The Representation Problem in Reverse Engineering

Spencer Rugaber and Richard Clayton
College of Computing and Software Research Center
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

Building models to understand software systems is an important part of reverse engineering. Formal and explicit model building is important because it focuses attention on modeling as an aid to understanding and results in artifacts that may be useful to others. The representation used to build models has great influence over the success and value of the result. Choosing the proper representation during reverse engineering is the representation problem. *This paper examines the representation problem by presenting a taxonomy of solutions. It also illustrates the issues involved in choosing a representation through an example reverse engineering task.*

1 Introduction

Chikofsky and Cross define reverse engineering to be "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 [CC90]." They go on to describe six key objectives of reverse engineering: controlling complexity, generating alternative views, recovering lost information, detecting side effects, synthesizing higher abstractions, and facilitating reuse.

The concept of *representation* is central both to the process and the objectives of reverse engineering. To be effective, reverse engineers must develop a process capable of dealing with the informal insights developed while understanding a program and of producing explicit representations of that understanding useful to accomplish the objectives listed above.

This paper distinguishes between the mental activities involved in understanding a program and the representations produced. The reverse engineer is responsible for developing mental *models* of what a program does. *Representations* and formal representation techniques are tools for modeling and for expressing the results.

Model making is an appropriate activity for understanding software systems. One reason is that software itself is a model and so is already in a schematic form, reducing the effort needed to construct further models. A second reason is that programs are constructed from known quantities, such as executable statements and data structures, reducing some of the mystery as to how a program is constructed and leading to models that can accurately reflect the software system under consideration. Finally, programmers, the people most likely to need a detailed understanding of a software system, are, to some extent, trained as model makers and so are likely to have both the skill to construct mental models and an appreciation of the value in doing so.

Understanding a software system by constructing mental models is an important part of any task involving a software system. A large part of current work in reverse engineering is concerned with making explicit the usually implicit comprehension activities and models involved in software tasks [Van92, Raj92]. Explicit knowledge representation results in greater awareness and attention to reverse engineering activities, better prediction of the activities' expected efforts and results, and a collection of well-defined system models valuable to every task being carried out on the software system.

The importance of building and manipulating explicit models points to a central issue in reverse engineering: the way models should be built and, in particular, the representations that should be used to build the models. In keeping with the implicit nature of most reverse engineering activity, current model representations, if they exist at all, are typically informal and ad hoc: text files containing extracted information, documentation, programmer's notes, or folklore. These representations are too ephemeral to be a stable base for explicit, organized activity. The *representation problem* in reverse engineering is concerned with determining what representations are suitable to the reverse engineering process and the circumstances under which they should be used.

This paper examines the general features of the representation problem. It presents a taxonomy of issues related to solving the representation problem. The issues are illustrated in a reverse engineering exercise to convert a COBOL program to a program written in a fourth generation language (4GL). Our observations from the example lead to suggested directions in future research aimed at solving the representation problem.

2 A taxonomy of issues in model representations

There are several ways of looking at representations of reverse engineering information. We have chosen an approach emphasizing three aspects: the information content captured by the representation, how the representation supports modeling, and how the representation is to be used.

2.1 Information content

There are two issues related to the information content of a representation. The first is the subject matter being represented. The second is the tradeoff between fidelity and abstraction.

2.1.1 Subject matter: Reverse engineers require an understanding of at least three aspects of a program in order to accomplish any of Chikofsky and Cross's objectives: 1) the computations performed by the program and how they are structured, 2) the rationale behind the computational structure, and 3) an understanding of how the program computations solve the application problem. For example, if a maintenance task is to repair a program fault, the fault must be located amongst all of the computations performed by the program. If the task is to make a significant enhancement, it may be necessary to understand why the program was structured the way that it was. If the task is to extract business rules from an existing information system, then the nature of the business and how the program helps accomplish it must be determined.

Rumbaugh, et al.[RBP$^+$91] have identified three important viewpoints for understanding the computations performed by a program: the objects manipulated by the computation (the data model), the manipulations performed (the functional model), and how the manipulations are organized and synchronized (the dynamic model). Most representation techniques used during software design emphasize one of these three views.

Representation	Examples
Object-Oriented Frameworks	[JF88]
Category theory	[Sri91]
Concept hierarchies	[Big89, Lub91]
Mini-languages	[Nei84, ABFP86]
Database languages	[CR86]
Narrow spectrum languages	[Web87]
Wide spectrum languages	[Wil87, WCM89]
Knowledge Representation	[Bar85]
Text	

Table 1: Application domain models

A way of representing the rationale for a program's computational structures is in terms of the design decisions they implement [ROL90]. Design is a rational process involving a series of decisions transforming a specification into an implementation. If reverse engineering can detect clues to the decisions made, it can use the clues to "untransform" the program back to a higher level. The detected clues can be organized around categories of design decisions. Among the decision categories are the following:

- Decomposition: breaking a problem into pieces;
- Specialization: dealing with commonality and special cases;
- State introduction: introducing structures to avoid recomputation or improve abstraction;
- Removing choice: tying down choices left free by the specification;
- Representation: choosing implementation mechanisms;
- Encapsulation/interleaving: controlling which computational structures are grouped together and which are distributed.

The third issue related to subject matter concerns the domain knowledge used to express the emerging model. Two important categories of domain knowledge are programming skills and application domain. Programmers use a collection of mechanisms for constructing their programs. These are often described as "plans" or "schemes" in program understanding literature [RW90]. There also is work going on in "domain analysis/engineering/modeling" to support forward engineering activities via reuse. Table 1 summarizes a variety of approaches taken by various projects involved in modeling application domains.

2.1.2 Fidelity and abstraction: Models constructed during reverse engineering are more abstract than the code from which they are derived. However, there are several questions related to the degree of abstractness appropriate to the emerging models: the amount of detail retained in the representation and, consequently, the completeness of the representation; the extent to which the representation captures the application semantics versus reformulating its code syntactically; and the "operationality" of the representation.

A question with regard to fidelity is whether the corresponding representation retains all of the information available in the source code. For example, are the indentations and comments from the code retained? Abstract syntax trees, a common representation, ignore such minutiae in favor of a syntactically organized representation focusing on constructs expressing the computations performed. A related issue is whether the representation retains informal annotations composed by the original designers, subsequent maintainers, or reverse engineers.

Syntactically oriented representations are limited to answering low-level questions about the design. For reverse engineering to be successful, a representation must be constructed describing the implementation architecture and how it relates to the problem being solved by the software. An example is thinking of a compiler as a pipeline of independent filter programs for lexically, syntactically, and semantically analyzing a program and then generating and optimizing the target code. The pipeline model of the compiler architecture is one of many emerging high-level idioms for software designs. Shaw discusses the issues related to this approach to software development [Sha89].

Semantically oriented representations can be classed as to how operational they are. The more operational representations serve as algorithms for some specific computational mechanism or abstract machine. Expressing a program in another programming language is an operational representation; so is a virtual machine description. Slightly less operational are representations employing well-understood mathematical constructs, such as sets, lists, and trees, similar to the approach taken by VDM and Z in expressing software specifications. Finally, the least operational representations abstract all the way back to a specification expressed in the predicate calculus. This is similar to the axiomatic semantics approach to formal specification.

2.2 Representational support for the modeling process

A representation for reverse engineering models should support the methods used to construct the models. Among the issues that must be addressed are the following. How does the representation organize and express hypotheses and guesses about a program segment's purpose before they are confirmed or refuted? How does the representation describe the intra- and inter-relationships among the derived design description fragments, the code, and the domain representation, particularly before the overall design architecture has become apparent? Does the representation support inferencing and querying by the reverse engineer? As the program/design is transformed and abstracted, are previous versions retained? Perhaps most important, to what extent does the representation support pattern matching (the search for abstract program constructs), which is often expressed in terms of syntactic patterns and logical constraints?

2.3 How will the representation be used?

Reverse engineering has many objectives, such as to generate or update system documentation, to support maintenance, or as a prerequisite for re-engineering. In the ideal situation, it would be desirable to have a single representation capable of supporting all of these objectives. However, the current state of the practice often leads to representations aimed at supporting a single objective. In addition to the high-level uses mentioned above, more specific output requirements may drive the choice of a representation. For example, does the representation need to be automatically manipulatable? The answer to this question might determine whether the representation is kept in symbolic or in graphical form. Or is the representation capable of supporting multiple alternative graphical views?

If the representation is going to promote reuse, then it is important to know the extent to which it supports manipulations to adapt the code, such as generalizations and specializations. The manipulations might also involve interleaving several abstractions to form a combined result. Of course, being able to find the abstraction in the first place is essential to representations intended to support reuse.

Finally, the representation needs to be extensible in the following sense. If the results of the reverse engineering are going to be used to support other activities such as maintenance and re-engineering, then it is desirable to be able to use the same representation for these activities as was used during the reverse engineering. This will likely require adaptations and extensions to the representation.

2.4 Summary

In this section we have analyzed the issues related to selecting a representation for use during reverse engineering. The issues have been broken down into categories depending on whether they relate to the information content of the representation, the reverse engineering process actually undergone, or the intended use of the reverse engineering information. Table 2 summarizes our taxonomy of issues related to solving the representation problem. Next we will present an example of how these issues have effected the representation choices on an existing reverse engineering project.

3 An Example

This section describes the models developed and the representations used during a reverse engineering exercise. The intention is to 1) illustrate our view of the importance of modeling as described in section 1; 2) give readers the opportunity to establish connections with their own experience; and 3) motivate portions of the taxonomy described in section 2.

3.1 The reverse engineering task

The motivation for this reverse engineering exercise was to determine the feasibility of moving third generation information systems from a mainframe COBOL environment to a 4GL running with a relational database management system in a distributed, open systems environment. To investigate this problem, a typical information system was reverse engineered to obtain its design. The design was used to construct a 4GL program capable of duplicating the reports generated by the original program. The program, called P13AGU, is part of a system for keeping track of and producing reports on the maintenance status and operational readiness of Army equipment. The P13AGU program produces a report summarizing the status of all equipment described in an input file.

3.2 Approach

Synchronized Refinement was the technique used to reverse engineer P13AGU [KR90, OR92]. Synchronized Refinement begins by constructing two related models of the software: one describing the program text and the other the application problem. Once the two models are in place, they are refined towards each other. The low-level, code model becomes more abstract, suppressing operational details while highlighting the design architecture. The high-level, application model becomes more specific, augmenting the problem description with solution details. The objective is to refine the two models so they meet and match "in the middle," forming a complete and plausible path from problem to solution.

Progress in Synchronized Refinement occurs in both directions beginning with the application model; the early stages of the code model are often little more than independent fragments of unknown value. The application model begins as an overview of the problem and a series of hypotheses or expectations about possible solutions. The expectations are represented as a concept hierarchy to indicate interdependencies. The current set of expectations that obtain in the application model suggest possible structures for the code model. When a plausible structure is located in the code, the associated parts of the two representations are linked by an annotation describing the relationship. Successfully linking the two models leads to suggestions for further refinements. The application model becomes more detailed to reflect the deeper understanding about possible solutions, and the code model becomes more abstract as details are suppressed.

Design decisions coordinate the two models. A design decision is denoted by programming language constructs indicating how a programmer has chosen to implement a particular design idea. The application model suggests what kinds of design decisions the code model might implement, and the source code contains manifestations of actual design decisions for which the application model must account. [ROL90] provides a characterization of useful categories of design decisions; section 2.1.1 of this paper contains a summary.

We chose to use Synchronized Refinement for a number of reasons. First, we are familiar with the technique and have used it successfully on similar problems in the past. Second, it uses many models, including not only the application and code models, but also a number of subsidiary models to support refinements of the two main models. Finally, it performs frequent and complex manipulations on models to keep track of progress during refinement.

- What is the information content of the representation?
  - What is the subject matter covered by the representation?
    - What views of the program's computation does the representation provide?
      - Data view?
      - Functional view?
      - Dynamic view?
    - How does the representation support the model building process?
    - What domain knowledge is represented?
      - Programming domain knowledge?
      - Application domain knowledge?
  - How does the representation deal with fidelity and abstraction?
    - How complete is the representation?
    - Is the representation aimed primarily at syntactic or semantic issues?
    - How abstract is the representation?

- How well does the representation support the modeling process?
  - Does the representation include expectations?
  - Can the representation cope with incomplete structures?
  - Does the representation support inferencing and querying?
  - Does the representation support version control?
  - Does the representation support pattern matching?

- How will the representation be used?
  - For automatic manipulation?
  - For graphical or textual viewing?
  - To support reuse and adaptation?
  - Can the representation be indexed?
  - How compatible is the representation with other tasks?

Table 2: Summary of representation issues

3.3 Synchronized Refinement models of P13AGU

The Synchronized Refinement of P13AGU starts with a high-level application model (section 3.3.1) and a low-level code model (section 3.3.2) and goes on to develop a number of subsidiary models (section 3.3.3).

3.3.1 The application model: The application model is a description of the problem the system is trying to solve. A successful reverse engineering effort must include a description of what a system does and how it does it. The initial application description used in Synchronized Refinement is expressed in terms of a concept hierarchy.

The initial application model is typically quite vague and incomplete. It is a statement of what, if any, expectations the reverse engineer has about the purpose of the program. The expectations for P13AGU consisted of two concepts: "equipment availability" and "report writing". The latter leads to expectations about page layout and line counting. As the code model develops, further expectations are added to the application model's hierarchy. Most expectations are either generated by code analysis or are elaborations of the original expectations. Occasionally it is necessary to amend part of the hierarchy when conflicting data are encountered in the program.

The application concept hierarchy is related to the code model in several ways. First, each concept in the application model is ultimately manifested in the code model, by both data and operations. When using Synchronized Refinement, annotations are kept relating each refinement in the application model to the related abstraction in the code model. Annotations record not only the appropriate lines in the program and the concept hierarchy, but also the type of design decision the original developer made in order to express the concept in the code.

3.3.2 The code model: Representations of the initial code models are necessarily low level since they are derived directly from the code. They can all be easily expressed using well-known techniques. Moreover, they are so generally useful that they are typically performed prior to any detailed reading of the code.

The two prominent features of the application model described in the previous section are the inputs read and the report generated by the program. P13AGU reads two input files. One file contains a single record, called the DATE record, describing the circumstances under which a report is requested. A model for the DATE record file was constructed directly from the FILE DESCRIPTION section of the source code and represented by a Jackson diagram [Jac75]. An analysis of the source code indicated that the file is only read once, so no repetition is involved. Later analysis indicated some of the DATE record fields are never referenced by the program. This led to refinement of the original model and edits to its representation.

The other input file was significantly more difficult to analyze. Superficially, it appeared to be a collection of homogeneous data records, each containing a fixed number of fields. There are no REDEFINES clauses, but analysis of the source code indicated that multiple records from the input file are read. In addition, there are several COBOL FILLER fields, indicating positions in the input record not relevant to the program. It became apparent later in the analysis that the model of the second input file was seriously deficient. In fact, the records in this input file are sorted to three levels, and the program depends on the ordering. This fact only became obvious after detailed analysis of the source code indicated how the records were being processed. This model was also represented by a Jackson diagram.

The complement to this representation of the program's input data is a model of the flow of control during program execution. P13AGU makes heavy use of PERFORM paragraphs to organize activities. A control flow graph of each paragraph was used to represent this information and quickly made it apparent that the input file had some structure not obvious from its FILE DESCRIPTION information.

Another model developed early during the exercise indicated where in the source code each named data item was declared, where it was used, and how it was referenced (for reading or writing). This information is typically represented in a cross reference listing. Despite the simplicity of a cross-reference representation of the code, it proved to be surprisingly useful, and it became essential when trying to unite the data and control-flow models into a comprehensive model of program behavior.

As an example, P13AGU makes heavy use of PERFORM clauses. Before a paragraph is PERFORMed, data are moved into named areas of WORKING STORAGE, and after the paragraph is complete, results are moved from other areas into program variables. This led to the expectation that these MOVE actions were implementing an argument list to be made available to the paragraph as if it were an independent procedure, taking arguments and returning results. Once this expectation was confirmed, the data items used for this purpose could be annotated and a more abstract version of the program constructed. It is interesting to note that, in this example, data flow analysis of the sort performed by an optimizing compiler had to be understood, but it did not have to be represented explicitly.

Another model used during this exercise determined the major components of the system and their relation to each other. One representation for this information is a call graph in which nodes at the top of the graph PERFORM nodes at the bottom. This representation enables the reverse engineer to determine a bottom-up ordering for modeling the purpose of each paragraph.

Table 3 summarizes the code models derived during this stage of analysis and the representation techniques used.

Model	Representation Technique
Input data	Jackson diagram
Control flow	Control flow graph
Variable-use	Cross reference
Program architecture	Call graph

Table 3: Models and representation techniques

3.3.3 Refined models: The models described in the previous two sections were developed early during Synchronized Refinement. As the process continued, the initial models were transformed and abstracted. Versions of the model were saved with a description of why they were created, the transformations performed to create them, the associated design decisions, and any other commentary the reverse engineer thought important. Approximately one hundred model transformations were cataloged during P13AGU's Synchronized Refinement.

Some transformations were lateral or regressive; that is, they either didn't advance a model's refinement or they "unrefined" it. For example, COBOL permits the contents of one memory location to be MOVEd to several other named locations in the same statement. These other locations are often not related conceptually. Replacing multiple MOVE statements by individual MOVEs allowed for further independent refinements. These extra MOVE statements were important in recognizing and forming the procedure-call refinements on PERFORM statements mentioned earlier.

Another class of language-specific transformations were related to limitations in COBOL control structures. On some occasions, modern control structures unavailable in COBOL were simulated using GOTOs. On others, a cascade of decisions and interleaved GOTOs were used to avoid duplicating a section of code. In practice, this might improve the efficiency of the program, but at the price of reduced code clarity. Transformations applied to the code model abstracted away from these details.

The original code was monolithic, all statements appearing in one file. There were no logical or cosmetic distinctions separating the various components other than labels indicating the beginning and end of each paragraph. Moving related code into separate files helped reconstruct a high level architecture. For example, there was one paragraph responsible for checking whether the value of a variable occurred in a table of legal values. The table, together with the checking code, were broken out into a separate file. This step was a prerequisite for moving from a design dominated by logical cohesion to one with a more functional organization.

Code hoisting and lowering are interesting lateral transformations. Because the design of the original system was dominated by logical cohesion, it was often the case that a section of code contained several operations or sub-functions that were not closely related. Separating these functions lead to more coherent design. One way to do this was to hoist a subfunction from the beginning of a paragraph to sites just before each of the places where the paragraph was PERFORMed. Likewise, code near the end of a paragraph was lowered to just after the point where the paragraph returned. These transformations enabled subsequent transformations to group operations together based on the data they manipulated (communicational cohesion).

There were also many cosmetic transformations. These included renaming variables to better indicate their purpose within the code, removing data items not referenced in the code, and introducing new functions to perform data retrieval from encapsulated objects. The value of these cosmetic transformations has been pointed out by Byrne[Byr91].

Each of the transformations reflected a refinement to the reverse engineer's understanding of the program. The technique used to represent the transformation was to indicate the original source lines and their replacement, expressed in pseudo code. Table 4 summarizes the kinds of transformations performed during this stage of reverse engineering.

Transformation	Purpose
Lateral/regressive	Separation of concerns
Control restructuring	Raise abstraction level
Segregation into files	Improve modularity
Code hoisting/lowering	Improve modularity
Cosmetic	Relate to Application

Table 4: Model transformations

3.4 Summary

This section has described the models and associated representations used when reverse engineering a program. The method used, Synchronized Refinement, attempts to cover the entire range of system description from problem statement through design to implementation. This range of focus results in a large set of models, diverse in both what is being modeled and how the models are being represented. The iterative nature of Synchronized Refinement encourages explicit model manipulations, analyzing the current model set and extending it to reach the next model set.

4 Global issues in model representations

The previous sections have considered requirements on model representations with respect to a single reverse engineering task. Explicit model making also provides value over time, as subsequent reverse engineering tasks can draw on the models created by tasks already performed. This section considers requirements for long-term support of reverse engineering model development.

An important issue in long-term support for model representations is the trade-off between representation diversity and the ability to integrate representations. Diversity arises because representations are not uniformly suited to all models; given some particular model, it will usually be easier to represent it in some forms than others. The need for a number of different models during reverse engineering leads to representation diversity.

The ability to integrate representations ensures that reverse engineers using a collection of previously created models will be able to extract maximum value from them with minimum effort. Although it may be that a model is used only in isolation, it seems more likely, as illustrated by the Synchronized Refinement example above, that models are integrated in various ways during reverse engineering. It should be both possible and easy to integrate previously unrelated models; failure to do so results in either a limited ability to carry out reverse engineering or duplicated effort as the reverse engineer re-creates the models in an integratable form.

Two mechanisms for achieving balance in the trade-off between representation diversity and integration are universal representations and software backplanes. A *universal representation* claims to provide more or less equal support for all models. Examples of universal representations are Lisp symbolic expressions and relational database tables. A more sophisticated approach aimed directly at the software representation problem is IDL [Lam87]. Universal representations move the trade-off balance away from diversity towards integration. A *software backplane* is essentially an interconnection and communication standard understood by all models [Pas88]. The interconnection standard insures the models can be linked together; the communication standard insures the models can be made to do useful things once linked together. Software backplanes try to compromise, moving one piece towards diversity and the other towards integration.

The degree to which each of the mechanisms should be used to balance the trade-off between representation diversity and integration is unclear. It seems unlikely that just choosing one over the other will be acceptable. Software backplanes are the more promising of the two mechanisms since they offer a chance to achieve a balance instead of a trade-off between diversity and integration. However, it is clear that software backplanes require much preparatory work and incur a fixed and perhaps not small overhead during model making. In addition, the number of notable systems built using software backplanes is small in comparison to the popularity of the metaphor. Universal representations, having more modest objectives, have been more successful, particularly in reverse engineering and software engineering in general. However, their value outside the research lab, where diversity in product, practice, and practitioner are a fact of life, is less clear.

5 Conclusion

We have argued that model making is a fundamental activity in reverse engineering but is generally done in an implicit and informal way, which tends to limit the value of model making to the short term. Explicit and formalized model making in reverse engineering results in models that continue to have value after the activity that resulted in their creation is completed. The representations used to construct models is of central concern when the models are to be long lived. After characterizing the general features of models and the ability of various representations to support their features, we have presented a supporting example of modeling in reverse engineering. We have also presented more general considerations for model representations, with emphasis on their overall architecture.

6 Acknowledgements

The authors gratefully acknowledge the support of the Army Research Laboratory through contract DAKF 11-91-D-0004 and the encouragement of Dr. Jay Gowens.

References

[ABFP86] Guillermo Arango, Ira Baxter, Peter Freeman, and Christopher Pidgeon. TMM: Software maintenance by transformation. *IEEE Software*, 3(3):27–39, May 1986.

[Bar85] David Barstow. On convergence toward a database of program transformations. *ACM Transactions on Programming Languages and Systems*, 7(1):1–9, January 1985.

[Big89] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7), July 1989.

[Byr91] Eric J. Byrne. Software reverse engineering: A case study. *Software—Practice and Experience*, 21(12):1349–1364, December 1991.

[CC90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[CR86] Y. F. Chen and C. V. Ramamoorthy. The C information abstractor. In *Proceedings COMPSASC 86*, pages 291–298. IEEE, 1986.

[Jac75] M. A. Jackson. *Principles of Program Design*. Academic Press, 1975.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June-July 1988.

[KR90] Kit Kamper and Spencer Rugaber. A reverse engineering methodology for data processing applications. Technical Report GIT-SERC-90/02, Software Engineering Research Center, Georgia Institute of Technology, March 1990.

[Lam87] David Alex Lamb. IDL: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9(3):297–318, July 1987.

[Lub91] Mitchell D. Lubars. Domain analysis and domain engineering in IDeA. In Ruben Prieto-Diaz and Guillermo Arango, editors, *Domain Analysis and Software Systems Modeling*, pages 163–178. IEEE Computer Society Press, 1991.

[Nei84] James M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineerings*, SE-10(5):564–574, September 1984.

[OR92] Stephen B. Ornburn and Spencer Rugaber. Reverse engineering: Resolving conflicts between expected and actual software designs. In *Proceedings of the Conference on Software Maintenance*, pages 206–213, Orlando, Flordia., November 1992.

[Pas88] W. Paseman. Architecture of an integration and portability platform. In *COMPCON Spring 88*, pages 254–8, San Francisco, California, February 29-March 3 1988. IEEE Computer Society.

[Raj92] Vaclav Rajlich, editor. *Workshop Notes — Program Comprehension*, Orlando, Florida, November 9 1992. IEEE Computer Society.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[ROL90] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr. Recognizing design decisions in programs. *IEEE Software*, 7(1):46–54, January 1990.

[RW90] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. Addison Wesley, 1990.

[Sha89] Mary Shaw. Large scale systems require higher level abstractions. In *Proceedings of the Fifth Workshop on Software Specification and Design*, pages 143–146. ACM SIGSOFT Notes, ACM, May 1989.

[Sri91] Yellamraju V. Srinivas. *Pattern Matching: A Sheaf-Theoretic Approach*. PhD thesis, Department of Information and Computer Science, University of California at Irvine, May 1991.

[Van92] Larry Van Sickle, editor. *Workshop Notes — AI and Program Understanding*, San Jose, California, July 12 1992. AAAI.

[WCM89] M. Ward, F. W. Calliss, and M. Munro. The maintainer's assistant. In *Proceedings of Conference on Software Maintenance*, pages 307–315, Miami, Florida, October 1989. IEEE Computer Society Press.

[Web87] Dallas E. Webster. Mapping the design representation terrain: A survey. Technical Report MCC STP-093-87, Micro-Electronics and Computer Technology Corporation, July 1987.

[Wil87] David S. Wile. Local formalisms: Widening the spectrum of wide-spectrum languages. In L. G. L. T. Meertens, editor, *Program Specification and Transformation*, pages 165–195. Elsevier North Holland, 1987.