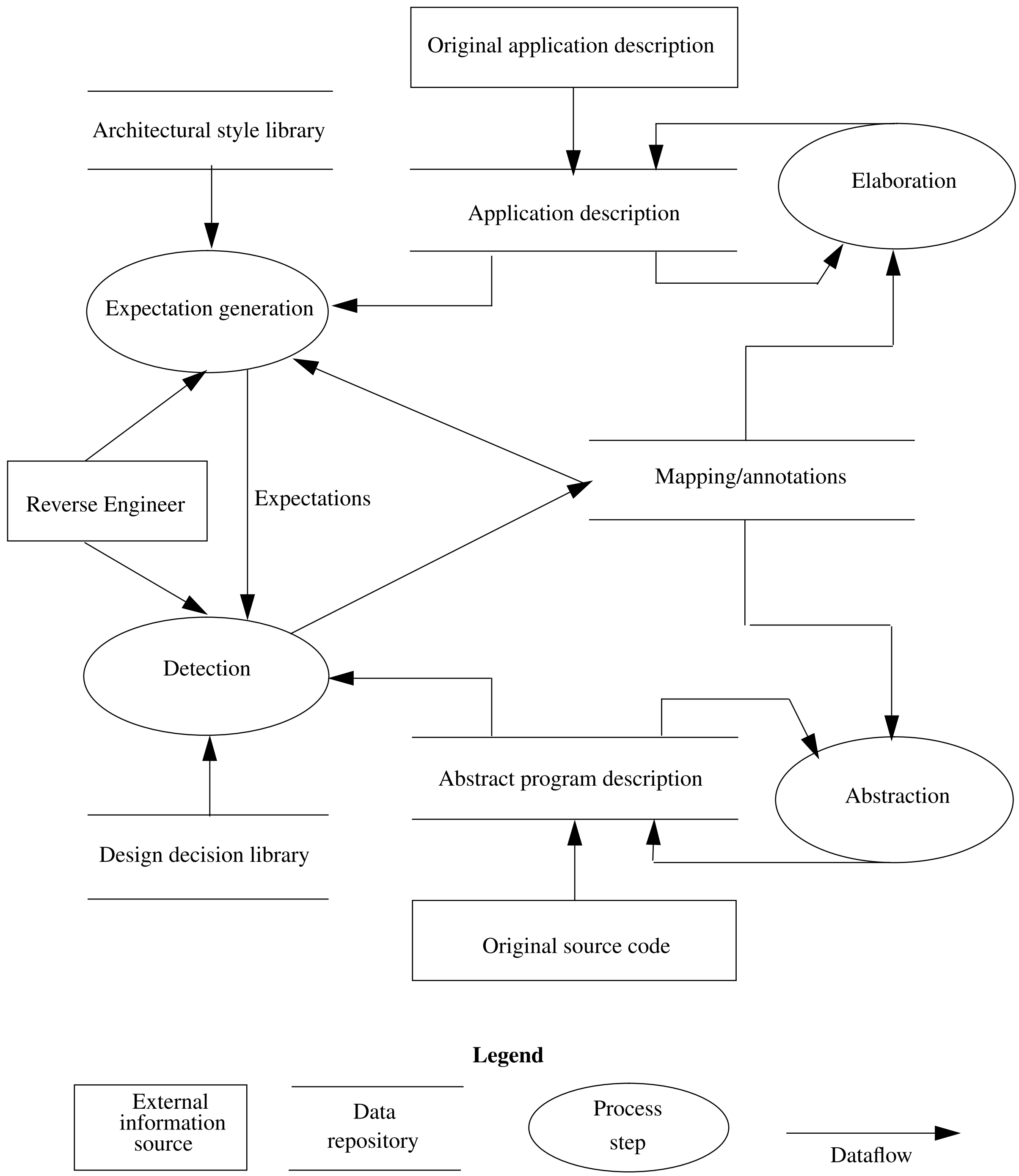

Figure 1: Data Flow Diagram for Synchronized Refinement

III. Case Study

As our case study, we chose to examine the Mosaic internet web browser, specifically version 2.4. We chose it because of its familiarity and accessibility. Our examination began with the building of an initial application-domain model as described in subsection A. Subsection B discusses our plans for code analysis. The issues raised by this study and our approach in general are described in Section IV.

A. Application-Domain Model

Application-domain modeling [12] 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. In the Mosaic case study, 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 more detailed information.

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) [16]. Third, the OMT Object Model is instantiated in a machine-manipulatable model using the Software Refinery tool suite.

The first step in creating the application-domain model is to extract objects 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 [7]) and fine-grained semantic definitions (e.g. style sheets [17]). Higher-level specifications, when they exist, are usually given in unhelpful forms, such as binary-only reference implementations (e.g. Arena [1]).

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. All the considerations that make a description informative also make it a good source for domain analysis: conciseness, completeness, clarity, and so on. In addition, to allow for machine-aided object analysis, it is helpful if the material is available in machine-readable form. For the case study, we chose Chapter 1 from [2].

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 some frequency analysis results. We performed a noun and verb analysis on the World Wide Web

Word	Count	Noun-verb digrams	Count
server	61	server sends	3
script	36	data return	3
browser	24	you receive	2
client	23	views HTML	2
user	14	user views	2

Table 1: Some word frequency analysis results

Word	Count	Noun-verb digrams	Count
information	12	server receives	2
scripts	11	server delegates	2
output	11	script called	2

Table 1: Some word frequency analysis results

descriptions to extract possible objects and associations. The analysis resulted in 25 objects, and 6 associations; see Figure 2.

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 Object Modeling Technique (OMT) [16] because of its familiarity in the software engineering community. OMT includes several graphical notations for specifying models. Its Object Model expresses the important domain objects and the inter-object associations. Figure 2 shows the OMT Object Model for the Internet web browser domain model.

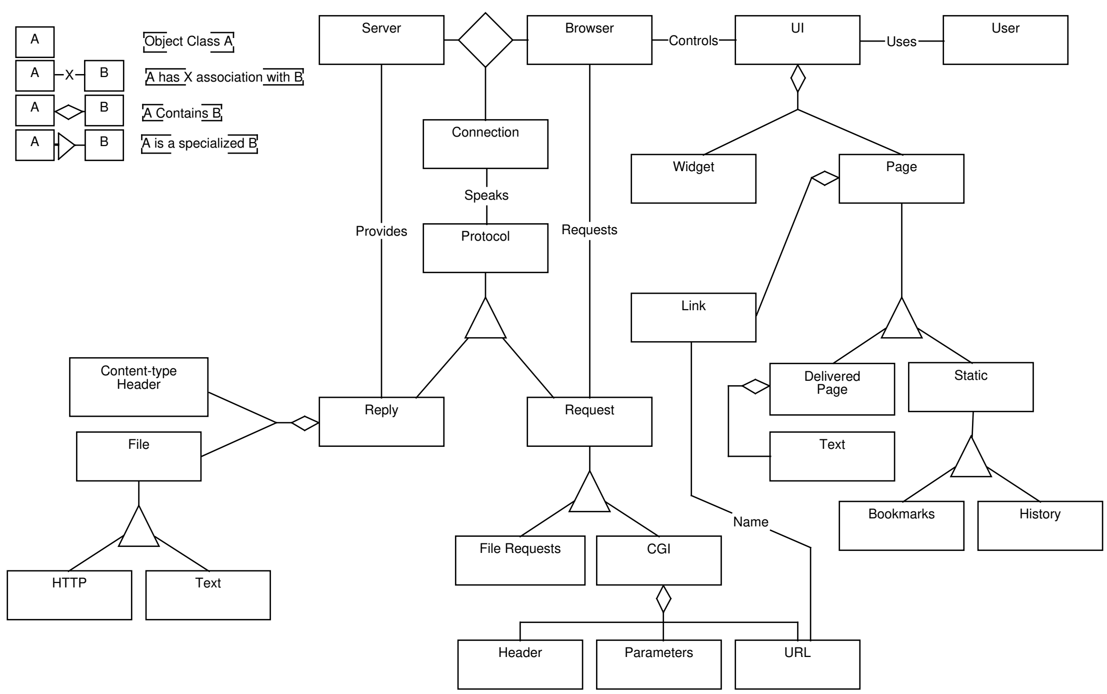

Figure 2: OMT Object Model Diagram for the Web Browser Domain Model

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 is one of the primary hypotheses of this project: that an object-oriented framework is a powerful and useful representation for application-domain information in reverse engineering. 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 tools as our base modeling tool. The Software Refinery is a large, Lisp-based suite of programs oriented towards program analysis and transformation. The Software 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 representing software information. In addition, we use the Dialect subsystem to build parsers that translate between Software Refinery internal and external textual representations of the Object Model.

The Software Refinery representation, particularly in the form provided by Dialect, does not map directly to the OMT Object Model. The Dialect representation is oriented towards attributed syntax trees, while the OMT Object Model is oriented towards more general graph structures. There are several approaches possible for resolving the mismatches between the Software Refinery representation and the OMT Object Model. The first is to ignore the Dialect parse trees and represent OMT Object Model associations as objects instead of as tree links. Using an object instead of a tree link to represent an association solves some of the more immediate representational problems: an object can easily represent association attributes, many-to-many attributes, and role names, things which are difficult to represent with just tree links. However, the more complicated problem of representing the relation between OMT Object Model objects and associations still remains. In addition, many of the Software Refinery tools for manipulating and analyzing Dialect trees are of little or no use for models that ignore tree links.

The second approach to resolving the mismatches is to modify the Software Refinery and Dialect representations to a form more appropriate to the OMT Object Model, which involves changing the Software Refinery and Dialect representations to allow arbitrary graphs. The open nature of the Software Refinery makes these kinds of changes possible, but the amount of machinery involved requires some degree of certainty before proceeding.

To illustrate the points made in the preceding paragraphs, we end this section with a brief tour through the current implementation of the Software Refinery representation of OMT Object Models. 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.

We chose to resolve the mismatch between the trees provided by the Software Refinery and the graphs found in the OMT Object Model by ignoring the trees. 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 describes Object Model structure.

There are four Refine objects defined to represent OMT Object Model entities:

```
var omt-om-entity: object-class subtype-of user-object
var omt-om-root: object-class subtype-of omt-om-entity
var omt-om-class: object-class subtype-of omt-om-entity
var omt-om-association: object-class subtype-of omt-om-entity
```

`Omt-om-entity` is the superclass for all other representation entities. `Omt-om-root` represents the whole Object Model; `omt-om-class` represents a class in the Object Model, and `omt-om-association` represents an association in the Object Model.

There are four mappings defined among the Refine entities given in the previous paragraph:

```
var omt-om-classes: map(omt-om-root, set(omt-om-class))
var omt-om-associations: map(omt-om-root, set(omt-om-association))
```

```
var omt-om-name: map(omt-om-entity, string)
var omt-om-associates: map(omt-om-association, set(string))
```

`Omt-om-classes` maps a root entity to the set of class entities defined in the Object Model represented by the root; similarly, `omt-om-associations` maps the root to the set of association entities defined in the Object Model. `Omt-om-name` maps a string name with an entity (which, via inheritance, could be a root, class, or an association). `Omt-om-associates` gives, for each association entity, the names of the class entities that are part of the association.

Refine needs to distinguish between mappings that structure the representation (such mappings are called **attributes**) and mappings that annotate the representation:

```
form declare-omt-om-tree-attributes
  define-tree-attributes(
    'omt-om-root, {'omt-om-classes, 'omt-om-associations})
```

In this representation, only the `omt-om-classes` and `omt-om-associations` mappings provide structure (and do so relative to the root); the `omt-om-associates` and `omt-om-name` mappings provide annotations.

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 of a object-model class is the keyword `class` followed by a string giving the class name; for example

```
class "a"
```

The Dialect rule to recognize a class is

```
omt-om-class:= [ "class" omt-om-name ] builds omt-om-class
```

Mappings may be used as non-terminals in the grammar; Dialect uses the range of the map to determine the parse type (a string in this case). The `builds` clause indicates that the parser should build an `omt-om-class` entity when the rule is successfully applied; the parser insures the given string is associated with the new `omt-om-class` entity via the `omt-om-name` map.

The external representation of an object-model association is similar to that of an object-model class:

```
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. The Dialect rule for recognizing an association is

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

The Dialect rules can use extended regular expressions; in this case the rule specifies one or more associates (that is, object-model class names) separated by a semicolon.

Finally, the Dialect rule for recognizing an external representation of an object-model is

```
omt-om-root  ::=
[ "(" omt-om-classes + ";" ")"  "(" omt-om-associations + ";" ")" ]
builds omt-om-root
```

When successfully applied, this rule also constructs the internal representation for the Object Model.

B. Code Analysis

High-level models are useful in the description and comprehension of complex constructs such as programs. Their utility is due primarily to their abstraction of the complicated details of the systems they describe. 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 field of building architecture is the difference between the blueprint of the classical Frank Lloyd Wright "Prairie House" and a particular instance of it. For example, there may be differences in the instantiation that are derived from pragmatic concerns due to the location site peculiarities.

Through Synchronized Refinement, a body of source code is matched to application-domain models. Through code analysis, an abstraction of the code is developed that is resolved with the expectations generated by the models. This is a iterative process reflected by the dataflow cycle in Figure 1 that flows from the **Abstraction** process to **Abstract program description** to the **Detection** process to the **Mappings/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 this particular instantiation of "Prairie House" that incorporates the peculiarities of this instantiation (i.e., the deviations from the "classical"). The following is a discussion of several code analyses that we are evaluating for application to the Mosaic source code.

1. Reflexion

A software reflexion model [11] 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 that is 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. Because of the incremental nature of SR, reflexion can serve a monitoring role, gauging the fidelity of the match.

2. Invocation Analysis

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

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

One advantage of using the Software Refinery is that analyses like the above are easily expressed in terms of Refine language data structures. A relatively straightforward pattern matching procedure can be used to translate analysis constructs into instantiations of domain or architectural objects. Because both are represented and stored inside the Software Refinery Object Base, the analysis process is more tightly integrated.

3. 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 doesn't natively support object oriented constructs [4]. *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 function 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 functions' formal arguments and return types and associating functions to the abstraction interface of one particular type. Additionally, datatypes serving as collections have to be identified. Implementation of this in the Software Refinery requires interaction with both the low level (parse tree/symbol table) and high level (who-calls, etc.) representations of the program. The Refine language makes both representations manipulable by many of the same operations and transformations.

4. Coupling

The amount of communication between modules can be used as a measure of the strength of the association between them. In short, a measure of the coupling between two modules or objects is indicated by the strength of this association [2]. 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 function call. Or it may be represented by a slot or field access of a structure or object.

5. Problems of Pragmatic Decisions

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 done above unless elements of the design decision can also be incorporated into the abstraction process. This is the justification for why design decisions also feed into the abstraction cycle in SR through the detection process. Decisions that might confound the derived representation are synchronized with the architectural and domain expectations to derive reflexional mappings for the abstraction process.

IV. Issues Raised

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 description of the program can bridge this gap. This raises three primary issues:

1. What is the relation between the application-domain model and the architecture?
2. How can architectural styles be connected to program descriptions resulting from standard code analyses?
3. 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?

A. The Relationship between the Application-Domain Model and the Architecture

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. A useful way of expressing these nonfunctional properties as well as the expected behaviors and functions is by using usage scenarios [8] that are described in reference to the architecture.

In application-domain programs, there are sets of architectural mechanisms that typically implement these properties, behaviors, and functionality. For example, these might be 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.

B. Architecture Style and Program Descriptions

In the other direction of analysis, 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 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 useful?

C. 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.

References

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

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

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

[4] 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.

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

[6] 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.

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

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

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

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

[11] 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.

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

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

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

[15] 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.

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

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