UNIVERSITY OF CALIFORNIA
Irvine

Domain Engineering for Software Reuse

Technical Report 88-27

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Information and Computer Science

by

Guillermo F. Arango

Committee in charge:

Professor Peter Freeman, Chair
Professor Richard Selby
Professor Peter Woodruff

1988

Contents

List of Figures

Abstract

Domain Engineering for Software Reuse

by

Guillermo F. Arango

A precondition for reusability is the existence of reusable information. There is a lack of systematic methods for producing reusable information. We propose a method for practical *domain analysis*, defined as the process of identification, acquisition and evolution of information to be reused in the construction of software systems for restricted classes of applications, or problem-domains.

The method for domain analysis is presented in the context of a *domain engineering framework*. A framework is not a theory and we do not offer a detailed, canonical scheme of how *every* type of domain analysis is or ought to be done. We have developed a set of principles providing coherence to a diverse set of findings about domain analysis. These principles (and heuristics) are used to develop instances of methods and representations, and to demonstrate their application in practice.

Reuse systems are viewed as composed of two parts, a performance component and a learning component. The operation of the performance component—the target reuser—is based on a *reuse infrastructure* specified by the learning component. Based on feedback from the target reuser, the learning component evolves a *model of the problem domain*, with the purpose of maintaining or enhancing the level of performance of the reuser. This viewpoint allows for an operational formulation of the domain analysis process as the systematic evolution of the reuser's infrastructure, where this evolution aims to attain and maintain a desired level of performance.

The methods proposed focus on two forms of performance enhancements—competence and efficiency—in the reuse process. Competence enhancements are achieved through model-driven information acquisition from experts and the incremental induction of a domain-specific language. Efficiency enhancements are the result of tuning a reuse infrastructure to explicitly defined patterns of reuse in the environment of the reuser.

Methods and representations are demonstrated in the context of first-order reuse—the reuse of software components.

Chapter 1

Introduction

Chapter summary. The context of this work is the research program for Reusable Software Engineering outlined by Peter Freeman in 1980. The focus is on the acquisition of reusable, domain-specific information for software construction. A conceptual framework, the *domain engineering framework*, is proposed as a foundation for the study of practical methods for the identification, acquisition and evolution of reusable information.

Chapter 1 summarizes the context and focus of this work and outlines the research questions and research assumptions for further reference. The chapter closes with a summary of results and an outline of the dissertation.

1.1 Background—reusable software engineering

Summary. In 1980 Freeman [Fre80] outlined a research program on *reusable software engineering*—the broad reuse of all information generated in the course of software development. The systematic reuse of products and processes in software development was defined as a software engineering problem. There are three technical aspects to reusability: the acquisition of reusable information, its representation, and its actual reuse within some problem solving task. The focus of this work is on the acquisition of application-specific information to be reused in the specification and construction of software systems.

There exists a widespread belief within the software engineering community that reusability is the key to substantial improvements in productivity and quality in software development [Fre87b], [Weg84]. In practice, this promise remains largely unfulfilled for technical and nontechnical reasons [Ber87]. The nontechnical barriers to reuse are psychological, sociological and economic (see, for example [Tra87a]), and

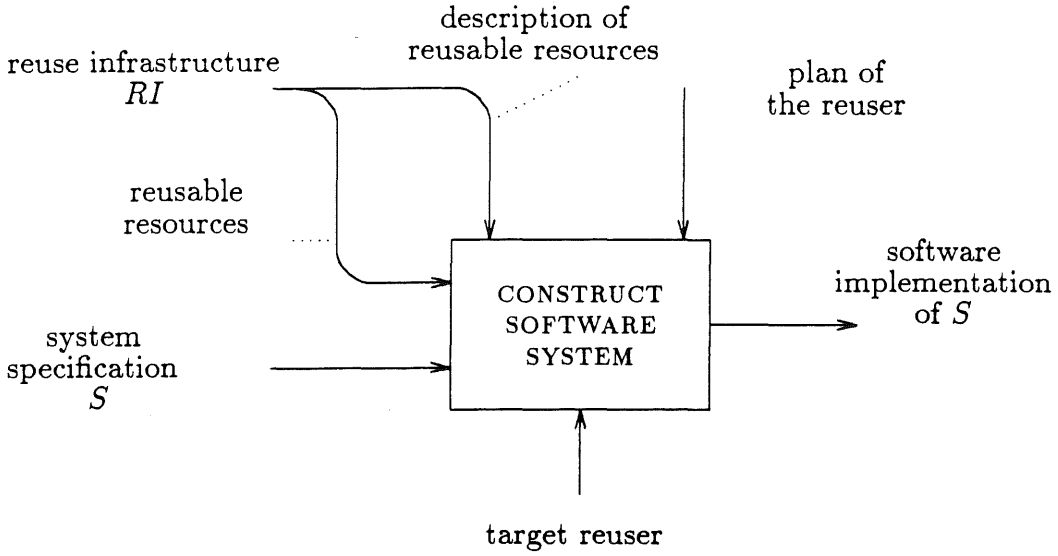


Figure 1.1: Reuse-based software construction

lie outside the scope of this research. The two technical reasons most often cited are the lack of effective methods for locating appropriate resources for solving a given problem and the lack of quality reusable resources [Tra87a] [Tra87b] [BR87]. These are distinct aspects of reusability. In the context of software construction (Figure[1] 1.1) the performance task—actual reuse—can be formulated as:

> Given a system specification S and a collection of reusable resources RI—a *reuse infrastructure*—how to locate, select, adapt; and integrate appropriate members of RI into an implementation (or plan for an implementation) of S.

The *infrastructural aspect* involves the systematic development and representation of reusable resources, that is

> How to identify, acquire, represent, analyze and evolve a relevant collection RI.

These two aspects of reuse are interdependent at a conceptual level and at a technological level. The acquisition of reusable resources depends on the task to be supported and on the technology of the reuser. Most past research on reusability has focused on the actual reuse task and representational issues. In the case of reuse

[1]The boxes and arrows diagrams follow the SADT [MM88] convention. Boxes represent activities; the subject matter is inside the box. The four sides of the boxes are called: Input (left), Control (top), Output (right) and Mechanism (bottom). Arrow connections make the interface between the subjects.

in software construction, for example, those efforts have resulted in automatic and semi-automatic application generators, library systems for software components, and object managers [Sep87] [iee87]. Research on representations led to the development of mechanisms and languages for supporting modularization, abstraction, and polymorphism [CW85] [LG86] [Sha84a].

This work is a contribution to the infrastructural aspect of reuse in software specification and construction. We focus on the acquisition of domain-specific information for the specification and implementation of restricted classes of problems, or *problem domains*. Examples of problem domains are: financial accounting, the control of industrial furnaces, missile navigation, patient monitoring in a hospital, inventory control, two-dimensional graphics. In this sense, this work is a step towards what Abbot [Abb87] calls "knowledge programming"—the construction of programs in a manner that makes domain-level knowledge visible in the process and the product.

1.2 The technological components of reuse

We use the expression *reuse environment* to refer to an organizational context in which information is systematically acquired and reused in software construction under the control of explicit management guidelines. Reuse environments include organizational and technological components. The technological components directly relevant to the reuse process are drawn from three major groups [Ara88a]:

1. Performance, or "actual reuse" technologies.
   Many approaches to reuse in software construction have been identified. For example: reuse by composition, as implemented by Ada [Boo83]; by inheritance [Mic88]; by transformation [Bar88] [Nei84] [PS83]; by replay [Bax87e] [SS83], by analogy [Car83][DS87] [Der83]. A performance technology supports the actual reuse of whatever resources are relevant. In some cases the reuse process may be completely mechanized, as is the case with application generators. In other cases, the reuse technology may be a combination of people and systems; for instance, programmers trained in reuse working in the Smalltalk environment [Gol83].

2. Environments for the management of reusable resources.
   Environments must provide representation formalisms for the description of reusable objects, as well as mechanisms for the creation, transformation, evolution, persistent storage, and viewing of descriptions. Smalltalk or CLF [CLF88] are examples of such environments.

3. Resource acquisition technologies.
   Technologies for managing and reusing resources are not sufficient to make reusability feasible. Systematic methods for the identification and capture of reusable information are needed.

The focus of this work is on a particular form of acquisition engineering, which we call *domain engineering*. Domain Engineering (DE) is concerned with the systematic identification, acquisition, and evolution of reusable information in restricted problem domains.

1.3 Problem definition

Summary. A reuse infrastructure is a collection of reusable items of information that can be employed by a reuser to specify and implement software systems. Our research problem can be summarized as: Given a partial, nonformal description of a problem domain, how can we identify items of information that could be reused in the specification and implementation of software systems in the domain.

"Software is often expected to provide a formal description of the very same properties for which mankind has yet failed to evolve and accept a satisfactory linguistic system ... We, the software makers, are contributing to the development of **application domain theories** by exposing, albeit in a painful and costly way, highly concentrated consequences of the inadequacies of available domain descriptions."[Tur86, p. 1078]

1.3.1 The domain engineering process

A reuse infrastructure must be in place for a software developer to practice reuse. A reuse infrastructure is a collection of reusable resources, appropriate for the task at hand, together with operators for locating and manipulating those resources. The structure of the reuser requires that reusable abstractions be represented (or, encapsulated) in particular ways. For example, if the target reuser is a system like Draco [Nei84], then the reusable infrastructure is encapsulated as formal grammars and libraries of tree-to-tree transformations. If the target reuser is a programmer using the Ada technology, a reusable infrastructure may include collections of Ada generics built according to DoD-STD-2167A, set in a storage and retrieval system such as [MO87], [PD87a], or [SW86]. As illustrated in Figure 1.1, the infrastructure supports and guides the execution of a reuse plan.

There exists a wide gap between the kinds and form of knowledge about a problem domain and the content and form of the information captured by a reuse infrastructure. For example, most knowledge on the problem domain is implicit and nonformal, while information captured in a reuse infrastructure must be represented formally. Bridging the applications-infrastructure gap is a difficult and expensive process. It involves identifying relevant information in the problem domain, defining a vocabulary for specifying systems in the domain, and packaging implementation knowledge in a form that is usable by a target reuser.

Given:

1. A partial, nonformal description of a class of problems, or problem domain.
2. A problem-solving task, e.g., software construction, specification analysis, fault diagnosis.
3. A reuser, the realization of a model of the problem solving task using a particular technology.

Perform:

1. *domain analysis* – the construction of a model of the problem domain to help achieve the goals of the problem-solving task.
2. *infrastructure specification* – the organization of the information in the model to fit the architecture of the reuser and the patterns of reuse in its environment.
3. *infrastructure implementation* – the encoding of that information to fit the implementation technology of the reuser.

Figure 1.2: The general domain engineering process

The bridging process must be decomposed into a system of well-defined activities and intermediate work products—a *domain engineering process*—supporting three fundamental concerns:

1. *Domain analysis*: the identification and acquisition of reusable information in a problem domain to be reused in software specification and construction. There are two aspects to domain analysis:
   - Conceptual analysis: the identification and acquisition of information required to *specify* systems in the domain.
   - Constructive analysis: the identification and acquisition of information required to *implement* those specifications.
2. *Infrastructure specification*: the modularization and organization of reusable information as required by a reuse plan; for example, libraries of subprograms, object bases, libraries of transforms, database schemas.
3. *Infrastructure implementation*: the design and encoding of the pieces resulting from the specification process using particular representations required

by the technology of the reusers; for example, procedural programming languages, object-oriented languages, source-to-source transformations, spreadsheet templates.

Figure 1.2 summarizes a *parametrized view*, of the general domain engineering process. Three parameters are identified: 1) a class of problems, 2) a problem-solving task, and 3) the technology of the reuser. Depending on the values of these parameters, different forms of the domain engineering process emerge.

1.3.2 Practical domain analysis

We summarize the essential difficulty of domain analysis by quoting from [Tur86]:

> "... an application domain, is often natural, or has a major natural component, or while man-made, has no discernible prior design. Thus, its properties are difficult to describe and the resulting descriptions are quite complex. It is because of these complexities that some software is intrinsically complex."
>
> "...Thus, software is inherently as difficult ... as science where it is concerned with the description of properties of nonformal domains."[p. 1078]

We argue against the possibility of practical procedures for capturing a "true" ontology and semantics of arbitrary problem domains. If such procedures existed they would be theory formation algorithms; i.e., formalizations of the scientific method. The history and philosophy of science show how elusive this goal is [RB60] [Dil81] [Had45] [Poi68] [Pol81] [Sup77].

We strive for systematic methods for *practical* domain analysis; that is, for the *incremental approximation* to a definition of an ontology and semantics for a problem domain. We propose that this approximation be characterized in terms of a notion of performance of a target reuser. The purpose of the approximation process is to achieve a desired level of performance.

> Difficult tasks succumb nonlinearly to knowledge. There is an even greater "payoff" to adding each piece of knowledge, up to some level of competence (e.g., where an NP complete problem becomes Polynomial). Beyond that, additional knowledge is useful but not frequently needed (e.g., handling rare cases)[LF87, p. 1173]

In this dissertation we outline a unifying domain engineering framework. A framework is not a theory; we do not offer a detailed, canonical scheme of how *every* type of domain analysis is or ought to be done. We have tried to develop a set of principles providing coherence to a diverse set of findings on the nature of the problem of domain analysis. These principles and heuristics, collectively called "domain engineering", will be used throughout the dissertation to develop methods and representations, and to demonstrate their feasibility in practice.

Given:

- a partial, nonformal description of a problem domain
- a model of a reuser as a learning system

Find: a systematic method to

1. *identify* information in the problem domain which, if available to the reuser in appropriate form, would allow it to attain a specified level of performance.
2. *capture* the information identified as relevant, and
3. *evolve* the acquired information to enhance or maintain the performance of a reuser.

Figure 1.3: The problem we are attacking

1.4 Thesis

"Pure" domain analysis is an instance of the theory formation problem routinely confronted by scientists or systems analysts. We propose a new formulation for the problem of "practical" domain analysis. The new formulation, distinct from "Which are the appropriate, reusable abstractions in a problem domain?" [DoD86][Nei81], is

> *How is a set of abstractions incrementally evolved to achieve a specified level of performance with a given target reuser?*

This, in contrast to the old question, is a thoroughly practical problem (Figure 1.3). By adopting this new viewpoint we move towards operational [2] definitions for practical domain analysis, and away from the view of domain analysis as an art—"a domain analyst is a machine for turning coffee into reusable information"[3].

> *Practical domain analysis methods must be based on a view of reusers as learning systems. On such a foundation, the solution to the practical domain analysis problem is a method for the systematic evolution of the reuser's infrastructure, to attain and maintain a desired level of performance.*

[2] A goal is operational for an agent in a particular environment if that agent, by performing some computation using the available information, can determine whether the goal is achieved [DB].

[3] Paraphrasing Paul Erdös, "A mathematician is a machine for turning coffee into theorems".

This view shares with the conventional view the assumption that problem domains exist, and the overall goal of identifying appropriate, reusable abstractions in problem domains. It differs from the conventional view in that it makes possible the development of systematic methods for practical of domain analysis.

Evolution of this research

This research was originally motivated by our work on the reuse of domain-specific information in the context of Draco [Nei84]. Our early experiences in developing Draco-domains demonstrated the need for practical methods for domain analysis.

A study of research done on the capture of problem-specific information in such areas as software specification, conceptual modeling and expert systems revealed the usefulness of an evolutionary view in the study of the domain analysis problem.

The insight that reuse systems must be modeled as learning systems indicated the need for representations and methods to record and enhance that learning process. In particular, it prompted us to make explicit the purpose, the content and the form of the information used at each step in the reuse process and its effects on the performance of the reuser.

In this research we have borrowed both from the methodological experience accumulated in software engineering, as well as from the growing understanding of the processes of knowledge acquisition and learning.

Our viewpoint

The attack on the problem of domain analysis we propose is biased by our goal to produce practical methods and by our software engineering background. We have developed some theoretical aspects of domain analysis to provide a foundation to the principles and assumptions embodied in the methods. The engineering paradigm—controlled approximation to satisficing solutions—is at the core of our conceptual framework. Finally, our software engineering background led us to model an infrastructure development cycle after the established software development cycle.

Other useful approaches to the problem of domain analysis could be proposed from other perspectives. For example, one could envision radically different proposals from an artificial intelligence perspective.

Summary of research hypotheses

The two fundamental assumptions of this work are that reuse in software construction is useful, and that domain-specific approaches are needed—we conjecture that there exists no universal characterization of the information that could be valuable to a reuse algorithm.

This research is based on the following hypotheses: 1) problem domains exist; 2) problem domains evolve gradually; 3) there are communities that need to develop large numbers of similar software systems within those domains; 4) there is expertise

in building systems in those domains; 5) reusers follow systematic approaches to reuse. These hypotheses are discussed in detail in Chapter 2.

The class of problem domains and reusers found in most real-life reuse situations satisfy these constraints.

1.5 Results

This work contributes to the practice of developing reuse infrastructures by providing:

- A domain engineering framework for the generation of reuse infrastructures, which distinguishes between the permanent and changeable components in a domain analysis process.
- An analysis of reusers from a systems viewpoint, and a classification of reuse systems based on the conceptual model of reuse they realize.
- An operational definition of practical domain analysis as a process of enhancing the competence and efficiency of a reuse system.
- Systematic methods for enhancing the competence of a reuse system through the evolution of a model of the problem domain.
- Systematic methods for enhancing the efficiency of a target reuser by the explicit modelling of patterns of reuse in its environment.

Methods and representations are demonstrated in the context of first-order reuse—the reuse of software components—in the domain of disk drivers.

The domain engineering framework should be judged on how well it describes real-world reuse situations and, in particular, on the insights it provides on the practice of domain analysis. The conceptualizations offered are an attempt to provide structure and a vocabulary to the field. At this stage, they can only be judged in the abstract on their soundness and coherence.

The absence of other domain analysis methods excludes the possibility of comparative analyses. The methods can be evaluated by how well they meet pre-defined requirements, and on the results of their application in practice. A domain analysis method should be systematic, incremental, and have an explicit scope of application. Further, it must be independent of the problem domain, and of the technology used to implement the target reuser.

The practicality of methods depends to some extent on having mechanical support for laborious, routine tasks. At this point, the lack of support environments is compensated by including people in all aspects of the problem-solving process. Hence, practicality is achieved at the price of a loss in traceability.

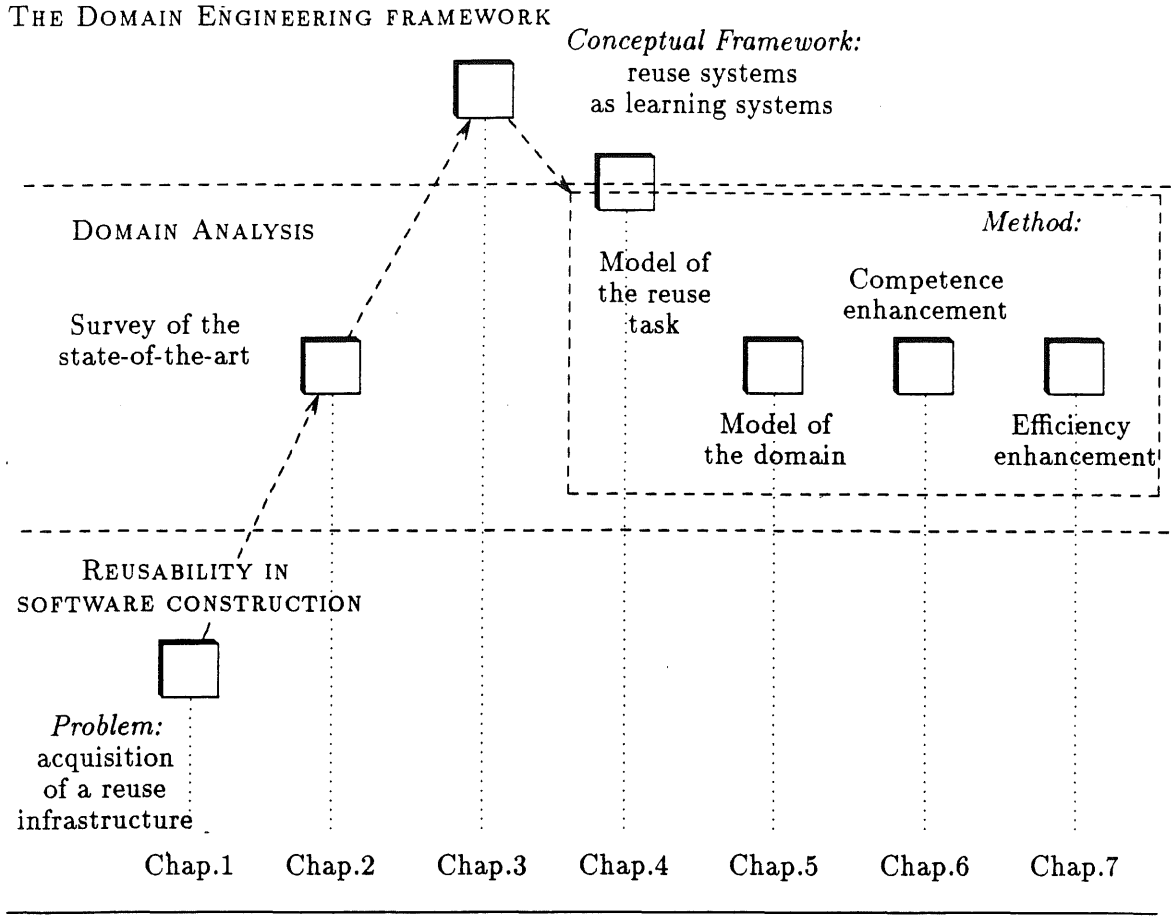


Figure 1.4: Road map to the dissertation

1.6 Outline of the dissertation

Figure 1.4 summarizes the organization of the dissertation. The dissertation discusses issues at three levels: reuse in software construction, domain analysis and the domain engineering framework.

Our research problem is at the reuse level; i.e., a precondition to perform reuse-based software construction is to have appropriate, reusable information. In this dissertation we argue that to satisfy that precondition, new methods—currently not available within Software Engineering—are needed. The study of these methods defines the Domain Analysis level.

The Domain Engineering Framework provides the conceptual foundation upon which the domain analysis methods are based.

Chapter 2 provides a context and motivation for the activity of domain analysis. Experiences in domain-specific reuse are examined to illustrate the issues involved

in identifying and capturing reusable information. A conclusion of Chapter 2 is that the state-of-the-art in domain analysis has little to offer in terms of a theory or practical methods.

Chapter 3 presents a conceptual framework for the study of reusers as learning systems. A basic vocabulary is introduced by describing reusers as systems. Learning takes two major forms: improving the competence of a reuser through the acquisition of reusable information, and improving its efficiency through an appropriate organization of the information. A domain analysis methodology for software construction is outlined. The framework identifies four parts in the definition of a domain analysis method: 1) a model of the reuse task, 2) a structure for a model of the problem domain, 3) methods for populating this structure with information, and 4) methods for specifying reuse infrastructures from that information. The balance of the document discusses in detail each one of those tasks for a particular model of reuse.

In the Figure, the topic of Chapter 4 is shown as crossing the boundary between the Domain Engineering level and the Domain Analysis level because it has a theoretical component and a component that is part of the Domain Analysis method. The theoretical component outlines a view of reusers in terms of Orders of Reuse. This view characterizes compositional and transformational reusers within a single framework and helps to make explicit the scope of the domain analysis methods. A first-order compositional model of software reuse is then selected to focus the presentation of the method.

Chapter 5 discusses a representation for models of domains to capture the semantics of reusable components for first-order reuse. Other models that could be used for second-order reuse, or in the context of other tasks are briefly surveyed.

Chapter 6 presents a family of methods for evolving the information in a first-order model of a domain. The methods realize a form of competence learning through a model-driven approach to information acquisition and the incremental induction of a domain-specific language.

Chapter 7 focuses on the organization and packaging of reusable information to enhance the efficiency of a reuse system. An analogy with the management of a memory hierarchy is developed, and the problem of packaging components is presented as a problem of defining a reuse *working set*. A precondition for defining the packages in a reuse working set is an objective characterization of patterns of reuse. A characterization of patterns of reuse as probabilistic classification trees is proposed, and a conceptual clustering method for generating the trees is discussed.

To facilitate the presentation of the ideas we have kept separate the reuse, domain analysis and domain engineering levels. This separation is apparent in the organization of the Chapters (Figure 1.4). The survey of the state-of-the-art in domain analysis in Chapter 2 motivates the DE framework discussed in Chapter 3. The methods derived from this framework are presented in Chapters 4 through 7. The reader interested in the discussion of methods may skip over Chapters 2 and 3

in an initial reading.

Chapter 2

A Context for Domain Analysis

Chapter summary. The problem of domain analysis is presented against the backdrop of reusability in software development.

Domain analysis takes place in the context of domain-specific approaches to software construction. Domain analysis is the front-end activity in the process of bridging the gap between the nonformal descriptions of problems in a domain, and the formal description of reusable components to support some specific task such as software construction.

A survey of domain-specific reusers reveals that: 1) domain analysis is a key enabling technology for domain-specific reuse; 2) domain analysis has the characteristics of a learning process; 3) domain analysis is driven by some model of the reuse activity; 4) the boundaries of a problem domain are nonformal and subject to negotiation; and 5) there are no systematic methods for domain analysis, nor is there a theory of domain analysis.

2.1 An historical sketch

Summary. Reusability gained stature as a topic of software engineering research only recently. With few exceptions, past research on reusability has focused on the problems of encapsulating information and on the operational aspects of reuse. Domain analysis is now being acknowledged as a key missing component in the family of technologies needed to make reusability practical.

The concept of software reusability can be traced back to the begining of computer programming—subroutine libraries were proposed for the EDSAC computer at the University of Cambridge in 1949—and it saw the light as a software engineering topic at the 1969 NATO conference that is usually regarded as the time and place where software engineering began. On that occasion, M. McIlroy [McI76]

called for the development of factories of reusable components. Little progress was made until the mid- to late-1970, when some organizations put to a test the promise of productivity through reusability [RL79]. In the late 1970's, the Ada initiative by the US Department of Defense, identified among the language requirements, support software reusability [Ich83] [DoD86].

In the early 1980's software reusability regained stature as a topic of software engineering research [Fre80]. The Workshop on Reusability in Programming sponsored by ITT in 1983 represents a major milestone in that process. At that Workshop, Freeman [Fre83] reported on a broad-spectrum research program for *reusable software engineering*. The early research on reusability focused on methods and mechanisms to operationalize reuse, on the representation of reusable components and on the organization of repositories of components. Comprehensive surveys are available from [ITT83], [Fre87b], and [Big88].

A significant step in this area was Neighbors [Nei81] on the Draco system, which introduced the term *domain analysis* to refer to the activity of surveying a class of problems with the purpose of developing problem-oriented specification languages. For some time the expression was used by a limited number of specialized research groups. In retrospect, the common thread among these groups appears to be the shared goal of automating some software development task. Some of these are, for example, domain-specific automatic programming [Bar85], specification analysis [Fic85] [Fic87a], application generator generators [Nei84], software portage [ABFP86]. These projects shared a prevailing view in the expert-systems area that mechanical reusers perform well when they work on restricted classes of problems and have access to problem-specific knowledge.

These experiences uncovered the dominant role of domain-specific information in the definition and organization of reusable resources. The term *domain*, used informally as a synonym for a class of related problems, was later adopted by the developers of libraries of reusable software components.

The importance (and difficulty) of identifying "appropriate" reusable resources soon became clear as a larger segment of the software engineering community (hereafter called the *reuse community*) attempted to realize the promise of reusability. One of the recommendations from the 10^{th} Minnowbrook Workshop on Software Reuse in July 1987, suggests "concentrating on specific applications and domains (as opposed to developing a general reusability environment)"[McG87]. The Rocky Mountain Workshop on Software Reuse [rmi87] acknowledged the lack of a theoretical or methodological framework for domain analysis [rmi87, p. 14].

Programming Languages, Databases and Artificial Intelligence

The identification and acquisition of relevant knowledge for the specification and implementation of software systems has been a concern of several disciplines such as artificial intelligence, semantic data modelling, requirements analysis in software engineering or the design of specification or programming languages. Each one of

these disciplines puts emphasis on different aspects of the problem of conceptual modelling. [BMS84] offers a fine collection of papers on the differences in perspective.

The task of defining reuse infrastructures sits at the intersection of these disciplines. Our perspective is somehow different from those of artificial intelligence, databases or programming language design. It is strongly colored by our goal: reuse-based software construction. Still, a discipline of domain analysis will greatly benefit from the experience and conceptual developments in those areas.

2.2 Domain-specific reuse

Summary. In preparation for examining some experiences in domain-specific reuse general issues are discussed. There appears to be a trade-off between the generality of the reusable information and the leverage it provides in software construction. The connection between this practical trade-off and research on extensible languages is discussed.

Research on the role of domain-specific information in software development is not new. It has a long and distinguished tradition in the evolution of computer languages, as well as particular kinds of software, such as application generators or expert systems. It has been restated many times in the literature: "... We have known all along that every program is in some sense, frozen domain knowledge" [Abb87, p. 666]. The process of software construction

> "appears to consist of a rich inventory of methods applied at various times and at various levels of abstraction. These methods appear to span a cascade of knowledge systems from the problem domain to the programming domain, and to employ knowledge and representations from various appropriate modeling domains"[Sta73].

2.2.1 Power versus Generality

Technologies for reuse in software construction fit in a spectrum ranging from the domain-specific to the general, Figure (2.1 has been adapted from [BR87].) Generality informally refers to the kinds of problem domains to which a particular technology can be successfully applied. While the diagram is not based on rigorous data, the distribution of the datapoints suggests a trade-off between power and generality. Power refers to a perceived "amplification" of a person's capabilities to generate software as compared with the leverage provided by a high-level programming language. Intuitively, amplification correlates with the perceived size of the "semantic gap" between the language used by a specifier and the (executable) implementation language in which the final product is encoded.

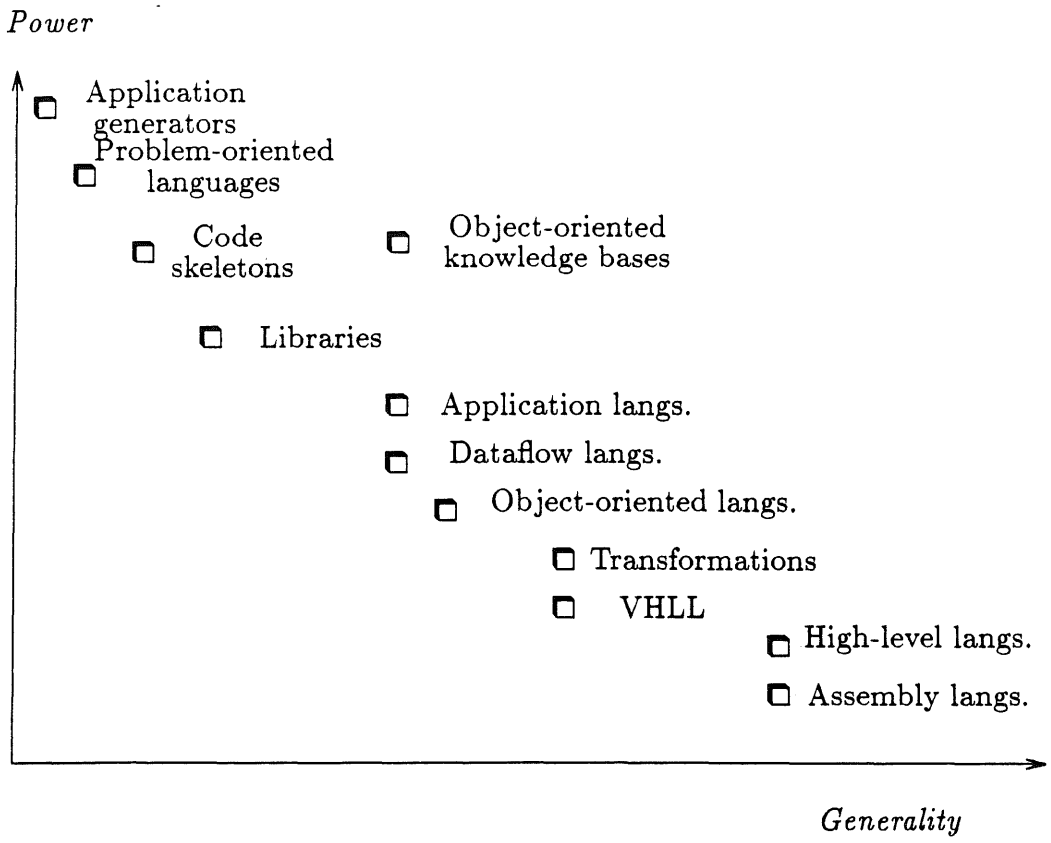


Figure 2.1: A characterization of reuse technologies [BR87] ©IEEE

For example, both compilers for high-level programming languages and application generators can be regarded as translators of specifications into executable programs. Compilers translate programs in the source language regardless of the problem abstractions encoded. Application generators have a narrow range of applicability: they can only be used to implement specifications for restricted classes of problems. For instance, parsers for a particular class of grammars, data-entry screens for some class of data-processing applications. Most commercial application generators—e.g., SBA [dJ80] [ZdJ77], Nomad [McC80], Ramis [Mat82], Dbase [AT81]—implement data-intensive applications usually involving data entry, file processing, and report generation. They offer users a problem-oriented interface and a restricted set of services; usually, database management and query languages, graphics and report generation [CG82] [HKN85]. Generators in systems programming; for example, parser or lexical analyzer generators, or generators of structured editors, are as restrictive. The homogeneity in the set of inputs to an application generator allows it to make strong assumptions about the system specifications and

about how they should be built. These assumptions are embedded in the generator.

Using the problem-space metaphor, the trade-off between power and generality results from two different approaches to reducing complexity in software construction: the compilation of macro-operators versus the restriction of the space of operators. A useful criterion in practice, has been to define restrictions to match problem domains. Recent research on wide-spectrum and extensible languages seems to be converging on the same view.

2.2.2 Extensible languages

Abstraction mechanisms in programming languages and formal specification techniques have evolved in an attempt to move programs closer to the problem domain. A trade-off in the design of software representations has always been to minimize *conceptual cost* at the stage of problem analysis and software specification by making problem aspects visible, and to minimize *execution cost* at the stage of software implementation.

In the late 1960s and early 1970s extensible languages were proposed to minimize the conceptual cost at the specification level. The aim of that research was to provide base notations to which programmers could add new notations and new data types [Sha84b]. The early emphasis was on supporting syntactical extensions to a base language found some serious obstacles which later led to a redefinition of "extension". Standish [Sta75] summarizes the reasons why the approach failed. He classifies potential extensions to a language into paraphrases, orthophrases and metaphrases. He argues that only paraphrases (extensions to suppress detail, promote conciseness or cover up for some irritating language feature) can be accomplished with a modest amount of effort by a user. Orthophrases and metaphrases, (i.e., features that are "orthogonal" to the language or that alter the interpretation rules of the language) require major surgery of the language processor, or major (and thus undesirable) deformations to syntax, conventions or style.

The so-called "failure" of extensible languages left behind a rich legacy of data abstractions and process abstractions that enhance "problem visibility" in specifications and programs [Abb87] [Bor85] [CW85] [Sha84a] [Win79]. In turn, those developments lead to what is called object-oriented programming (OOP). OOP systems revive the old dream of extensible languages—the user can extend (or create) the objects needed for the task at hand. "Many applications can be designed by straightforwardly examining the problem domain, identifying the objects found there and their behaviors and deciding how to implement each behavior in the computer. As a result, the designer is not forced to restate his problem in computer-based terms ..." [Cox86]. Such promises are overly optimistic. The task of examining the problem domain to determine relevant objects and their associated behaviors is most emphatically *not* a straightforward process.

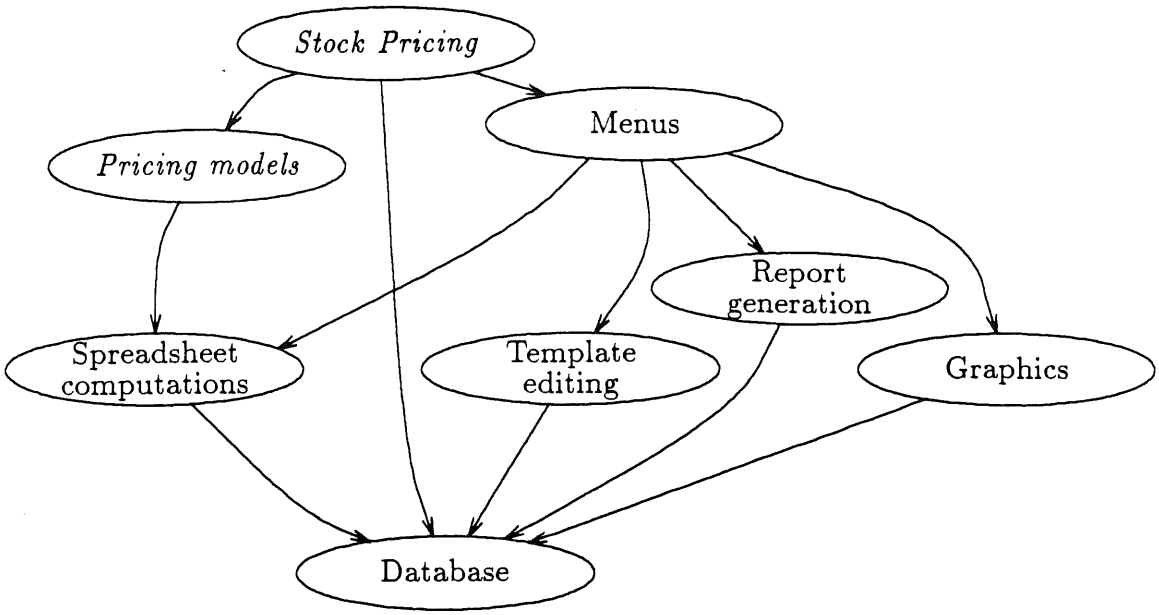


Figure 2.2: Example: A domain network for spreadsheet applications

2.2.3 Domain networks and local formalisms

The problem of creating useful extensions to languages has been recently attacked from a different viewpoint. The earlier notion of base language has been abandoned in favor of more flexible network structures of "domain languages" or "local formalisms" linked by cooperating sets of transformations.

Research on Draco [Nei84] [Fre84] introduced the notion of domain language networks which represent a trade-off between the restriction and extension views.

A Draco domain language goes in the opposite direction of "extending" a base language. The goal is to "contract" a language so that it captures only the constructions needed to specify restricted *classes* of systems with high potential for reuse. Independent domain languages are articulated by a domain language network. The relations between constructs in different languages are implementation transformations (or, "refinements" in the jargon of Draco).

Figure 2.2.3 illustrates a fragment of a domain language network. Each oval represents a domain-specific language, whose constructs are implemented using constructs from other subsidiary languages (pointed by the arrows). For instance, constructs in the Menus domain may be refined in terms of constructs in the Template Editing domain or in the Report Generation domain. This reductionistic process stops when all the language constructs used to represent a system are drawn from "executable domains". Domain networks capture the result of the analysis of classes of problems as problem-specific languages (nodes), and of implementation knowledge in the reusable refinements (arcs). Two forms of reuse are supported. The first

form is the direct reuse of language constructs in the specification and in the implementation instances of systems; for example, systems for evaluating Stock Prices. A second form of reuse occurs when the domain language network is extended in different directions. For instance, the nodes labelled *Stock Pricing* could be replaced by *Financial Accounting* and the node *Pricing Models* by *Cash Flow Models, Budgeting Models, Breakeven Analysis Models*, etc. The extension for Financial Accounting would reuse the network of subsidiary domains (nodes labelled in roman font in the Figure) in the implementation of models in Financial Accounting.

The CIP project [Gro84] implements specifications written in a wide-spectrum language, CIP-L, by using networks of (algebraic specifications of) data types, related by formal transformations.

[Wil86] proposes a similar framework to allow users to extend a language by narrowing the context in which individual extensions are well-defined. The extensions, called local formalisms, are related to one another by sets of formal transformations. Each local formalism defines an independent "semantic axis" which widens the overall spectrum of the language.

All these are attempts to overcome the practical limitations found in the *definition facilites* of extensible languages. The problem of generating orthogonal extensions to a language, is addressed by creating a flexible architecture of languages—e.g., domains, local formalisms—articulated by transformations that capture the semantics of their interactions. Still, these extensions are *not* easy to define, but their development cost could be amortized by repeated reuse.

2.3 Case studies in domain-specific reuse

Summary. Selected experiences in domain-specific reuse are surveyed. The cases discussed include the development of software engineering standards and the capture of domain-specific knowledge to support different aspects of the software development cycle. These experiences are a source of insights and illustrate important issues in domain analysis. A persistent theme, the lack of systematic methods for domain analysis, is identified.

In preparation for discussing the problem of domain analysis, we review selected case studies in domain specific reuse. The survey reveals that:

- Domain analysis is a key enabling technology for domain-specific reuse
- Domain analysis has the characteristics of a learning process
- Domain analysis is driven by some model of the reuse activity
- There are no systematic methods for domain analysis, nor is there a theory of domain analysis

- The boundaries of a problem domain are nonformal and subject to negotiation

Domain-specific approaches to reuse go beyond software construction. For example, Good [Goo82] has studied the reuse of "problem domain theories"—specifications of the functions and properties that characterize an application—to reduce the cost of software verification. His experiences with the GYPSY system [AGB+86] suggest that the cost of verification could be greatly reduced if reusable problem domain theories were developed.

Gerhart [Ger83] shows that for substantial problems, verification proofs require a collection of theories for the system model, data types, algorithms, and implementations of algorithms and data structures. The AFFIRM project calls these "domain theories". Gerhart acknowledges that their development is a challenging task requiring a large portion of the resources currently devoted to proving. Yet the theories are highly reusable if developed properly [Ger83, p. 110].

The PROUST system by Soloway and others [WLJ85] [SE84] has demonstrated how collections of domain-specific programming plans can be reused for mechanically creating explanations of and debugging Pascal programs.

We now survey some reuse projects that provide insights on the nature of domain analysis.

2.3.1 The development of the CORE graphics standard

The CORE graphics standard [NvD78] was developed to support software reuse by increasing the portability of graphics software. An ACM SIGGRAPH Graphics Standards Planning Committee was formed in 1974. Progress was slow for the first two years due to the existence of different paradigms [Kuh70] for graphics systems. "In recent years wide areas of agreement have been reached between the different schools (of thought), making it possible to consider the development of graphics standards"[NvD78, p.367]. Interestingly, this agreement resulted more from economic and market conditions—raster displays won over storage tube displays—than from consensus on a unified theory of graphics design.

A stumbling block remained. It was not the lack of a body of knowledge on graphics but the lack of what Newman and Van Dam call a "methodology", design rules or strategies based on that body of knowledge on how applications are developed. That obstacle was removed by studying "the structure of application programs in order to understand how to design software systems to support them ... the advantages of separating the essential picture-generating functions (the "core") from other functions..."[NvD78, p.372].

In summary, progress was made once, 1) competition between paradigms subsided (for reasons outside the control of the analysts), and 2) a model for the construction of graphics applications became explicit. Useful domain analysis heuristics were identified in the process of identifying a "core" of graphic functions:

1. Inclusion criteria: include only those "logical capabilities which can be easily supported by most existing graphics hardware and cannot be easily built on top of other Core System capabilities"—a notion of basis or generating set.

2. Accepted practice: accept features that are generally accepted practice, standards would not be accepted if too far ahead of the state-of-the-art.

3. Well-structuredness, cleanliness and explainability: when faced with several sets of features that implement the same semantic capabilities, select those that are well-structured and whose effects are obvious.

2.3.2 Draco—An application generator generator

The Draco system is an application generator-generator [Nei81] [Nei84] [Fre84] [Nei88a]. The contribution of the Draco project to software reusability is twofold, it demonstrates: 1) the key role of problem-specific information in the organization of reusable information, and 2) that *if good domain descriptions can be designed* then a relatively simple reuse mechanism suffices for performing software construction. The Draco paradigm for software construction can be considered a confirmation, in the field of reuse, of the aphorism: "in the knowledge lies the power." (The trade-off between the richness of the domain-specific information and the sophistication of the reuser will be discussed in detail in Chapter 4, Figure 2.3.2) The utility of the Draco technology depends on the availability of a domain analysis technology [Ara88a]. It was Neighbors who introduced the expression *domain analysis* in the vocabulary of software reusability:

> "Domain analysis differs from systems analysis in that it is not concerned with the specific actions in a specific system. It is instead concerned with what actions and objects occur in all systems in an application area (problem domain). This may require the development of a general model of objects in the domain... [Nei81, p. 20].

Neighbors did not propose a theory or technique to perform domain analysis. He did, however, distinguish between two activities: domain analysis and domain design. The design stage—what we call infrastructure design—involves casting the results of the domain analysis in a form that can be usable by the Draco technology:

> "A problem domain is a collection of objects and operations, but to specify a problem domain to Draco a few other things must be included. In particular, a domain language parser, a domain language prettyprinter, source-to-source transformations for the domain, and components for the domain must be specified to create a usable Draco domain" [Nei81, p. 40].]

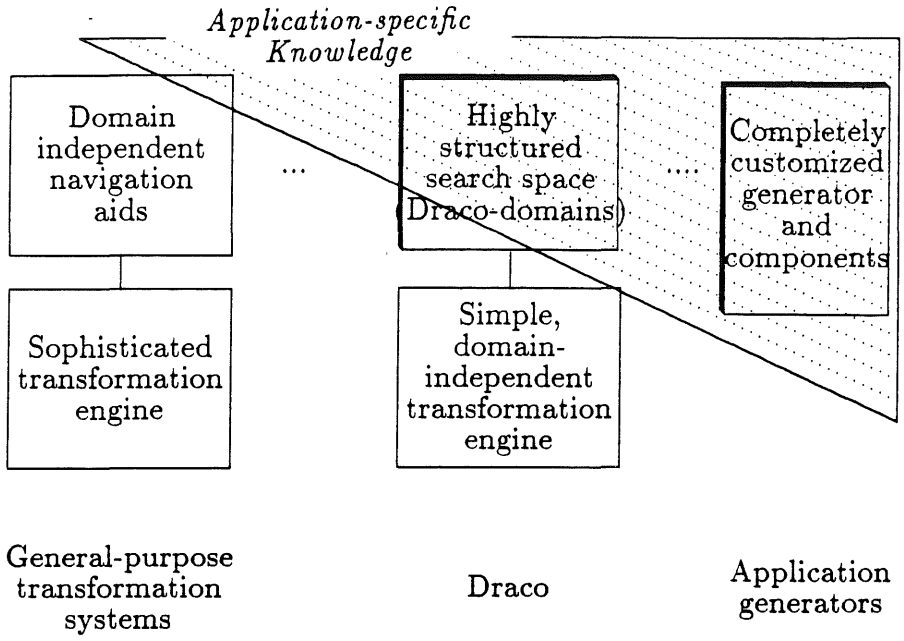


Figure 2.3: The trade-off of Draco

2.3.3 "Little languages" and application generators

Bentley [Ben86] points out that programmers and computer users in general are surrounded by "little languages." These are very restricted languages, that capture abstractions in very restricted problem domains, such as, interfaces with operating systems or devices, edition or formatting of text, organization of bibliographic references.

Programmers, explicitly or implicitly, use little languages to abstract patterns of operations that must be done again and again. Sometimes the perceived benefits justify the design of programs to process such languages. These application generators tend to reflect short-term goals of individuals rather that long-term reusability goals of a project or organization [Cle87]. They result from personal experiences that are seldom documented or captured in any systematic manner. Furthermore, they tend to result from insights spontaneously gained on the job, rather than from the systematic application of methods.

One notable exception is the development of the TEX text formatting system which is based on rigorous theoretical developments and for which there exists rather detailed accounts of the development process (for example, see [KP81] for an elegant presentation on breaking paragraphs into lines.) Evidence of the quality of the resulting product is the fact that "plain TEX" has become the basic domain language in a network of text and picture formatting languages, such as, LATEX,

$\mathcal{AMS}$-TEX, and local dialects. On top of these, extensions or "styles" are routinely built: book style, article style, report style, and so on.

Some domain-specific languages have been very successful in practice, from early examples such as the Automated Tool Programming [Pat72], or APT, language for numerical control machining, to today's so-called fourth generation languages for business applications. Such successes have stimulated interest in the development of application generators. Recognizing what domains are mature for a generator is a difficult task. Cleaveland [Cle88] suggests two heuristics: 1) has a formal or informal notation surfaced within the user community, and 2) are there identifiable patterns or regularities in the applications generated in a user community?. The existence of a common vocabulary is taken as a manifestation of a shared understanding within a community of prospective users of the application generators. Looking for things that are done again and again, is a way of identifying opportunities for reuse. No additional guidelines are proposed.

2.3.4 Libraries of domain-specific components

The indices to libraries of reusable components constitute a form of language. Building an index requires that we identify a vocabulary for describing the objects in the library and a syntactic structure for library queries. Different projects have recently advocated the use of domain-specific libraries of components to facilitate the tasks of identification and retrieval.

[PD85] [PD87a] present a comprehensive analysis of software classification. Prieto's faceted scheme classifies software components using *terms* from six *facets* which can be considered as describing different viewpoints. Three of them describe the functionality of a component—function, objects, medium—and the remaining three the environment of the component—system type, functional area, and setting. It has been observed that the functional facet provides the most valuable information for retrieval, and [PD87c] calls for the application of domain analysis techniques for generating *domain-specific vocabularies for indexing* library items.

[SW86] proposes a different approach to processing library queries in the context of the Eclipse Integrated Project Support Environment (IPSE). Sommerville and Wood argue that keyword-based techniques are insufficient as a method for describing software components. They propose libraries including six types of objects: functions, procedures, declarations, objects, abstract data types and subsystems. These objects are catalogued using *software function frames*. In contrast with Prieto's keyword-based techniques, catalogue entries are described using Schank's Conceptual Dependency Graphs. A query to the collection is an English description of the problem using a *problem domain vocabulary*. A domain vocabulary is needed to enable a successful matching against the descriptors in the collection. No suggestions on how to define domain vocabularies are given.

[Cam84] outlines a proposal for a major library construction project. The purpose of the CAMP Project (Common Ada Missile Packages) was to demonstrate

"the feasibility and value of reusable Ada parts in real-time, embedded, mission-critical, DoD applications" as well as to test the existence of commonalities among systems in a real-time domain; in this case, missile guidance and navigation.

The study, conducted by McDonnell Douglas Co. under the sponsorship of the US Air Force, resulted in the specification of a library of reusable Ada parts for missile flight applications. The specifications of ten missile flight software systems were analyzed to identify commonalities. The analysis resulted in the identification and architectural design of some 250 to 300 parts that were subsequently implemented. [Cam87] classifies the components produced into: missile operation parts (navigation, guidance and control), general operation parts equipment interface parts, mathematics (vector and matrix algebra, geometric operations, etc.) and data types.

A process of cataloguing and testing of the library of parts is under way. The project reportedly adopted a systematic approach to the analysis of the missile navigation domain. The project documentation is classified.

2.3.5 Specification acquisition in KATE

The traditional paradigm for requirements analysis views the process as one of translating of user intentions, rephrasing them in terms of a language in which properties such as internal completeness or consistency can be assessed by syntactic means. Fickas [Fic87a] [Fic85] proposes an interactive approach in which both users and analyst are involved in supplying parts for the requirements document. This approach presumes that the analyst has *domain knowledge,* which allows for a more thorough, semantic analysis of the user's intentions. An automated analysis assistant, KATE, has been built.

The development of KATE makes an interesting case study from the point of view of domain analysis because it illustrates some important issues: the influence of the model of the task in the definition of an ontology of the problem domain, the "Crack Analyst" approach to knowledge acquisition, and an unavoidable learning process.

A model of the requirements analysis task explicitly defines the categories of information needed by the system: 1) a conceptual model of the problem domain (common objects, operations and constraints), 2) a model of how the environment affects the embedded system and 3) a model of typicality in the domain—*usage scenarios* [Fic87a].

Domain-specific information for the system was collected using the "Crack Analyst" approach, in which a very capable individual with a deep understanding of the needs of the particular reuse process, single-handedly identifies and acquires all the domain-specific information needed to support reuse. A strategy derived from the analysis of interviewing protocols was applied to elicit information from experts. The strategy involves: 1) "short answer" questions, 2) example-based questions to support problem acquisition, debugging, general argumentation and negotiation,

and 3) summarization-based questions—to reach agreements between analyst and expert and to trigger new lines of acquisition and analysis [Fic87b].

The development of the KATE assistant illustrates a broad learning process, including the evolution of: 1) a model of the reuse-based task, 2) the kinds of reusable information considered relevant, and 3) the implementation technology of the system. KATE grew in several generations [Fic87a]. The Critic-1 of a first-generation KATE performed a critique of a specification based on a conceptual model of the domain and known examples. Experimentation with that system uncovered the need for an explicit representation of typically undesirable situations. A more advanced Critic-2 was developed, and the model of the domain was augmented to distinguish between good specification components and bad specification components. It later became apparent that specifications are bad, only in relation to particular goals or policies. Thus, the ontology of the problem domain was extended to include policies, and relations between basic concepts and policies. A more sophisticated Critic-3 was developed to use the new types of information. Concurrently with this process of revision, different implementation technologies were employed in each generation: a production language, an object-oriented database, and finally a sophisticated expert-systems prototyping environment.

2.3.6 XPLAIN and EES—Explainable Expert Systems

Some reuse systems operate on several kinds of domain-specific knowledge, and this poses an even greater challenge to domain-analysts. The construction of explainable software is a case in point. One way of explaining the behavior of programs is to paraphrase the code of the program or traces of its execution into English. This kind of explanation provides justifications for *what* the program does but not for *why* it is reasonable to do so. Answering Why-questions requires knowledge of the problem domain—goals, appropriate problem-solving heuristics, etc. and such knowledge is normally not available as part of the program code.

The Explainable Expert Systems project, EES [NSM85] and XPLAIN [Swa83] are examples of a paradigm for developing diagnosis expert systems that can answer Why-questions. XPLAIN and EES demonstrate the reuse of domain knowledge for the double purpose of construction of a system and the explanation of its behavior to support validation.

To enable explanation, the reuser must have an explicit representation not only of the components it uses to construct the system but also of the methods used in the construction, and of the reasons justify the application of the methods. Swartout's XPLAIN system adopts a formal model of the task of developing an expert system—refinement by a hierarchical planner. XPLAIN maintains two separate forms of domain-specific information: descriptive information on how the domain works, and "domain principles", how to employ its domain descriptive information to achieve particular goals such as disease diagnosis or therapy administration. In EES, domain principles were enhanced to represent problem-solving strategies that are used by

the program writer to drive the refinement process. Associated with the principles there are "trade offs" and "preferences" that guide the application of the principles.

EES and XPLAIN are examples of systems that perform reuse of domain-specific information at separate levels: factual information of the problem domain, plans for the generating diagnosis rules using that information, and goal structures to guide in the application of plans. There exist no systematic methods for acquiring information for such "higher-order" reusers (Section 4.2). The difficulty of the task is compounded by the need to uncover the interactions among information at different levels. To compensate, such systems operate on very restricted, well-understood problem domains.

Summary

In summary, practical domain analysis methods are a precondition for domain-specific reuse. Learning and negotiation are essential aspects of the process. There are no theories or systematic methods for performing domain analysis.

2.4 The problems-infrastructure gap

Summary. To make domain-specific reuse possible, we must bridge the conceptual and representational gap between partial, nonformal descriptions of a class of problems and collections of (formal) reusable components for software construction. We focus on a front-end activity of the bridging process: domain analysis. Since the available definitions of the task of domain analysis are non-operational, we propose a new operational definition.

2.4.1 Bridging the gap—An example

There exists a wide gap between the form in which knowledge about a problem domain is found in real-life situations, and the form and organization of reuse infrastructures. We illustrate aspects of the gap with an example. Let us consider parsing sentences for a subset of a natural language; that is, recognizing sentences such as "John washed the car" or "Did the red barn collapse?" that could be used to query a natural language data base.

The previous sentence is typical of intensional, incomplete definitions of a problem domain, and presumes quite a bit of shared knowledge on the part of the reader. To make the jump to a reuse infrastructure dramatic, Figure 2.4.1 illustrates fragments of a reuse infrastructure for that problem domain, in the Draco system [Nei84].

The parsing problem is solved by means of augmented transition networks (or, ATN). An ATN language is shown (lower right-hand window) as one node in

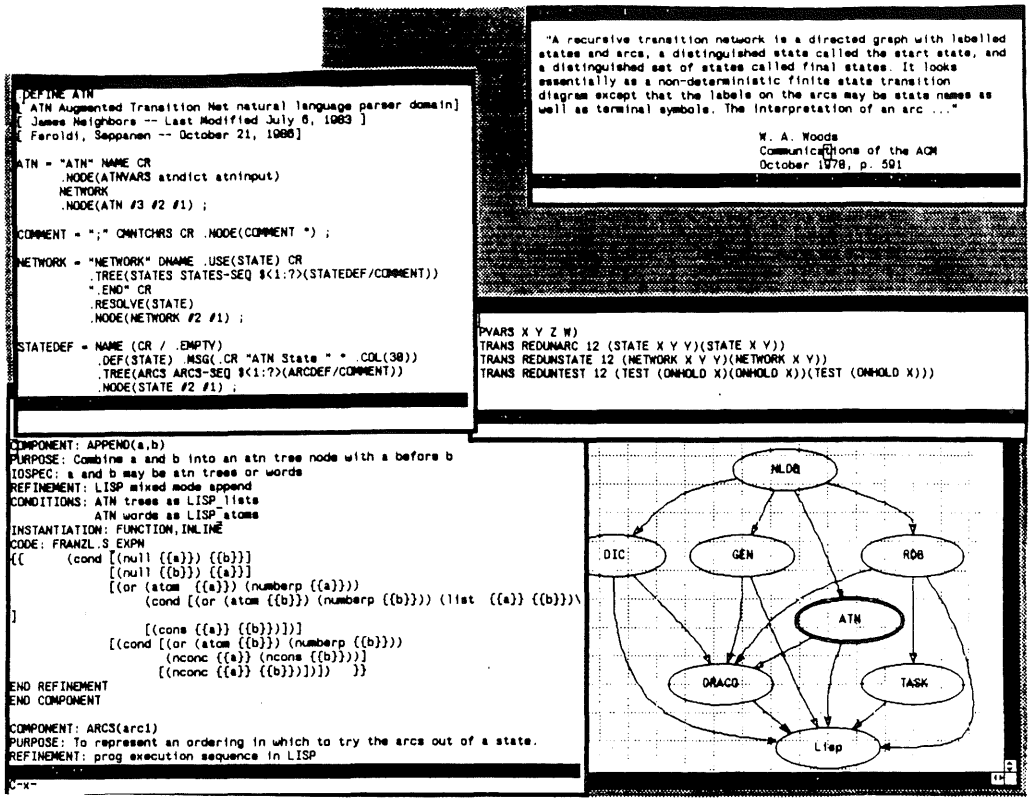


Figure 2.4: The applications-infrastructure gap

a domain language network, which also includes languages for the specification of natural language interfaces (NLDB), dictionary structures (DIC), relational data base operators (RDB), etc. The remaining windows in Figure 2.4.1 show fragments of a a definition of the ATN domain: a parser description for the ATN language (ATN.DEF window), of Draco-refinements (ATN.REF window) and of Draco-transformations (ATN.TFM window). The Draco system, reusing this infrastructure—parser, refinement and transformation libraries—can mechanically implement specifications encoded in the ATN language.

The natural question is: How were all those pieces derived from the statement of the problem domain? The gap is spanned by the following steps:

1. the precise description of problems in the domain,
2. the description of methods for solving those problems,
3. the formalization of software specification and implementation knowledge to meet the requirements of the technology of the target reuser,

4. the encoding of that information into a reusable form (resulting in the representations shown in the Figure).

Typically, steps 1 and 2, the *domain analysis* steps, involve surveying the sources of expertise in the problem domain. In the example, [Woo70] is a classic reference; an excerpt from Woods' description is shown in the upper right corner of Figure 2.4.1. Woods must be credited with the theory formation step. Step 3 is a design process constrained by the technology of the target reuser. Neighbors [Nei81] calls this step "domain design". Step 4 is a software implementation process constrained by the particular technology of the target reuser.

Bridging the problems-infrastructure gap is a difficult and expensive process. Evolving an infrastructure can be as difficult. Our experience shows that without a justification of how the infrastructure was derived (i.e., a blue print of the bridge) understanding and evolving a reuse infrastructure is a hopeless task, comparable to debugging or maintaining low-level code without having specifications or design documents, and only some general statement of the requirements of the system.

2.4.2 A critique of current definitions for domain analysis

The commonly held view of domain analysis is that

> "A domain analysis is an attempt to identify the objects, operations and relationships between what domain experts perceive to be important about the domain" [Nei88b].

Most approximations to a definition of domain analysis derive from Neighbors' original. While they provide some intuition of what domain analysis is about, they are insufficient from the point of view of the practice of domain analysis:

- they are not operational definitions,
- there is no clear statement of goals, nor a termination predicate that could answer the question: "Are we done with the analysis of this domain?"
- they provide no measures to help assess progress in domain analysis.

To overcome these limitations we must adopt a different viewpoint towards the problem. An operational formulation of the domain analysis is developed in Chapters 3 and 4. In preparation, we summarize our research assumptions.

2.5 Assumptions about the nature of the problem

Summary. We summarize the research hypotheses which are the foundations of this work. These hypotheses strongly influence the methods discussed in later chapters. Four hypotheses are listed: 1) the existence

of real-life problem domains with specific properties, 2) the persistent but gradual evolution of problem-domains, 3) reuse-intensive software construction environments, and 4) the availability of sources of expertise in the problem domain.

The "Domain" hypothesis

A problem domain is defined by consensus, and its essence is the *shared understanding* of some community. As a result of this consensus a language with shared semantics may emerge. This is a precondition for domain analysis. One of the goals of domain analysis is to make that language and its semantics explicit. We now summarize the intuitive notion of *problem domain* [Sha77, p. 525]:

> In a given community, items of real-world information come to be associated as bodies of information or problem domains having the following characteristics:
>
> 1. deep or comprehensive relationships among the items of information are suspected or postulated with respect to some class of problems,
> 2. the problems are perceived as significant by the members of the community, and
> 3. there exists the knowledge to produce solutions to the problems.

No restrictions are imposed on the number, complexity or kinds of problems, nor on the amount, kinds or level of detail of the real-world information involved.

The "Gradual Evolution" hypothesis

Software artifacts are models of some aspect or process in the world and "must, therefore, be changed to keep pace with the needs and the potential of changing environment" [Leh80, p. 216]. The empirical laws of software dynamics [Leh80] [Leh81], apply to reusable infrastructures:

> Law of continuing change. A program that is used and that, as an implementation of its specification, reflects some other reality, undergoes continuing change or becomes progressively less useful.
>
> Law of increasing complexity. As an evolving program is continuously changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain it or reduce it.

We conjecture that evolution is gradual, that in most cases a reuse infrastructure can be incrementally adapted to match changes in the reuse patterns in the environment

and on the kinds of information that are considered relevant in the domain. On the average, the cost of engineering a reuse infrastructure must be amortized over an extended period of time [BDGP87]. As a consequence, infrastructure "maintenance" becomes unavoidable. We conjecture that evolution will be a dominant economic factor in the development of reusable components.

The "Large Scale of Reuse" hypothesis

Initial costing models of reuse (e.g., [BDGP87]) suggest that large numbers of systems must be developed to realize the potential of reusability for improving the productivity. "A domain analysis is only useful if many similar systems are to be built so that the cost of the domain analysis can be amortized over all the systems" [Nei81, p. 141]

Hence, we assume that some communities have a need to develop large numbers of similar systems within certain application areas.

The "Knowledge Engineering" hypothesis

We propose to restrict the domain analysis activity to the capture of *available* expertise. In this sense, domain analysis is an instance of knowledge engineering [Fei77].

There are multiple sources of information about a problem domain—people, textbooks, journals, existing software systems. People are the only known agents who are consistently successful in blending together problem-specific and software engineering knowledge in software construction.

We assume the existence of experts in the problem domain who can produce examples of system specifications and who can provide a rational reconstruction of the implementation of those specifications. Boehm's observation [Boe87] that people are one of the most significant factors on the quality of the software development process squarely apply to this case: the quality of the results of the domain analysis process depends on the quality of the domain experts.

We postulate the need for a role, the *domain analyst*, who conducts the process of eliciting, representing, and analyzing information obtained from experts. In the case of complex problem domains, expertise may be distributed among many people. In such cases the domain analyist may need to interact with more than one expert and to articulate the inputs received from them into a consistent domain network. Our view differs from Neighbors' [Nei81] who assumes that expert and analyst are one and the same, i.e., the Crack Analyst view.

2.6 Summary

This chapter discussed the context and motivation for the study of domain analysis and the assumptions on which we base our work.

Domain-specific information drives many aspects of the process of software development. We have surveyed some instances of domain analyses to support different aspects of software construction. With the exception of the preparation of community-wide standards, domain analysis has been performed singlehandedly by very capable individuals. *No methods have been reported.* There is consensus in the reuse community on the need for a theory of domain analysis, and for practical methods.

Chapter 3

The Domain Engineering Framework

Chapter summary. Our survey of the state-of-the-art of Domain Analysis revealed a lack of theory or practical, systematic methods for doing domain analysis. A Domain Engineering (DE) framework provides a foundation for an operational definition of the task of Domain Analysis.

Actual reusers are defined as systems. A vocabulary for the description of reusers is introduced. It will be used throughout the dissertation. The DE framework views reusers as learning systems. This view makes possible an operational re-formulation of Domain Analysis as the *incremental evolution of a Model of a Domain to attain and maintain desired levels of performance for a target reuser.*

We outline a DE strategy for arriving at a domain analysis method:

1. make explicit a model of the reuse task,
2. define a structure for a model of the problem domain,
3. define methods, based on 1. and 2., for populating this structure with relevant information, and
4. define methods for specifying reuse infrastructures from the information in the model.

This strategy is demonstrated in detail in Chapters 4 through 7.

3.1 Definition of a reuse system

Summary. Reusers are described as directed-systems—systems with identifiable input and output variables—embedded in a reuse environment. This characterization provides a basic vocabulary to describe

reuse systems that is used in the balance of the document. This definition is extended to learning systems in the next Section.

Reusers are embedded in a *reuse environment.* For the purpose of this discussion, a reuse environment is an organizational context in which information is systematically acquired and reused in software construction under the control of some management guidelines.

3.1.1 The reuse system

A reuse system P, (Figure[1] 3.1) is described by a set of primary traits [Kli85], abstracted by state variables:

- a model of the reuse task, MoT
- a pool of technologies to support reuse, T_R
- a reuse infrastructure, RI
- a class of specifications, S
- a class of implementations, I_S

A *Model of the reuse Task* is defined as a rigorous characterization of how reuse is performed, including a specification of the form of the reusable objects, of operators for manipulating those objects and their applicability conditions, and of a control strategy for achieving the goal of implementing a specification through reuse. Ideally, the value of the MoT would be an algorithm; for example, Draco [Nei81]. In practice, people are involved in the reuse process, and the value of the MoT becomes a combination of algorithms and informal guidelines, as illustrated in the example below.

The model of the task is realized by a set of support technologies, T_R. These may include people, languages, tools.

A *reuse infrastructure* is defined as a collection of reusable abstractions that can be instantiated to specify and implement systems in a particular problem domain. The reusable information must be such that, 1) it is of the type required by the MoT, and 2) it is encoded in a form that allows for manipulation by the technology of the reuser. A reuse infrastructure, RI, could take many forms depending on the values of the MoT and T_R. For instance, a formal grammar and a library of formal transforms for the Draco system, or, for a programmer, a library of catalogued Ada generics built according to DoD-STD-2167A. The formal transforms could be appropriate, in principle, for a programmer, but humans are not well-equipped to

[1] Figure 3.1 is a relabelled version of Figure 1.1. The new labelling will be referred to throughout the dissertation .

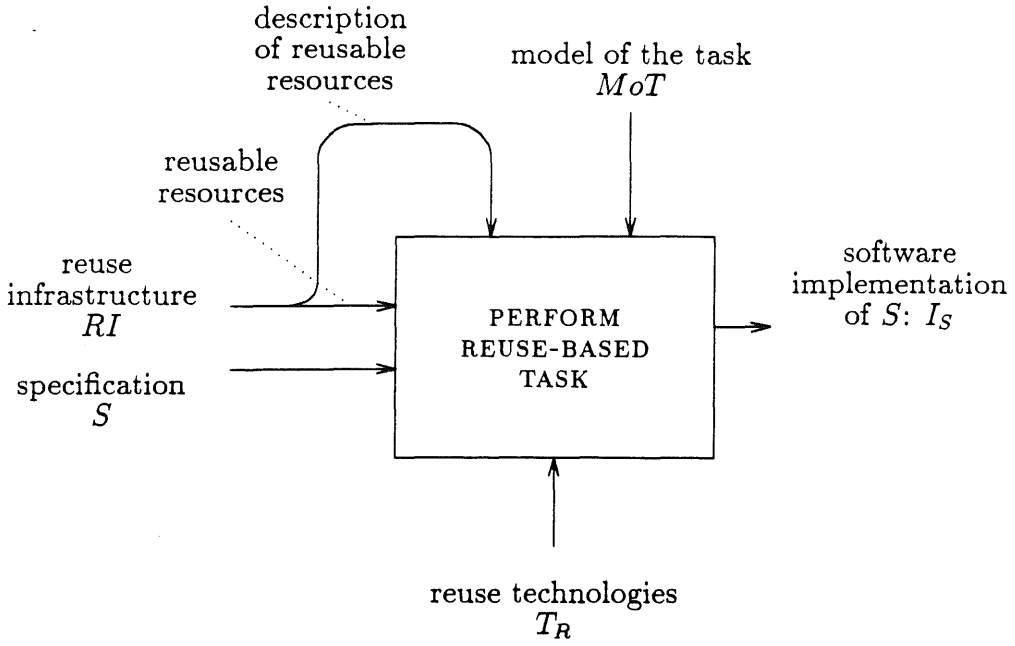


Figure 3.1: The reuse system $P = \langle MoT, T_R, RI, \{S, I_S\}\rangle$

manually apply (thousands of) them efficiently. Thus, in practice, a library of transforms would not be an appropriate reuse infrastructure for a programmer.

We illustrate these definitions with an example.

Example—The GTE Assets Library System

The environment of the GTE Assets Library System [PD87b] is GTE Data Services, in which a management structure, the Assets Management Program, has been set up to create, maintain and make available a collection of reusable assets; in particular, software components. The task to be supported is reuse-based software construction.

A "Help Support Group" in charge of training provides guidelines for reuse. The Model of the Task is typical of the reuse of libraries of components:

1. Encode specification as a library query using a faceted scheme [PD87a].
2. Submit query to the library system.
3. Select a "best candidate" from the set of candidate components with the help of thesauri and heuristic metrics describing conceptual distances between terms in each facet.
4. Adapt (i.e., a programmer "copy-and-edit"s) the best candidate to fit the structure of the system under construction.

5. Integrate the adjusted component into a system.

The second and third steps are well-defined. The remaining steps are performed by people under programming guidelines or reuse guidelines and on the basis of their own judgements. These steps are not formal, and little support for reuse can be offered.

The supporting technologies, T_R, include software engineers, programmers programming languages, and software development tools.

The reuse infrastructure, RI, is a collection of software components. The part of T_R that supports the management of this collection (we might call it T_{ri}) is a relational data base management system on a personal computer. RI could be further decomposed into a Catalogue part and a Shelf part corresponding to the two aspects of the Assets Library. **End of example**

Figure 3.1 represents the reuser as a performance system P. Features that are sufficient to characterize a reuse system are abstracted in terms of state variables. P is a directed system, i.e., the set of its variables of a directed system can be partitioned into two categories: input variables (RI, MoT, S, T_R) and output variables (I_S, implemented software systems). Output variables are those whose value is determined from within the system, $I_S = MoT(S, RI)$. The *state of a reuse system* is characterized by the value of each of the state variables.

Instances of reuse systems can be compared by comparing the values of their state variables. Time is used to distinguish between different observations of the same attribute of a reuse system. Variables may be decomposed into sub-variables representing more or less independent attributes of the reuse situation. For instance, the T_R in the previous example can be decomposed into T_r (with values such as, programmer, application generator), T_{ri} (with values, classification scheme, relational database, thesauri), T_s (with values, library query language). This decomposition asserts that T_R is a (neutral) system, whose state variables are T_r, T_{ri}, T_s. In [Ara88a] we referred to these as *assemblies of technologies.*

3.1.2 The environment

The factors that determine the values of the input variables of the reuse system constitute its environment. We characterize the environment of a reuse system in terms of four features:

- The reuse task—e.g., software specification, specification analysis, software construction, implementation maintenance, fault diagnosis. These tasks may be realized in different ways. The MoT makes explicit the model of reuse applied.

- The problem domain, characterized in terms of, 1) a set of information sources and 2) a collection of instances of problems. Sources and instances may change

over time. The reuse infrastructure must evolve to match the evolution of the problem domain.

- Available technologies. We distinguish between technologies to support the execution of the actual reuse task, the acquisition, and the management of reusable resources. The pool of technologies to support actual reuse include library systems, programming languages, transformational systems, application generators. Technologies to support the management or resources include library systems, version control systems, software maintenance aids, and others. The technologies to support acquisition are yet to be defined.

- Cost-benefit models. This is an underdeveloped area in reusability, in part because of the lack of enough empirical data. [BDGP87] is a first attempt to define the conditions under which it is economical to invest resources in the development of an infrastructure.

The adoption of a reuse system requires a technological and organizational assessment. There is little or no experience on defining the reusability needs of an organization, and assembling a set of technologies to address those needs [Ara88a]. Few reuse experiences have been reported in the literature [RL79] [PD87b][Mat87] [Cam87]. This is an underdeveloped area of reusable software engineering.

We have defined an initial vocabulary for the description of reuse systems. In the next section we extend the definition of reuse systems to include a learning component.

3.2 Reusers are learning systems

Summary. Reuse systems are viewed as learning systems. Learning is defined in terms of two properties, competence and efficiency. This view is the key to an operational definition of domain analysis.

The purpose of a reuse system $P = \langle MoT, RI, T_R, \{S, I_S\}\rangle$ implemented by an assembly of technologies T_R [Ara88a] is,

- to construct satisficing implementations, I_S, for all specifications in S by reusing information from a reuse infrastructure RI under the control of MoT, and
- to construct the implementations as efficiently as possible.

Other purposes could be specified, for instance, to produce implementations that execute efficiently in terms of storage or CPU time, or, produce implementations that are safe. We do not exclude these or other properties of the implementations themselves or of the reuse-based implementation process. For our current purposes only measurable properties of the state of the reuse system are acceptable.

The *state* of a reuse system $P = \langle MoT, RI, T_R, S, I_S \rangle$ at time t was defined as the values of the state variables at time t (Section 3.1.1).

A measurable property of the state of P is defined as a Boolean or real-valued, time-dependent function $f(t, MoT, RI, T_R, \{S, I_S\})$. Examples of measurable properties of interest for this work are:

- *Coverage.* If the system P, in state $\mathcal{S}$, can produce implementations for a set of specifications S we say that P *covers* S, $S \subseteq C_P(\mathcal{S})$. The size of the largest set S is used as a measure of coverage. From now on, we assume a reuse system P, and use the notation C instead of C_P.
- *Size of the reuse infrastructure.* Measured as the number of items of information in RI (e.g., the number of generic components in a library). A reuse infrastructure is said to be *parsimonious*, if every item in the infrastructure is necessary for the implementation of at least one specification in $C(\mathcal{S})$.
- *Implementation cost.* Measures of cost are (technology-dependent) functions that associate a value to the resources expended on the average in the implementation of a specification.

The number of specifications covered by a reuser could be infinite. In practice, coverage or cost are measured over a selected set of sample specifications, a *benchmark set.* The members of a benchmark set are selected among those specifications that are deemed relevant to the purpose of the measuring exercise. For example, a benchmark for measuring the difference in coverage between two systems may

include specifications that exhibit a diversity of problem features, average number of variants, or some notion of "typicality". Benchmarking is a standard approach to assessing the performance of software or hardware systems.

3.2.1 Goal state

To assess the performance of P, we introduce an ideal *goal-state* $\mathcal{G}$. A goal-state is a particular configuration of values of the state variables of P.

A notion of distance, δ, over values of measurable properties of the states of P is needed to compare the current state of reuse system with respect to a goal state. These measures apply to two states of the same system as well as to two different systems.

We now define three distance functions relevant to the purposes of this work. We assume that the values of state variables MoT and T_R for two states of the reuse system are identical. The reason for this assumption is that differences in *competence, parsimony* or *efficiency* can be attributed *exclusively* to differences in the values of RI. We later argue for the legitimacy of this assumption.

Competence. Given two states of a reuse system, $\mathcal{S}$ and $\mathcal{S}'$, the distance in competence is defined to be the cardinality of the difference between their covered sets:

$$\delta_c(\mathcal{S}, \mathcal{S}') \;=\; |C(\mathcal{S}) \ominus C(\mathcal{S}')|$$

Intuitively, δ_c, is the number of specifications that can be implemented by one and only one of the systems.

Parsimony. Given two states of a reuse system $\mathcal{S}$ and $\mathcal{S}'$ such that $C(\mathcal{S}) = C(\mathcal{S}')$, the difference in power between the two infrastructures is defined by:

$$\delta_p(\mathcal{S}, \mathcal{S}') \;=\; size(RI_{\mathcal{S}}) - size(RI_{\mathcal{S}'})$$

Intuitively, given the same coverage, a smaller infrastructure is regarded as more powerful and perhaps more desirable.

Efficiency. Given a cost function f that depends on the technology of a reuse system, the difference in efficiency between two states of $\mathcal{S}, \mathcal{S}'$ is defined as

$$\delta_e(\mathcal{S}, \mathcal{S}') \;=\; f(\mathcal{S}) - f(\mathcal{S}')$$

over some benchmark set. The definition of the cost functions, f, is a management issue, and depends on the technologies of the reuse system. In a mechanical transformational system it could be measured as length of the derivations, or CPU time expended. In a reuse system implemented by programmers and a database of reusable components, cost could be measured in terms of hours of programmer time and number of database accesses.

The benchmark set may be designed to reflect patterns of reuse in the environment of the reuse system, based on such properties as recency of reuse, frequency of reuse over some period of time or some measure of typicality.

3.2.2 Performance as convergence to a goal

The performance of a reuse system is a measure of the degree to which it satisfies a goal. Separate goal states can be defined to establish standards of competence or efficiency. We measure the performance of a reuse system P in state $\mathcal{S}$, relative to a goal-state $\mathcal{G}$, using a selected distance between $\mathcal{S}$ and $\mathcal{G}$.

P satisfies a competence goal (formalized as a goal-state $\mathcal{G}$) if $C(\mathcal{G}) \subseteq C_P(S)$. The degree to which P may satisfy a competence goal, i.e., a relative measure of competence, can be obtained as $1 - \frac{\delta_c(\mathcal{S},\mathcal{G})}{C(\mathcal{G})}$. This measure is analogous to relative sparseness of a logical complex as proposed by [MS83, p.339] to measure the degree to which a logical characterization generalizes over a set of observed instances.

P satisfies an efficiency goal (formalized as a goal-state $\mathcal{G}$) if the cost of implementing some selected benchmark set is less than cost of using $\mathcal{G}$, for some cost function, f.

To improve the performance of a reuse system we must modify the appropriate system variables to reduce their distance along the desired dimension. A performance function ω, defined over pairs of states:

$$\omega : state \times state \rightarrow [0 \ldots 1]$$

is used to represent *the degree to which a state approximates a specified goal-state relative to other known states.* Given a set of known states $\mathcal{Y}$, the performance function is expressed in terms of a chosen distance function δ (e.g. competence, efficiency) by the formula [Kli85]:

$$\omega(\mathcal{S},\mathcal{G}) = \frac{\delta_m(\mathcal{S},\mathcal{S}') - \delta(\mathcal{S},\mathcal{G})}{\delta_m(\mathcal{S},\mathcal{S}')} = 1 - \frac{\delta(\mathcal{S},\mathcal{G})}{\delta_m(\mathcal{S},\mathcal{S}')}$$

where,

$$\delta_m(\mathcal{S},\mathcal{Y}) = max\ \delta(\mathcal{S},\mathcal{S}') \text{ for} \mathcal{S}' \in \mathcal{Y}$$

Intuitively, if a state $\mathcal{S}$ satisfies the performance goal, the difference $\delta(\mathcal{S},\mathcal{G})$ is null, and $\omega(\mathcal{S},\mathcal{G})$ is 100 percent. If the $\delta_m(\mathcal{S},\mathcal{S}') = \delta(\mathcal{S},\mathcal{G})$, the performance of $\mathcal{S}$ is null.

This definition of performance function provides an objective basis to assess progress in the incremental evolution of the state of a reuser towards a goal-state. As a problem domain evolves, the changes are reflected characterized as changes in the values of the variables $\{S, I_S\}$.

Pragmatics

It may be expensive for a reuse system to achieve a goal state. In practice it could be cost-effective to get the reuse system to a state that is "close enough" to the goal state.

Given a goal state $\mathcal{G}$ and performance function ω, and two states of a system $\mathcal{S}$ and $\mathcal{S}'$, $\mathcal{S}$ is said to be *goal-oriented* [Kli85] with respect to $\mathcal{S}'$ if and only if

$$\omega(\mathcal{S},\mathcal{G}) > \omega(\mathcal{S}',\mathcal{G})$$

that is, if the reuse system in state $\mathcal{S}$ performs better with respect to $\mathcal{G}$. The value of $\Delta\omega(\mathcal{S},\mathcal{S}'|\mathcal{G}) = \omega(\mathcal{S},\mathcal{G}) - \omega(\mathcal{S}',\mathcal{G})$ is called the *degree of goal-orientation* of $\mathcal{S}$ with respect to $\mathcal{S}'$. These measures, coupled with a procedure for cost-benefits analysis provide guidance and stopping conditions for the evolution of a reuse system, in particular, given the conditions of equal values for MoT, T_R and S, for the evolution of the reuse infrastructure of the system. For example, the measures can help select among competing candidates libraries of components or compare the coverage of two collections of components for some benchmark set.

3.2.3 The learning component in a reuse system

The process of convergence to a goal cannot be modeled within the reuse system as defined above. We propose an enhanced view of the reuse system. Figure 3.2 is an enhancement of Figure 3.1 in Chapter 2. We will focus our discussion on the *performance component* P , the *learning component* L, and their relation.

The purpose of L is the (goal-directed) generation of new values for the goal-related variables of P with the intention of increasing the performance of the reuse system. The learning component is defined by the following traits (Figure 3.2):

- a model of the problem domain, MoD,
- a method for increasing the competence of the reuser, L_c,
- a method for improving the efficiency of the reuser, L_e,
- the expertise available in the environment of the reuse system, Exp, and
- the feedback received from the actual reuser, a *Reuse Log*
- a set of technologies T_L to realize the learning component, that is, to support the representation of the MoD as well as the application of the methods.

Additional components could be defined as abstractions for methods to improve the performance of P along other dimensions. We have chosen competence and performance as the most relevant for the purposes of discussing domain analysis.

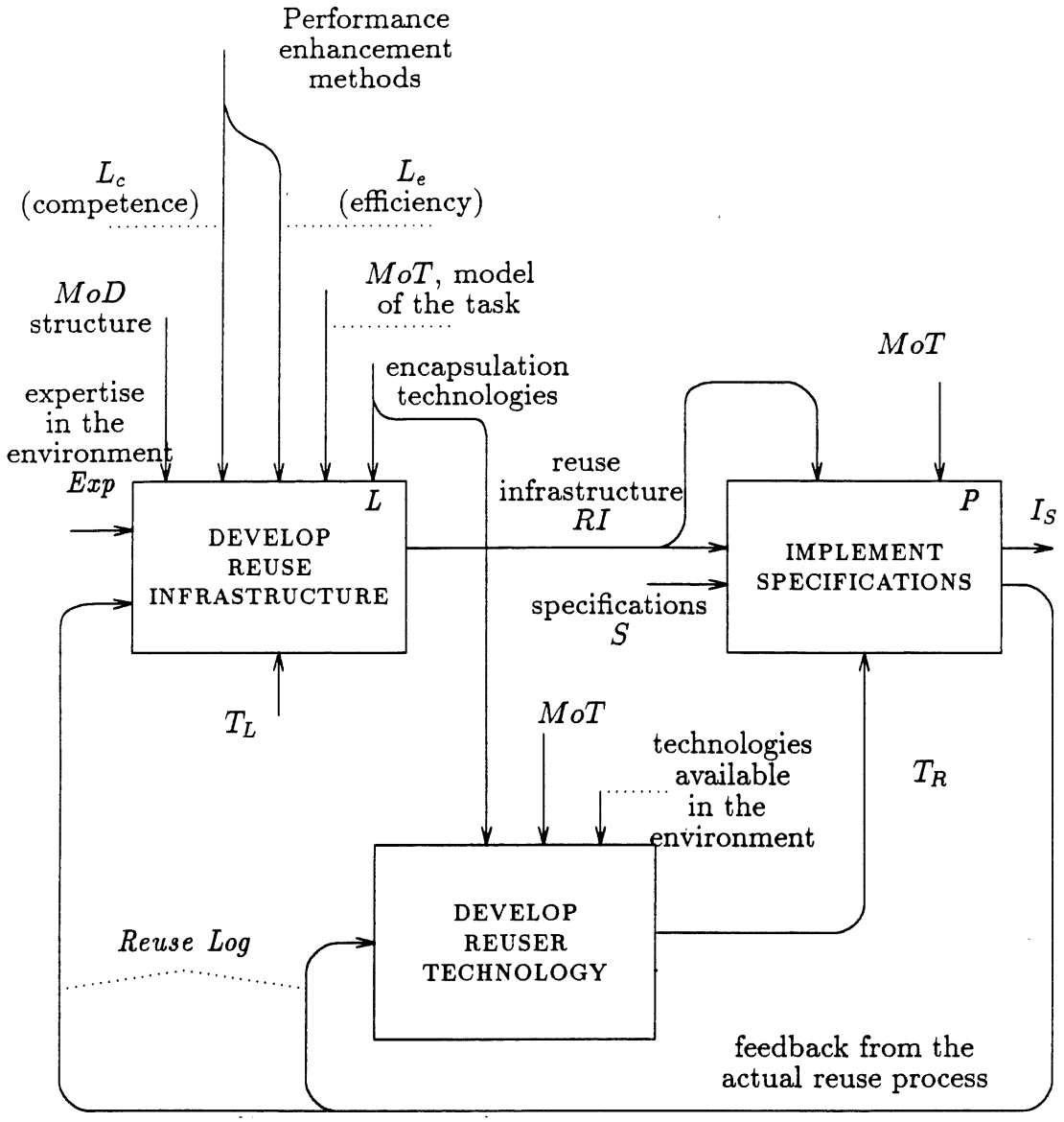


Figure 3.2: The reuse system as a learning system

The MoD structure is a data structure for representing reusable information about the problem domain. We distinguish between the MoD *structure* for representing information, and the MoD *state*, defined by the information captured at a given point in time.

We have chosen to represent a model of the problem domain, independently of the reuse infrastructure RI for practical reasons. A reuse infrastructure is a particular representation for information in the MoD dictated by the technology of the reuser, T_R. The constraints imposed by those representations hinder the process of analysis with the goal of learning. An analogous argument is made in software development for distinguishing between representations for system specification and programming languages for implementations.

The outputs of L are values for the reuse infrastructure of P.

The variable Exp abstracts problem domain expertise available in the environment of the reuse system. The mechanisms in L_c acquire information from Exp and evolve the information in the MoD.

The values of RI, the reuse infrastructure, are derived from the state of MoD. We have called this process *infrastructure design*. Methods L_e evolve the information in the reuse infrastructure based on feedback from the reuser. For the time being, we informally define the Reuse Log as a record of the specifications submitted to the actual reuser.

Broadly speaking, the values for L_c and L_e are learning methods. "Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same tasks or tasks drawn from the same population more efficiently and more effectively the next time" [Sim83].

3.3 The infrastructure development cycle

Summary. We identify three major activities in a reuse infrastructure development cycle: 1) the analysis of the problem domain for the purpose of capturing information relevant to the reuse task, 2) the specification of some subset of that information to match the architecture of the reuser and patterns of reuse in its environment, and 3) the implementation of the specified parts. These steps can be compared with the the major phases in the conventional software lifecycle. In particular, Step 3) and to some extent, step 2) are covered by conventional software engineering approaches.

We identify three major steps in an infrastructure development cycle: domain analysis, infrastructure specification and infrastructure implementation. This decomposition of the activities performed by the learning component in the reuse system (Figure 3.2.) Figure 3.3 summarizes

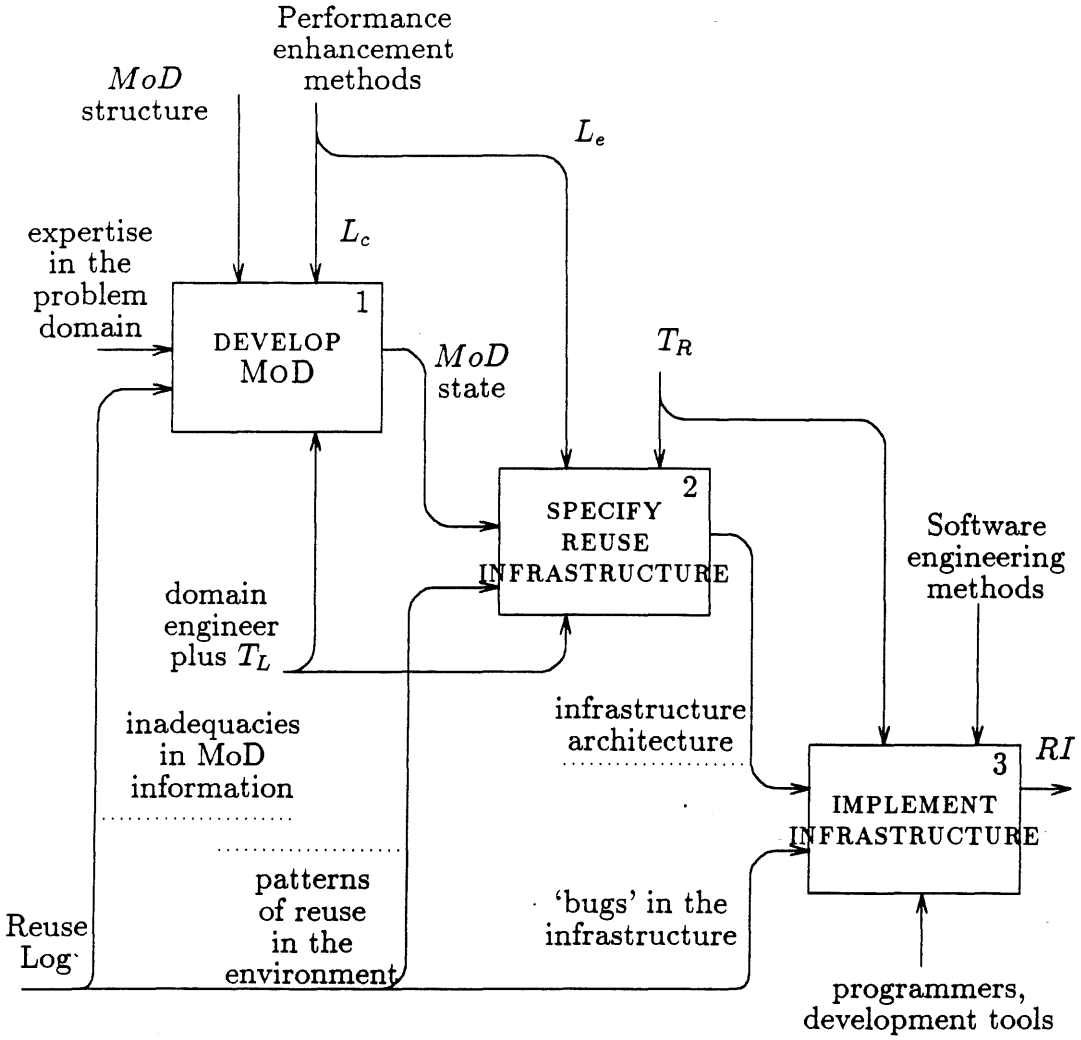


Figure 3.3: Infrastructure development cycle

This decomposition is a "convenient fiction", analogous to the waterfall view of program development: requirements analysis, specification, design and coding. As in the case of program development, the decomposition is useful to distinguish between the pursuit of different goals.

Domain analysis aims at capturing relevant information to achieve a desired level of competence in the reuser.

The purpose of infrastructure specification is to organize the information in the reuse infrastructure to "tune" the operation of the reuser to the patterns of reuse in its environment. The goal of the specification task is to achieve a desired level of efficiency in the reuser.

Finally, the implementation task is concerned with the actual implementation of the various pieces of the infrastructure specified. The implementation maps specifications into different technologies—Ada, C++, domain-specific languages, software templates—depending on the technology of the reuser. Infrastructure design and implementation are conventional software design and implementation tasks.

The diagram in Figure 3.3 emphasizes the data-driven aspect of the development cycle, and it could be interpreted as a conventional waterfall process. In practice, it is an iterative process with feedback information flowing upstream analogous to the program development lifecycle.

As an illustration of the kinds of research questions that need to be answered, let us consider the following: Given a class of similar problems, how do we identify reusable problem-specific information. How can we modularize such information and package it into reusable components. If a component is changed, what is the impact on our ability to specify and construct systems for that class of problems? If the class of problems evolve, how can we propagate such changes to the collection of components to maintain the level of competence of the reuser? How can we organize a collection of reusable component to improve the efficiency of the actual reuse process?

Similar questions are routinely asked in the context of conventional software engineering: How can we acquire the requirements for a program? How can we design a program to satisfy some set of requirements? How does a change to the implemented program affect its required behavior? If some requirements change, what is the impact on the implemented system? Which system architecture would improve the performance of the program?

Over the years, software engineering developed into a body of principles, methods, representations, and tools to help bridge the gap between the description of needs (expressed as requirement specifications) and machine-processable representations of solutions (expressed as computer programs) in a controlled, systematic manner. In the field of reusability, we face an analogous "needs-solution" gap. There is a need for an engineering perspective, to help bridge the problems-encapsulations gap in a controlled, systematic manner.

Two fundamental research issues need to be addressed; 1) the definition of a

conceptual framework to enable the description and analysis of the engineering of a reuse infrastructure, and 2) the design of systematic methods to realize the process.

3.4 The domain engineering framework

Summary. An engineering framework is needed to systematically bridge the problems-infrastructure gap in reusability. An analogy with the conventional software development cycle is direct. The focus of our work is on the front-end activity of domain analysis.

By analogy with software engineering, the aim of domain engineering is to provide methods and representations to help in

1. the construction and analysis of models of problem domains to support software specification and construction;

2. the transformation of those models for particular purposes; for example, enhancing performance, enhancing efficiency, or specifying instances of reuse infrastructures.

Our focus is on domain analysis and infrastructure specification, the upstream activities in the infrastructure development cycle, rather than on the downstream activity of infrastructure implementation.

Figure 3.4 outlines a strategy for defining the relevant controls in the learning component of the reuse system, as summarized in Figure 3.2.

The major parameter in the design of domain analysis methods is the model of the task, MoT. The model of the task defines the information requirements of the performance component P, and will be used as a foundation for organizing the process of acquiring information. The MoD structure is directly derived from the MoT subject to the constraints of available representation technologies (T_L). The design of a MoD structure interacts with the design of the MoD evolution procedures L_c and L_e. The MoD structure must facilitate the application of the performance enhancement methods.

3.5 The operationalization of domain analysis

Summary. The domain engineering framework enables an operational definition of the domain analysis task that avoids the objections raised in Section 2.4.2. Domain analysis methods become the means to regulate the performance of a reuser P. Regulation is achieved by the goal-oriented evolution of a model of the problem domain.

Given: a description of a reuse situation characterized by a reuser, P, and its environment.

Produce: the methods and support technology for the learning component of the enhanced reuse system. The logical dependency between parts suggests the following stages:

1. Define the environment of P, i.e., characterize a
   - model of the reuse task, MoT,
   - problem domain,
   - set of technologies that implement the reuser, T_R, and those available to support the operation of the learning component being designed, T_L,
   - costs-benefits model to determine under which condition it is economical to invest in the development of a reuse infrastructure.
2. Define the kinds of information required by a reuser operating under MoT.
3. Design a MoD *structure* to: 1) capture the kinds of information required by the MoT, and 2) support the application of evolution procedures L_c and L_e.
4. Define competence enhancement methods, L_c, based on the structure of the MoT, to operate on MoD information.
5. Define efficiency tuning methods, L_e, based on the structure of the MoT and the patterns of reuse in the environment.
6. Produce a *scheduling policy* for the evolution of reuse infrastructures according to cost-benefits analysis in the environment of P

To develop/evolve a reusable infrastructure:

1. Evolve the MoD to meet the performance requirements on the reuser.
2. Upon demand, specify an infrastructure or propagate changes from the MoD to an existing one.

Figure 3.4: The domain engineering strategy

Given: a reuse system characterized by

- a model of the reuse task, MoT,
- a problem domain,
- the requirements of the environment defined by the reuser's feedback, *Reuse Log*,
- a MoD structure for representing information about the problem domain, and
- a family of methods L_c (for enhancing the competence of the reuser) and L_e (for enhancing the efficiency of the reuser.)

Apply:
methods L_c and L_e to evolve the state of the MoD to approximate desired competence and efficiency levels in the reuser

Figure 3.5: A reformulation of the task of domain analysis

In Section 2.4.2 we pointed to the fact that the current perspectives on domain analysis are not operational and that they do not offer a termination predicate or a measure to assess progress. The view of reuse systems as learning systems enables a reformulation of the domain analysis task that avoids these problems.

The purpose of practical domain analysis is to evolve the state of the MoD—the knowledge of the problem domain available to the reuse system—so that the reuse system attains a specified level of performance within resource constraints (Figure 3.5).

The Fixed MoT/technology Assumption. We assume that the values of all the state variables that define a reuse system (see Figure 3.2) and its environment remain constant except for the definition of the problem domain, as characterized by: 1) a collection of specification/implementation pairs $\{S, I_S\}$, and 2) the feedback from the reuser, *Reuse Log*.

Under this assumption, the performance of a reuser, P, depends directly on the quality of its reuse infrastructure, RI, (and if we assume a correct implementation process,) on the quality of the MoD. Thus, the performance (competence and efficiency) of P can be tuned using the controls on the learning component, methods L_c and L_e (Figure 3.2).

The goals of the domain analysis process can be explicitly defined as a goal-state for the reuse system. For example, a competence goal is defined as a benchmark set $\{S, I_S\}$. Convergence to that goal can be measured using coverage measures as

defined above.

More complicated schemes can be derived by allowing other variables to change. The example of the evolution of the KATE system discussed in Chapter 2 illustrates a process in which the values of RI, MoT and T_R were allowed to change in each generation of the system, and were tuned to each other. The approach followed by the KATE project is justified by the exploratory nature of that research. In in real-life software development situations the Fixed MoT Assumption is safe. Software development systems are complex systems involving objectives policies, information, people, procedures, and tools, all of them embedded in a particular technological and organizational culture [Fre87a]. Once such a system is in operation, the very complexity of the structure represents an obstacle to change. Imagine, for example, a software development organization working under DoD-STD-2167A and committed to the Ada technology.

3.6 Summary

In Chapter 2 we concluded that there is a need for a conceptual foundation for domain analysis. The domain engineering framework discussed in this Chapter is an initial attempt to fill that void.

The view of reusers as directed-systems must be enhanced to include a notion of performance and a learning component to regulate performance using feedback information. A domain engineering strategy has been presented that identifies steps for the definition of a learning component.

Our focus is on the definition of domain analysis methods. We propose an operationalization of practical domain analysis as the evolution of the model of a domain which captures semantics of a domain-specific language. We define the inputs, controls and supporting mechanisms for the learning component in the reuse system. Domain analysis methods play the role of controls. The methods should be systematic, incremental, independent of the technology of the reuser, and have an explicit scope of application.

Systematic. The economics of reuse suggest that the development of a reuse infrastructure is a multi-year, multi-person process. The success of the application of a method should not depend on a Crack Analyst.

Incremental. The hypotheses of partial knowledge and gradual evolution (Section 2.5) require that domain analysis methods support incremental approximations to desired goal states.

Independent of the reuser's technology. The process of domain analysis is expensive and it is desirable to make the information reusable in as many situations

as possible; it is also desirable to isolate it from changes in the technology of the reuser.

Explicit scope of application. We should be able to state explicitly the conditions under which domain analysis methods can be successfully applied. Under the domain engineering framework, the scope of a method is defined by the family of reuse system that matches the specification used to constrain the design of methods.

Road map to Chapters 4 through 7

In the next four Chapters we "step down" to the Domain Analysis level in the dissertation (see Figure 1.4 for an outline), and discuss the application of the domain engineering strategy to the design of domain analysis methods.

In Chapter 4 we make the transition between the theory and the methods by identifying kinds of MoT. Models of reuse-based software construction are examined using the notion of Orders of Reuse and a particular one, a first-order interconnection model, is selected to make the presentation concrete.

Domain analysis realizes a form of incremental language acquisition. The competence of the reuser depends on the richness of this language and on how well the language expresses the class of specifications of interest in the environment of the system. The efficiency of the reuser corresponds to the cost of processing the language, (i.e., parsing specifications and producing implementations.)

Having selected a value for MoT, a MoD structure is presented in Chapter 5. From the point of view of domain analysis as language acquisition, the MoD structure can be regarded as a domain-independent or "universal grammar" for the representation of reusable, first-order information.

Chapter 6 presents a particular realization of the L_c component in the reuse system. That is, a family of methods to evolve a MoD state. The goal of that evolution is to induce a more satisfactory domain-language. The competence of the reuser is enhanced by incremental language acquisition constrained by the model of reuse selected in Chapter 4.

Finally, Chapter 7 discusses L_e, methods for analyzing the efficiency of a reuser and tuning its operation to the patterns of reuse in its environment.

Chapter 4

Models of the Reuse Task

Chapter summary. This chapter serves chiefly to prepare for the discussion of practical domain analyses methods in Chapters 5 through 7. The model of the reuse task, or MoT, is a key control parameter in the process of domain analysis. The MoT, 1) identifies the kinds of information relevant to the reuse process and provides guidelines for its acquisition, and 2) determines the scope of applicability of the domain analysis methods derived under the domain engineering strategy.

The purpose of this Chapter is, 1) to demonstrate the first step in in Domain Engineering strategy—the identification of MoTs; 2) to define the scope of domain analysis methods associated with typical MoTs; and 3) to present a rationale for practical domain analysis methods to support first-order reuse.

A notion of *orders of reuse* is introduced to categorize reusers independently of their implementation technologies. The study of the order-related aspects of reuse systems distinguishes between "compositional" and "generational" reusers, and is used to define the scope of different forms of domain analysis. We argue that the difficulty of acquiring reusable information grows with the order of the reuser.

An analogy is presented between the process of reuse and of language comprehension and generation. We propose a view of practical domain analysis as a form of language acquisition. Our notion of competence for a reuser is similar to that of linguistic competence. Chapters 5 and 6 refine and elaborate this view.

4.1 Introduction

Summary. An extensional definition of a problem domain as a set of specification/implementation pairs is proposed. Reuse systems are

represented by equations of the form $MoT(S, RI) = I_S$ where the technology-dependent features have been abstracted away. Domain-specific reuse is defined.

We propose the following extensional characterization of problem domain:

A problem domain D is *characterized as* a collection of pairs of specifications of software systems and their implementations, $D = \{ (S, I_S) \}$.

In practice, if the set $\{ (S, I_S) \}$ is infinite, the problem domain is characterized using a subset considered to be "representative" of the larger class. The members of that subset will be called *exemplars* of the domain D. This definition does not imply that other factors that shape our intuition of the concept of problem domain, such as, organizational goals, people, or the technical literature, are excluded as sources of information. Those are factors in the environment of the reuse system that determine what it is to be "representative" and provide complementary information, justification, etc.

4.1.1 Domain-oriented and domain-specific reuse

In Section 3.1.1 we defined a reuse system as a tuple of four elements $P = \langle MoT, T_R, RI, \{S, I_S\} \rangle$; for the purpose of the discussion we abstract away the implementation technology of the reuser, T_R. A *model of reuse* is then represented as a function MoT:

$$MoT(S, RI) = I_S$$

that is, each implementation, I_s, is the result of applying the reuse function MoT to a specification $s \in S$ and an infrastructure RI. S in $\{(S, I_S)\}$ is the set of specifications covered by the reuse system.

A reuse system is said to be *domain-oriented* if the structure of MoT, or RI, or both are determined by the definition of the set $\{(S, I_S)\}$.

Some popular forms of reuse are *not* domain-oriented. For instance, the "copy-and-edit" actions performed routinely by programmers, are an example of non-domain-oriented reuse. New software is created by modifying existing software. In practice, the modification operators applied and their bindings are not captured explicitly and their relation to particular classes of programs is unclear.

The size of the class S that a reuse system can implement measures its degree of *domain-specificity*. There is a lot of latitude in the degree of domain-specificity of a reuse system. The expression "general purpose" reuse has been used to refer to reuse systems where the class S is very large.

As an aside, the degree of domain specificity of a reuse system is orthogonal to a notion of "size" of the domain itself, e.g.,most people would agree that Airline Reservations is a "bigger" problem domain than Accounts Receivable. This intuitive notion of size refers to the volume and complexity of the information required to

specify and implement applications in the domain. Size, in this sense, correlates with the number of nodes and the complexity of domain network structure (or, from Wile's [Wil86] perspective, on the number local formalisms and the complexity of their links) necessary to construct software in the domain.

The following examples of domain-oriented reuse systems illustrate varying degrees of domain-specificity, starting with the most general purpose systems: programming languages and their compilers. The grammar of a programming language (e.g., Ada, Pascal, Lisp) formally defines a class of specifications S. The compiler for the language transforms the specifications $s \in S$ into implementations I_s. The compiler for the language realizes a MoT. Its structure is determined both by the source language, S, and by the target language of implementation. Programming language compilers are usually general purpose, domain-oriented reuse systems.

Draco is an application generator-generator. It is domain-oriented—the class of specifications that can be implemented is determined by the grammar of the Draco-domain languages. Domain language networks are parameters to the process of software construction using Draco, the degree of domain-specificity varies with the family of grammars available to the system. The libraries of components used by Draco (RI) provide operational semantics to the domain languages in the network. The MoT is a transformation mechanism based on tree-to-tree substitutions.

Application generators illustrate a trend towards increasing domain-specificity. Application generators can be regarded as compilers of very restricted languages.

The IMS system [Bax87d] is an instance of a very domain-specific reuser. IMS reuses a transformational derivation for the implementation of a specification s to create a new derivation for a specification s', that is slightly different from s. The MoT realized by IMS has a domain-independent aspect which operates on generic derivations using *permutation* and *truncation* operators, and a domain-dependent aspect, encoded as goal/plans structures for capturing performance information on the implementations. The RI is extremely specific: the transformational derivation of s; i.e., there is a special RI for each s. The class of specifications that IMS can implement constitutes a small neighborhood of the specification s that is being evolved.

4.1.2 Information distribution and evaluation time

The examples discussed above illustrate that domain-specific information can be distributed between the MoT and RI components in the reuse equation $MoT(S, RI) = I_S$ in many ways.

One end of the spectrum is illustrated by compilers for programming languages and application generators, in which all the knowledge on how to generate implementations I_S for specification in S, rests on the MoT, and RI is nil. The other end of the spectrum is illustrated by reusers which "assemble" or "refine" components, where the domain-specific information is captured exclusively by RI, the library of components, as it is the case with the Draco system

The issue of distribution is important. It emphasizes a distinction between reusable, domain-specific information that is evaluated at reuse time—in the *process* of constructing a piece of software—and domain-specific information that is *not* evaluated at reuse time—reused in the product. In the next Section we characterize this distinction using a notion of orders of reuse.

4.2 Orders of reuse

Summary. Orders of reuse is a criterion for categorizing reuse systems. The order of a reuser is an indication of the "depth" of the implementation knowledge explicitly available to the reuse system in some form of *RI*. First-order information is captured in solution components and operators over solution components. Second-order information corresponds to plan-components for the generation of solutions and operators over plans, and so on. The notion of orders parallels a similar categorization of planning knowledge found in artificial intelligence.

4.2.1 Definitions

The notion of *order of reuse* is independent of the actual operators used to manipulate the reusable information (permutation, concatenation, truncation, transformation, etc.). Figure 4.1 offers a simplified summary.

In the context of software construction, a *first-order reuser* operates directly on explicit, implementation-level information. First-order information consists of software fragments in some appropriate form; for example, Cobol programs, Ada generics, abstract data types, spreadsheet templates, Smalltalk objects.

A *second-order reuser* operates on (second-order) information on how to generate, adjust, or organize software components into a desired system. Second-order information is at a *meta-level* with respect to the first-order information embedded in the components. A second-order system reuses *plans*—temporally organized patterns of intended actions—for the generation of an implementation. It may create plans from plan components by composition or transformation of plan parts. Examples of such plan structures are, for example, program derivations [SS83], formal developments [Wil83], or derivation histories [Car83] [ABFP86] [Bax87a].

Third-order reusers reuse strategies for the generation of plans to subsequently derive implementations. The meta-level/level distinction made between plans for generating implementations and software components applies here to strategies, (or goal structures) and plans. IMS extended with the Production Control Language [Bax87c] or the generator of Explainable Expert Systems [NSM85] are examples of third order reusers. The generation of animated graphics using "reality-based" programs [Ame87] that build on rules of physics like those that govern reality is another example of third-order resue.

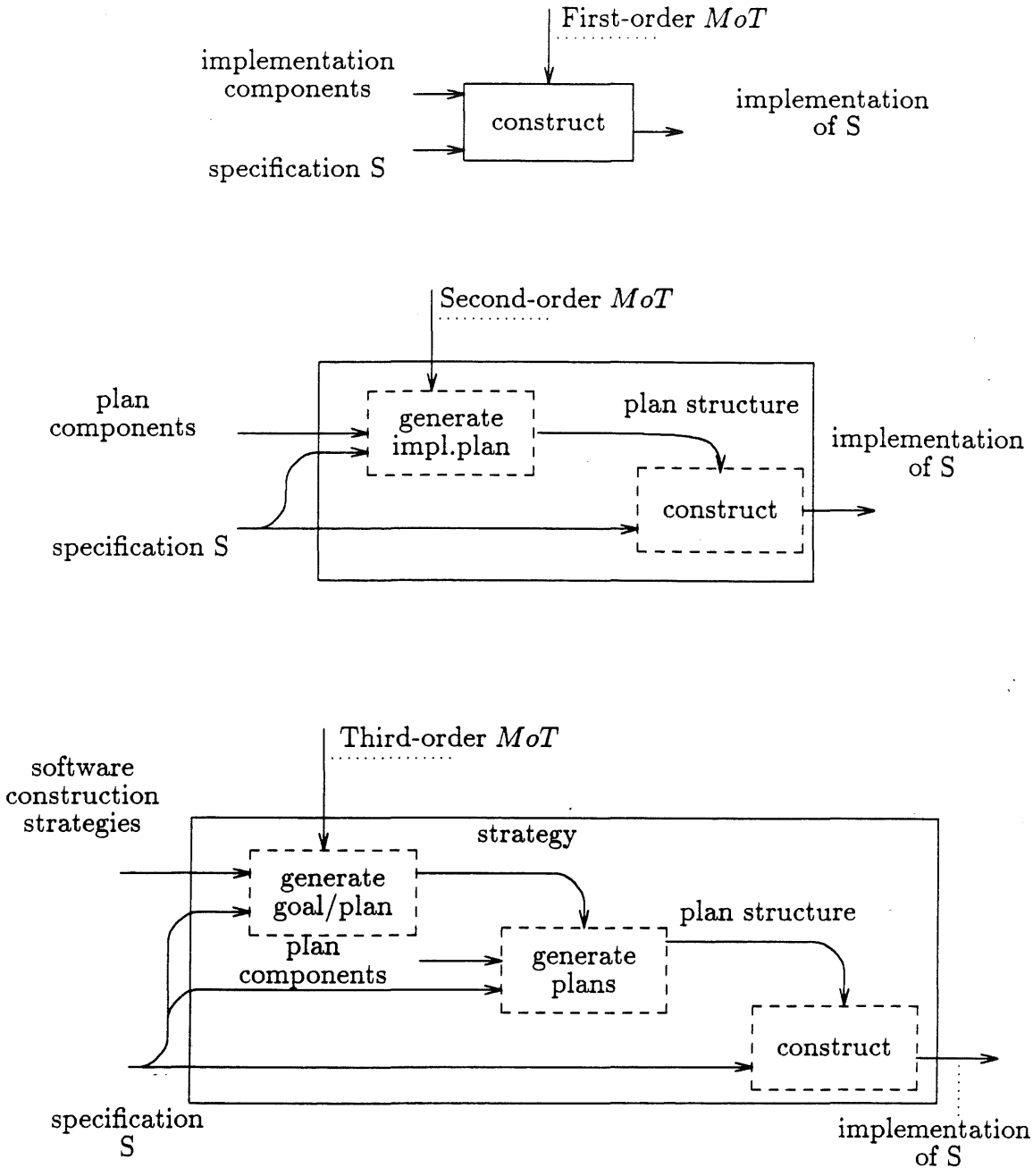


Figure 4.1: The first three orders of reuse

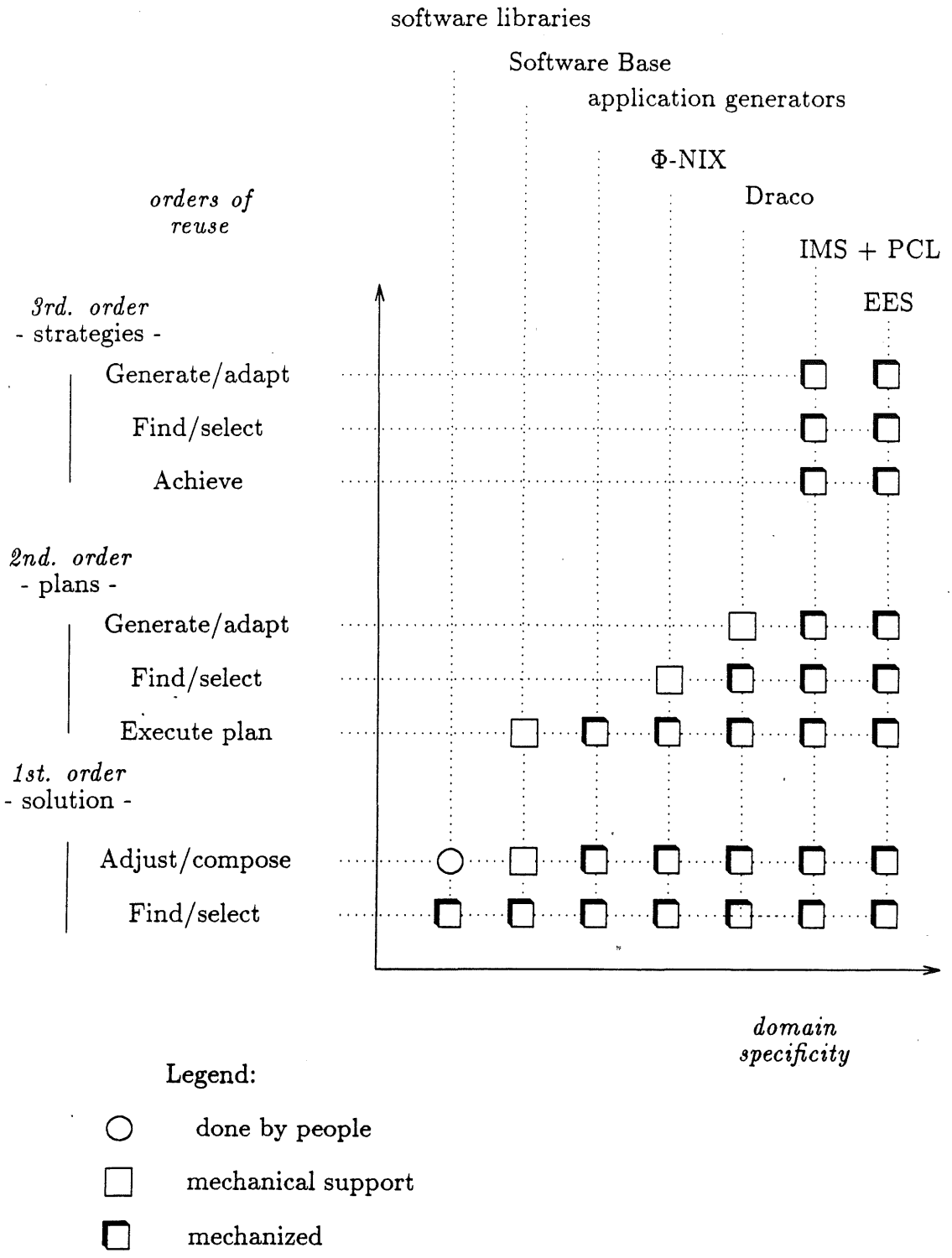


Figure 4.2: A sample classification by order

4.2.2 Examples

A small sample of reuse systems is categorized with respect to the order of reuse they perform (or enable) in Figure 4.2. The Figure illustrates the correlation between domain-specificity and order of reuse. The correlation is not accidental. Higher-orders reusers require "deeper" knowledge about the problem domain, e.g., not only knowledge of what are solutions in the domain, but how to generate them, how to generate the generators, and so on. Our lack of knowledge on the nature of the software process forces us restrict the problem domain to achieve acceptable levels of competence. Under those restrictions, the reuser can make strong assumptions about which are the useful implementation plans and how to generate them.

4.2.3 Orders and "depth" of implementation knowledge

A notion of "depth" of problem solving knowledge can be found in the literature on artificial intelligence—e.g., shallow vs. deep, low-road or heuristic vs. high-road [Cha83] [Mic82]. Kahn [Kah84] distinguishes a spectrum of approaches in diagnostic expert systems from those that apply shallow evidential models to those that manipulate a causal model until the simulated behavior matches that of the real system. Layers of problem-solving knowledge are made explicit in some planners. The control in Stefik's MOLGEN [Ste81] switches between three distinct levels or "spaces": the planning space, the design space and the strategy space. The planner resorts to the higher spaces to help resolve constraints at the lower levels.

Such distinctions have an impact on the design of domain analysis techniques:

1. Ease of knowledge acquisition.
   Instances of compiled knowledge are more readily available than explanatory theories of why things are the way they are. In fact, there exist no universally accepted models of how to implement a specification, or of how domain-specific and programming knowledge interact. Thus, shallow, (first-order) knowledge is more available than deeper (higher-order) knowledge, and can be acquired at considerably less expense. First-order knowledge, however, has its shortcomings.

2. Robustness vs. knowledge brittleness.
   Shallow problem-solvers, "operate on a high plateau of knowledge and competence until they reach the extremity of their knowledge; then they fall off precipitously to levels of ultimate incompetence" [LPS86] [LF87]. The same happens to people, but their performance tends to degrade more gracefully, as the fall is cushioned by layers and layers of weaker, more general models that underlie their specific knowledge. Thus, the deeper the knowledge available to the reuser, the better it performs on "boundary cases", and the better the justifications about the resulting implementations.

3. Explanation.
   To be able to "explain" a software system we must be able to produce a rationale of *how* it came to be the way it is, together with *why* it is reasonable for it to be so. Explanation requires meta-level knowledge of the process. How-explanations can be produced by making explicit second-order information, e.g., the algorithm or derivation of the implementation. Why-explanations require even higher-order information, e.g., the goal structures that determined the choice of derivation.

The robustness and explanatory power of a reuse system grow with its order. So does the difficulty and cost of populating RI with the necessary information. This notion of "depth" differs from that in [Nei88b], which refers to how many layers of implementation detail should be made explicit as part of a domain analysis. That interpretation corresponds to view of software implementation as a reductionistic process [LST84]. It is a first-order notion, closer to degree of operationalization [Mos81].

4.2.4 The reuse equation revisited

Using the notation introduced in Section 4.1.1, a domain analysis to support first-order reuse:

$$MoT(S, RI^{(1)}) = I_S$$

must acquire the information needed to produce a first-order infrastructure—$RI^{(1)}$, reusable components that are *not* evaluated at reuse time—and some indexing scheme for locating these components. In the case of pure, second-order reuse, $RI^{(1)}$ is nil, and $MoT = T(S, RI^{(2)})$ thus, the equation may be rewritten as:

$$(T(S, RI^{(2)}))(S) = I_S$$

For example, in Draco the components in $RI^{(2)}$ are tree-to-tree transforms called "refinements". The MoT is composed of a transformation engine T and a second-order infrastructure $RI^{(2)}$ of reusable components (transforms) and component selection (transformation tactics), which are evaluated at reuse time. In the case of hybrid application generators,

$$(T(S, RI^{(2)}))(S, RI^{(1)}) = I_S$$

that would also be the case of an Ada compiler, in which, $RI^{(1)}$ is a library of compilation units (packages, subprograms) and $RI^{(2)}$ consists of a single, procedural transformation, the compiler itself.

A domain analysis to support second-order reuse must capture two distinct kinds of information together with the interactions between them. There are differences in the complexity and in the method needed between, for example, defining a library of Ada generics, a library of transformations, or the structure of an application generator. In preparation for making these distinctions precise, we examine the "compositional" and "generational" approaches to reuse.

4.3 Composition, interconnection and transformation

Summary. Research on reusability has yet to produce a taxonomy of methods of reuse. Common expressions such as, reuse "by composition" or "by generation" are not backed by formal definitions. We need to make precise those forms of reuse to establish their information requirements, and to set goals for the task of domain analysis.

[BR87] divides reuse technologies into composition and generation technologies. A definition of what is composition is not given, but compositional technologies are characterized as those that operate on "passive" elements. Generation technologies are described as those in which the components are "woven into the fabric of a generator program". Generation technologies are "not easy to characterize because the reuse components are not easily identifiable as concrete, self-contained entities"[BR87, p.42]. The expression, reusable pattern, is used to identify those resources. The authors conclude, "we cannot easily characterize the principle whereby those patterns are reused, except to say that it is a kind of reactivation of patterns."

All reuse systems use a combination of compositional and generational approaches. Given two orders n and and $n+1$, reuse at the level $n+1$ is generational with respect to the level n. In some cases (e.g., in transformational systems) reuse is *compositional* within the level $n+1$ itself. Compilers and some application generators implemented as a single, procedural transformation at level $n+1$ being the extreme case.

We now categorize reuse systems according to *how* the reusable information is manipulated.

4.3.1 Context free composition

The *compositionality principle* as originally stated by Frege [Fre77] reads:

> The meaning of a compound expression is composed from the meanings of its parts.

This principle is used in the definition of denotational semantics [Sto77]. Semantic valuation functions map syntactic constructs in a program to abstract values which they denote. These functions are recursively defined: "the value denoted by a construct is specified in terms of the values denoted by its syntactic subcomponents" [Sto77, p. 13].

These definitions presume a language L whose meaning is defined in terms of the elements of a model I, i.e., with model-theoretic semantics with interpretation in I. The expressions in the language L are *composed from parts*, which creates the need for a syntax for L. The principle requires that we give meaning to the parts,

the basic symbols in L. Following [JvEB82], the general form for the syntactic rules for composing parts in L:

> From expressions $E_1, \ldots, E_k$ build the composed expression E, where $E = S_j(E_1, \ldots, E_n)$.

which requires that we specify a syntactic rule, S_j, and the parts $E_1, \ldots, E_k$. Expressions E_i are divided into groups called sorts, signatures or categories depending on the branch of semantics. [JvEB82] remarks that in applying the compositionality principle one should not look at the language L but at the "derivational history"; i.e., the parse trees of expressions in L. From this viewpoint, the syntax is considered to be a set of categories $\{c_i\}$ together with rules $\{S_j\}$ which are operators of the form:

$$S_j : c_1 \times \ldots \times c_k \rightarrow c_0$$

Basic expressions are also needed for starting the construction process.

An example

To illustrate the definition of compositional reuse, we assume a toy specification encoded in a language L for the specification of disk drivers:

DCB, READ-SECTOR FAST, WRITE-SECTOR FAST□

The specification reads: "implement a disk driver with the following features: a Disk Control Block structure, and two operations: read and write disk sectors on a fast disk". A fragment of a grammar G_L for language L could be:

G $\xrightarrow{1}$ DCB input;
G $\xrightarrow{2}$ DCB input output;
input $\xrightarrow{3}$ READ-SECTOR FAST |...;
input $\xrightarrow{4}$ READ-SECTOR SLOW |...;
output $\xrightarrow{5}$ WRITE-SECTOR FAST |...;

Where each production labelled $\xrightarrow{j}$ corresponds to a grammar rule S_j. A parse tree for the specification in the example is shown in Figure 4.3. The implementation function must be defined for the terminals in the language. We represent implementations as boxes labelled with the name of the symbol they implement. The first line below is read: The function $\mathcal{M}$ applied to the terminal symbol DCB evaluates to a program $\boxed{\text{DCB}}$ that is an implementation of the semantics of Disk Control Block represented by DCB.

$\mathcal{M}$(DCB) = $\boxed{\text{DCB}}$
$\mathcal{M}$(READ-SECTOR) = $\boxed{\text{READ-SECTOR}}$
$\mathcal{M}$(FAST) = $\boxed{\text{FAST}}$

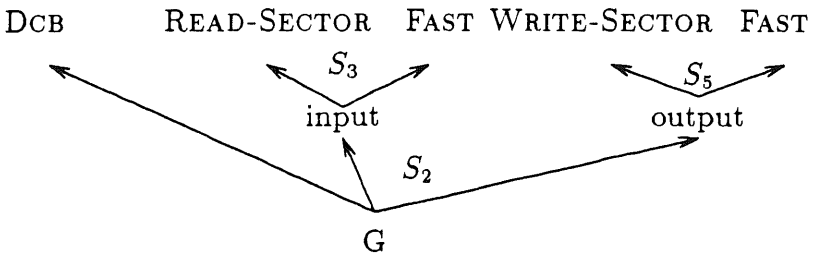


Figure 4.3: A parse tree for DCB, READ-SECTOR FAST, WRITE-SECTOR FAST

$\mathcal{M}$(SLOW) = [SLOW]
$\mathcal{M}$(WRITE-SECTOR) = [WRITE-SECTOR]

The composition operators describe the meaning of non-terminals in G_L as a composition, $\otimes_i$, of the implementation of the children in the parse tree. Thus, the meaning of the non-terminals can be expressed as:

$\mathcal{M}$(WRITE-SECTOR, FAST) = $\mathcal{M}$(WRITE-SECTOR) $\otimes_5\mathcal{M}$(FAST)
$\mathcal{M}$(READ-SECTOR, SLOW) = $\mathcal{M}$(READ-SECTOR) $\otimes_4\mathcal{M}$(SLOW)
$\mathcal{M}$(READ-SECTOR, FAST) = $\mathcal{M}$(READ-SECTOR) $\otimes_4\mathcal{M}$(FAST)
$\mathcal{M}$(DCB, input, output) = $\mathcal{M}$(DCB) $\otimes_2\mathcal{M}$(input) $\otimes_2\mathcal{M}$(output)
$\mathcal{M}$(DCB, input) = $\mathcal{M}$(DCB) $\otimes_1\mathcal{M}$(input)

Terminals in the parse tree are replaced by their implementations. The subtrees are recursively replaced by implementation through the execution of composition operators $\otimes_i$.

$\mathcal{M}$(WRITE-SECTOR, FAST) = [WRITE-SECTOR] $\otimes_5$ [FAST]
$\otimes_5$(A,B) = if B = [FAST] then [WRITE-SECTOR-FAST]
$\mathcal{M}$(READ-SECTOR, FAST) = [READ-SECTOR] $\otimes_4$ [FAST]
$\mathcal{M}$(READ-SECTOR, SLOW) = [READ-SECTOR] $\otimes_4$ [SLOW]
$\otimes_4$(A,B) = if B = [FAST] then [READ-SECTOR-FAST]
else if B = [SLOW] then [READ-SECTOR-SLOW]
$\otimes_2$(A,B,C) = if B = [READ-SECTOR-FAST]
and C = [WRITE-SECTOR-FAST]
then [concatenated A,B,C]
else signal-"Inconsistent specification"

As the example shows, the composition process is context free. Each composition operator has access only to the implementations of the parts to be composed. Conditions such as "if B = [SLOW] then ..." can only be used to locally ensure the consistency of an implementation, but cannot verify contextual aspects of program-

ming languages such as the correct scoping of variables.

To introduce context sensitivity in the composition process, *global information* on the structure of the specification must be processed at construction time. This violates the intention of the definition of composition. We will use the term, "interconnection" to refer to that approach.

4.3.2 Interconnection

To compensate for the limitations of context free composition, global information on the structure of a specification must be made available to the software construction mechanism. An interconnection MoT operates on information on

- first-order parts (or "components", although the process of software construction is not strictly compositional), and
- on the relations, or dependencies between them.

For example, the **with** clause in Ada makes interconnection information explicit to the compiler. The **#include** directive in C indicates to the C preprocessor the context for the compilation of a program.

Perry [Per87a] has classified module interconnection information into three types called the unit, syntactic and semantic interconnection models. The unit interconnection model defines a *depends-on* relation between units such as modules or files, for example, the **with** clause in Ada and compilation units. The syntactic interconnection model describes relations among the syntactic elements in the programming languages, such as *calls, is-called-by, is-argument-of.* Instances of this model are used to support change management in Gandalf, Interlisp or CLF.

The semantic interconnection model attempts to capture relations about how the objects that comprise a system are meant to be used. This is accomplished by associating with the predicates such as pre- and post-conditions, and defining a *satisfies* relation on them. Other context sensitive aspects of the implementation process are captured by the notions of *obligations* and *exceptions.* These are conditions, emerging as side-effects of operations, that must be met eventually, e.g., closing a file that has been open, de-allocating buffers, etc.

From the point of view of orders of reuse, interconnection information is in-between first-order (solution) information and higher-order (generation) information. A reuser that realizes a *strict* interconnection model performs first-order reuse of components and evaluates interconnection information (at reuse time) to guide the selection of the appropriate pieces to integrate or to "adjust" the results of the composition.

From the point of view of domain analysis, first-order information is acquired, but enriched with interconnection relations. Interconnection information is necessary to capture context sensitive aspects to be used at implementation time.

Examples

First-order reusers usually apply some form of the interconnection model. In some cases this is done informally, as when programmers keep in their heads a record of how different pieces of an implementation relate to each other. In other cases, the interconnections are made explicit, but are verified manually. Instress, the module interconnection language in the Inscape environment [Per87b] directly supports the semantic interconnection model.

Mittermeir and Oppitz [MO87] have proposed an analogue of the database paradigm for program reuse where "data" is substituted for "programs". A Software Base system, supports the storage, retrieval, verification of security and integrity constraints and partial composition of program modules. The modules are classified by Task-type and Program-type. The initial specification of a system is an Application Lattice, a "composition skeleton" consisting of a set of typed module-slots and enriched by interconnection assertions. Lattices specify the typical structure of a system, the assertions specify the types and order of execution of the component programs—integrity and scheduling constraints. Application composition—completing the application lattice—is like formulating a query against the SB. The result is the (manual) join of several uniquely determined SB entries that fulfill the assertions in the lattice definition.

The refinement subsystem in Draco [Nei84] is basically compositional but interconnection information is introduced in the form of (context-sensitive) applicability conditions for the refinements.

4.3.3 Transformation

A *transform* is a formula, $p, c \rightarrow a$, where p is called a pattern, c are applicability conditions and a is an action. Transforms are usually applied to formal specifications. If the pattern matches some part of the specification, and the applicability conditions are met, then the bindings defined by the pattern are instantiated in the action a and the action is performed. There exist many forms of transforms. In some, the action part of the rule, a, is a syntactic pattern that is substituted for the pattern p; in other cases it is a procedure to be executed, or a description of a goal to be achieved by some mechanism. [Bax87e] discusses several criteria for categorizing transformation systems: the form and richness of transformable specifications, the capability of the specification language to describe final implementations, the power of individual transformations, grouping of transformations, mechanisms to control the application of transformations, and the ability to record and replay derivation histories (a second-order feature).

Composition operators are particular forms of a general definition of transform where: the specification has the form of a parse tree; the pattern p identifies parts of the tree, and the rule $p \rightarrow a$ associates a pattern with a composition operator, $\otimes_i$, which executes under the conditions defined above.

The reuse equation for pure transformational systems is:

$$(T(S, RI^{(2)}))(S) = I_S$$

where reusable components within the MoT, $RI^{(2)}$, are transforms which are evaluated at reuse time, under the control of navigation tactics. For example, Burstall and Darlington [BD77] have proposed a "folding" and "unfolding" strategy for the transformational implementation of recursive equations as iterative programs. Thus, S is the class of specifications that can be represented as recursive equations. I_S is some subset of the class of iterative problems. The MoT is a three step process [BD77, p. 48]:

1. Make any necessary *definitions*.
2. *Instantiate*.
3. For each instantiation *unfold* repeatedly. At each stage of unfolding:
   - Try to apply *laws* and *where-abstractions*.
   - *Fold* repeatedly.

$RI^{(2)}$ is the set of syntactic transformations called: definition, instantiation, unfolding, folding, abstraction and laws, which are independent of any particular recursive specification.

The tactics determine which transforms to apply where and when, and produce a transformational derivation, a plan for the generation of an implementation of s. The plan is *composed* (e.g., sequential application) out of plan components, such as, individual transforms or pre-existing derivations.

From the point of view of domain analysis, two kinds of second-order information must be captured: plan components and tactics for plan composition. Examples of tactical information are the performance tactics in Draco, the dependency networks and performance goal structures in IMS+PCL [Bax87c], or the formal developments in Paddle[Wil83].

Special cases: application generators and compilers

Compilers and application generators are reuse systems in which a powerful translation procedure is being reused. They are special cases of transformation systems in which $RI^{(2)}$ consists of a single, procedural transform. In principle, the specification, submitted to the system, s, is composed of two parts: one that specifies the functional aspects of the specification, $s(f)$, and another one that parametrizes the application of the procedural transform, $s(p)$, for example, setting compilation options, choice of implementation guided by performance constraints or target implementation language. The engine T is the processor running the compiler or application generator.

$$(T(s(p), \langle \text{ compiler } \rangle))(s(f)) = I_s$$

Summary

From the point of view of identifying requirements for domain analysis methods, we distinguish the following aspects of reusable information.

- Composition model :
  the components capture first-order, context-free, domain-specific information. Components are not evaluated at reuse time.
- Interconnection model:
  first-order domain-specific information captured only by components, context information captured by explicit representation of interconnections between components. Interconnection information is evaluated at reuse time.
- Transformation model:
  transforms are second-order components evaluated at reuse time. "Navigation" tactics are required to guide the evaluation steps. Special case: single procedural transform (e.g., some application generators and compilers).

We will use the term, *first-order*, to refer to the first two models of reuse.

4.4 Implications for domain analysis

> **Summary.** Based on the distinctions made about MoTs in the previous section we identify different forms of domain analysis for each order of reuse. An analogy between first-order reuse and the processing of a language provides insights about domain analysis as a form of language acquisition.

A domain analysis for a first-order reuser must produce two kinds of information: information on how to decompose specifications in S, maps of the form $s \rightarrow I_s$ associating implementation fragments to specification fragments, and information on the implementation map to be evaluated at reuse time.

A domain analysis for a transformational reuser must acquire information on how to describe specifications, on transforms to be evaluated at reuse time, and on tactics to guide the transformation process.

4.4.1 Acquiring first-order information

The task of first-order reusers can be summarized as:

1. produce a syntactic decomposition (or, parse tree) of a specification,
2. map the atomic elements in the decomposition to implementations,

First – order Reuse		**Language processing**
MoD object definitions	...	Vocabulary in the language
Models of applications	...	Sentences in the language
Decompostion relations	...	Grammar productions
MoD state	...	Grammar, concrete syntax
MoD Representation Language	...	Abstract syntax[DGKLM84]

Domain Analysis		**Language acquisition**
Purpose : enhance competence of reuser	...	Purpose : enhance linguistic competence
Method : acquisition and induction over exemplars	...	Incremental grammar induction

Figure 4.4: Term correspondence: Domain analysis/Language acquisition

3. compose (or assemble) an implementation guided by the decomposition structure (and perhaps context information).

Step 1 is a recognition step. Steps 2 and 3 are generation steps. To parse a specification, and to generate an implementation, the reuser must have knowledge of which are the appropriate decompositions, of the relation between parts and of the implementation map.

Analogy—first-order reuse and language processing

First-order reuse can be compared to processing sentences in some language. Figure 4.4 summarizes the correspondence between terms in this analogy. [DGKLM84] defines an abstract syntax as a set of syntactic categories used to describe the structure of the sentences in a language together with composition operators that describe how sentences in one syntactic category are constructed in terms of more primitive sentences. MoDL plays the role of an abstract syntax as it defines the set of objects that are considered abstract representations of well-formed specifications. [WA86] discusses the analogy between abstract syntaxes and object bases.

Grammars are convenient abstractions for modeling the process of language comprehension (as parsing) and generation. The interconnection model of reuse embedded in MoDL is also an abstractions of the process of mapping a specification into an implementation. A model of the domain is an abstraction of the knowledge of the user in the same sense that a grammar is an abstraction of the knowledge of language. It allows the reuser to recognize specifications and to generate implementations form the decomposition of a specification.

This information captured in a MoD determines the competence of the actual reuser P. A similar notion of competence, well-known in linguistic theory, originated with the work of Chomsky. [Dal76] defines *linguistic competence* as the set of learned principles that a person must have in order to be a speaker of a language. It is distinguished from *linguistic performance*, the translation of this knowledge into action. Performance accounts for competence as well as other factors such as memory, perception, fatigue, distraction. etc.

An underlying assumption of that view is that competence can be described independently of the processes of comprehension and production [WC83, p. 10]. Context-free grammars are an example of such separability.

The linguistic perspective provides insights about the generation of a domain-specific language in domain analysis. The reuser P is the system that recognizes specifications and generates implementations. The analogy can be extended to domain analysis (for first-order reuse) and the process of language acquisition as grammar induction.

"Pure" domain analysis is a form of learning that we do not know how to characterize, much less to approach systematically. Such learning is a part of the routine work of systems analysts, requirements analysts and knowledge engineers. In most cases, learning occurs at a personal level, and can only be reused by the subject, (or, indirectly, by those who hire the subject). The resulting systems definitions, requirements specifications or knowledge bases capture only a "snapshot" of the information gathered, but not the experience accumulated by the analyst over many such exercises.

A systems analyst with experience in some class of applications can be regarded as a walking Model of (that) Domain. The competence and efficiency of an analyst develops from the analysis of a number of instances of problems in the domain leading to the definition of a vocabulary and semantics for specifications in a *domain-specific language*. The aim of a practical domain analysis method is to realize this form of learning systematically.

The structure of a MoD is derived from the MoT realized by the target reuser. Having a fixed MoT, the competence depends only on the MoD structure. This assumption corresponds to the separability assumption mentioned above.

Given a goal-state $\mathcal{G}$ for the reuse system characterized by some set $\{S, I_S\}$, the MoD must capture the information necessary for describing the specifications S, and producing the associated implementations I_S. The competence of a reuse system is enhanced by evolving the MoD of the system, so that each new state captures a more general form of the domain language; i.e., provides a better coverage of S.

4.4.2 Summary

The task of acquiring and evolving a first-order MoD involves:

Input: specification components for encoding instances $s \in S$, and instances of the specification-implementation map I_s.

Outputs: a model of the domain capturing information for the encoding and implementation of specifications in the domain.

Method: induction over instances of specifications and implementation maps

Support: MoD, a knowledge representation structure to, 1) capture the information on reusable first-order components, and 2) support induction.

The purpose of the MoD is to provide a unified representation with well-defined semantics so that instances of specification or implementation components can be subject to classification and generalization.

4.4.3 Acquiring second-order reusable information

Transformational systems

A similar view applies to the acquisition of second-order reusable information if we view the reuser as a constructor of a transformational derivations with transforms as first-order components.

Input: specification components and transformational derivations of implementations.

Outputs: a model of the domain capturing information for the encoding of specifications and the generation of transformational derivations (transforms and transformation tactics).

Method: induction over instances of specifications and derivations.

Support: a knowledge representation structure to, 1) capture the information on transforms, and 2) support induction over derivations.

Representation formalisms for capturing derivation have been proposed; for instance, the Production Control Language in [Bax87c], transformational developments in [Par86], a design calculus in [Sin87], formal developments in [Wil83]. These languages provide the support needed to induce grammars for second-order information.

Some cases of analysis of derivations and induction of generic transformational tactics have been reported in the literature. [SA88] reports on general techniques for the synthesis of algorithms for sorting, search, and graph marking, and other restricted problem domains, using problem spaces as a framework for analysis.

[Par86] provides a detailed analysis of transformational development in the domain of recognition algorithms, parsing algorithms and transformation of context-free grammars. While the generalization process itself is not addressed, many generic transformational tactics or plans are identified. For example: "divide and conquer", "generator incorporation" "dynamic programming" techniques, "partial evaluation" techniques, "finite differencing", etc. [Par86, p. 218].

In summary, formalisms for the description of transformational developments can be used as basic language for representing instances of exemplar transformational implementations. Once the exemplars are encoded in a unified language with well defined semantics, analysis and induction techniques can be applied to them.

Application generators

Application generators, AG, have been categorized as a special case of transformational systems, in which $RI^{(2)}$ consists of a single, powerful, procedural transform.

Specializing the outline above, a domain analysis for application generators must capture at least two kinds of information: 1) a language for the specification of applications, and 2) a procedure for translating specifications into implementations.

[Cle88] suggests that a process of domain analysis must be conducted to acquire the specification language. No hints are available in the literature on the systematic development of the generator itself: "the designer writes a program that translates the specification language into the desired product"[Cle88, p. 31]

We believe that the inductive method outlined above also applies to the definition of the generator itself. The literature offers some evidence that such a process does take place; for instance, [CG82] [HKN85] identify properties of the structure of generators for business applications; [Cle88] identifies structural components such as lexical analyzers, parsers, semantic analyzers and code- or table-driven code generators. The results of such inductive process have been partially captured in the form of application generator-generators such as Draco [Nei84], Stage [Cle88] or SSAGS [P+82].

Using the reuse equation, the *domain designers*[1] survey the set of known translation mechanisms (T_k) and domain-specific implementation techniques ($RI_k^{(2)}$):

$$(T_k(s, RI_k^{(2)}))(s) \;=\; I_s$$

combinations of possible values are evaluated, and the selected combination is assembled into a generator program

$$(T_k, RI_k^{(2)}) \rightarrow \langle AG \rangle$$

AG becomes a monolithic translation function from the language of specifications into the language of implementations:

[1] A term introduced by Neighbors [Nei81] to refer to the role of the person in charge of developing a Draco-domain—a language parser, and libraries of refinements and transformations. More recently, Cleaveland [Cle88] applied the term to the designers of application generators.

$$AG(s(p), s(f)) = I_s$$

The field of translation mechanisms is a well-researched field; the field of domain-specific implementation techniques is not; and the integration of both is not trivial. This is a reason why, in practice, domain designers produce generators only for restricted and well-defined problem domains.

4.5 Summary

The purpose of this Chapter was to identify instances of the model of the reuse task to define the scope of a domain analysis method. We have categorized MoTs in terms of the order of reuse they realize, and we use the notion of orders to examine the composition, interconnection and transformation models of software construction. First-order reuse is based on composition of implementation parts, possibly constrained by context information. Higher-order forms of reuse are based on the generation of implementation parts at reuse time.

We have presented an analogy between first-order reuse based on an explicit model of the reuse task, MoT, and the process of language processing based on a definition of language structure. This analogy links the process of domain analysis of first-order information with forms of language acquisition.

We propose that practical domain analysis be realized as a process of incremental acquisition and induction of domain-specific information from exemplars. Competence enhancements in a reuser can be realized by incremental induction of the specification language under the constraint that all specifications expressible in the language can be implemented.

Road map to Chapters 5, 6 and 7

The scope of the domain analysis methods is defined both by the model of reuse chosen, and by the limitations imposed by the MoD representations.

In Chapter 5 we propose a structure for first-order MoDs: a statement of requirements and the definition of a meta-language to describe MoDs. The MoD Representation Language is to domain-specific information like the BNF notation is to a formal grammar.

In Chapter 6, a method for practical domain analysis based on the incremental induction of MoDs is presented.

Finally, Chapter 7 discusses efficiency enhancements in the execution of the reuse task.

Chapter 5

First-order Model of a Domain

Chapter summary. A structure for a first-order MoD is presented. This is the second step in the domain engineering strategy outlined in Figure 3.4.

The primary purpose of a first-order MoD structure is to support the representation of the semantics of first-order information, and the application of induction procedures over that information.

It serves other purposes as well: it is a persistent repository of domain-specific information relevant to a reuse system; it defines the boundaries of the competence of the reuse system; it acts as a specification for reuse infrastructures, and it has the potential for being used as a tool for communication and training.

The state of a MoD is defined by the information encoded in the MoD structure at a given time. The MoD state consists of representations for parts, parts relations and parts decomposition that correspond to a grammar for recognizing specifications in the problem domain. Implementation relations associate to these parts information that a reuser may employ to generate executable versions of the specifications.

A language for representing first-order information is presented together with definitions for well-formedness, integrity, completeness and pertinence for the information in a MoD state.

5.1 Introduction

The MoD structure defines a language, MoDL, for the representation of the semantics of specifications in the problem-domain and for the representation of information on implementations. The state of the MoD plays the role of a grammar for the recognition of specifications in the domain and for the generation of implementations.

Our method for practical domain analysis involves acquiring and generalizing domain-specific information expressed in the MoD language. Thus, the primary purpose of a first-order MoD structure is to support the representation of the semantics of first-order information, and the application of induction procedures over that information. MoDL fulfills the role of a "meta-language" with respect to the domain-specific grammar that is the aim of the domain analyst.

5.1.1 Requirements on a MoD structure

Requirements derived from the MoT

A structure for a MoD is proposed on the assumption that the model of software construction, or MoT, is domain-specific first-order reuse. The MoD structure must provide support for representing the semantics of problem-specific abstractions—statements in the domain language—as well as the decomposition and implementation of these abstractions. The following items of information are required to support first-order reuse:

- A vocabulary for the specification of data and processes in the problem domain.
- Decomposition relations and context sensitive constraints over the parts in a decomposition. In particular:
  - Structural constraints describing relations among the parts
  - Communication or dataflow constraints between parts
  - Timing, or scheduling constraints between parts
- Implementation components for data and processes in the problem domain.
- Interconnections between implementation parts.

Support for the analysis of competence, L_c

MoDL captures the semantics of a domain-specific language for specification of software systems. The competence of a reuser is enhanced by identifying patterns of reusable information that allow to specify and implement solutions to a larger number of problems within the domain. In a first-order model of reuse, competence is enhanced through acquisition of new information and the classification and generalization of existing information. Hence, the structure of a MoD must provide representations for items of information that facilitate:

- The acquisition of new information.
- The identification of similarities between items of information that are relevant to the purposes of reusability.

- The definition of classes of similar items and of generalization relations among classes of items.

Support for evolution

Assessing the impact of a change within a model of a problem domain is difficult because of the complex interdependencies of the information in the model. Our experience in "maintaining" Draco-domains created by others has demonstrated that it is an extremely difficult task even in restricted problem areas. In small problem domains, it often seems easier to design a new model from scratch than to understand which parts need repair and how to propagate changes. To perform a controlled evolution of the MoD we need to be able to answer the following questions efficiently:

- *What* is the purpose of this item of information?
- *How* did it get to be a part of the MoD?
- *Why* is it reasonable to have this abstraction in the model?

The purpose of an item of information in a MoD is defined by its role in the specification or implementation of exemplars in the problem domain—its *semantic definition.* Answers to how and why questions constitute the *justification* for a concept definition.

Neighbors [Nei88b] has recently suggested the need for *domain rationales.* We define the *rationale for a MoD state* as the collection of justifications for the concept descriptions in a state of the MoD.

5.1.2 Constraints derived from T_L

The set of technologies available to support MoD evolution also constrains the form of the MoD structure. For the purpose of demonstrating domain analysis methods, we assume

- that a person, the domain analyst, applies the methods.
- a formal representation language for encoding a model of the problem-domain with "natural language escapes" as a concession to domain analysts.
- a database system to store the descriptions of concepts in the problem domain. We assume that the database management system will relieve the analyst of the routine tasks of locating and retrieving concept descriptions with specified properties [Rob86] [Hyp88].

5.2 A first-order MoD structure

A MoD is a collection of related concept descriptions, or objects. The types of objects and relations are selected to meet the problem-specific, task-specific and evolution requirements.

5.2.1 The MoD graph

The structure of the MoD can be described as a graph. The nodes in the graph are concept descriptions. Labelled arcs in the graph represent relations between concept descriptions.

The arcs in the MoD graph are categorized into four classes. S-class arcs represent information relevant to the *specification* of systems in the domain. I-class arcs represent information relevant to the *implementation* of specifications in the domain. G-class arcs represent classification (*instance-of*) and generalization (*is-a*) relations between concept definitions. The J-class arcs relate concept definitions with justifications (e.g., design assumptions, problem domain goals, rationales for a design plan)

Each class of arcs determines a sub-graph, a projection of the total MoD making explicit a particular viewpoint on the concept nodes. Within each graph, subsets of the represented relations may be chosen to define more specialized viewpoints such as structural, dataflow, or timing relations, design assumptions, problem-level goals, etc. We now summarize the four viewpoints that coexist in the structure:

1. *The S-graph*, or specification graph, captures the semantics of a domain-specific vocabulary needed to specify and implement some typical system. The answer to the question, "What does this S-level concept description mean?" is the set of concept descriptions related to the subject by S-type arcs—its *definition neighborhood.* For example, if the subject describes an activity, these relations include: the inputs and outputs of activity, the conditions under which it starts or stops, conditions that must be true before it starts or after it is completed. S-graph objects and relations are presented in Section 5.3.

2. *The I-graph*, or implementation graph, captures relations relevant to software implementation. The I-relations of a specification concept describe a plan for the implementation of a specification, relations to the implementations of other objects, and design trade-offs and assumptions embedded in the implementation. Examples of I-relations are: implementor (between S-object and implementation objects), obligation/satisfaction (between implementations of related S-objects), and preconditions for an implementation plan. I-graph objects and relations are presented in Section 5.4.

3. *The G-graph*, or generalization graph, encodes relations between concepts and instances of those concepts and specialization relations between classes of

concepts. G-graph relations are discussed in Section 5.5.

4. *The J-graph*, or justification graph. is used to make explicit assertions about specification or implementation objects that are needed to understand and evolve a MoD. J-relations provide the evidence, or support for the belief in concept descriptions that is needed to answer How and Why questions. J-graph relations are presented in Section 5.6.

5.2.2 Representation

A first-order MoD is a collection of model definitions (Figure 5.1). Each model is a representation of an exemplar in the problem domain. It captures information on how to specify some typical problem and on how to implement it. A model consists of a collection of parts which are either specification objects or implementation objects.

Specification information is represented in terms of three kinds of objects: entities, activities and assertions. The adequacy of these categories for the description of software applications has been confirmed empirically by a large number of methods and languages used in software engineering [Lei87], and by theoretical developments [AFS+86] [BMW84] [CW85] [LST84] [Gog86].

Implementation objects capture domain-specific design information and implementation plans, and concrete implementations in a programming language.

Both the models and the objects are related by parts-of, instance-of and subclass-of relations. Models and objects—the nodes in the graphs—are represented as bundles of property-value pairs. The property names correspond to those of the relations with other objects, the labels on the graph arcs.

Attribute-value representations are common in knowledge representation in artificial intelligence, and have also been applied to the representation of requirements in software engineering [Gre84]. An attribute-value representation facilitates the definition of procedures for classification and generalization based on explicitly represented attribute values.

Finally, the need to support the controlled evolution of the MoD requires justification assertions on concept descriptions.

A full grammar for MoDL is included as Appendix B. The language is based on the Requirements Modeling Language, RML, whose semantics are described in [Gre84]. RML has been extended and the formal representation has been augmented with natural language descriptions.

5.2.3 Case study: A disk drivers domain

We introduce the representation language using an example from a real-life problem domain, the domain of disk drivers in an operating system. The domain is small, but it exhibits the features of problem domains found in real-life software development

```
⟨MoD⟩ ::= { ⟨ model-definition ⟩ } +
⟨model-definition⟩ ::= ⟨ m-subject ⟩ ( { ⟨ factual-properties ⟩} )
                                        in {⟨ m-classifier ⟩}+
                                        isa {⟨ m-superclass ⟩}+
                                        with {⟨ m-def-properties ⟩}
⟨ m-subject ⟩ ::= ⟨ user-defined-model ⟩
⟨ m-classifier ⟩ ::= ⟨ model ⟩
⟨ m-superclass ⟩ ::= ⟨ model ⟩
⟨ m-def-properties⟩ ::= ⟨ m-property-category⟩{⟨ m-def-property⟩}
⟨ m-def-property⟩ ::= ⟨attribute⟩ : ⟨ m-property-value⟩
⟨  n-property-category ⟩ ::= id| part | constraint | supported-by
⟨   -property-value⟩ ::= { "[" ⟨object ⟩ "]" }⟨ object ⟩ [ ⟨bindings ⟩ ]
                                | ⟨ model ⟩ [ ⟨bindings ⟩ ]
⟨ user-defined-model ⟩ ::= ⟨ upper-case-identifier ⟩

⟨ model ⟩ ::= { ⟨ object-definition ⟩ } +
...
```

Figure 5.1: Fragment of a grammar for MoDL (Appendix B)

situations: richness of information, intricate implementation detail, technology dependencies, shifting customer and developer goals, economic constraints. We have use fragments of the representation of a disk driver in that domain to illustrate aspects of MoDL.

The environment of this actual reuse situation is a company which develops and markets systems software for micro-computers. One of its products, SDOS, is a time-sharing operating system. SDOS provides a file system which assumes the presence of disk drivers for performing sector I/O to the specific hardware on which the operating system happens to run. The reading and writing operations on disks presume that the disks are formatted in a way readable by the electronics. Thus, there has to be software for formatting disks.

Fragments of this domain are further developed in Chapters 6 and 7 and [Ara88b].

5.3 Specification information

Figure 5.2 illustrates the definition of a model for an exemplar in the problem domain, the Conrac Logical Floppy disk-driver. The model definition identifies the application (**id**) and describes its major parts (**parts**). The model consists of a collection of activities, entities and assertions.

CONRAC-LOGICAL-FLOPPY-DRIVER *in* MODEL

id

date: "3-11-88"
source: "SDOS Systems Implementor's Guide"
implementation: "IOCONRACLF.ASM"

supported-by TALK-TO-CLF and PRE-FORMATTED and SAFETY, ...

parts

initialization: [TALK-TO-CLF] CLF-INIT
read: [TALK-TO-CLF]CLF-READ
write: [TALK-TO-CLF] CLF-WRITE
wait-till-done: [SEPARATE-INITIATION-COMPLETION]CLF-WAITDONE
status: [ESCAPE-DOOR]CLF-STATUS
escape: [ESCAPE-DOOR]CLF-CONTROL
sector-buffer: CLF-BUFFER *isa* DD*entity
device-description: DDD*CLF-DEVICE-CONTROL-BLOCK
sector-info: DDD*CLF-RDSI
disk-info: DDD*CLF-DISKINFO
...

Figure 5.2: Definition of an exemplar in the Disk Drivers domain

The activities, usually associated with "entry-points" in the domain of disk drivers (e.g., controller initialization, read and write sector) describe the operations performed by the driver. The entities describe structures operated upon by the driver (e.g., disk control blocks, buffers, controller interface information). Assertions at the model level describe the goals for the application, e.g., TALK-TO-CLF: "Implement entry points compatible with Conrac Logical Floppy hardware"; design decisions, e.g., ESCAPE-DOOR "Leave an 'escape door' for accessing the controller's hardware from the operating systems, to implement functions not foreseen at the time of Operating System design"; or structural properties.

Each model definition is analogous to the definition of a particular sentence (or, class of sentences) in a language for the description of disk drivers. Each object definition introduces a term in the domain vocabulary.

We collectively refer to entities, activities and assertions definitions as S-objects. The Entity-Activity-Assertion framework used here is an extension of RML [Gre84]. It offers a neutral, general purpose framework for representation that matches naturally with many popular analysis and design methodologies in software engineering. From his experience at the Rocky Mountain Workshop [Nei88b], Neighbors reports that people used many kinds of representations for describing the results of a domain analysis: data-flow diagrams, entity-relationship diagrams, semantic nets, object diagrams, and class hierarchies with inheritance. All these are compatible with the

Entity-Activity-Assertion representation chosen.

MoDL could be enhanced in a number of ways. Specialized, local syntaxes could be adopted for representing particular classes of assertions, for example, Petri-nets for defining synchronization properties. These extensions would enhance the usability of the language if adequate data-base support could be provided. Classification and generalization relations would have to be defined within descriptions in those specialized languages.

5.3.1 Entities in the problem domain

Entities represent the "things" in the world—structures, persons, places, equipment, etc.—that are part of a description. Figure 5.3 illustrates a partial representation of an entity in the domain of DISK-DRIVE-DESCRIPTION, or DDD* [1]. The entity DEVICE-CONTROL-BLOCK, or DCB, is an instance of a S-entity in the class of entities in the domain of disk drives descriptions.

Property categories are represented in boldface. The DCB is described in terms of its **parts**, the activities that produce it (**producer**) or modify it (**modifier**) and a design plan for its implementation (**implementor**). The **parts** of the DCB are labelled: doneflag, lasterror, name, etc. The values of these properties are objects of type DONEFLAG, ERROR-STATUS-CODE, DEVICE-NAME-REF and so on. The value of the properties in the **id** property category documents the conditions under which the model was incorporated in the MoD.

A description for the class of objects ERROR-STATUS-CODE appears in the lower half of the Figure 5.3. It shows ERROR-STATUS-CODE as a class in the meta-class of DDD* domain entities, and in particular, as a subclass—the *in* relation—of the class of parts of a DCB, DCB-PART. The **constraint** property category includes two constraints over the values of the property 'code' in the subject. The value of the property 'code' is set by INT*INTERRUPT-ROUTINE—an activity in the INT* domain—and the value is zero when the routine executes successfully or non-zero otherwise.

Other property categories of entities descriptions are: **association**, whose values are objects associated with the subject which change over time; **invariant**, assertions that are true of the subject for as long as it exists, **initcond** and **finalcond**, assertions that are true at the time an object becomes an instance of the class described by the subject or ceases to be an instance of the class; and **consumer**, activities that consume the instances of the subject. Collectively, these property categories provide sufficient information for the description of first-order reusable entities.

[1] Notation: The "*" notation attaches a prefix to the name of an object to indicate its domain, e.g., DD*X stands for object X in domain of Disk Drivers, or INT*Y, object Y in the domain of INTerrupt services. When absent, the objects are presumed to be in the domain of the subject.

CLF-DEVICE-CONTROL-BLOCK *isa* DDD*S-entity *with*

id

model: CONRAC-LOGICAL-FLOPPY-DRIVER

source: "SDOS Systems Implementor's Guide, p.69"

parts

doneflag: DONEFLAG

lasterror: ERROR-STATUS-CODE

name: DEVICE-NAME-REF

...

park: DCB.parkcylinder

steprate: DCB.steprateinfo

precomp: DCB.writeprecompinfo

rightnormalizetrack: RIGHT-NORMALIZE-TRACK

producer

p1: SDOS-sysgen

p2: DD*DISK-DRIVER

implementor

imp: DCB-DESIGN

ERROR-STATUS-CODE *isa* DDD*S-entity-class *isa* DCB-PART *with*

parts

code: integer

constraint

c1: code = 0 *iff* successful-completion(m)

c2: code $\neq$ 0 iff failed(m)

producer

p: SDOS-sysgen

modifier

m: INT*INTERRUPT-ROUTINE

implementor

imp: ERROR-STATUS-CODE-DESIGN in DCB-PART-DESIGN

Figure 5.3: DCB: example of S-entities in the Disk Drivers domain

5.3.2 Activities in the problem domain

Activities represent actions and events in the problem domain. Activities happen between a *start time* and *end time*; at all times in-between they are said to be active. We use the CLF-WRITE activity in the definition of the CLF driver in Figure 5.2 to illustrate the representation of an activity in the problem domain (Figure 5.4).

The following property categories are used for describing activities: **input, output, control, precond, postcond, trigger, stopcond, parts,** and **implementor.**

The values of the properties in category **control** represent S-entities that participate in the activity but that are not removed by the activity from their property value class—the subject is not a **consumer** of them. In the case of CLF-WRITE, the values of the LSN component of the Resident Disk Sector Information entity (DDD*RDSI), the Disk Control Block for the CLF device (DDD*DCB), and a disk-sector buffer (CLF-BUFFER) are the controls which define which sector in the disk must be transferred to the buffer and how to locate them.

The execution of the CLF-WRITE activity is triggered by an operating system-level request. Precondition must be met for CLF-WRITE to execute, for example, the disk controller must be free, the semantics of the predicate ready are defined elsewhere as an assertion object READY. As a result of the execution, two conditions may occur: either the disk is discovered to be writeprotected, in which case a message is returned to the operating system, or the transaction completes. If it does complete, then it is the case that the contents of the buffer have been written, and the information written on the disk has been verified.

The implementation of the subject is described by the implementation object CLF-WRITE-DESIGN and CLF-WRITE-PLAN described in Figure 5.5.

5.3.3 Assertions in problem domain

Assertions are used, 1) to define the semantics of S-objects by defining constraints on structure and behavior, and 2) to state facts about the problem domain or software design that justify the concept descriptions in the model. For example in Figure 5.4:

constraint

pass-msg: out *iff* DETECTED(WRITE-PROTECTED-DISK)

states that the output 'out: WRITE-PROTECTED' occurs if and only if it has been detected that the disk in the disk is writeprotected. In Section 5.4 we discuss types of assertions related to implementation objects and in Section 5.6 those used to justify object definitions and to capture useful background knowledge.

Assertions in MoDL are encoded using the same structure as S-entities and S-activities, but with the following property categories and values: **argument**, the free variables in the assertion; **suffcond**, assertions giving sufficient conditions for being a member of the class; **neccond**, an assertion that is true; when the subject

CLF-WRITE *isa* S-activity-class *isa* DD*S-activity *with*

id

model: CONRAC-LOGICAL-FLOPPY
alias: WRITE-SECTOR, DISK-WRITE
source: "SDOS Interface Vectors, p. 157",
"SDOS Systems Implementor's Guide, p. 148"

supported-by

purpose: TALK-TO-CLF and RELIABILITY
and ALWAYS-COMPLETE-OPERATION ...

control

sector-number: DDD*RDSI.LSN
buffer: CLF-BUFFER
dcb: DDD*DCB

trigger

driver-invocation: SDOS*DISK-WRITE-REQUEST

precond

good-RDSI: VALID(RDSI)
good-DCB: VALID(DCB)
device-ready: READY(CLF-CONTROLLER)
interrupts-operational : OPERATIONAL(INT*INTERRUPT-SYSTEM)

postcond

transaction-initiated: CLF-WAITDONE.**precond**

output

out: WRITE-PROTECTED

constraint

fail-to-complete: DETECTED(WRITE-PROTECTED-DISK)
pass-msg: out *iff* FAIL-TO-COMPLETE

implementor

p: CLF-WRITE-DESIGN

Figure 5.4: CLF-WRITE, example of an S-activity in the Disk Drivers domain

is true, **defn**, an assertion class, every instance of which gives a necessary and sufficient condition for the subject being true.

The assertion sub-language includes a first-order logic language with predicates: *in*, instance of; *isa*, subclass or specialization; =, equality, and <, time ordering. Terms are formed using property names, the function PVAL, property selection; RANGE, property value selection; *start* and *stop*, times for activities; and predicates to assert time constraints—*During, Next, Before,* and *Samebegin.* We extend the sub-language to include arithmetic equations, and we informalize it by allowing natural language strings. These "escapes" are a concession to practicality and readability, the semantics of a natural language statement are defined by the interpretation made by the reader.

5.4 Implementation information

Information on how to implement S-objects is represented using implementation objects, or I-objects. We distinguish between two kinds of objects: DESIGN and PLAN. Samples corresponding to CLF-WRITE are shown in Figure 5.5.

The definition in the upper half of the Figure corresponds to a DESIGN object that documents a design act *recovered* from an implementation ('spec' property) of CLF-WRITE with the help of an expert. The result of this act is an implementation **plan** for CLF-WRITE: CLF-WRITE-PLAN DESIGN and PLAN are MoDL built-in objects. Design plans describe time-ordered sequences of intended actions. The plan actions are represented as activities in other (subsidiary) domains.

The CLF-WRITE-DESIGN description relates a specification-level object, CLF-WRITE, with one (or more) possible implementations of that operation, CLF-WRITE-PLAN. The implementation is designed to meet two **design goals**, represented as assertion objects. For example, ALWAYS-COMPLETE-OPERATION", which states that,

> "A driver always completes a write operation in a finite amount of time, successfully or not".

Constraints on the implementation of a write operation are made explicit. For example TIME-OUT states that,

> "Failure to complete a physical operation due to hardware problems or failure to synchronize with the hardware must be detected by setting time limits for the completion of the operation"

If the operation is not completed successfully within an established time-window, it must be declared a failure. The **decision** property category enables the representation of design heuristics and justifications for the implementation adopted. For instance, the SPLIT-RESPONSIBILITY-1 assertion states that,

CLF-WRITE-DESIGN *isa* DD*DESIGN *with*

id spec: CLF-WRITE

design-goals

g1: ALWAYS-COMPLETE-OPERATION
g2: MEET-TECHNOLOGY-CONSTRAINTS
g3: RELIABILITY
g4: PERFORMANCE
g5: ...

constraints

separation: SEPARATE-INITIATION-COMPLETION
time-limit: TIME-OUT
warning-coupling:
shared-by(INT*INTERRUPT-HANDLER, CLF-READ, CLF-WRITE, WAITDONE)

decision

sep-concerns: SPLIT-RESPONSIBILITY-1 and SPLIT-RESPONSIBILITY-2

plan

p: CLF-WRITE-PLAN

CLF-WRITE-PLAN *isa* DD*PLAN

id

design: CLF-WRITE-DESIGN
source: "IOCONRACLF.ASM, p. 112"

technology

implementation-domains: DDT*
type-of-controller: Device-controller = CONRAC-CONTROLLER

obligation

must-finish-transfer: [SEPARATE-INITIATION-COMPLETION] CLF-WAITDONE

plan [SPLIT-RESPONSIBILITY-1]

step-0: SET-CMD-TYPE(CONRACWRITE)
step-1: CHECK-LEGAL-LSN(RDSI)
step-2: if 'bad' then SIGNAL(BAD-LSN) and ABORT
step-3: LOCK-INTERFACE(CLF-INTERFACE)
step-4: CHECK-MAP-CHANGE
step-5: if 'changed' then
SET-MAP-ALGORITHM(DDD*CLF-INFO.CLFMAPALGORITHM)
step-6: [INT*INTERRUPT-RESOURCE-DEALLOCATION
and DESIGN.SEP-CONCERNS]
ALLOCATE(CLF-INTERFACE)

[SPLIT-RESPONSIBILITY-2]

TRIGGER-READ/WRITE(CLF-WRITE-INTERRUPT-CODE)

Figure 5.5: CLF-WRITE, example of DESIGN and PLAN object descriptions

> "Task Level Code (DDT) code does not touch the hardware. Only computes and issues requests to interrupt level code"

The CLF-WRITE-PLAN implements the CLF-WRITE operations using activities in the Disk Driver Taskable code domain, DDT*, e.g., SET-CMD-TYPE(CONRACWRITE), CHECK-LEGAL-LSN(RDSI). Sometimes **precond**itions must be met for the plan execution to be successful. For each precondition in a PLAN object, there must be other PLAN objects that satisfy it, the **satisfies** property category. Similarly, the **obligations** defined by a PLAN object must be met by some other PLAN object. This ensures the global integrity of each implementation of the model.

Other representations for plans to assemble first-order components have been proposed, for example, plans in the Programmer's Apprentice [Ric81], dataflow schemas in IDeA [Lub86], programming plans in PROUST [WLJ85]. For the purpose of evolving and inducing more general forms of first-order implementations, the frame-based representation of S-objects/I-objects chosen appears to be adequate.

5.5 Classification and generalization

The G-graph represents the specialization/generalization and instance/class relations among MoD objects (S- and I-objects). Problem objects are organized into an ISA classification hierarchy with inheritance. Figure 5.6 shows a fragment of the G-graph in the Disk Drive Description domain, including instances and parts of the Disk Control Block entity.

MoD objects are descriptions of instances and of classes. [Gre84] introduces two ISA constraints. The extensional *isa* constraint states that every instance of a subclass is in the superclass:

$$isa(C, D) \wedge in(x, C, t) \Rightarrow in(x, D, t)$$

i.e., if C is a subclass of D and x is an instance of C at time t, then x is also an instance of D at time t.

Classes are also defined by a set of properties. The values of those properties, definitional property values, or *dpv*, are classes of S-objects—entities, assertions or activities, depending on the property. The intensional *isa* constraint states that the definitional properties of the superclass are inherited by the subclass:

$$[isa(C, D) \wedge dpv(D, i) = F \wedge F \neq \perp] \Rightarrow [\exists E\ dpv(C, i) = E \wedge E \neq \perp \wedge isa(E, F)]$$

That is, if C is a subclass of D and the definitional property value of property i for objects of class D is an object in F then there must exist a subclass E of F such that the definitional property value of property i for objects of class C is an object in E. These constraints define the notion of inheritance as conventionally used in object-oriented programming [Mic88].

The definition of *in* and *isa* relations among models and objects reflects the result of learning on the part of the analyst. Class definitions are used to capture com-

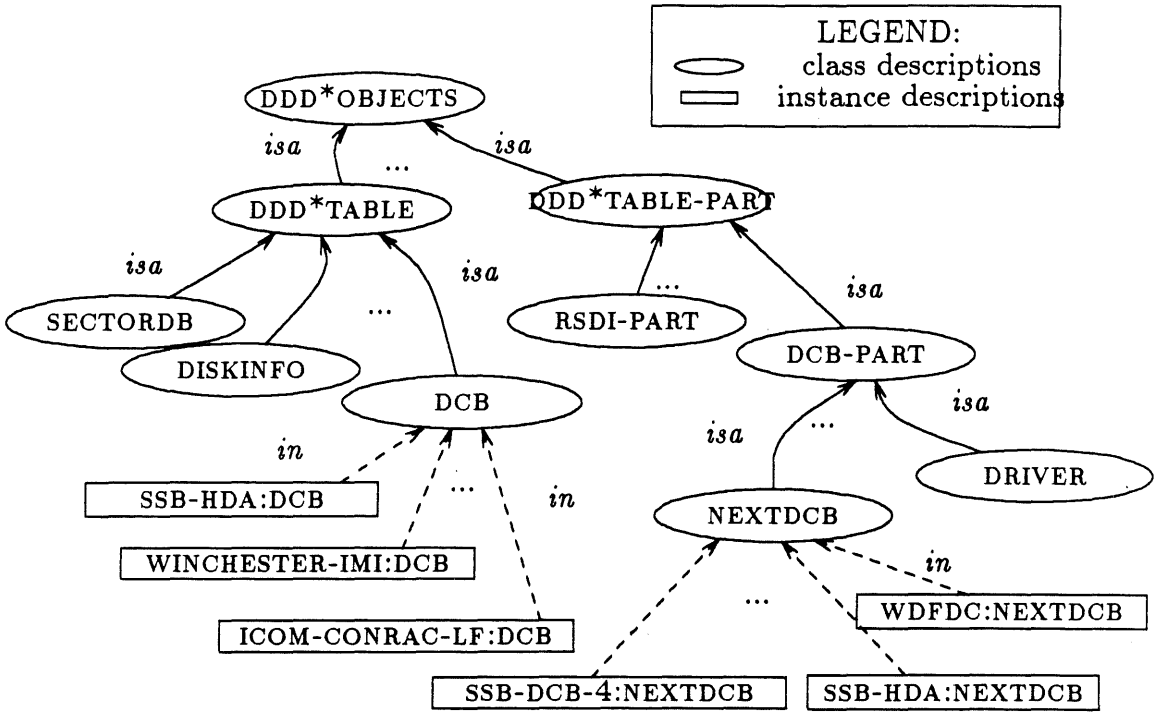


Figure 5.6: Fragment of G-graph in the DDD* domain

monalities discovered among concepts in the problem domain. The generalization of concept definitions enhances the number of models that can be specified.

The generalization of S-level objects must be accompanied by the upgrading of their associated implementation objects. Care must be taken to ensure that the more general implementation objects maintain *compatible interfaces*—**precond**, **postcond**, **obligation** and **satisfies** relations—with other more specialized versions. We adopt Perry's [Per87b] definitions of strict and upward compatibility. Given two versions of a PLAN: D_1 and D_2, D_2 is *strictly compatible* with D_1 if:

$$D_1.\textbf{precond} \supseteq D_2.\textbf{precond}$$

$$D_1.\textbf{postcond} \supseteq D_2.\textbf{postcond}$$

$$D_1.\textbf{obligation} = D_2.\textbf{obligation}$$

Given two versions of a PLAN: D_1 and D_2, D_2 is *upward compatible* with D_1 if:

$$D_1.\textbf{precond} \subseteq D_2.\textbf{precond}$$

$$D_1.\textbf{postcond} \supseteq D_2.\textbf{postcond}$$

$$D_1.\textbf{obligation} = D_2.\textbf{obligation}$$

5.6 Justifications

Object definitions in the MoD must be justified to provide a basis for revision during MoD evolution. To be able to answer, How did this concept definition come about? and, Why is it reasonable to have this abstraction in the model? we distinguish between two kinds of justification information. The How-information associated with an object is defined by properties in the **id** category. Why-information associated with an object is captured by *justification* assertions.

Properties in the **id** category include: *alias*, *date* of incorporation to the model, *source* of the definition (e.g., a reference to an implemented application, a manual), the *expert* consulted, etc. (see Figure 5.2). If an object definition was derived from other definitions a reference to the ancestors and the derivation method would be included, as in:

id ...
method: ClosingIntervalRule
property: p_i
ancestors: $a_1, a_2, ..., a_k$

which reads: induced using the ClosingIntervalRule (Figure 6.15) on the value of property p_i, over the set of ancestor objects $a_1, a_2, ..., a_k$. The **id** properties are properties of a concept definition and *not* of the concept described.

DOMAIN	TYPE of ASSERTION	EXAMPLES
Disk Drivers		
	Problem-level	Talk-To-CLF, Safety, Performance
		Meet-Technology-Constraints
	Design-level	
		Escape-Door
		Driver-Initialization-Preconditions
	Environment/Technology-level	
		Lack-of-Memory, Pre-Formatted
		No-Stealing-Diskettes
		Multi-Tasking, Constant-OS
DD Taskable Code		
	Problem-level	
		Appropriate-use-of-disk-I/O
		Read-Bootstrap-Condition
		Interface-as-Resource
	Design-level	
		Separate-Initiation-Completion
		Split-Responsibility-1
		Implement-Disk-Drives-Method
	Environment/Technology-level	
		Sector-Sequential-Access-Efficiency
		Short-Access-Time, Constant-NSTP
		Map-Performance-Difference
Interrupt Services		
	Design-level	
		Interrupt-Resource-Deallocation
		Interrupts-Not-Interruptable
		Measuring-Delays
	Environment/Technology-level	
		Long-Operations, Stopped-Motor
		Prevent-Erosion
Disk Drives Descriptions		
	Problem-level	
		Device-Description, No-Invariance
		Signal-of-Completion-Needed, Local-Data
	Design-level	
		Separate-RDSI-from-buffer-body
		Naming, Enumerating-Devices
		Unified-Signal-of-Completion
	Environment/Technology-level	
		DMA-Historic-Reason

Figure 5.7: Fragment of background assertions used in justifying the CLF model

Figure 5.7 illustrates various kinds of justification assertions in the context of a domain network for the construction of disk drivers [Ara88b].

A justification relation associates MoD-objects with justification objects. For example, in the definition of CLF-WRITE (Figure 5.4) the property "justification" in category **id** invokes the following assertions in support of the definition:

supported-by (TALK-TO-CLF and RELIABILITY
and ALWAYS-COMPLETE-OPERATION ...

When assertion objects are refered to justify particular attributes or values within a definition, we use brackets as an abbreviated notation, as in

...
plan
[SPLIT-RESPONSIBILITY-1]
step-0: SET-CMD-TYPE(CONRACWRITE)
step-1: CHECK-LEGAL-LSN(RDSI)
...

in the definition of the CLF-WRITE-PLAN in Figure 5.5. Similarly, justification assertions are used to represent design goals or heuristics, and implementation assumptions (as shown in 5.5), or properties of the environment of the drivers, for example, NO-STEALING-DISKETTES states that

> "Assume that nobody will take diskettes away without explicitly notifying the OS".

as well as other types of background knowledge, for instance the design heuristic SEPARATE-RDSI-FROM-BUFFER-BODY states that, "RDSI's and the body of the buffers are allocated separately" which is a left-over from older days: DMA-HISTORIC-REASON

> "To support early DMA transactions buffers would start at page boundaries"

Justification objects have as value an assertion whose arguments can range over any type of object in MoDL. The relations supports, supported-by, negative-supports, negative-supported-by are used to define *justification networks* for MoD objects. Examples of such networks are discussed in Section 6.4.3.

5.7 Properties of a MoD state

Summary. Well-formedness, integrity, sufficient completeness and pertinence of the information in a MoD state are defined.

5.7.1 Well-formedness

Well-formedness is a syntactic property of a MoD state. A MoD state is well-formed if all the model and object descriptions in the MoD are well-formed.

> **Well-formedness Rule.** A concept description in the MoD is well-formed if it can be derived from the grammar of MoDL.

The MoD grammar for concept descriptions, based on the grammar for the Requirement Modeling Language, RML [Gre84], and is presented in Appendix B.

5.7.2 Integrity

The problem of integrity is the problem of ensuring that the data in the MoD is accurate. Inconsistency between two object descriptions representing the same "fact" about the problem domain is an example of lack of integrity.

Integrity constraints similar to those defined for relational databases—entity and referential integrity [Dat82]—apply to states of a MoD. A rule for the integrity of definitions states that no object definition can have a null name. All definitions in a state must be distinguishable. Definition names, e.g., DISK-CONTROL-BLOCK, CLF-WRITE, SECTORDB, provide unique identification. This rule relates to a basic precondition for first-order reuse: specification implementation parts must be *named.*

> **Integrity Rule for Concept Definitions.** Every concept definition must have a unique name.

Similar considerations lead to a rule for referential integrity. It is often the case that the value of a property in an object definition refers to a property value of another object definition. For instance,

control
sector-number: DDD*RDSI.LSN

is asserted in the definition of CLF-WRITE, and refers to the value of the assertion 'lsn' component in the definition of RDSI object in the Disk Drives Description domain. Clearly, the value of RDSI.LSN must exist for CLF-WRITE to be well-defined.

> **Referential Integrity Rule.** The property value of object definitions in a MoD state is either null—the property does not appear in the definition—or it is the name of a class of object definitions in the MoD same state.

In particular, the referred object may be the subject itself. The RML representation reserves the identifier *self* to refer to the subject. As a default, if an object name is not specified, self is assumed.

A third constraint, *implementation integrity*, is peculiar to this kind of database. This constraint derives from the requirement that the MoD state capture sufficient information to implement any specification that can be represented with S-objects in the state.

> **Implementation Integrity Rule.** For every S-object definition s in the MoD there exists at least one I-object describing an implementation of the object s. Furthermore, **preconditions** and **obligations** defined for the subject are satisfied by I-objects implementing the remaining S-objects in each model containing s.

We now discuss two properties of a well-formed MoD state: completeness and pertinence. Completeness is defined *relative* to a set of exemplar applications in the domain.

5.7.3 Sufficient completeness of a MoD

The notion of sufficient completeness is based on our hypothesis regarding problem domains (Section 2.5). The completeness of a model of a domain is judged only with respect to a known class of problem descriptions.

A MoD is sufficiently complete relative to a set of applications if, 1) each one of them can be specified as a collection of S-object definitions, and, 2) an implementation can be composed using the associated I-objects.

(This definition parallels that of sufficient completeness for a logical theory [Yue87]: a consistent logical theory is sufficient complete with respect to a set of formulas G, if and only if for the set of axioms contained in the theory, there exist proofs for all the formulas in G.)

5.7.4 Pertinence of object definitions

Sufficient completeness requires that the collections of S- and I-objects in the MoD have a set of concepts sufficient for describing the semantics and implementation of a given class of applications. *Pertinence* deals with the other side of the coin. The objects defined should be necessary as well: for all object definitions o in the MoD there exists at least one model definition m such that o is necessary for specifying, implementing, or justifying m.

5.8 Summary

We presented a representation language for first-order, domain-specific information to be reused in software construction. The language supports:

- the representation of domain-specific information for the specification and implementation of first-order reusable components,

- the representation of classification and generalization relations, and
- the justification for the information in the MoD.

Domain-specific information on data and processes in a domain is represented in terms of specification objects of three kinds: activities, entities and assertions on both. Implementation information is captured as design acts and plans.

Items of information are represented as frames of property-value pairs, or MoD objects. The selected properties define semantics relevant to first-order reuse, and facilitate the identification of similarities and the application of generalization procedures. Explicit constructs are provided for the representation of classes and generalization hierarchies with inheritance.

Finally, different kinds of assertion objects are introduced to provide justification for MoD objects—problem and performance goals, design goals, design rationales. The relations defined in MoDL allow us to organize justifications into dependency networks.

A grammar for the representation of MoD states defines well-formedness conditions on first-order domain-specific information. Integrity constraints analogous to those in relational databases ensure the accuracy of the information captured by the MoD.

Chapter 6

Evolution of Models of Domains

Chapter summary. The purpose of domain analysis is to enhance or maintain the level of performance of a reuser. This is achieved through the incremental evolution of the MoD associated with the reuser. Each evolution step is an atomic transaction that maintains the well-formedness and integrity of the MoD state. The effect of an evolution step is to modify the competence of the reuser. The purpose of a sequence of evolution steps is to approximate a desired level.

The execution of an evolution step involves:

1. Identify an opportunity for a change (or, evolution trigger) that may result in an improvement in the competence of the reuser.
2. Acquire a definition of the desired change.
3. Apply the change to the MoD state.
4. Propagate the effects of the change to satisfy the integrity constraints on the MoD state

We distinguish between monotonic evolution (new information is added to the MoD, and existing information is retained) and non-monotonic evolution (existing information is removed from the MoD.)

An evolution step involves acquiring information from a domain expert and revising existing concept descriptions. The scheduling of the evolution steps depends on an external cost-benefits analysis. The MoD data base may be populated using the same incremental steps. However, for practical reasons, it is advantageous to set-up a MoD structure in an initial, non-incremental phase.

6.1 Introduction

Summary. A reuse infrastructure evolves for the following reasons: inadequate information in the MoD, an inappropriate specification or a faulty design or implementation. This chapter focuses on the first form of evolution. Chapter 7 discusses aspects of the second form of evolution. A MoD is evolved to enhance the competence of a reuser. The evolution may be monotonic or it may require the revision of previously held beliefs about the problem domain.

6.1.1 Triggers of evolution

Evolution triggers are events that suggest the need to change the state of the MoD. Evolution triggers can be the result of the analysis of the MoD or of a reuse infrastructure derived from the MoD. The analysis may be static or dynamic. The purpose of a static analysis of a state of the MoD is to verify that the MoD is well-formed and that the MoD integrity constraints are satisfied. A dynamic analysis requires that the MoD or the reuse infrastructure be put to work.

We distinguish between external triggers (those generated in the environment of the reuser) and internal triggers, which are a side-effect of the analysis and transformation of the state of the MoD (Figure 6.1). Sources of external triggers are:

- Changes in the "common knowledge" of the domain. For instance, experts discover new useful concepts; or, they become aware of the relevance of some piece of knowledge they have been using implicitly; or, new problems are defined to be part of the domain.
- Feedback from the target reuser (Figure 6.2):
  - flaws in the design or implementation of the reuse infrastructure,
  - inappropriate problem specification,
  - inadequate domain-specific information in the MoD.
- Hypothetical Reuse, a form of emulation of a MoT-driven reuser by the domain analyst (Section 6.5.2.)

The sources of external triggers are the domain experts volunteering new information —as *teachers*; the domain analyst performing hypothetical reuse and the actual reusers —as *critics*.

Internal triggers result from, 1) the application of general purpose classification and induction heuristics to the information in the MoD, or 2) from an underlying process of analogical reasoning wherein the analyst uses information on a problem

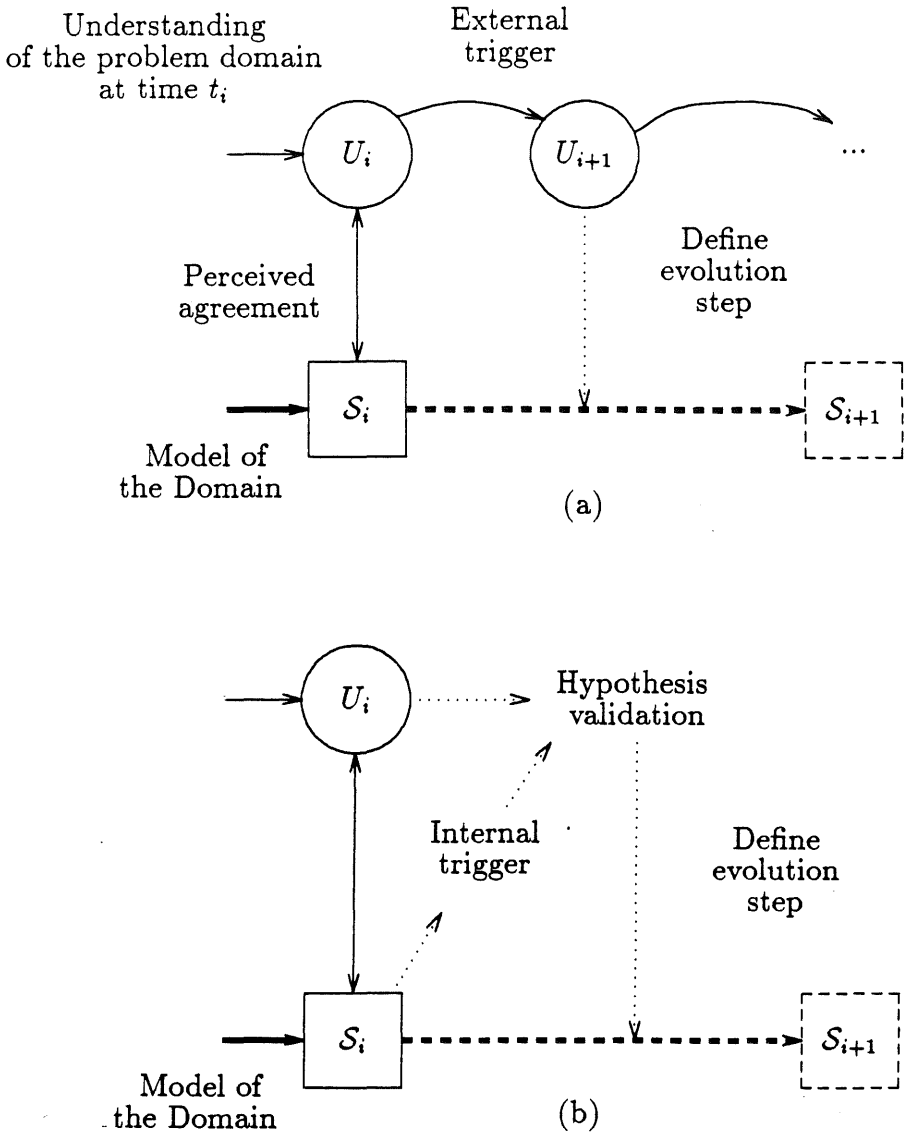


Figure 6.1: External (a) and internal (b) MoD evolution triggers

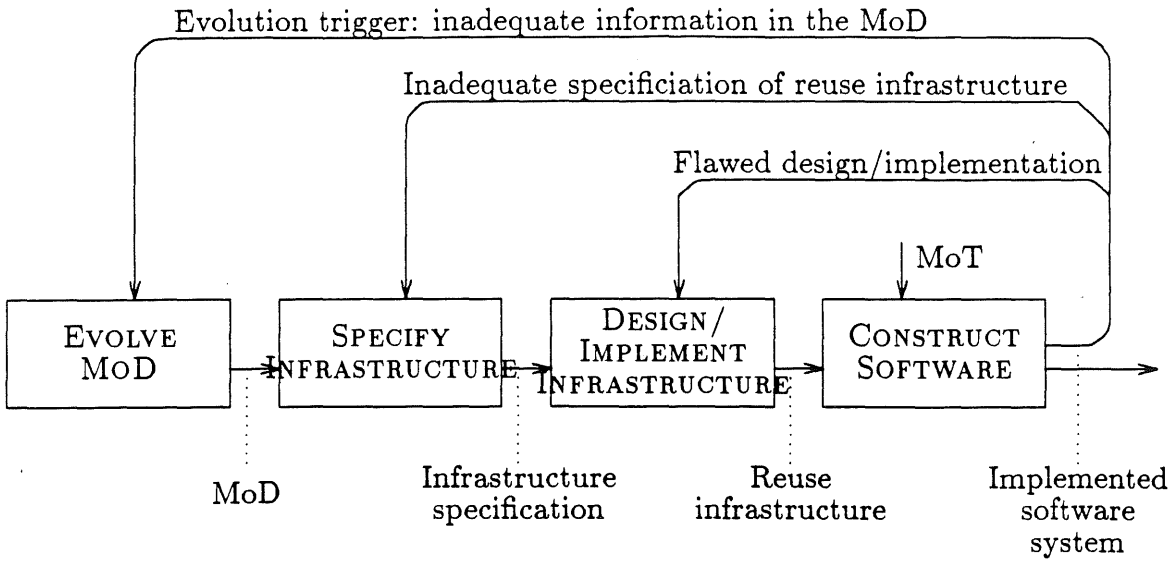


Figure 6.2: Forms of infrastructure evolution

to relate it to previous experiences, when a match is found, the analyist draws upon that previous problem experience to structure the current problem. Both forms are discussed below.

The search for triggers and the generation of hypotheses have been identified as the most effective problem-solving strategies in systems analysis [VD83].

In this Chapter we are concerned only with changes to the state of the MoD and in particular, to those changes that enhance the competence of the target reuser. Some aspects of the evolution of infrastructure specifications are discussed in Chapter 7. The evolution of designs or implementations falls in the category of a software debugging or maintenance situation and it is not discussed here.

6.1.2 Competence enhancements

In Chapter 3 we proposed that practical domain analysis realizes a form of incremental learning. We now focus on the definition of methods for enhancing the competence of the reuser, L_c. A measure of the competence of a reuser is the class of specifications it can recognize and implement. A MoD encodes information to describe an implement some set of specifications S. For the purpose of our analysis, we will ignore the fact that the target reuser operates on a projection of the MoD, the reuse infrastructure. We assume that S defines the domain of competence of the reuser[1]. Thus, changes to the state of the MoD affect the competence of the

[1]In Chapter 7 this assumption will be relaxed, by recognizing that, 1) in practice, the MoD and RI are usually separate, 2) the competence of a target reuser is usually a subset of the class S, and

reuser.

The set S is an extensional definition of a domain-specific language. The object definitions in the MoD capture the semantics of the vocabulary in that language. Individual specifications are instances of known "sentences" in that language (see for example, the definition of CLF-DISK-DRIVER in Figure 5.2.)

A desired level of competence in a reuser is specified as a goal-state $\mathcal{G}$ with a specified value for $S_{\mathcal{G}}$. A *difference in competence* between the current state and the goal-state was defined as a difference in coverage between them (Section 3.2.1):

$$\delta_c(\mathcal{S}, \mathcal{S}_{\mathcal{G}}) \;=\; |C(\mathcal{S}) \ominus C(\mathcal{G})| = S \ominus S_{\mathcal{G}} = \{s_1, s_2, \ldots s_j, s'_1, \ldots s'_k\}$$

The difference in competence is characterized by the set of specifications that are implementable in one and only one of states. Two sets of specifications are distinguished: the Plus set, $\{s_1, \ldots s_j\}$, and the Minus set, $\{s'_1, \ldots s'_k\}$. The Plus set is the set of the specifications that are covered in state $\mathcal{G}$ but not in the current state $\mathcal{S}$. The Minus set is the set of the specifications that are covered in the current state but not in goal-state $\mathcal{G}$.

Approximating a goal-state $\mathcal{G}$ corresponds to reducing the number of exemplars in the Plus and Minus sets:

1. Adding relevant information to cover the Plus set, $\{s_i\}$—monotonic evolution.
2. Removing information needed only to cover members of the Minus set, $\{s'_i\}$—non-monotonic evolution.

An *evolution step* is the process that results in removing one or more members from difference set:

Evolution step: current-state $\mathcal{S} \;\rightarrow$ new-state $\mathcal{S}'$

so that,

$$\delta_c(\mathcal{S}', \mathcal{G}) \;<\; \delta_c(\mathcal{S}, \mathcal{G})$$

6.2 A method for MoD construction

Summary. A method for the construction of a MoD is proposed. The first phase is to outline the problem domain by defining the problem-specific goals that systems in the domain must satisfy, and selecting a representative set of specification/implementations pairs. The second phase consist of the recovery of the design in those implementations to outline a domain network. The third phase is a continuous cycle of incremental evolution. Each evolution step consists of: exemplar acquisition, MoD-exemplar integration and validation.

An incremental method for MoD evolution generates successor states of the current state that reduce the difference in competence between the current state and a goal-state:

3) the reuse infrastructure can be augmented to affect the efficiency of the reuser.

1. Preparatory phase: outline of the problem domain
   (a) Outline of the goals of software developers in the domain (Section 6.2.1)
   (b) Selection of a "representative" set of problems in the domain (Section 6.2.2)
   (c) Outline of a domain network (Section 6.2.3)
2. Evolutionary phase: (Section 6.2.4)
   (a) Exemplar Acquisition (Section 6.3)
   (b) MoD-exemplar Integration (Section 6.4)
   (c) Validation of the information captured in the MoD (Section 6.5)

Figure 6.3: Overview of a method for MoD construction

Given: a MoD state $\mathcal{S}_i$ with a coverage $Coverage(\mathcal{S}_i)$, and a well-defined exemplar s_k.

Find: a new MoD state $\mathcal{S}_{i+1}$ such that $Coverage(\mathcal{S}_i) \cup s_k \subseteq Coverage(\mathcal{S}_{i+1})$.

Exemplars are the units of information acquisition. On the one hand, exemplars provide the opportunity to acquire "knowledge in context", i.e., not only reusable abstractions but also information on how they are used and why. On the Other hand, they provide a focusing mechanism for the acquisition of information. The experience in systems and requirements analysis and in knowledge engineering for expert systems has shown practical advantages in organizing the process around examples or case-studies (e.g., [Fic87b] [FAM+85] [Hof87])

In practice, the evolution step: $\langle \text{exemplar} \rangle \circ \mathcal{S}_i \rightarrow \mathcal{S}_{i+1}$, involves the application of a sequence of *evolution operators* on object definitions, for example: add, merge, generalize, remove, repair, specialize, split.

The evolution steps preserve the integrity of the database. The state of the MoD may not be well-defined at intermediate points in the execution of the step.

Preparatory phase

The construction of the MoD could be conceived as a purely evolutionary process which starts with a "blank MoD", and populates it with relevant information trough incremental evolution steps. A purely incremental models is equivalent to hill-climbing in the space of possible MoD states, using reuser competence as an

evaluation function. The improvements in competence due to internal analysis suffer from the shortcomings of hill-climbing—getting stuck on local optima, ridges or plateau—but they can be compensated by "jumps", i.e., external triggers.

However, for practical reasons it is appropriate to start MoD construction with a preparatory phase (1. in Figure 6.3) in which some basic structures are established. These structures set-up a framework for incremental evolution.

The goals of the preparatory phase are similar to those found in knowledge engineering guidelines, (e.g., knowledge definition phase [FAM+85], initial preparation [Reb81], getting background knowledge [Pre87]): the identification of case studies, familiarizing the domain analyst (or knowledge engineer) with the terminology of the domain, outlining the relevant classes of knowledge to be acquired, and the general structure for organizing that knowledge.

In our case, three are work-products of the preparatory phase:

- an initial outline of problem-specific goals that must be satisfied by systems in the domain,
- a list of exemplars, and
- an initial "road map" of the kinds of software design knowledge required to implement the systems in the domain—an outline of a domain network.

The case studies used in this chapter

We illustrate aspects of the method with examples taken from two real-life domains. One of them, the disk drivers domain, was introduced in Chapter 5. The second one, UnixInit—for Unix Initialization domain [TA88]—deals with the problem of customizing the operating environment of a user of the Unix operating system. Specifications in the UnixInit domain are called `.cshrc` or `.source` files in Unix. A user's `.cshrc` file includes commands that customize some aspects of the user's operating environment each time the file is executed.

UnixInit was selected as a case study because it presents features that are complementary to those of the disk drivers domain. Trivial implementation relations. UnixInit is a simple, directly decomposable problem domain. By decomposable we mean that domain goals can be satisfied by simple aggregation of components (i.e., parametrized Unix shell commands). Large number of data-points—approximately 70 samples of real-life `.cshrc` files—to study the distribution of concept abstractions and instances.

The Disk Drivers domain, on the other hand, is an instance of a restricted but quite complex real-life domain. It is very rich in domain-specific implementation information. Parts in disk drivers are highly interconnected. We were able to analyze samples of disk drivers constructed for (basically) the same operating system over a period of 10 years. One focus of this study was to study the effect of evolution triggered by real-life external causes. It allowed us to study both the evolution

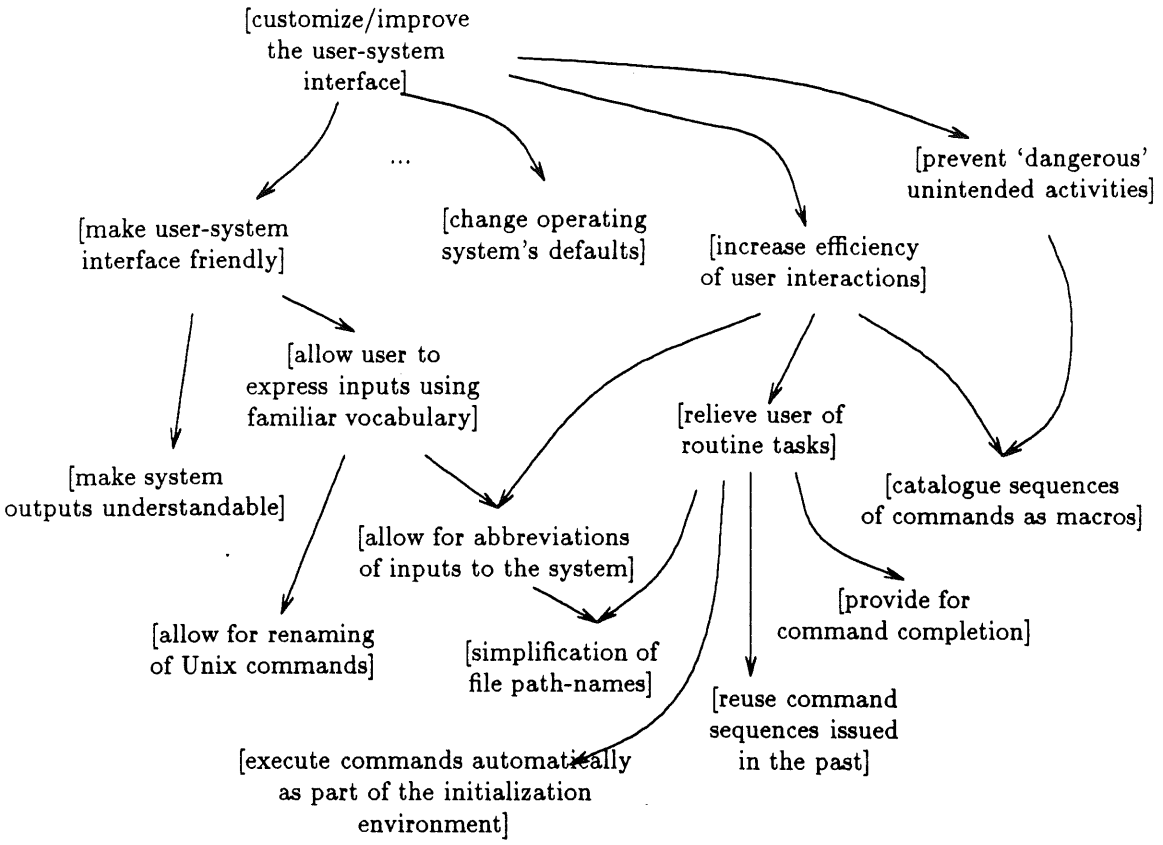


Figure 6.4: Fragment of a goal net for UnixInit

of the driver's specifications and design under changes of device and controller technologies, and to trace the learning on driver design.

6.2.1 Outline a goal-dependency network

An outline of the goals of software developers in the domain can be captured as a goal dependency network (GDN). The GDN GDNs provide an anchor into the reality of the community affected by the problems, and is the basis for creating justifications of why concept definitions are in the MOD. This information is also used to guide the classification and generalization of MoD object descriptions as well. We refer to it as an outline because it is a working document to be refined and evolved.

Goals in the problem domain are desribed by Justification objects in the MoD language. Figure 6.4 illustrates a sample fragment of a GDN for UnixInit. At some point the goals are related to the models or specification objects they justify. For instance, in the case of UnixInit: a goal such as "[allow for renaming of

Unix commands]" could be mapped directly to a parametrized component provided by Unix: "`alias <new-name> <old-name>`"; while "[execute commands automatically]" maps to more complex forms such as "`source <command-file>`" where a command file must be defined. In Section 6.4.3 we present a more detailed example of a justification network in the domain of disk drivers and its use in supporting MoD evolution.

The acquisition of information for the GDN, is a continuous process during domain analysis. GDN information is the result of asking the experts *Why*-questions (e.g., Why is this an interesting topic? Why is this concept needed? Why should we place such constraint on this behavior?)

6.2.2 Problem selection

A problem domain is outlined as a list of exemplars, $\{S, I_S\}$, together with a collection of sources of knowledge. The quality of the sources of knowledge—literature, systems documentation, experts—has a direct bearing on the quality of the results of the domain analysis process. Selecting a collection of representative exemplars for a problem domain is both a technical and a managerial problem [DoD86] [NvD78]. From a technical viewpoint, the exemplars should be selected to improve the quality of the resulting MoD and the efficiency of the domain analysis process. In practice, we encounter a phenomenon that has important consequences in this respect.

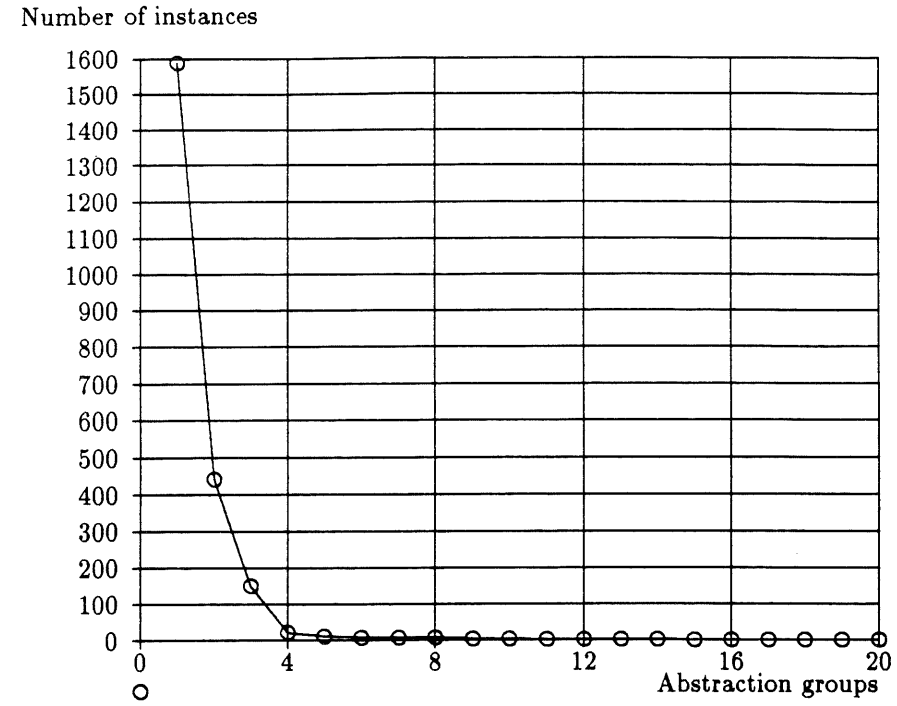
The First-Twenty-Percent phenomenon

Boehm's observation that many software phenomena follow a Pareto distribution: 80 percent of the contribution comes from 20 percent of the contributors [Boe87] seems to apply to the relation between reusable problem abstractions and exemplars. This distribution correlates with the "conceptual cohesiveness" of domain.

Figure 6.5 illustrates this effect with data from the UnixInit study. Initialization commands were grouped into abstraction groups, according to the domain goals they serve, for instance: renaming, saving history of commands, checking on mail, etc.

The graph at the top of Figure 6.5 plots the number of instances of abstractions found in the study of 68 applications against a classification of abstraction groups. The graph at the bottom plots the number of applications that make use of abstractions from a particular group. As the graphs show, most applications use a very small subset of abstractions (1 to 4) many times, while most of the other abstractions are used infrequently (less than 10 instances from abstraction groups 5 to 20 were counted.)

Figure 6.6 shows the growth in the kinds of abstractions identified with respect to the instances of applications analyzed, in chronological order. The progression shows that the great majority of the applications (range 12 to 62) are built using the same abstractions, and that 10 percent of the applications studied (range 62-68) contribute instances of all the abstractions seen. (This progression was the result of



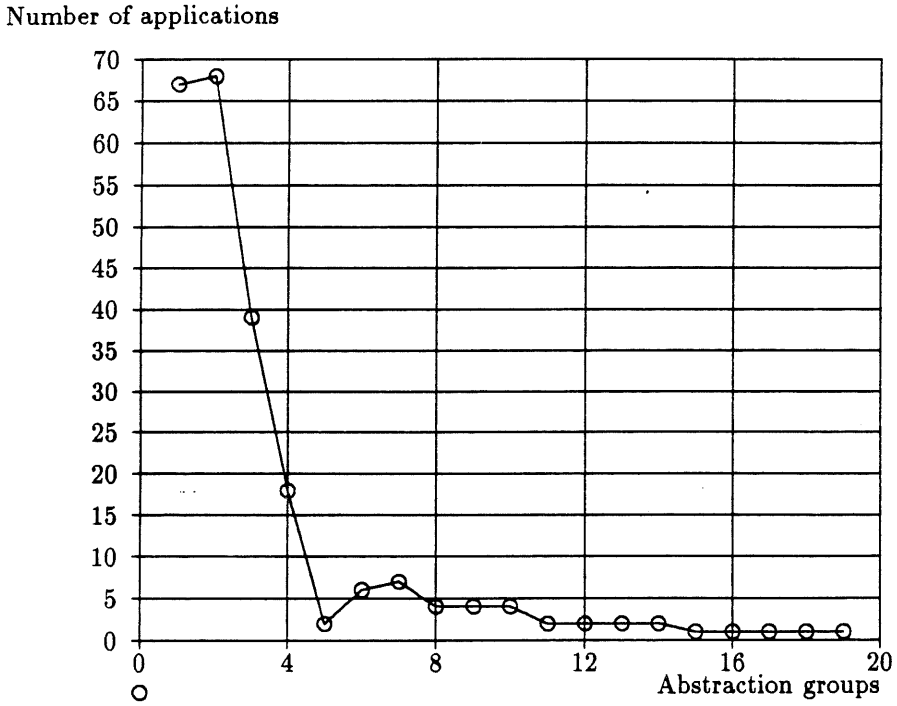


Figure 6.5: Distribution of abstractions in UnixInit domain.

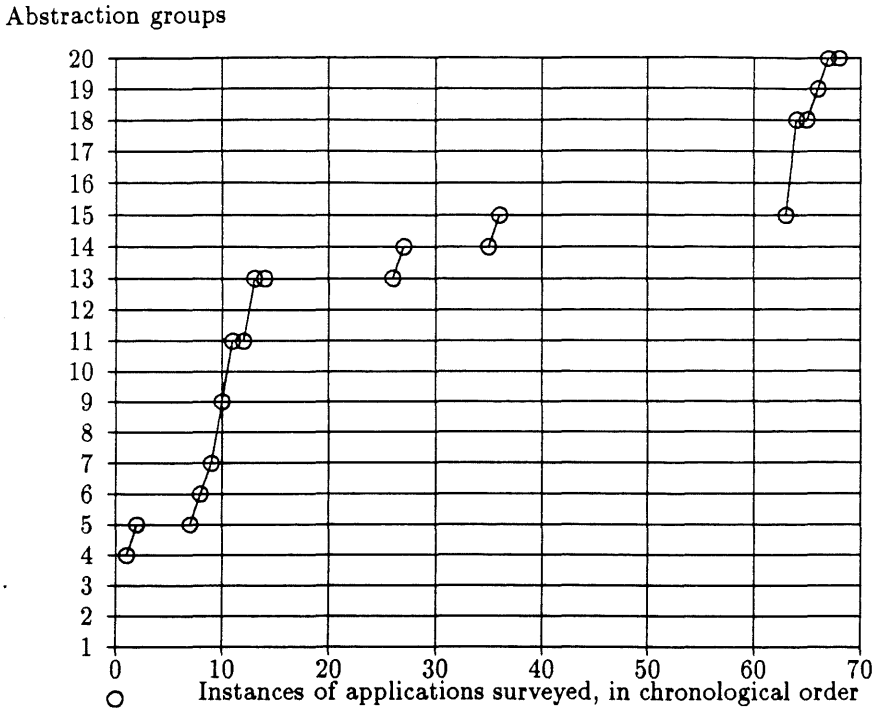


Figure 6.6: Monotonic growth in the population of reusable abstractions

a conscious decision to investigate first the smaller and apparently simpler `.cshrc` files.)

These distributions have practical consequences for the organization of the process of information acquisition and the selection of exemplars for analysis. At the begining of a domain analysis project, it is clearly advantageous to select as exemplars as many members of that *twenty percent* that can contribute instances of most of the interesting specification and implementation abstractions. This effect has also been identified in the acquisition of knowledge for expert systems [LPS86].

The properties that characterize a First-Twenty-Percent family of exemplars appears to be domain-dependent. In the UnixInit domain, there is a direct correlation between the sophistication of the user and the number ofabstractions employed. In that case study, the First-Twenty-Percent exemplars were provided by users that could be labelled either as "hackers", or as "computing support personnel". In the domain of disk drivers, most systems require a basic set of structures (e.g.,Disk Control Blocks) and processes (Read, Write, Map logical sector number to physical sector number, etc.) Variations relate to key aspects of the technology of the physical devices: "intelligence" of the controllers, availability of Direct Memory Access channels, reliability of the media, etc.. The First-Twenty-Percent must include exemplars of disk drivers from a set of representative technologies. In the domain of spreadsheets for financial accounting, the financial and accounting models are

fairly standard across companies. In this case the source of variability is the type of company. Different types of companies have different "charts" and "feed" different data to the models.

> **Heuristic.** A study on the sources of *variability* in the relation between typical specifications and implementations must be conducted in preparation for selecting representative exemplars for an initial outline of a domain. Exemplars should be chosen to cover as many different variants as it is cost-effective.

Coverage and width

The term "width" has been sometimes used to indicate "how big is a domain" [Nei88b]. The decision of which abstractions to include and which to exclude is crucial and will limit the range of systems which can be built from the model later [Nei81, p. 16].

If width is a trade-off between generality and specialization of the model of the domain, then such decision must be based on objective cost criteria. For instance, How often do we have to implement variants of a particular application, What would be the opportunity cost of not having appropriate components to build some kind of systems.

Another interpretation of width might be called "definitional width". Once a concept description has been declared to be *in* the domain, we must specify the semantics of the concept as completely and as precisely as possible. The view we have adopted defines concepts in terms of other concepts in the domain. For example, in the domain of disk drivers, the definition of a READ-SECTOR operation, requires a statement in the MoD Representation Language which includes other concepts such as, SECTOR, CYLINDER or SYSCALL. Intuitively the concept of an operating system SYSCALL is foreign to the disk drivers domain, however, being the trigger for a READ-SECTOR it must be included in the definition. Where do we draw the boundaries of the universe of discourse? Rather than a question between generality vs. specificity of the domain, it is a question of avoiding an infinite regression in a definition process. We adopt a pragmatic answer

> **Heuristic.** Concepts in the model of the domain are implementable or definitional. The representation of definitional concepts consists of references to agreed upon sources of meaning—manuals, textbooks, systems documentation.

The situation is not unlike normal conversation, where so many things are left unsaid, and meaning is still conveyed because of implicit shared knowledge.

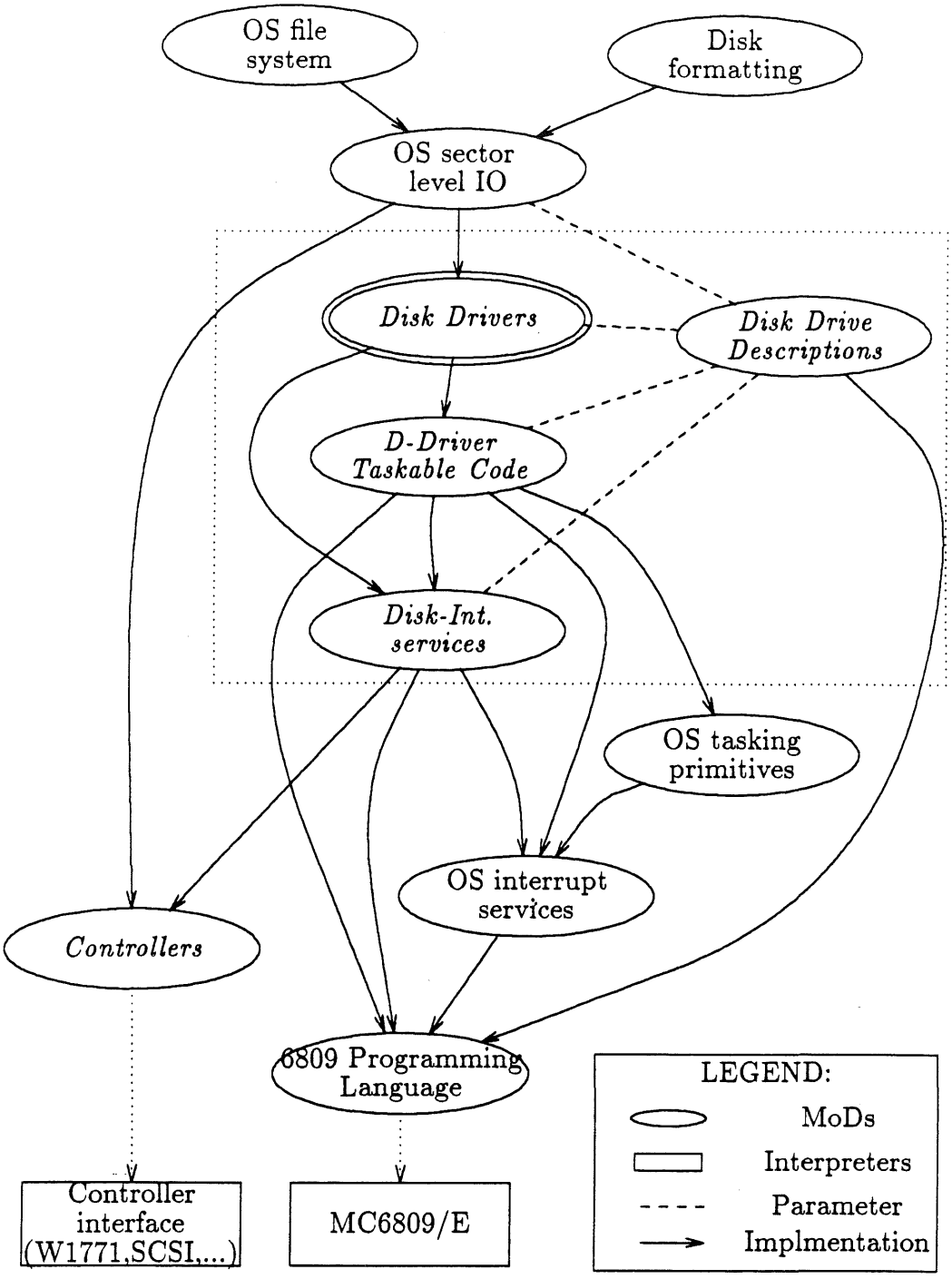


Figure 6.7: A domain network in the domain of disk drivers

6.2.3 Outline of a domain network

The notion of domain language network was discussed in Section 2.2.3 as a mechanism for defining domain-specific languages. A similar notion is implicit in the horizontal and vertical compositions in Goguen's Library Interconnection Language (LIL) [Gog86] to support first-order reuse. Neighbors [Nei81] used them to modularize the transformation bases used by the Draco system.

From the point of view of constructing MoDs, domain networks provide an opportunity for reuse *within* the process of domain analysis. Pre-existing sub-networks of MoDs offer to the domain analyst domain-specific languages in which to encode the design plans for new domain languages. The principles involved in the design of reusable networks are not well understood. We use the expression *domain engineering-in-the-large* to refer to the activity of designing new networks out of reusable MoDs, in direct analogy with *programming-in-the-large* [DK76]. It appears that many of the features that enhance the reusability of conventional software components—e.g. low inter-component coupling, high-cohesion, implementation information hidding—can be extended directly to MoD structures. We do not elaborate on this topic here. [Ara88b] presents instances of component couplings via subsidiary domains that limit the possibility of sub-net reuse.

Outlining a network

The task of outlining a network parallels that of developing a "skeletal structure" for the model of a problem-domain [Reb81] or, a "first-pass" database [Hof87] in the development of expert systems.

The inputs include: 1) a definition of the goal of software developers in the domain under (the "root domain"), 2) a collection of implemented systems in the domain, and 3) (possibly) pre-existing MoD that may become associated to subsidiary positions in the network. Those domains usually characterize structures and algorithms that are used by large numbers of software implementations, they have been called: implementation domains, computer science domains or technology domains.

The fourth, key input is the expertise of a domain expert. Based on the existing implementations, and on the software design experience of the expert, the network of justifications is expanded with information on (broad) design approaches. The enriched justification network provides a framework for clustering related abstractions into candidate domains. Biases introduced at this stage color the subsequent acquisition steps.

The process of outlining the domain network is guided by the informal application of a design recovery process along the lines described in Section 6.3.1, follow by a justified, or goal-directed clustering process.

Figure 6.7 represents a domain network in the domain of disk drivers. The examples of object definitions discussed in this document are confined to the domains

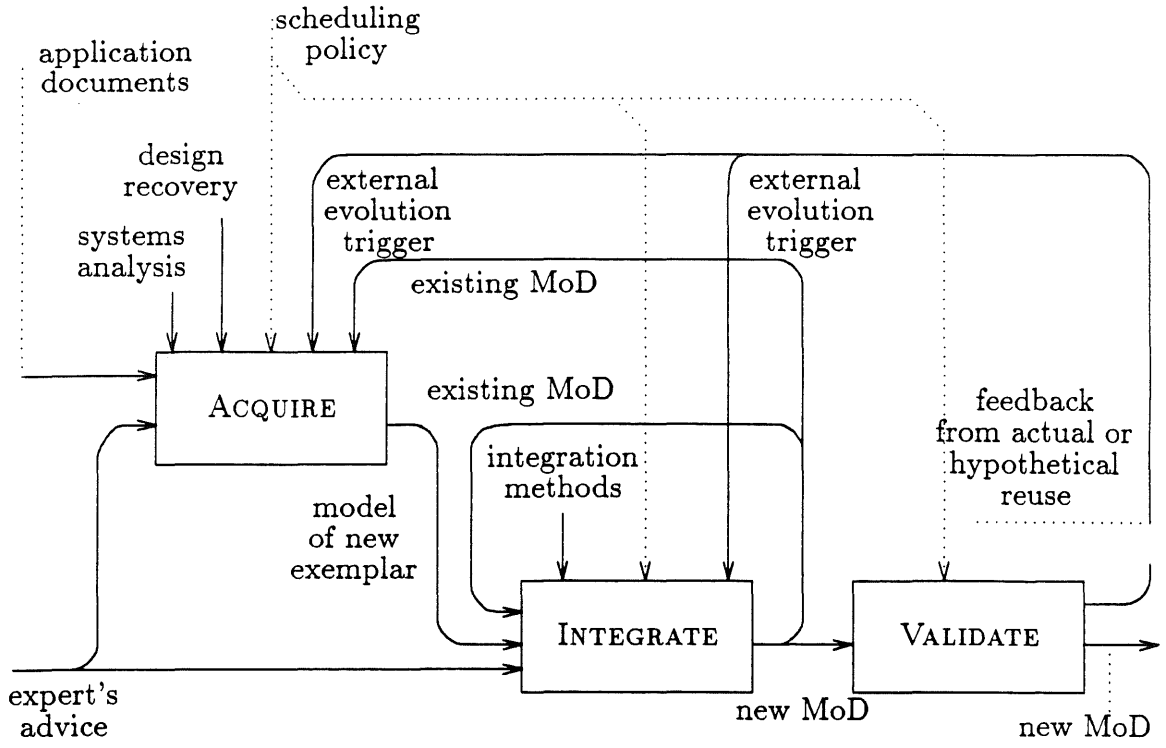


Figure 6.8: MoD evolution: The incremental step

enclosed by the dotted rectangle in Figure 6.7. Each node in the network represents a MoD for a particular problem domain. Specification objects in the domain of Disk Drivers are implemented in terms of objects in the domain of Disk Driver Taskable Code. Objects in that domain are implemented using objects in the domains of Disk Interrupt Series, OS Tasking Primitives and in the 6809 Programming Language. The objets defined in the domain of Disk Drives Descriptions (e.g., Disk Control Blocks, Resident Sector Information Block, Disk Information, etc.) parametrize several domain (dashed lines in the Figure.) [Ara88b] offers a detailed description of objects in these domains.

6.2.4 The incremental evolution step

The problem of MoD construction was defined as one of incremental evolution of a MoD, based on the integration of new information with an existing MoD (Section 6.2.) The unit of acquisition for new information is the definition (expressed in the MoD representation language) of an exemplar application in the problem domain.

Figure 6.8 summarize the three major activities in the process of defining, applying and validating an incremental evolution step. The details of each one of these

activities is discussed in the remaining Sections in this Chapter.

6.3 Acquisition of exemplars

Summary. The acquisition step takes as inputs an implementation of system, and the advice of an expert in the domain. The results of the acquisition step are: a specification for the system in the MoD language and a "recovered" implementation represented using implementation and justifications objects. A model-driven approach is proposed to guide the acquisition of a specification. Design Recovery is proposed as a method to acquire design information.

The acquisition of domain-specific information by using exemplars corresponds to the "analysis of familiar tasks" strategy in knowledge engineering [Hof87], in which a domain expert is requested to specify and recover the design of an existing system.

The formal structure in the MoD language (Appendix B) together with the integrity constraints defined in Section 5.7 provide a syntactic model to identify relevant information for the description of an exemplar. The structure of the MoD representation language allows for a direct mapping from other languages that the expert might be more familiar with such data-flow diagrams, entity-relationship diagram, SADT [MM88], JSD [Jac83], Forest [PFAB86], etc. The information already encoded in the MoD provides additional leverage for acquiring and verifying information on new systems by analogy with previously analyzed systems.

6.3.1 Design recovery

Successful implementations are valid instances of models in the domain. Implementations are the tangible result of a design process that the analyst needs to recover.

Design recovery is a reverse-engineering technique whose aim is to recover from an implementation some of the software design information employed in the original construction. We view design recovery as a process of generating explanatory hypotheses from software implementations using design- and justification-object definitions. The technique has applied successfully in practice [ABFP85] [ABFP86] and [Ara88b]. Starting with the reuse equation,

$$MoT(S, RI) = I_S$$

design recovery can be summarized as

$$DR : MoT, I_S \rightarrow S, RI$$

In other words, the result of a design recovery on an implementation I_s is both a plausible specification s together with a set of components, RI, that could have been used to carry s into an implementation I_s. The components can be thought of as parts of an *explanation of the implementation under a particular MoT.*

The components in RI are an essential part of the explanation. When the design recovery is performed by an expert, the expert defines the design objects. As the analysis of a domain progresses and the library of objects definitions grows, the expert is called to help when the available objects fail to provide a complete explanation. In such cases, we have a focused knowledge elicitation situation, driven by the design recovery process:

$$DR : MoT, I_S, RI_{known} \rightarrow S, RI_{acquired}$$

This is precisely the case we are interested in. The extreme case in which *all RI* is presumed to be known, is the object of study of explanation-based approaches to machine learning [DM86] [MKKC86]. The PROUST system [AS85] [WLJ85] uses a similar approach to understanding and debug programs coded in Pascal.

An example

We introduce the process of design recovery with an example. Figure 6.9 shows a real implementation of a WRITE-SECTOR entry point in a floppy-disk driver. The fragment of a listing is an implementation in the "6809-Programming-Language" domain (Figure 6.7). The handwritten notes on the listing outline abstractions in the "D-Driver Taskable Code" domain. They correspond to the plan for CLF-WRITE (Figure5.5).

The first time a design recovery is performed on a WRITE implementation, the explanation information must be acquired from an expert or some other source (Figure 6.9). Once the information has been captured as definitions of PLAN, DESIGN and specification objects supported by a scaffolding of justification objects at different levels in the domain hierarchy—the definitions aid in the recovery of design information in other instances of implementations of WRITE. Figure 6.10 from [Ara88b] illustrates an actual sample of the process. The bracketed parts mark the differences between the old object definition and the new definition. The notation indicates: "+" attribute-value pairs to be added, "-" attribute-value pairs to be removed, "i" objects which do not change at the specification level but whose implementation change, "d" attributes whose value is an objects whose definitional neighborhood changes. The lower half of the figure summarizes the new plan for the Winchester disk WRITE (IMI), recovered from a Conrac floppy driver WRITE (CLF). The lower half of Figure 6.11 compares the two plans. From the comparison of only the plans it is apparent that some parts do not match ("...?..." in the Figure). Every mismatch between an implementation under study and the existing plans uncovers an opportunity for learning something new. Design recovery becomes a focusing mechanism for knowledge elicitation (Section 6.3.3) [Ara88b].

Design recovery is an abductive process

The design recovery process is based on two assumptions. First, the implementation under analysis was in fact "designed", i.e., there exists a design plan and justifi-

```
...
82EE 8602    185: CLFWRITE  LDAA  #CONRACWRITE   set up command
82F0 2002    186:           BRA   CLFOPSET
82F2 8602    187: CLFREAD   LDAA  #CONRACREAD
82F4 36      188: CLOPSET   PSHA                ; SAVE COMMAND
82F5 DE06    189:           LDX   DCBPOINTER
82F7 EE2B    190:           LDX   DISKINFO:SECTORDB,X
82F9 A603    191:           LDAA  RDSI:LSN+1,X      check
82FB E604    192:           LDAB  RDSI:LSN+2,X      legal
82FD SW06    193:           LDX   DCBPOINTER        LSN
82FF E01D    194:           SUBB  DISKINFO:NLSN+2,X
8301 A21C    195:           SBCA  DISKINFO:NLSN+1,X
8303 2506    196:           BCS   CLFSETUP1
8305 31      197:           INS        ; THROW AWAY THE COMMAND
8306 BD94AD  198:           JSR   SDOS+SDOS:ERROR   signal Bad LSN
8309 040E    199:           FDB   ERR:ILLLSN
830B CE8196  200: CLFSETUP1 LDX   #CLFINTERFACE ; WAIT FOR INTERFACE FREE
830E BD8508  201:           JSR   ALLOCATERESOURCE  lock interface
8311 DE06    202:           LDX   DCBPOINTER
8313 32      203:           PULA                ; GET THE COMMAND TYPE BACK
8314 B7819B  204:           STAA  CLFREADWRITE ; SET THE OPERATION
8317 A616    205:  check    LDAA  DISKINFO:MAPALGORITHM,X
8319 E617    206:  if MAP   LDAB  DISKINFO:MAPALGORITHM+1,X
831B E131    207:  changed  CMPB  CLFMAPALGORITHM+1,X ; SAME AS OLD?
831D 2604    208:           BNE   CLFSETUP2     ; NO, SET THE MAPALGORITHM
831F A130    209:           CMPA  CLFMAPALGORITHM,X
8321 270C    210:           BEQ   CLFSETUP3     ; YEP
8323 CE8454  211: CLFSETUP2 LDX   #CLFISETMAP
8326 BD825F  212:           JSR   CLFDOIO       ; CAUSE THE MAP TO GET SET
8329 CE81    213:           LDX   CLFINTERFACE ; ALLOCATE INTERFACE AGAIN
832C BD8508  214:           JSR   ALLOCATERESOURCE  select new MAP
  832F       215: CLFSETUP3 EQU   *
832F CE8345  216:           LDX   #CLFCMDFEED
8332 BD8270  217:           JSR   CLFSTARTIO ; ENTER INTERRUPT SERVICE CODE
...
```

Figure 6.9: Design recovery step—the CLF-WRITE-PLAN

```
{d IMI-WRITE-PLAN d} in {+ IMI-PLAN +}  in DD*-PLAN  with
id
--- design: {d IMI-WRITE-bDESIGN d}
technology
---  implementation-domains: DDT*
precond
--- type-of-controller:  Device-controller = 'IMI 7710 Winchester'
obligations
--- must-finish-transfer: [Separate-Initiation-Completion]
                          {i TRIGGER-READ/WRITE i}
parts
[Split-Responsibility-1]
--- step-0: SET-CMD-TYPE {d (WDCWRITECMD) d}
{+  step-wd-1: SET-RETRY-COUNT(5)
--- step-wd-2: DCB.lasterror <- ERR:NONE   +}
{d  step-3: LOCK-INTERFACE(IMIINTERFACE)      d}
--- step-1: CHECK-LEGAL-LSN(RDSI)
--- step-2: if 'bad' then SIGNAL('badlsn')  and ABORT
{- [Winchester-Controller-Handles-Map]
    step-4: CHECK-MAP-CHANGE
--- step-5: if 'changed' then
                SET-MAP-ALGORITHM(IMI-INFO.clfmapalgorithm)
--- step-6: [INT*Interrupt-Resource-Deallocation and design.Separation]
             REALLOCATE-INTERFACE                -}
[Split-Responsibility-2]
--- step-7: TRIGGER-READ/WRITE(imi-write-interrupt-code)
```

Summary:

SET-CMD-TO-WRITE(WDCWRITECMD)
SET-RETRY-COUNT(5)
RESET-ERROR-STATISTIC
ALLOCATE-RESOURCE(IMIINTERFACE)
CHECKLEGAL-LSN(RDSI)
if 'bad' SIGNAL(BADLSN) and EXIT
TRIGGER-READ/WRITE(imi-interrupt-code)

Figure 6.10: Design recovery step—the IMI-WRITE-PLAN

GENERIC-WRITE-PLAN('write-command-parameter', 'interface-type')
SET-CMD-TYPE(cmd-for-controller(controller))
SET–RETRY-COUNT(reliability-level)
VALIDATE-LSN and if bad(RDSI) then ABORT
ALLOCATE-RESOURCE(resources-for-controller(controller))
DCB.lasterr ← ERR:NONE
HANDLE-MAP-CHANGE(mapcode(DCB))
DCB.lasterr ← ERR:NONE
TRIGGER-READ/WRITE(generic-write-interrupt-code(controller))

CLF-WRITE-PLAN		**IMI-WRITE-PLAN**
SET-CMD-TO-WRITE(CONRACWRITE)	⟷	SET-CMD-TO-WRITE(WDCWRITECMD)
...?...		SET-RETRY-COUNT(5)
...?...		RESET-ERROR-STATISTIC
...?...		ALLOCATE-RESOURCE(IMIINTERFACE)
CHECKLEGAL-LSN(RDSI)	⟷	CHECKLEGAL-LSN(RDSI)
if 'bad' SIGNAL(BADLSN) and EXIT	⟷	if 'bad' SIGNAL(BADLSN) and EXIT
ALLOCATE-RESOURCE(CLF)		...?...
CHECK-MAP-CHANGE		...?...
if changed(MAP-CODE) SET-NEW-MAP		...?...
TRIGGER-READ/WRITE(...)	⟷	TRIGGER-READ/WRITE(...)

Figure 6.11: WRITE.**plan** generalization

cations to be recovered. Second, the implementation was produced using a model of software construction process compatible with that of the target reuser, i.e., the design recovery method proposed here cannot produce a first-order infrastructure $RI^{(1)}$, from the analysis of a transformational implementation.

The process of recovering the design from implementation results in a plausible specification and an explanation relating specification and implementation which captures justified design information. According to the domain network view of software construction, explanations are multi-steps structures, linking implementation patterns in a subsidiary domain with objects in a higher-level domain. For the purpose of the presentation of the method we will concentrate on a canonical recovery step between any two domains in the network (Figure 6.12).

The recovery of designs proceeds abductively. Abductive inferences can be summarized as:

Given a set of facts Y and rule $X \rightarrow Y$, infer "plausible X"

or, in our case

Given: I_s and $\langle s \rangle$**implementor.plan**:I_s.
Propose: s as a plausible specification for I_s,
and the value of $\langle s \rangle$**implementor** as the justification for I_s.

We assume that there exists a model of the domain that provides: 1) a set of specifications $\mathcal{S}$ (represented as objects of type S- in the MoD); 2) a set of implementation plans $\mathcal{I}$ (represented as objects of type PLAN in the MoD); 3) a set of design descriptions $\mathcal{D}$ (represented as objects of type DESIGN in the MoD); and 4) a set of model definitions $\mathcal{M}$. Figure 6.13 outlines a method for performing a *canonical design recovery step*. Design recovery involves:

1. *Identification of implementation components.*

2. *Hypotheses generation.* The purpose of the generation sub-step is to identify patters of the implementation that relate to known[2], specification-level abstractions.

3. *Hypotheses selection.* The purpose of the filtering step is to preserve only those hypothesis components that can be articulated into known specification models or parts of models.

4. *Hypotheses validation.*

[2]In this context "known" is equivalent to "explicitly defined in the MoD".

Given:

- A MoD including:
  - a set $\mathcal{S}$ of specification objects in a domain D,
  - a set $\mathcal{I}$ implementation objects (PLAN objects in the MoD language, composed of actions in a set of subsidiary domains D_i)
  - set $\mathcal{D}$ of design descriptions (DESIGN objects in the MoD language), and
  - a set $\mathcal{M}$ of model definitions.
- An implementation I_s represented in terms of concepts in the subsidiary domains $\{D_i\}$

Find: a MoD-based explanation for I_s consisting of:

- one or more models (or parts of models) $m \in \mathcal{M}$,
- for each m, a (possibly recursive) MoT-specific derivation of I_m represented in terms of decomposition (**parts**) and implementation (**implementor.plan**) relations taken from $\mathcal{D}$ and $\mathcal{I}$.

Figure 6.12: The design recovery problem

1. Given I_s identify constructs in the subsidiary domains D_i.
2. Hypothesis generation:
   (a) Identify instances of PLAN.**part** $\in \mathcal{I}$ in I_s. Define a set $\mathcal{I}^+$ of "known" implementation objects found in I_s.
   (b) For each object $i \in \mathcal{I}^+$ propose as hypotheses components, $\mathcal{S}^+$, the set S-objects $h \in \mathcal{S}$ in D, such that exists a relation $[\langle h \rangle$.**implementor.plan**: $c] \in \mathcal{D}$
3. Hypothesis selection. While there are elements in $\mathcal{S}^+$.
   (a) For each hypothesis component $h \in \mathcal{S}^+$, find the set $\mathcal{M}$ of model definitions m that include h.
   (b) Recursively:
      i. For each m find the types of specifications in the definition neighborhood of h.
      ii. Find those hypothesis components in $h' \in \mathcal{S}^+$ that match those types.
      iii. Bind the types in the neighborhood to the candidate components h'.
   (c) If the S-relations defined for h are satisfied for at least one h', and the assumptions of the corresponding PLANS are consistent, and the justifications of the corresponding DESIGNS are consistent, then, propose m as a hypothesis.
   (d) Remove h from $\mathcal{I}^+$.
4. Submit hypotheses for validation to a domain expert.

Figure 6.13: The canonical design recovery step

6.3.2 A covering set model

The model presented here is an adaptation to design recovery of the Generalized Covering Set model[3] developed by Reggia and others [RN84] [RPNP85]. The GSC model was produced to formalize abductive hypotheses generation in diagnosis [RNY83]. The key relation between both situations is a correspondence between the concepts *causation* and *implementation.*

A design recovery problem, DR, is defined as a 4-tuple:

$$DR = (\mathcal{M}, \mathcal{I}, \mathcal{D}, \mathcal{I}^+)$$

where, $\mathcal{M}$ is the set of known models and parts of models in the domain, $\mathcal{I}$ is the set of known implementation objects (PLAN), $\mathcal{I}^+$ is the set of instances of implementation objects identified in the implementation under analysis. $\mathcal{D}$ represent design descriptions $\mathcal{D} \subseteq \mathcal{S} \times \mathcal{I}$. The non-empty set of all implementations of an S-object s_i is denoted:

$$implementations(s_i) = \{i \in \mathcal{I}, \text{ s.t. } (\langle s_i \rangle.\textbf{implementor.plan}{:}i) \in \mathcal{D}\}$$

The non-empty set of models and parts of models are implemented by an implementation object i is denoted by,

$$specifications(i) = \{s \in \mathcal{S}, \text{ s.t. } (\langle s_i \rangle.\textbf{implementor.plan}{:}i) \in \mathcal{D}\}$$

These definitions are extended to sets of specifications and implementations using set union:

$$implementations(\mathcal{S}_I) = \bigcup_{s_i \in \mathcal{S}_I} implementations(s_i)$$

and,

$$specifications(\mathcal{I}_J) = \bigcup_{c_j \in \mathcal{I}_J} specifications(c_j)$$

Given $DR = (\mathcal{M}, \mathcal{I}, \mathcal{A}, \mathcal{I}^+)$, a subset $\mathcal{E} \subseteq$ parts($\mathcal{M}$) is defined to be an *explanation* for $\mathcal{I}^+$ if, $\mathcal{I}^+ \subseteq implementations(\mathcal{E})$, i.e., if $\mathcal{E}$ covers $\mathcal{I}^+$, and $\mathcal{E}$ is parsimonious.

Using the Covering Model we define the solution to a design recovery problem as the set of all explanations for the set $\mathcal{I}^+$.

6.3.3 Information acquisition driven by design recovery

Figure 6.14 summarizes how design recovery is used to identify information missing from the MoD and to provide guidance in asking questions to the domain expert. Given an implementation I_s, the analyst attempts to recover the design. If a complete, satisfactory explanation can be build using MoD object definitions it means that the current state of the MoD has all the information required for the specification and implementation of the application, i.e., there is nothing to be learned. From the point of view of MoD construction the more interesting cases are those where the explanation cannot be completed. The point where an explanation breaks, defines a focus for information acquisition consisting of: an implementation I_s, a partially

[3] A covering problem can be stated as: for a finite set $\mathcal{I}$ of elements and a family $\mathcal{S}$ of subsets of $\mathcal{I}$, a cover $\mathcal{E}$ of $\mathcal{I}$ from $\mathcal{S}$ is a subfamily $\mathcal{E} \subseteq \mathcal{S}$ such that $\bigcup(\mathcal{E}) = \mathcal{I}$. $\mathcal{E}$ is called minimum if its cardinality is as small as possible.

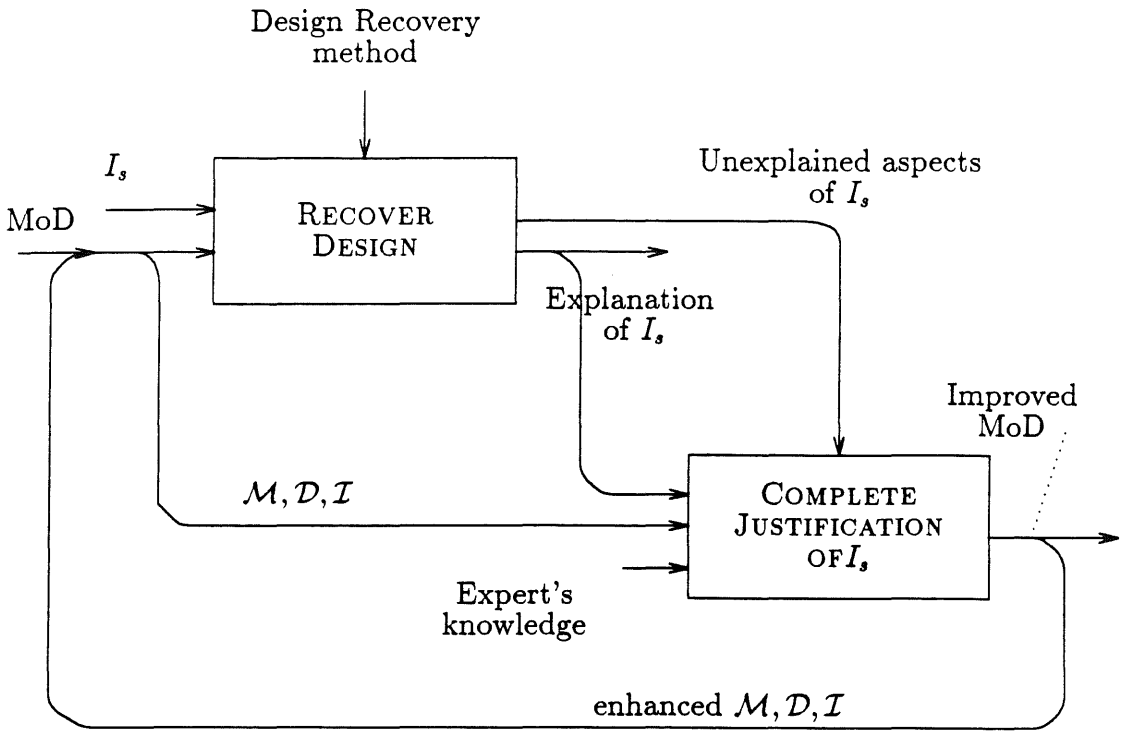


Figure 6.14: DR-driven information acquisition

constructed explanation $\mathcal{E}_{incomplete}$, and a collection of known definitions $\mathcal{M}, \mathcal{D}, \mathcal{I}$ that for some reason cannot be applied to the current situation.

At each breaking point the expert is requested to add new definitions to $\mathcal{M}, \mathcal{D}, \mathcal{I}$ or to generalize or adapt existing definitions. The example shown in Figure 6.11 illustrates the results of such process. An existing plan for describing the implementation of a WRITE-SECTOR entry point in a floppy disk-driver was used to explain the implementation of a similar entry point in a Winchester disk driver. The explanation breaks at several levels. The figure shows discrepancies at the level of the Disk Drivers Taskable code domain. In these cases, the expert is called-in to "repair" the explanation.

6.3.4 Related work in model-driven knowledge acquisition

The methods for information acquisition just described are driven by a model of representation and problem-solving. The assumptions built into model-driven approaches restrict their scope of applicability, but provide guidance for the identification and verification of relevant information.

Instances of model-driven approaches for requirements elicitation have been proposed by Fickas [Fic87a] and Partsch [Par83]. Fickas uses a pre-encoded domain theory to provide guidance (see Section 2.3.5). Partsch uses heuristics based on the structure of algebraic data types, for example: "First specify the constructor operations, then all remaining operations that have an effect on the constructed objects". Most practical requirements analysis methodologies use some abstract model of how to represent requirements to drive the process of acquiring them [Lei87].

Model-driven approaches are also used in the design of "knowledge acquisition assistants" for expert systems. Conceptually, they are similar to the approach proposed here. For instance, MORE [GKM85] and MOLE [EM86] were designed to aid in the construction of knowledge bases for expert systems using an evidential approach to diagnosis; SALT [MMW85] [MM86] for a "propose-and-revise" expert system. The Roget system [Ben84] has been designed to be more generic, and to provide assistance in knowledge acquisition for a number of diagnostic problem-solving tasks. Bennet's "conceptual structures" in diagnostic problem-solving have a role equivalent to our MoTs. The Roget system must first acquire information to identify the conceptual structure of the target expert system, then it selects appropriate acquisition strategies.

Each one of those systems defines knowledge representation structures analogous to our MoD, but information is typed differently depending on the purpose of the expert. For example, MORE collects hypotheses, symptoms, tests, or conditions, linked by "paths" representing causal relations. Domain-specific knowledge in SALT is organized as a dependency network with design constraints as nodes and relations such as "contributes-to", "constraints", "suggests-revision-of".

6.4 MoD-exemplar integration

Summary. A MoD state evolves by integrating (the object definitions in) the representation of an exemplar with (the object definitions) in the current state. The new state must satisfy the well-formedness and integrity constraint defined for MoDs. The execution of an evolution step may uncover new (internal) evolution triggers. To capture the many evolution situations that may arise we describe MoD evolution in terms of three types of abstract relations between evolution triggers, evolution operators and states in the MoD—definition, restriction and effect. Evolving a MoD corresponds to managing these relations so that the predicate "well-defined(new-MoD-state)" holds at the end of each evolution step.

The incremental evolution of a MoD state as a sequence of integration steps between a model of an application in the domain and the current state was defined in Section 6.2. MoD evolution is constrained by the properties required of

the resulting state (as defined in Section 5.7.) Evolving a MoD state while satisficing those constraints is a complicated process. It often requires the acquisition of additional information from experts, cost-benefits analyses, negotiation, or the revision of existing object definitions. This makes it difficult to synthesize procedural descriptions of MoD evolution.

6.4.1 Evolution relations

To capture the essence of the evolution process, we propose a description based on three relations: *definition*, *effect* and *restriction*. The relations are defined over the following sets:

- $\langle state \rangle$, the set of MoD states.
- $\langle operations \rangle$, the set of tuples of evolution operators bound to particular concept definitions.
- $\langle triggers \rangle$, the set of evolution triggers.

MoD states are represented using the MoD representation language. The evolution operators (defined below) represent elementary manipulations of concept descriptions in the MoD state such as: *add*, *remove*, *split*, or *generalize*. An evolution operator consists of a *name*, (e.g., *generalize*) and a set of *operands* representing object definitions or sets of object definitions. An example of an operator with its operands bound is:

generalize (CLF-WRITE-PLAN, IMI-WRITE-PLAN, GENERIC-WRITE-PLAN)

Evolution triggers (introduced in Section 6.1.1) can be characterized as a partially defined evolution operators, with either the name or some of the places left un-defined, for example:

generalize (CLF-WRITE-PLAN, IMI-WRITE-PLAN, ?)

A tuple of evolution operators $\langle e_1, e_2, \ldots e_k \rangle$ is called an *evolution step*. An evolution step is executed when the operators in the tuple are bound to concrete object definitions and applied to a state of the MoD.

Definition establishes a correspondence between MoD states and evolution triggers, and evolution operators:

Definition: $\langle states \rangle \times \langle triggers \rangle \times \langle operators \rangle$

The relation puts a MoD state and a set of evolution triggers in correspondence with a tuple of evolution operators that may realize the evolution step suggested by the triggers.

Effect establishes a correspondence between MoD states, evolution operators, and MoD states and triggers:

Effect: $\langle states \rangle \times \langle operators \rangle \times \langle states \rangle \times \langle triggers \rangle$

The effect relation puts a MoD state and a tuple of evolution operators in correspondence with a new MoD state (resulting of applying the operators to the previous MoD state) and a set of new (internal) evolution triggers identified as a result of the evolution step.

Restriction establishes a correspondence between MoD states, and evolution triggers:

Restriction: $\langle states \rangle \times \langle triggers \rangle \times \langle triggers \rangle$

The relation puts a MoD state and a set of evolution triggers in correspondence with a subset of triggers.

We illustrate these definitions with the approach to MoD construction outlined in Section 6.2. We assume a MoD state $\mathcal{S}_i$, and a new exemplar s_k. In practice, the submission of a $s_k = \{o_1, o_2 \ldots o_n\}$ defined as a set of n objects, defines a set of *external* evolution triggers, where object definitions o_i. are known, but the names of the evolution operators are not.

An analysis of the state of the MoD may reveal that some of the object definitions o_i are already part of the state. In those cases, the corresponding triggers t_i can be dropped—a restriction relation is defined:

$$(\mathcal{S}_i, \{t_1, t_2 \ldots t_n\}, \{t_1, t_2 \ldots t_k\}),\ k \leq n$$

A definition relation puts each trigger in correspondence with an evolution operation, depending on the state of the MoD. For example, *add* or *specialize*. These operations are ordered chronologically as a tuple $\langle e_1, e_2, \ldots e_k \rangle$. Finally, the application of the tuple to the state $\mathcal{S}_i$ defines an effect relation:

$$(\mathcal{S}_i, \langle e_1, e_2, \ldots e_n \rangle, \mathcal{S}_{i+1}, \langle \{t'_1, t'_2 \ldots t'_j\} \rangle)$$

As a side-effect of the application of the evolution operations, a new set of *internal triggers*, (for instance, opportunities for generalizing over similar instances of some concept definitions) may be discovered.

To evolve a MoD is to manage relations of the types just defined under the constraints of well-formedness and integrity of MoD states. A description of MoD evolution is completed in the next three sections by presenting:

- a set of evolution operators (Section 6.4.2),
- guidelines for the definition of the evolution relations (Section 6.4.3), and
- guidelines for the scheduling of evolution (Section 6.4.4).

6.4.2 MoD evolution operators

The MoD evolution operators are defined in terms of their *effect*, bindings, secondary agenda, and the new internal triggers they uncover. By secondary agenda we mean, a record of "obligations" generated by the application of the operators. For instance, some objects in the definition neighborhood of an object just added to the MoD may not be in the MoD yet. When an evolution step is completed, the agenda must

be empty. We use the notation T, to represent a set of triggers and A, to represent the agenda of obligations.

Addition

$$\text{Add: } \mathcal{S}_i, o_j \rightarrow \mathcal{S}_{i+1}, A, T$$

The effect of the *add* operator on the current state and an object definition o_j, is to produce a new state that differs from the previous one in that o_j has been included in the hierarchy of MoD objects, i.e., classified according to its the property values. Classification in a taxonomic structure proceeds from the root of the hierarchy down. At each level in the hierarchy, the new definition is either found to be an instance of one or more existing classes, leading to a re-classification among the children classes, or it becomes a (singleton) class of its own. The definition of *isa* in Section 5.5 does not preclude multiple inheritance.

In the process, the agenda A is updated, 1) by adding references to objects in its definition neighborhood that need to be included in the MoD, and 2) by removing resolved references to the subject.

If we take, for example, the WRITE operation in the Disk Drivers Domain, the integration of the Winchester driver to the MoD requires that a new activity object: IMI-WRITE be added to the MoD, together with an associated DESIGN and PLAN. Further, because of the difference with existing implementation plans, the IMI-WRITE-PLAN introduces additional structures such as RETRY-COUNT and ERROR-STATISTICS-COUNT. All these become annotations in A. At a different level in the network, the Disk Drivers Taskable Code (DDT*) domain, new activities must also be defined— SET-RETRY-COUNT and RESET-ERROR-STATISTICS—to maintain implementation integrity.

The addition of the new definition also uncovers new opportunities for evolution. Typically, the evolution triggers suggest more additions, either in the form of new objects or generalization over existing classes, e.g., information on 128 sectors per track for the CLF driver and 512 sectors per track for the IMI-7710 driver suggest a more general notion of Number of Sectors per Track, NSPT, with an expanding range of values.

Generalization

$$\text{Generalize: } \mathcal{S}_i, \{o_j\}, o_g \rightarrow \mathcal{S}_{i+1}, A, T$$

The generalization operator relates a class of objects definitions $\{o_j\}$ with a generalized definition, o_g. Generalization (see Figure 6.15) may proceed over: 1) property values of one or more properties, 2) multiple level-property values: one or more property values simultaneously 3) structure, that is, relations between property values (i.e., the property is an assertion on other property values.)

State $\mathcal{S}_{i+1}$ differs from $\mathcal{S}_i$ in the definition of o_g. In case of a generalization, the agenda A does not register any new requirements. There may be, however,

Generalization heuristics: *Notation*: P is an set of properties-value pairs describing MoD objects. $X ::> Y$ means that X is a definition for concept named Y. $X_1 \angle X_2$ means that the object description X_2 is a generalization of the object description X_1.

- Dropping condition:
  $P \wedge p ::> C \angle P ::> C$
- Adding alternative:
  $P_1 ::> C \angle P_1 \vee P_2 ::> C$
- Extending reference:
  $P \wedge [p = V_1] ::> C \angle P \wedge [p = V_2] ::> C$, where $V_1 \subseteq V_2 \subseteq domain(p)$.
- Closing interval:
  $(P \wedge [p = v_1] ::> c) \wedge (P \wedge [p = v_2] ::> C) \angle P \wedge [p = v_1..v_2] ::> C$
  if p is a linear property.
- Climbing the generalization tree:
  $(P \wedge [p = v_1] ::> C) \wedge (P \wedge [p = v_2] ::> C)...(P \wedge [p = v_n] ::> C)$
  $\angle P \wedge [p = v_g] ::> C$,
  if p is a structured property and v_g is the most-specific generalization of $v_1, v_2, ..., v_n$ in the ISA hierarchy,
- Suppression of detail rule (or, turning constraints in to variables)
  $p[a] \wedge p[b] \wedge ...p[n] ::> C \angle \exists x, p[x])] ::> C$
- Turning conjunction into disjunction:
  $p_1 \wedge p_2 ::> C \angle p_1 \vee p_2 ::> C$
- Extending quantification domain:
  $\exists(I_1)x, p[x] ::> C \angle \exists(I_2)x,\ p[x] ::> C$, where $I_1 \subseteq I_2$.
- Inductive resolution:
  $(P \wedge [p_1] ::> C) \wedge (\neg P \wedge [p_2] ::> C) \angle [p_1 \vee p_2] ::> C$
- Extension against rule:
  $(P_1 \wedge [p = v_1] ::> C) \wedge (P_2 \wedge [p = v_2] ::> \neg C) \angle [p \neq v_2] ::> C$
- Constructive generalization (requires the acquisition of p_2)
  $(P \wedge p_1 ::> C) \wedge (p_1 \Rightarrow p_2) \angle P \wedge p_2 ::> C$

Figure 6.15: Generalization heuristics

new opportunities for generalization, for instance, by applying the Climbing the Generalization Tree heuristic (see Figure 6.15), these are recorded in T.

A trigger that suggests a class of objects as candidates for generalization is a reference to the generalization operator, and to a class of objects, but does not specify o_g. For instance, applying the Climbing-the-generalization-tree heuristic, all the objects in the definition neighborhood of the members of $\{o_j\}$ are now candidates for generalization. For example, if the definition of an entity e is generalized to e', it might be of interest to generalize some of the activities a that had e as an input to activities a' that can take e' as inputs.

The *definition* of a generalization operation given a trigger is not a trivial step and often requires the acquisition of justification information. Figure 6.16 illustrate a Generic-DESIGN description resulting from the generalization of CLF-WRITE and IMI-WRITE into GENERIC-WRITE. The generalization of an implementation plan, requires that the new version be strictly or upward compatible with the instances. Generalized compatible implementations can be designed as truly generic PLANs, by coercion over inputs and outputs, or by overloading the plan with alternative implementations [CW85].

Merging

$$\text{Merge: } \mathcal{S}_i, \{o_j\} \rightarrow \mathcal{S}_{i+1}, A, T$$

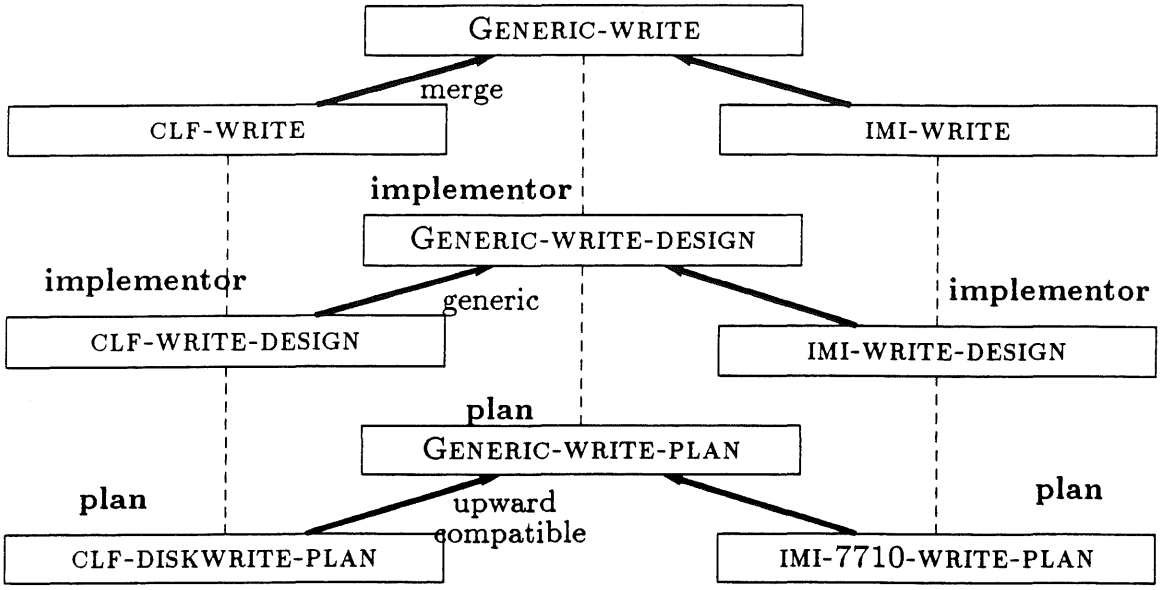
The effect of a merge operator is to create a definition for an aggregate object, e.g., the merge a collection of drive-related information into a composite entity DISK-INFO in the Disk Drive Description domain. The inverse operator: Split is used to achieve the opposite effect of decomposing an object definition into two or more definitions.

Removal

$$\text{Remove: } \mathcal{S}_i, o_j \rightarrow \mathcal{S}_{i+1}, A$$

The effect of the removal of an object definition is a new state where the subject, o_j, has been eliminated from the taxonomy of MoD definitions, and where the agenda A has been updated with references to objects in the definition neighborhood of o_j. Each of those objects becomes a candidate for removal or repair. Their definitions must be revised to maintain the integrity of the MoD state. The effect of a removal may ripple over large portions of the MoD. The analyst may conclude that it is too expensive to perform and advice against performing a particular evolution step, (i.e., in effect, a restriction is established to eliminates that trigger from consideration).

An example of this situation arose in the analysis of the disk drivers domain, when a decision was made to eliminate two explicit entry points: WAITDONE and LSNTOPSN for justified performance reasons [Ara88b]. The impact of removing these two object from the MoD propagated even to other domains in the network, in particular the superordinate OS Sector Level I/O domain.



Generic-WRITE-DESIGN *isa* DD*DESIGN *with*

id *spec*: Generic-WRITE
ancestors: CLF-WRITE-DESIGN, IMI-WRITE-DESIGN
method: generalization

plan p: Generic-WRITE-PLAN

design-goals
g1: "The faster the better"
g2: "Reliability and principle of least surprise"
g3: "Do not tie-up the hardware unnecessarily"

rationale
r1: "Design meta-plan: driver's structure can be divided into two tasks:
part1. the part that does not touch the hardware
part2. the part that (almost exclusively) talks to the HW"

constraints
c1: "initiate a single sector WRITE operation"
c2: "self.g2: a time-out must be set for each physical operation"
r1: "(self.g3) CHECK-LSN before ALLOCATE RESOURCE"
r2: "do RESET-ERROR-STATISTICS after allocating resource,
it does not make sense to do it before CHECK-LSN"

Figure 6.16: DESIGN generalization (internally) triggered-step: WRITE generalization

Repair

$$\text{Remove: } \mathcal{S}_i, o_j \rightarrow \mathcal{S}_{i+1}, A, T$$

The effect of a repair of an object definition is a new state where the subject, o_j, has been substituted by a new object definition o'_j (consequently its position in the MoD taxonomy may have changed). The agenda A is updated with references to the objects in the definition neighborhood of o_j. Each of those objects becomes a candidate for repair.

6.4.3 Monotonic and non-monotonic evolution

A MoD state provides a "closed-world" representation for the problem domain. Only information that is believed to be *relevant* in the problem domain is represented explicitly as a state $\mathcal{S}$ of the MoD. We distinguish between two kinds of relevant statements: those that assert true facts, $\mathcal{S}^+$, and those that assert facts that are not true $\mathcal{S}^-$.

A MoD evolves as new information is asserted of the domain. The new information, I, appropriately represented, is incorporated into $\mathcal{S}$. If $\mathcal{S}_i$ denotes the state $\mathcal{S}$ at time t_i, we say that the MoD evolves monotonically from state $\mathcal{S}_i$ to state $\mathcal{S}_{i+1}$ if $\mathcal{S}_i \cup I \subseteq \mathcal{S}_{i+1}$ is consistent. In other words, monotonic evolution augments of the model while maintaining the truth value of the information already available.

We say that a MoD evolves non-monotonically when the resulting $\mathcal{S}_{i+1}$ is inconsistent. That is, incorporating I introduces conflicts with existing information that must be resolved. This may result in I being rejected as an appropriate addition to the model, in pre-existing information being revised, and/or in new information being added. The relations among object definitions in the MoD are the basis for the propagation of change in the model. In summary, resolving inconsistency requires transfer of some information from $\mathcal{S}^+$ to $\mathcal{S}^-$ and vice versa. The purpose of this traffic is to define a new consistent state for the MoD.

Monotonic evolution—a form of grammar acquisition

There are similarities between the definition of MoD construction as the monotonic integration of models of exemplars into an existing MoD followed by the generalization, and the the process of "grammar inference" as it has been approached in linguistics [Gol67] [KK76] and artificial intelligence [CF82] [And77] [LC87] [VB19].

Some of those approaches (e.g., [KK76]) formulate a overly general hypothesis grammar and then refine it by applying simplification heuristics and by collecting new example of sentences (i.e., negative training instances). Other approaches are semantically based (e.g., [And77] [App83]) and use the semantics of the language to constrain the search for plausible grammars. Our method to MoD evolution uses a combination of both techniques.

The refinement of over-generalizations appears to be a good technique for structuring the dialogue with the domain expert. Gold [Gol67] called this learning situation an "informant presentation". The informant—in our case the domain expert—has the capability to produce positive and negative examples that can be used to determine the true language. [KK76] applied this technique exhaustively to context free grammars. In our case, the analyst proposes (over) generalization of a model or object definition to the domain expert, and the expert is challenged to poke holes at it, i.e., to produce instances of that are illegal, or to make explicit assertions about the problem domain or of design methods that falsify the generalization.

The semantic approach is illustrated by the Language Acquisition System, LAS [And77]. LAS incrementally induces a grammar for a small subset of natural language consisting of noun phrases. LAS "knows" about some principles for structuring phrases, for instance,
Noun-phrase → { morphemes } { Modifier } Noun { morphemes } { Modifier }
Modifier → Proposition { Modifier }

These rules indicate that a noun phrase consists of zero or more non-meaning-bearing morphemes, followed by an optional embedded list of prepositional modifiers, followed by an obligatory noun, and so on [And77, p. 143]. Using this kind of knowledge, and information on the roles of the terms in the sentence, LAS can induce a grammar to recognize and generate a given set of sentences. In our case, the definitions in the MoD language fulfill a similar role, allowing to recognize instances of applications in the domain, and providing constraints to the generalization of object definitions.

Non-monotonic evolution—MoD revision

Resolving inconsistencies requires that we identify it sources, that we revise them, and then we propagate the changes through the database to recursively remove any additional inconsistencies introduced by the change. As the MoD state evolves incrementally, inconsistencies result from the latest item of information asserted. Removing an inconsistency involves:

- retracting the latest assertion made, or
- changing the truth status of existing information, i.e., transfers between $\mathcal{S}^+$ and $\mathcal{S}^-$, or
- introducing additional information.

These actions are typical of a truth or belief maintenance system. Conceptually, the non-monotonic evolution the MoD poses a problem of truth maintenance or belief revision. Belief revision in MoDs as we have described differs from the conventional formulations of truth maintenance [Doy79] [dK84] in several respects:

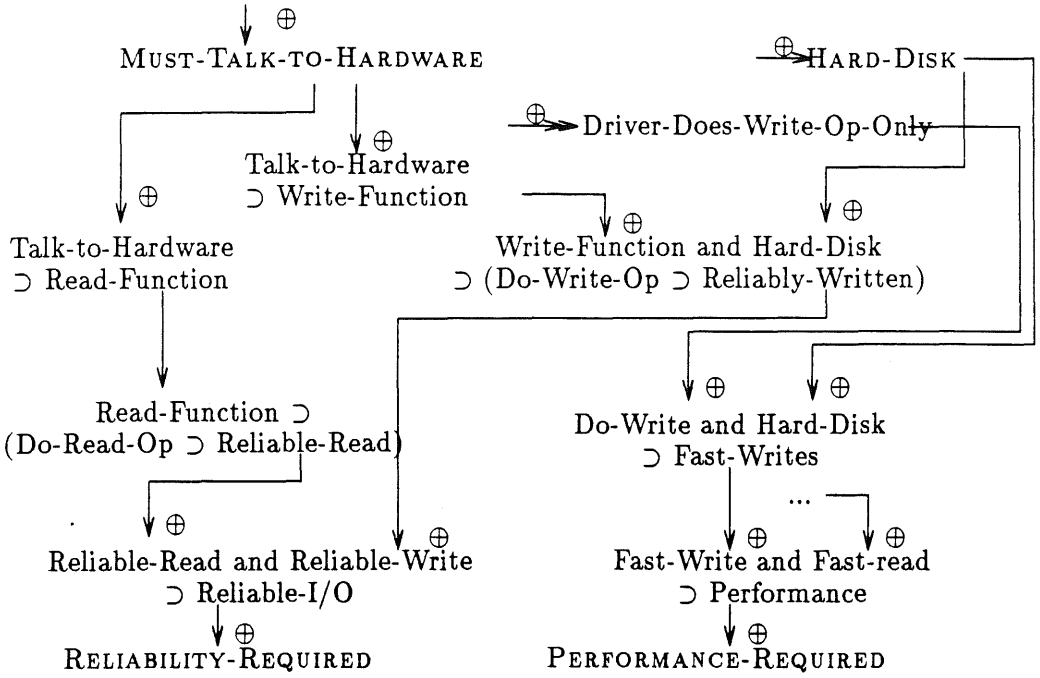


Figure 6.17: A justification network in the Disk Drivers domain supporting only hard disks.

the degree of formality in representation, the kinds of information represented and the degree of knowledge about the reasoning process that the TMS mechanism can use to organize the traffic between $\mathcal{S}^+$ and $\mathcal{S}^-$. The resolution of inconsistencies can be based on general knowledge of the logical system used (e.g., propositional calculus), or on domain-specific rules that indicate to the system under which circumstances which resolution steps are appropriate [Pet87]. In our case, we solve the problem by using an oracle—the domain expert—and each transfer of information between $\mathcal{S}^-$ and $\mathcal{S}^+$ becomes a knowledge acquisition event. In this sense, maintaining the integrity of the MoD becomes yet another mechanism for focusing the task of information acquisition.

To clarify the TMS view on non-monotonic evolution we now discuss an example in the domain of disk drivers. We adopt a "historic" perspective and trace the evolution of the domain as it actually happened.

Figure 6.17 shows a sample justification network in the domain of disk drivers at an early date, when only Winchester disks were supported. The arrows labelled "$\oplus$" represent supporting relations between facts. The "$\ominus$" label indicates negative support. Premises are those facts that are asserted of the domain without justification (pointed by arrows without support). RELIABILITY-REQUIRED and PERFORMANCE-REQUIRED were two goals supported by the design decisions made

at the time. The network captures dependencies between facts in the domain, the developer goals, and their beliefs about appropriate disk driver design.

The need to support a new storage media, floppy disks, lead to catastrophic changes in the assumptions and beliefs underlying the design of disk drivers at the time. Writing on floppy disks was not a reliable operation. Figure 6.18 captures the evolution step. The support relations that become untrue have been re-drawn as dashed arrows, and the new relations associated with the introduction of floppy disks as premises have been drawn as thick arrows. The assumptions on reliabile writes had to be revised. As a consequence, a new notion of *verification* had to be introduced.

Figure 6.19 makes presents the state of the justification network at the time when a *verify* operation was added to the drivers. Immediately after each write-sector operation, the driver would read the sector back and verify its integrity. This design decision adds *negative* support to the "Fast-Write" and thus, operates against the goal of having efficient I/O. The trade-off made by the designers at the time was to give priority to the reliable operation of the devices rather than to better performance at the risk of losing data.This situation is captured in several design assertion objects [Ara88b],

> **Floppy-disks-require-write-read-verify** "Floppy disks require that each sector written to the disk be read back for verification purposes in order to insure a reliable transfer. (Hard disks don't)"

> **Combined-Write-Verify-Function-Prevents-Efficient-Burst-Writes** "The old style WRITE logic prevented efficient burst writes on floppy disks because only one write call to the driver was made; at that point, the driver, to do its job properly, had no choice but to write the sector, and to do the verify, before returning control to the OS. Thus burst writes appeared to the disk as WRITE(n), VERIFY(n), WRITE(n+1), VERIFY(n+1), etc. A WRITE(n) followed by a VERIFY(n) takes, due to disk geometry, a complete disk revolution, and is thus (NSPT $\approx$18 times) slower than the theoretical optimum. This was barely tolerable in the old systems."

Eventually, the lack of performance on disk-writes became a major issue that affected the perceived value of the operating system. This situation lead to a major revision of design assumptions and design decisions—a new generation of drivers was born—that is captured by the justification network in Figure 6.20.

Deciding which items of information to transfer between $\mathcal{S}^-$ and $\mathcal{S}^+$ is a non-trivial issue to be negotiated between the actors in the domain engineering process. The revision of a problem-level goal or of a software design decision may result so many changes to the MoD state that cost of MoD evolution is higher than the

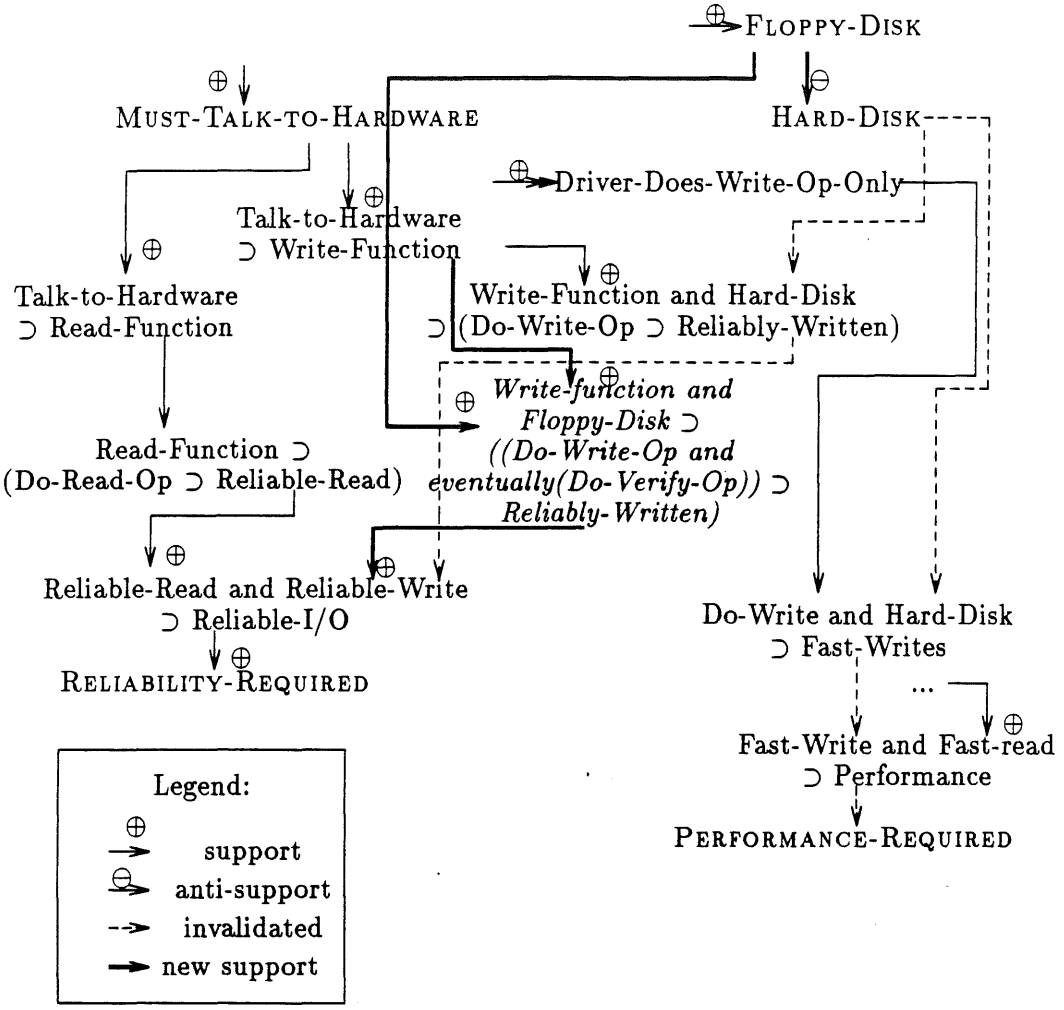


Figure 6.18: Non-monotonic evolution—introducing support for floppy disks.

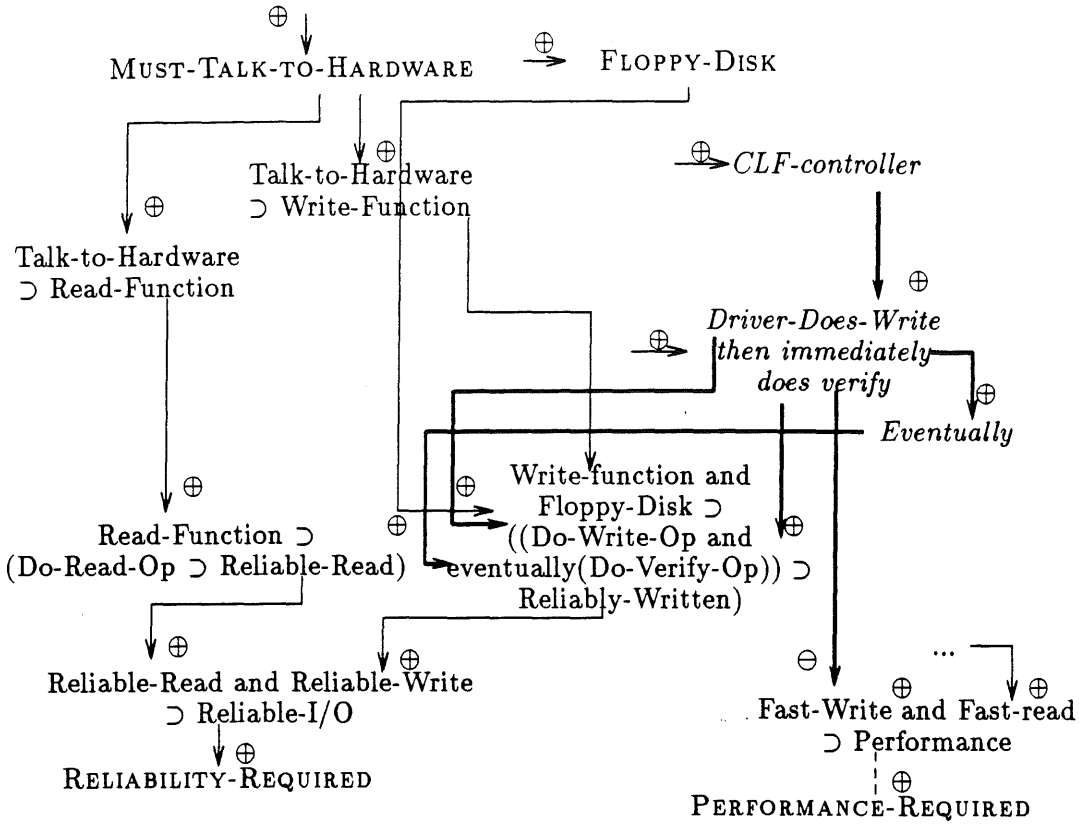


Figure 6.19: Reliability given more priority than Performance

potential benefits of reuse. For instance, in the case we have just discussed, the changes required to support efficient *and* reliable operations on floppy disks rippled through large areas of the domain network, and led to the redesign of parts of the operating system itself. A similar situation arose when attempting to support double-density floppy disks with a first, single-density track. In the first case, the need for better performance became so pressing that it made the evolution steps worthwhile. In the second case, it was decided (based on the history of requests of drivers for such types of floppies) that it was not cost-effective to develop them.

In our discussion of non-monotonic evolution we have explored the problem of making explicit the impact of changes in the structure of the MoD. We have discussed the role of justification networks in making explicit the assumptions held by experts and designers and the relation between design decisions and domain goals, and the use of a TMS-like approach to assess the impact of changes. Once a domain object has been targeted for removal or substitution, the associated justification, definition and implementation relations provides a network over which to propagate changes.

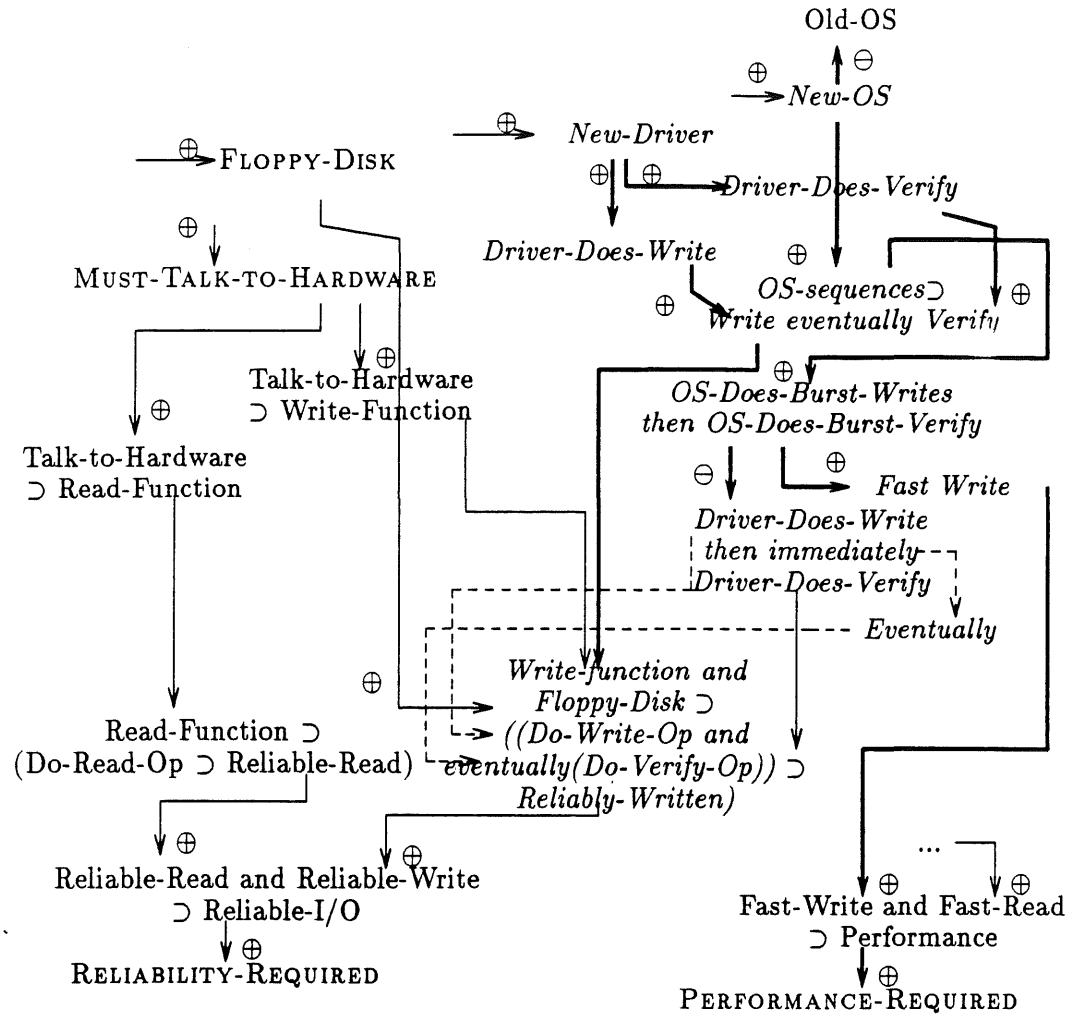


Figure 6.20: Achieving both Reliability and Performance

6.4.4 The scheduling of evolution

The order of evaluation of evolution triggers has an impact on the economics and the quality of the results of a domain analysis. In principle, the larger the number of triggers evaluated, the higher the probability of finding relevant ways of improving the coverage of an MoD. In reality, there exist economic constraints on the task of domain analysis that force us to evaluate a limited number of triggers.

We may invest in the search for internal triggers—an *active* evolution regime—or wait for external triggers—a *reactive* regime. Also, uncovered triggers may be evaluated incrementally or in batch mode.

Our limited experience suggests that there exist technical and economic advantages in starting a domain analysis in active, batch mode, and switching later to a reactive, incremental mode. The batch mode takes place mostly during the preparatory phase in the domain analysis method. The incremental mode is realized by subsequent MoD-exemplar integration steps.

A heuristic for choosing between regimes suggest that attention be paid to the number of new object definitions provided by the environment. The chronological sequence in Figure 6.6 can be used to illustrate the switching between regimes. In retrospect, a practical heuristic would suggest to switch regimes when derivative of the "new object definitions" curve decreases— after 15 applications have been examined, in Figure 6.6—and operate in reactive mode until there are indications of that a "new region of the domain" has been discovered—when applications 65 to 68 are found, in Figure 6.6.

If most of the exemplars chosen in the initial phase are in the Twenty-Percent that contributes eighty percent of the reusable abstractions, then the switching point occurs fairly early in the domain analysis process, and the more inexpensive reactive mode of evolution predominates until a new catastrophic point is reached.

6.5 Validation of acquired information

Summary. Static analysis techniques based on the structure of the MoD can uncover only some forms of inconsistency and incompleteness. A dynamic analysis is required to validate the information acquired. The feedback cycle between the learning and performance components in the reuse system provides a natural validation loop. In practice, the loop may be short-circuited using an informal "hypothetical reuse" technique.

Static analysis techniques focus on the dormant MoD or infrastructure. The basis for the analysis is the syntax of the MoD language, and the MoD integrity and sufficiency rules.

The "synchronization" between the semantics of S-objects as defined by S-level

assertions, and the operational semantics defined by PLANs can be verified depending on the formality of the representations and the resources available [Bax87b].

Still these forms of verification cannot cannot answer questions about the completeness or adequacy of available models to achieve the competence goals set for a reuser: Would the target reuser be able to successfully compute $MoT(S, RI) = I_S$?

That question requires a *dynamic analysis*. An infrastructure RI must be specified using the information in the current MoD state, and an instance of RI must be built and exercised. It requires a *critic* to evaluate the results of computing the reuse function. Feedback from the critic (Figure 6.2) points to inadequacies of three types: poor implementation of the infrastructure, poor specification (i.e., the information *projected* from MoD onto infrastructure is insufficient), and lack of "appropriate" information. This last group triggers the revision of the state of the MoD.

The actual reuser is the natural candidate to play the role of critic (see next Section). In practice, an informal emulation of the reuser is useful (Section 6.5.2).

6.5.1 Validation by actual reuse—the TMM experience

We have successfully applied this form of validation by actual reuse using the fact that we could use a mechanical reuser to do the task. The TMM (Transformational Model of Maintenance) project reported in [ABFP85] and [ABFP86]. The aspect of that project relevant that our discussion involved the construction of a domain network for generating software implementations using the Draco system. The problem domain was that of application generator-generators similar to Draco. The structure of a domain network was outlined and the specification and implementation objects for each domain (in that case Draco-domain languages and libraries of Draco refinements) were generated mostly through design recovery on the Draco code "guided" by the knowledge of the persons involved: domain-specific knowledge about application generators (the "domain" of the system to be generated), general software design experience, and general knowledge of Lisp dialects and operating systems.

Using the resulting Draco-domain network, a first version of Draco system was mechanically generated using Draco. The first attempts did not result in satisfactory implementations. A record of flaws of the three types listed above was produced in each case and after a revision of the network, the process was repeated. With each cycle the focus shifted from implementation bugs to conceptual bugs (i.e., domain analysis and infrastructure specification). Each cycle involved three to four mandays for the analysis of the faults detected in the previous run and for proposing revisions to the definitions in the domain network, and approximately 10 hours of VAX-750 to produce each new implementation. The process of convergence to an adequate infrastructure required of approximately a ten cycles. Adequacy was defined by a set S to be implemented successfully. For the purpose of the TMM investigation S had one member, the Draco system itself.

The TMM project showed the practical value of design recovery on a particularly complex problem domain and the utility of the generate-test-revise approach to tuning a domain network as a way to systematically approximate adequate MoDs.

It could be argued for some technologies the generate-test-revise approach would be unbearably expensive. A technique equivalent to an "early prototyping" of the reuse infrastructure is useful to discover the most obvious mistakes in the conceptual definition of theinfrastructure. We call it *hypothetical reuse*, to contrast it with actual reuse.

6.5.2 Hypothetical reuse

Hypothetical reuse tests the adequacy of the information captured by the MoD while ignoring component design and implementation issues. The domain analyst plays the role of critic by emulating the reuser in the execution of the MoT. It involves two phases:

- Specification. Formalize the description of a given application in terms of the domain-specific vocabulary offered by the MoD, and
- Hypothetical implementation. Assemble an implementation using the PLANS associated with the objects used to specify the application.
  - Verify interconnections. Verify that preconditions and obligations of the parts are met by other parts in the implementation.
  - (Informally) verify the synchronization of specification and implementation semantics. For each S-object translate the assertions involving the subject into assertions of S-objects in subsidiary domains, and verify that the PLANs associated to the subject preserves its semantics [MT84].

The specification stage allows the domain analyst to uncover errors in the definition of the semantics of the domain language: missing terms, overly restrictive or incomplete definitions of objects, etc. The hypothetical implementation stage offers the opportunity to identify missing interconnections, unfulfilled preconditions or obligations, some implementation errors. The quality of the inspection depends on the thoroughness and ability of the inspector.

Hypothetical reuse appears to be cost-effective for uncovering major conceptual deficiencies in the MoD before implementing an infrastructure. It is not a substitute for the actual implementation *and execution* of the resulting software. A similar technique, hypothetical synthesis, was employed in the Φ-NIX project as a means to identify the kinds of knowledge required by an automatic programming system [Bar85].

6.6 Summary

The evolutionary view of domain analysis reflects the fact that the population of problems perceived as important by a given community, and the knowledge of the community evolves over time. Further, one would expect that the knowledge and expectations of the target reusers would evolve as a consequence of the application of reuse. We may summarize these observations as a Hypothesis of Partial Knowledge.

> The actors in the domain engineering process – problem domain and software experts, domain engineers and reusers – have fragmentary knowledge of the problem domain and solution methods. Consequently, *MoDs are bound to be incomplete, biased, time-dependent representations of beliefs.*

Setting aside pragmatic distinctions between MoD preparation and MoD evolution, there exists a single underlying process: *incremental approximation* to a desired level of reuser competence through the transformation of the MoD state. The evolutionary view captures the process in terms of:

- A space of MoD states.
- Objective criteria for selecting a direction of evolution
- Criteria for defining new evolution steps.
- Procedures for the applying the steps
- Criteria for scheduling evolution – choosing when to apply an evolution step and chosing when to stop.

There are advantages to viewing MoD evolution as a step-wise process,

- It provides a conceptual framework to describe the practical aspects of domain analysis and to communicate our understanding of the process.
- As the process becomes explicit, it becomes an object of study in itself. It then becomes possible to make assessments of performance and to compare alternative approaches.
- Provides a rationale and a chronological trace to justify each state of the MoD.

We have produced a limited demonstration of the utility of a TMS-like approach to make systematic the non-monotonic aspects of evolving a MoD. The approach appears to have value as a means to capture the evolution of design rationales in general. We have not explored this path in depth yet.

Justification networks are important in the design of domain-network out of pre-existing domains. We have barely scratched the surface of these and other domain engineering-in-the-large problems.

Chapter 7

Specification of a Reuse Infrastructure

Chapter summary. In Chapter 6 we examined one aspect of the proposition "domain analysis is a form of incremental learning": enhancing the competence of the reuser through classification and induction of domain-specific information and belief revision. In this Chapter we examine another aspect of that proposition: improving efficiency of the reuser.

The efficiency of a reuser depends on, 1) the number of analysis steps that must be performed to recognize a specification as implementable (parsing the specification), 2) the number of composition steps that must be performed, 3) the number of component interconnections that must be enforced, and 4) the difficulty of "completing" an implementation (i.e., building and integrating parts that have not been captured during the domain analysis process). The efficiency of a reuser can be improved by pre-computing some of these steps.

We use the term "package" to broadly refer to a partial implementation of a composite specification. Packages must be completed to fit the particular features of a given specification (e.g., parts must be added, parameters must be instantiated). Packages behave like macro-operators that improve the efficiency of the reuse process but do not affect its competence.

The results presented in this chapter are:

- a method for acquiring and representing information on patterns of reuse, and
- procedures that use that information to answer the packaging question—What information should be packaged together?

Given:

- A model of a domain MoD, and
- A restriction on the MoD, characterized by
  - a set of domain-specific goals, or,
  - a collection, S, of specifications, or
  - a collection, I_S, of implementations.

Find:

- A restriction on the domain language of MoD that allows to specify a class of systems S' that achieve the proposed goals, or that is a superset of S, or of the systems implemented by I_S.
- A consistent family of implementation plans sufficient to implement the specifications in S' using the operators available to the reuser.

Figure 7.1: Specifying a component basis

7.1 Component basis

We use the term *component specification* (or component, for short), to refer to a MoD object definition together with its associated DESIGN and PLAN definitions and all justifications objects that support these definition.

Given a reuser R and a class of specifications S, a collection of components $\mathcal{B}$ is called *component basis* for S with respect to R if:

- the class of software systems that can be specified and implemented using components from $\mathcal{B}$ and the operators available to R is a superset of S, and
- the members of $\mathcal{B}$ cannot be generated from other members of $\mathcal{B}$ using the operators available to R.

A basis provides a first-order reuser with a restricted set of elementary parts from which to construct implementations. From a MoD many bases can be derived to fit the needs of different reuse environments, or different reuse patterns within a single environment. The definition of a basis is a restriction on a MoD state. The restriction may result of specifying a set of domain-specific goals, a collection of exemplar specifications, or a collection of exemplar implementations. The result is a sub-language of the MoD language, the "basic language" for a particular environment.

The process of defining a restriction to the MoD can be viewed as a marking of the object definitions in the MoD objectbase guided by the MoD integrity rules. Initially, the concept descriptions in the database are unmarked. Depending on how the restriction on the MoD is defined, the marking process proceeds "top-down" by selecting a set of specifications, or "bottom-up" by selecting a set of implementations.

Given a MoD state that satisfies the well-formedness and integrity constraints defined in Section 5.7, many bases could be selected. If we proceed top-down from the specification level

1. A set of exemplar specifications are selected and the constituent S-objects are marked.

2. (To reduce the number of terms in the vocabulary, and perhaps also implementation costs) if a set of specification objects present in different specifications are found to be special cases of a more general object, mark the more general object.

3. For each marked concept description, mark its definition neighborhood.

4. For each marked concept description, mark its implementation closure.

The result of this process is a collection of components that define the semantics of a domain-specific language, together with a description of plans for the implementation of each construct in the language. These components constitute a specification for a reuse infrastructure that must be submitted to a software designer for its actual implementation. Issues related to the design of reusable first-order components in the context of the Ada language are discussed in [Boo83] [B+86] [Gau86a] [Gau86b] [McC86].

7.2 Efficiency in first-order reuse

Summary. An analogy between the task of first-order reuse and the operation of a memory hierarchy is discussed. The analogy suggests that the problem of reuser efficiency can be described in terms of a Reuse Working Set of aggregate components, RWS. The components in the RWS are usually referred to as "packages". In later sections, the question of, What information should be packaged together is answered by providing methods that realize a package selection and replacement strategy.

7.2.1 Augmentations to a component basis

To implement a specification a first-order reuser must,

1. decompose (or, parse) a customer specification into its basic elements,
2. identify appropriate implementing components (i.e., those that meet interconnection constraints),
3. adjust the implementation components,
4. interconnect the implementation components,
5. If the specification is on the "boundary" of the domain, i.e., there are parts that have not been captured during the domain analysis, there is a need to complete the implementation by producing new parts.

One way of making this process more efficient is to pre-compute some of the steps. In other words, a sequence of steps prescribed by the MoT of the reuser can be substituted by a macro-operator that achieves the same effect. The results of decisions made and of operations applied are stored to avoid having to re-compute them at reuse-time.

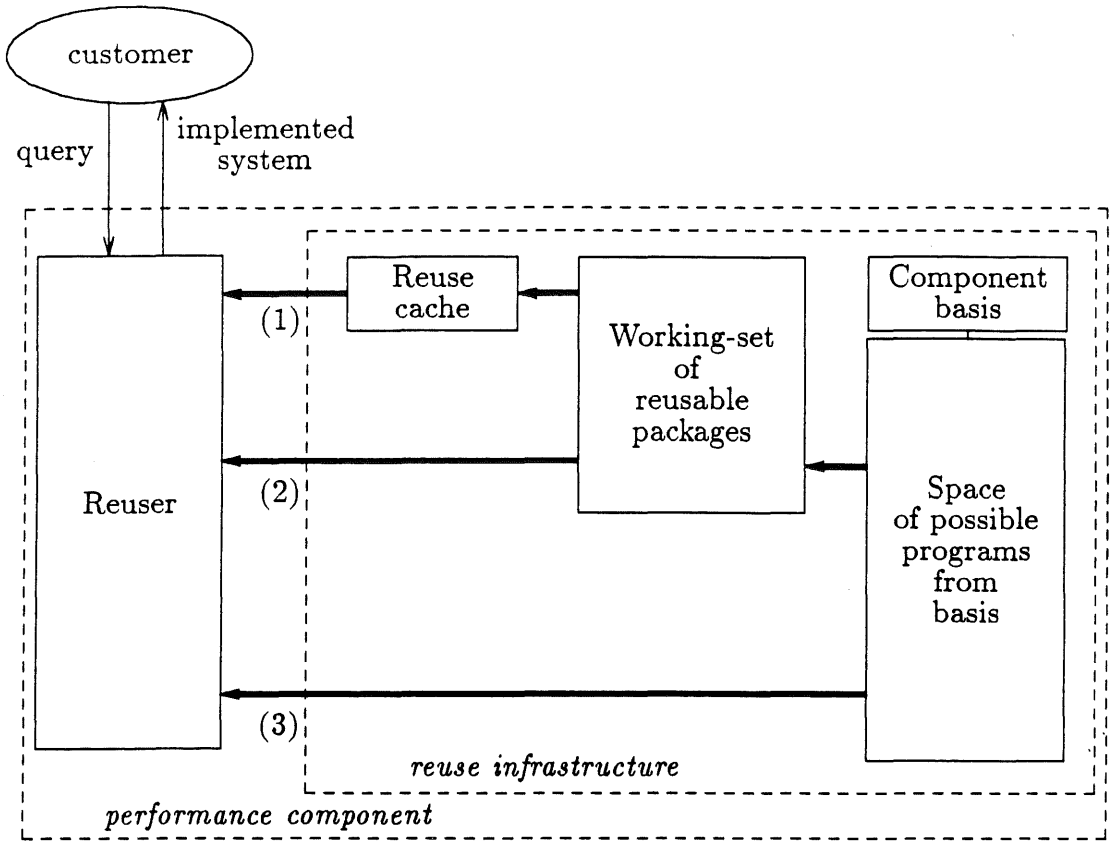
Each one of these pre-packaged components, is an augmentation to the component basis for the purpose of enhancing the efficiency of the actual reuse process. The motivation is economic, based on the assumption that, 1) the cost of reusing a package is lower than the cost of implementing a specification using basic elements, and 2) if a package is reused often enough, the savings from its reuse are greater than the cost of building it.

When specifying a reuse infrastructure we must decide which augmentations are worth implementing. In preparation to answer these questions we introduce an analogy between a compositional reuser and the management of a memory hierarchy.

7.2.2 The memory hierarchy analogy

The task of a first-order reuser can be compared with the management of a memory hierarchy with levels of storage having different access times. A basic trade-off in the organization of a memory hierarchy is the size and cost of the memory versus the access time. The goal of the hierarchy manager is to define a relation between the frequencies of access to units of information, and their placement in the hierarchy, such that programs attain adequate average performance at a reasonable cost.

We construct an analogy between a memory system and the performance component of a reuser (Figure 7.2) based on similarity of goals: support efficient access to memory and support efficient access to implementations. The memory manager views a program as a sequence of memory references. The reuse system views its environment as a sequence of specifications that it must implement, or *queries* submitted by a customer to the reuse system. For a given reuser, R, we define a *reuse event* as an attempt to implement a specification reusing the available infrastructure. A record for a reuse event is the (possibly incomplete) specification that was submitted to the reuser. A reuse event may be successful or not.



memory	....	space of known implementations
memory address	....	specification
memory reference	....	construct an implementation
history of memory references	....	reuse log
memory page	....	reusable package
working set	....	set of reusable packages
memory cache	....	set of instantiated packages

Figure 7.2: The memory management analogy

The reuser can construct implementations by adjusting and interconnecting basic domain-specific components (arrow 3 in Figure 7.2); may select a partially implemented package (arrow 2), or a fully implemented package (arrow 1). Each arrow has an associated construction cost that depends on the degree of aggregation of the parts; for example, the cost of completing a package by instantiating paramenters is expected to be lower than having to compose the implementation using basic components.

Both the memory manager and the reuser systems attempt to improve *average efficiency*. The packages in the cache can be regraded as a restriction, or a window, on the class of all domain-specific programs that can be constructed using basic elements (which in turn is a restriction over the set of all programs that can be constructed using some general purpose language).

On the memory side of the analogy, the principle of (temporal and spatial) *locality of references* is central to the successful operation of a memory hierarchy. The principle states that there is a tendency for a program to reference those pages that were referenced in the near past. Denning [Den68] defined the *working set* of a program at a time t for a window T, as the set of pages which have been referenced in the interval $(t - T + 1, t)$. A useful heuristic to improve the efficiency of the memory system is to keep the working set of a program in the levels of the hierarchy with faster access times.

Our Domain and Gradual Evolution Hypotheses (Section 2.5) are analogous to the principle of locality of reference, and are as essential to the performance of a reuse system. These hypotheses state that the queries to a domain-specific reuser *are* similar, and that the population of queries tend to change slowly over time.

On the reuse side of the analogy, a *reuse log* for a reuser R, $Log_R(T)$ is a record of reuse events over a period of time T. The log includes an entry for each reuse event consisting of the specification submitted together with an indication of whether an acceptable implementation was produced. If the implementation process does not succeed, a description of the failure must be recorded to provide feedback to the learning component in the reuse system. A failure occurs when part of a specification cannot be parsed or implemented.

Packages in a reuse infrastructure are like memory segments, they capture the similarities and semantic dependencies between classes of specifications.

We define a *reuse working set* for the environment of a reuser R at time t, as a collection W of packages, such that *most* queries in $Log_R(T)$ can be implemented by completing members of W.

We define a *reuse cache* for the environment of a reuser R, as a collection H of completed packages (i.e., fully implemented software systems), such that there exists a high probability that queries in $Log_R(T)$ can be implemented by members of H as they are.

In summary, the task of evolving a reuse infrastructure is comparable to that of managing of a memory hierarchy. The domain engineer must select what infor-

Given:

- A reuse infrastructure, consisting of a components basis, a reuse working set, and a reuse cache,
- a reuse log,
- a customer specification Q

Do:

1. Compute a package function Π such that, $\Pi(Q)$ implements Q.
   (a) Compare Q with past submissions in the reuse log,
   (b) Find a class C of specifications similar to Q
   (c) Complete the query producing $Q_{completed}$
   (d) Define $P = \Pi(Q_{completed})$ to be the package with the least reuse-cost used to implement the members of C.
2. If the cost of reusing P is less than the estimated cost of specifying and implementing Q from basic elements, else return P,
3. else, implement Q using basic elements.

Figure 7.3: Reusing packages

mation to package to improve the average efficiency of the reuser according to the patterns of reuse in the environment.

7.2.3 Reuse within a working set strategy

Figure 7.3 summarizes the process of attempting to implement a specification by reusing packages. The reuser attempts to identify the specification as a member of the class of specifications implemented by some existing package, i.e., to compute the function Π.

In the process of computing Π, the reuser may "acquire" system parts that were not originally specified, that is, a package may offers more than the customer requested. For instance, on the basis of past experience, a reuser may "guess" that additional features will be needed, even if the customer does not know about them. For instance, a specification for a floppy disk driver may include entry points for initialization, read and write operations, but no "escape" window to allow access to the controller's interface from the operating system level. The reuser may decide that the customer needs may best be served by a program that also includes an

Given:

- A model of a domain, MoD, and
- a history of the reuse environment, Log_R

Find:

- a description for patterns of reuse as a classification of the specifications in the Reuse Log, $Pattern = \{C_1, \ldots C_k\}$
- a packaging map Π, from C into the class of implemented (or, partially implemented) software systems, $\Pi(C_i) \rightarrow P_i$, such that $\{P_i\}$ improves the efficiency of the reuser.

Figure 7.4: Packaging—the problem of specifying a reuse working set

"escape" entry point, based on the the domain hypothesis and statistical experience accumulated in Log_R.

The Reuse Working Set as a sliding window over the space of potentially reusable packages, must be tuned to the patterns of reuse in the environment. The history of past reuse events is used as a predictor for future queries. The problem of specifying a reuse cache is summarized in Figure 7.4.

Two questions must be answered,

- How to classify past reuse experiences to uncover relevant patterns of reuse (the collection of C in Figure 7.3), and
- How to define the range of the packaging map, Π.

7.3 Identifying patterns of reuse

Summary. Packaging components improves the efficiency of a first-order reuser only if the packages in the reuse working set match (i.e., provide adequate implementations for the specifications in) the patterns of reuse in the environment. In order to implement a *packaging strategy*, an explicit characterization of patterns of reuse is needed. In this section we present one, based on the probabilistic clustering of specification features. This technique uses the Reuse Log as a predictor for future patterns of requests on the reuse system. The output of the process is a probabilistic classification tree on requests from environment.

Our motivation for defining patterns of reuse is to provide an objective basis for specifying generic packages. The overall goal in the design of a collection of packages is to minimize the average cost of reusing them, and to minimize the cost of implementing the packages. Quoting from Parnas' classic paper on the design of program families:

> "Experience has shown that the effort involved in writing a set of specifications can be greater than the effort that it would take to write one complete program. The (modularization) method permits the production of a broader family and the completion of various parts of the system independently but a significant cost. *It usually pays to apply the method only when one expects the eventual implementation of a wide selection of possible family members* [Par76, p. 7]"

In the foregoing discussion we assume that for each system specification submitted to the reuse system—an event recorded $Log_R(T)$—there is a *descriptor* consisting of a set of attribute-value pairs. In the domain of disk drivers, the attributes used identify features of the technology of the devices: "density", "number of R/W heads", "type of controller interface" (see Figure 7.5). In the UnixInit domain [TA88], the attributes employed to describe specifications are the customizations that must be encoded in the `.cshrc` file, e.g., "renaming", "maintaining records of command sequences", "executing commands from a file" (see Figure 7.7).

The attributes are identified during domain analysis, and correspond to features of the specifications that differentiate between variants in the domain. This approach has been advocated for a long time as a means of designing software systems to ease maintenance, for example, [Par72] [Par76][PCW83].

We aim at identifying clusters (classes or families) of specifications from $Log_R(T)$ such that:

- Members of each cluster satisfy similar combinations of requirements
- Members of different clusters satisfy dissimilar combinations of requirements

We have chosen to describe a cluster as a set of attribute-value pairs in which attributes may have several values. Each cluster descriptor is a "summary specification" for a possible package. Given a class C_i of specification descriptors, a ***possible package*** $P_c(C_i)$ is a descriptor for a package that could be use to implement the members of the C_i. The set of possible values associated with the attributes in the descriptor define the variation of external parameters that need be supported it the packages were to be implemented.

The success in computing the packaging function Π at reuse time, depends on having packages with a high probability of satisfying a customer request. Thus, a clustering function should "properly guess" attribute values not specified by the customer.

instance: COLOR COMPUTER

	Attribute name	Attribute value
attribute-1 :	type	(floppy)
attribute-2 :	heads	(1)
attribute-3 :	density	(double)
attribute-4 :	number sectors per track	(18)
attribute-5 :	number cylinders per unit	(35)
attribute-6 :	sector size	(256)
attribute-7 :	numbering within tracks starts with	(1)
attribute-8 :	numbering sectors on second side	(doesn't apply)
attribute-9 :	uniform density	(yes)
attribute-10 :	intelligent controller	(no)
attribute-11 :	Direct Memory Access	(no)
attribute-12 :	interface-type	(W1771)

Figure 7.5: Sample descriptor for a disk drive

We define a *pattern of reuse* in an environment over a period of time T as a partition of the events in the reuse log $Log_R(T)$ into classes with the following properties:

1. they tend to maximize intra-category similarity, and inter-category dissimilarity among reuse events.

2. similarities and dissimilarities are defined in economic terms, e.g., cost or reuse, cost of implementation.

7.3.1 A strategy for package selection

Figure 7.6 summarizes three major steps in the identification of packages using as input a history of reuse in the environment, Log_R.

Identification. Specification descriptors in the reuse log are clustered to outline a collection of possible packages. The identification step, a clustering task, depends on the attributes used to describe specifications, a measure of quality of the resulting clustering (Section 7.3.2) and background knowledge acquired from a domain expert.

Revision. The classification tree proposed by the identification step is revised by a domain expert. Discrepancies between the pattern of reuse encoded in the tree and the expert perceptions define opportunities for knowledge acquisition. The information acquired from the expert complements that acquired in previous domain analysis steps, and may result in new attributes, or additional background knowledge that are used to define a new pattern of reuse.

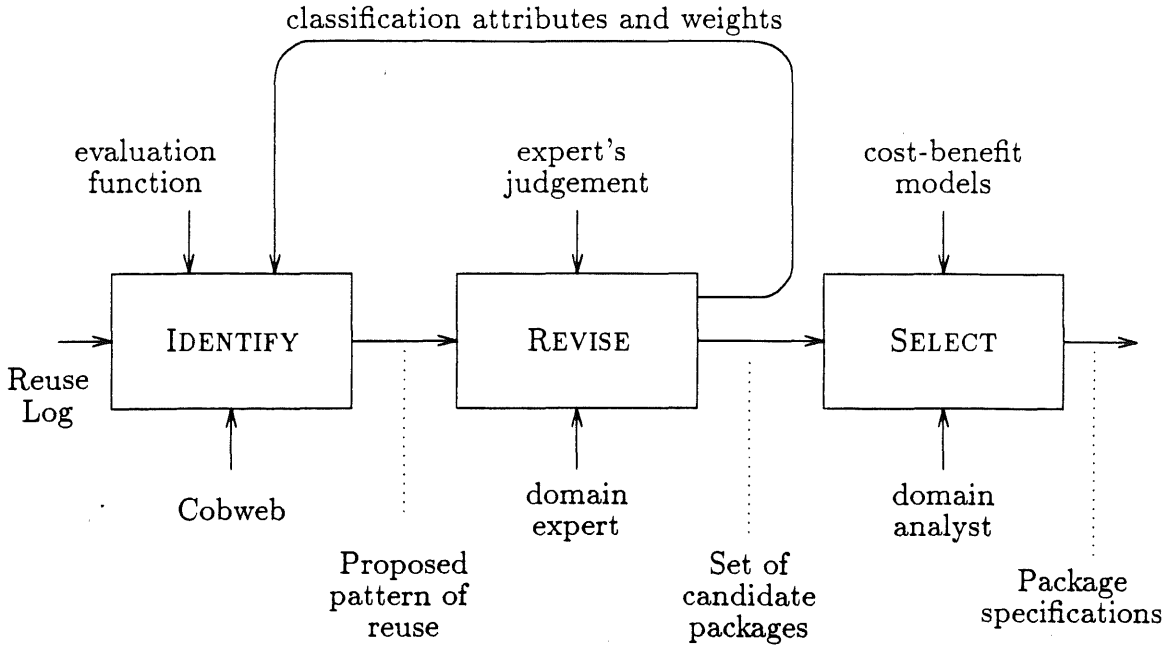


Figure 7.6: Steps in package identification

Selection. Some possible packages are selected for implementation based on trade-offs between the expected savings from future reuse and the estimated cost of implementation.

We now examine each of these stages in more detail.

7.3.2 Category Utility as a measure of clustering quality

The quality of the prediction depends on the historical data, Log_R. Based on the Scale of Reuse Hypothesis (Section 2.5) we assume that a new specification descriptor is generated with each new reuse event, and that large numbers of instances are available.

We have chosen a classification heuristic, Category Utility, CU, as a measure of clustering quality. Category Utility was introduced by Gluck and Corter [GC85] to predict the basic level in classification hierarchies. Hierarchical classification typically proceeds from the root of the hierarchy to the leaves. Empirical studies of humans suggest that they do not proceed in strict top down fashion but that they "enter" the hierarchy by some preferred classes, these constitute the so-called "basic level". A number of measures have been proposed to explain "basic level" phenomena. These measures are functions of the internal structure of categories and tend to reward intra-category similarity and inter-category dissimilarity.

Our interest on these measures is not their psychological significance but their utility to generate clusterings that maximize the *predictiveness* of categories by rewarding clusterings which maximize intra-category similarities and inter-category dissimilarity [Fis87a]. That is, given a classification hierarchy, and a description of an instance as a collection of attribute-value pairs: $[a_1 = v_{1j_1}, \ldots, a_k = ?, \ldots, a_n = v_{nj_n}]$, predict (or, infer) $a_k = v_{kj_k}$.

Category Utility measures the increase in the expected number of attribute values that can be correctly guessed, given knowledge that an object is in a category C_k over the expected number of values that can be correctly guessed without such knowledge. In preparation for presenting the definition of CU, we define a *collocation measure* for a value v_{ij} with respect to a category C_k [Jon83]:

collocation$(v_{ij}, C_k) = P(C_k|a_i = v_{ij})P(a_i = v_{ij}|C_k)$

the term $P(C_k|a_i = v_{ij})$ is called *cue validity*, and the term $P(a_i = v_{ij}|C_k)$ is called *category validity*. A category C_k, in a classification hierarchy that maximizes cue validity among ancestors and descendents favors inter-category dissimilarity or the predictiveness of values. Categories that maximize category utility, favor intra-category similarity or the predictability of values. The measure of collocation provides a balance between these two properties of a category definition. [GC85] propose a weighted collocation measure of a class C_k:

$$\sum_i \sum_j P(a_i = v_{ij})P(C_k|a_i = v_{ij})P(a_i = v_{ij}|C_k) = P(C_k) \sum_i \sum_j P(a_i = v_{ij}|C_k)^2$$

The term $P(a_i = v_{ij})$ weights the importance of individual values, thus assigning more importance to increase the class conditioned predictability and predictiveness of frequently occurring values. Category Utility is defined as:

$$P(C_k) \sum_i \sum_j P(a_i = v_{ij}|C_k)^2 - \sum_i \sum_j P(a_i = v_{ij})^2$$

where $P(C_k) \sum_i \sum_j P(a_i = v_{ij})^2$ is the expected number of correct guesses without knowledge of a partition, and $P(C_k) \sum_i \sum_j P(a_i = v_{ij}|C_k)^2$ is the expected number of correct guesses with knowledge of C_k. Given a collection of instances that can be partition in $C_1, C_2, \ldots, C_n$ classes, in many different ways, the chosen partition is the one that maximizes

$$CU(\{C_1, \ldots, C_n\}) = \frac{[\sum_{k=1}^{n} P(C_k) \sum_i \sum_j P(a_i=v_{ij}|C_k)^2] - P(C_k) \sum_i \sum_j P(a_i=v_{ij})^2}{n}$$

Fisher [Fis87b] has produced a clustering procedure, Cobweb, that, given a collection of instances, produces a hierarchical classification tree maximizing CU at each level. The classification tree is a probabilistic concept tree. Probabilistic concept trees are different from conventional classification trees in that classification information is not represented in logical form but as attribute-value probabilities. They differ from discrimination or decision trees in that information is not associated with the arcs of the tree but to the nodes.

Clustering based on CU falls in the category of what [Ste87] calls a Type-3

conceptual clustering process, that is, the measure of quality is attribute-based but no interpretation (or, concept characterization) is provided for the resulting clusters. We do not describe the Cobweb function here. The interested reader may refer to [Fis87a]. In the next section we illustrate a simple application of the procedure to a set of instances of UnixInit descriptors.

An example from the UnixInit domain

Figure 7.7 illustrates some of the 26 attributes used to describe applications in the UnixInit domain, and sample descriptors for a subset of applications collected from a real-life Unix environment [TA88]. Each row in the table represents an application descriptor. For the purpose of this example, the values of each attribute are restricted to nominal values "y" and "n", to indicate: "yes", the service represented by the attribute (column) is required by the subject, or "no" it is not required. For example, the first row, labelled `Q-086`, shows that the `.cshrc` file was specificed to provide the following services: Renaming (a_1), Maintaining records of user-system interactions (a_2), etc. while other services, such as Save command history after logout (a_3) or, Enable filename completion (a_{11}) were not specified. In general, a variety of nominal or real values were used in a study of patterns of reuse in the UnixInit domain [TA88] [AT88].

Figure 7.8 illustrates a tree produced by clustering the instances in Figure 7.7 using category utility as the evaluation function. Classes are represented in the figure as ovals, labelled with the number of instances classified under them. Rectangles represent classified instances. One particular node is expanded to illustrate the information available from a node in the probabilistic concept tree. For each attribute that is used to describe instances classified under the node, the probability of each of the values is available. For example, the probability that any of the instances classified under node 33 provides a Renaming service, a_1, is 1.0, (i.e, all of them do, the column corresponding to a_1 in Figure 7.7 contains all 'y').

7.3.3 Classification trees of possible packages

Patterns of reuse are represented as a probabilistic concept tree. The nodes in a probabilistic concept tree represent categories of specification descriptors. Each category is encoded as collections of attribute-value pairs with associated probabilities for each value (see, for example, the node expansions in Figures 7.8 and 7.9.) The definition of relevant attributes and values results from the process of domain analysis. Categories are defined using Category Utility to evaluate the quality of the partitions. The probabilistic concept tree defines categories that maximize intra-category similarity and inter-category dissimilarity. Moreover, category utility can be viewed as a measure that favors partitions that maximize inference abilities. In this, conceptual clustering based on CU differs from other approaches to conceptual

Sample of attributes used to classify UnixInit applications (attributes = service provided).

a_1 : RENAMING
a_2 : MAINTAINING-RECORD-COMMAND-HISTORY
a_3 : SAVING-COMMAND-HISTORY-AT-LOGOUT
a_4 : EXECUTING -COMMANDS-FROM-FILE
...
a_{16} : ACCESS-TO-BULLETIN-BOARDS
a_{17} : CHECKING-ON-MAIL
a_{18} : ENABLING-NOTIFY-END-OF-JOBS
...
a_{26} : CHANGING-AUTOMATIC-TIMING-OF-COMMANDS

Descriptor name	Attribute values																								
	1	2	3	4	5	6	7	8	9	10	12		14		16		18		20		22		24		26
	.	.	.	.	.	.	.	.	.	.	.		.		.		.		.		.		.		.
Q-068	y	y	n	n	n	n	n	y	y	y	n	n	y	n	n	n	n	n	n	n	n	n	n	n	n
Q-069	y	y	n	n	n	n	n	y	y	n	n	n	y	n	n	n	n	n	n	n	n	n	n	n	n
Q-071	y	y	n	y	n	n	n	y	y	n	n	n	y	n	n	n	n	n	n	n	n	n	n	n	n
Q-074	y	y	n	n	n	n	n	y	y	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n
Q-075	y	y	n	n	n	n	n	y	y	n	y	n	y	y	n	n	y	n	n	n	n	n	n	n	n
Q-076	y	y	n	n	n	n	n	y	y	n	n	y	n	n	y	y	y	n	n	n	n	n	n	n	n
Q-078	y	y	n	n	n	n	n	y	n	y	y	y	y	n	n	y	y	y	y	y	y	y	y	n	n
Q-079	y	y	y	y	n	n	n	y	n	n	n	n	y	y	n	y	n	n	n	n	n	n	y	n	n
Q-083	y	y	n	y	n	n	n	y	y	n	n	n	y	n	n	n	n	n	n	n	n	n	n	n	n
Q-085	y	y	n	n	n	n	n	y	y	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n
Q-089	y	y	n	n	n	n	n	y	y	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n
Q-090	y	y	n	n	n	n	n	y	y	n	n	n	n	n	n	y	n	n	n	n	n	n	n	n	n
Q-093	y	y	n	n	n	n	n	y	y	n	n	n	y	n	n	n	n	y	n	n	n	n	n	n	n
Q-094	y	y	y	y	n	n	n	y	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	y	n
Q-096	y	y	n	y	n	n	n	y	n	y	n	n	n	n	n	n	y	n	n	n	n	n	n	n	n
Q-100	y	y	n	n	n	n	n	y	y	n	y	n	y	n	n	n	n	n	n	n	n	n	n	n	n
Q-102	y	y	n	n	n	n	n	y	y	n	n	n	y	n	n	y	n	n	n	n	n	n	n	n	n
Q-107	y	y	n	n	n	n	n	y	y	n	y	n	n	n	n	n	n	n	n	n	n	n	n	n	n
Q-111	y	y	n	n	n	n	n	y	y	n	n	n	y	n	n	n	n	n	n	n	n	n	n	n	n
Q-113	y	y	n	n	n	n	n	y	y	n	y	n	n	n	n	y	n	n	n	n	n	n	n	n	n
Q-115	y	y	n	y	n	n	n	y	y	n	y	n	n	n	n	n	n	n	n	n	n	n	n	n	n

Figure 7.7: A sample of specification descriptors in UnixInit

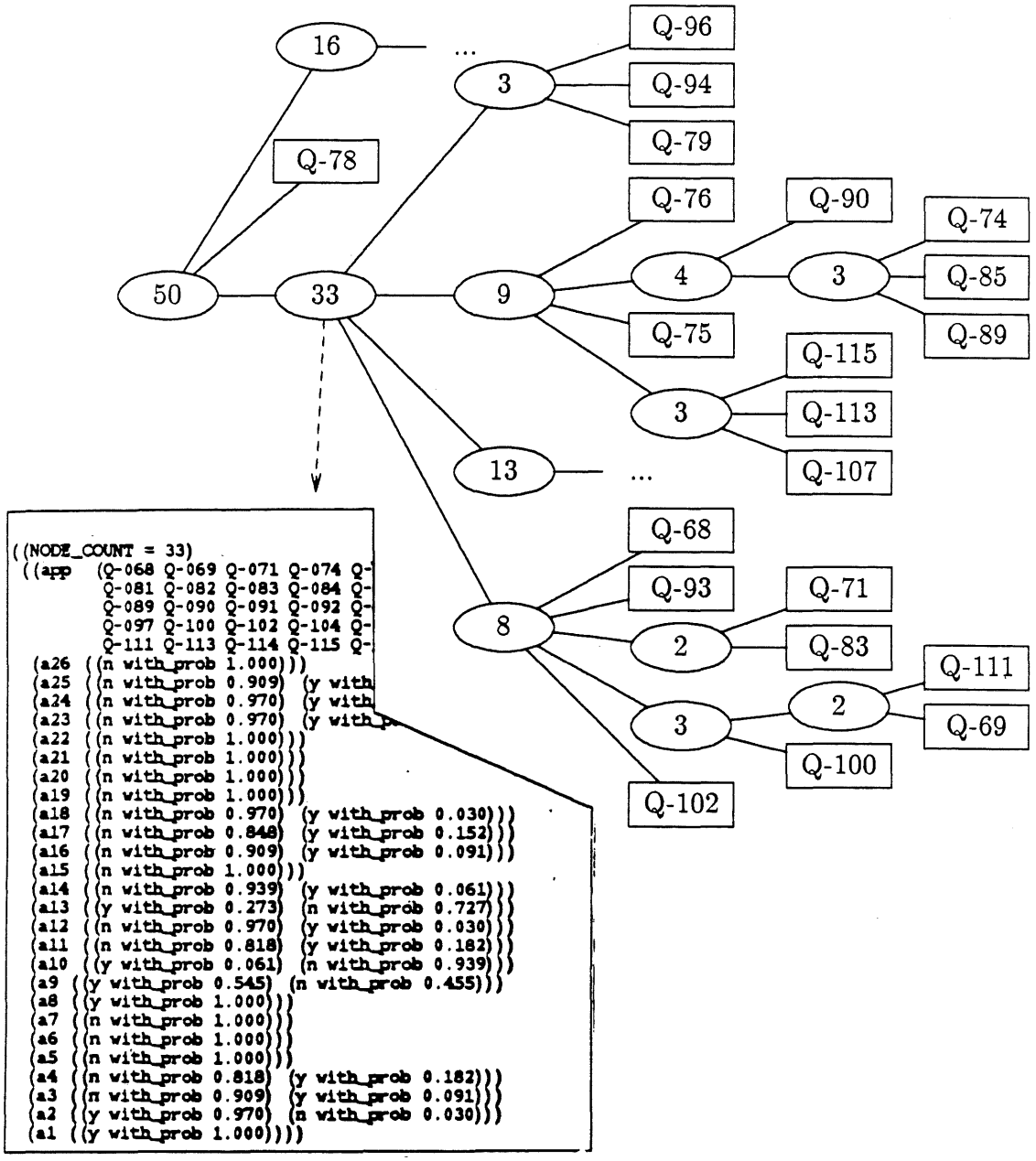


Figure 7.8: A CU classification of applications in UnixInit domain

clustering that stress "understandability" or "simplicity" of class description as a measure of quality of partitions [MS83] [SM86] [Ste87].

7.3.4 Introducing classification biases

The direct use of CU to instances in the UnixInit domain is appropriate because the addition or deletion of a service (in the case of "yes/no" values) has a relatively fixed cost. With few exceptions such as the "alias" command, Unix-shell commands can be added or deleted from the file without affecting other commands, i.e., the implementation of different services is relatively independent. The cost of adding a new service corresponds to the expense of researching in the Unix Manuals to find the right kinds of Unix-shell command to satisfy some initialization need and editing the `.cshrc` file.

In other problem domains, such as the Disk Drivers domain, all attributes are not created equal. Changes in the value of some attributes in a specification descriptor lead to drastically different implementations with very different associated costs. For example, in the implementation of disk drivers, the impact of having a Direct Memory Access (DMA) mechanism versus not having one, is greater than the impact of variations on the Number of Sectors per Track (NSPT). The latter can be abstracted away as a variable to be used to compute disk capacity or physical sector numbers. In the former case, large portions of the code of the driver are affected because the driver is directly responsible for the real-time transfer of disk sectors to and from the device. We elaborate on this example below.

The clustering procedure must have knowledge of the relative significance of attributes from the point of view of package construction. There exists a spectrum of alternatives for package implementation, ranging between:

- a "conjunctive" implementation: the package provides an implementation skeleton including only those features that are present in the members of a cluster with probability 1.0, and
- a "disjunctive" implementation: the package provides implementations for all values of the specification attribute.

Because features in the package specification have such a strong impact on the form of the implementation, in the first case the package is reduced to a bare-bones skeleton that provide little reuse leverage. In the second case, supporting that degree of genericity may result in an extremely complicated implementation, or in an "overloaded" implementation—an aggregation of practically independent implementations. Neither of these extreme alternatives is desirable.

Weights as an indication of "saliency"

To introduce the notion of saliency in the clustering process, we have experimented with a variation of category utility, *weighted* CU [AT88], defined as:

```
    ...
((NODE_COUNT = 4)
    (a12 ((caelus with_prob 0.25) (type-imi with_prob 0.25)
          (scsi with_prob 0.5)))
    (a11 ((no with_prob 0.75) (yes with_prob 0.25)))
    (a10 ((no with_prob 0.25) (yes with_prob 0.75)))
    (a9 ((yes with_prob 1.0)))
    (a8 ((reset with_prob 1.0)))
    (a7 ((0 with_prob 1.0)))
    (a6 ((256 with_prob 0.25) (512 with_prob 0.75)))
    (a5 ((400 with_prob 0.25) (350 with_prob 0.25) (150 with_prob 0.5)))
    (a4 ((32 with_prob 0.75) (24 with_prob 0.25)))
    (a3 ((unique-CAELUS with_prob 0.25)
        (unique-IMI with_prob 0.25) (unique-20MB  with_prob 0.5)))
    (a2 ((2 with_prob 0.25) (7 with_prob 0.25) (8 with_prob 0.5)))
    (a1 ((winchester with_prob 1.0))))
  NODE_CHILDREN_ARE
      (((DEF-CAELUS_Cartridge with_prob 0.25)
        (DEF-IMI_7710 with_prob 0.25)
        (DEF-SCSI_20_MB_Winchester with_prob 0.5)))
    (PARTITION_SCORE = 1.458333333333333)
    ...
```

Figure 7.9: Encoding of a descriptor for a possible package

$$P(C_k)\sum_i W_i \sum_j P(a_i = v_{ij}|C_k)^2 - \sum_i W_i \sum_j P(a_i = v_{ij})^2$$

where the weights W_i provide a measure of the relative importance of the attributes for the purposes of the classification. Fisher [Fis87a] proposes a similar device to identify the *saliency* of attributes when classifying instances using incomplete information. In that case, the values of weights W_i correspond to the probability of an attribute value being observed. We use of weights to:

1. Make explicit the expert's perception of the impact of variability on the structure of implementations. This perception is summarized in a Feature Domination (FD) Graph, which captures relations between kinds of information that are known to vary between instances of applications in the domain.

2. Relations in the FD graph are encoded as differences in *weights* to be used by the clustering algorithm.

3. Finally, the weights are scaled so that they properly affect the computation

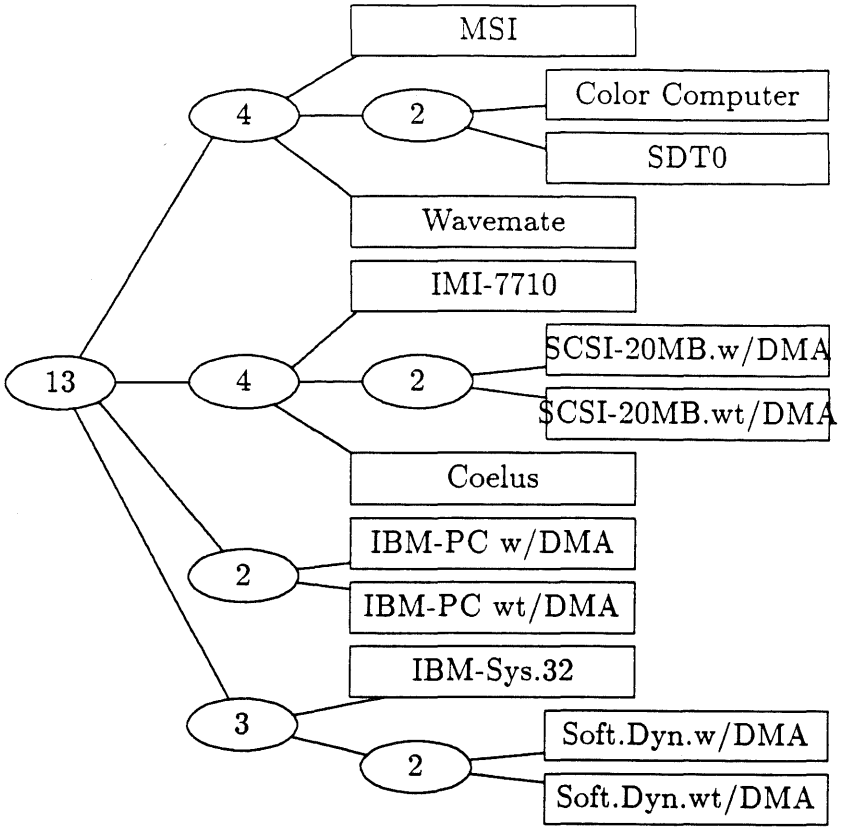


Figure 7.10: Disk drive descriptors clustered using CU

of the evaluation function.

The FD graph summarizes an aspect of the domain expert's experience in developing implementations in the domain that is not part of the MoD. Figure 7.11 shows a small FD graph in the disk drivers domain. It states that the difference in difficulty between implementing a driver using an "intelligent" controller vs. a "non-intelligent" controller is greater than that between supporting different types of controller interfaces, which is in turn greater than the difference in difficulty in implementing a driver with or without a DMA mechanism.

Heuristic. Choose as attributes for specification descriptors those features of the specifications that are known to vary and whose variations affect the structure of the implementations and indirectly, the cost of reuse.

Heuristic. Select those classification attributes for which changes in their values have a strong impact on the implementation of the applications. The larger the perceived difference between attribute values the

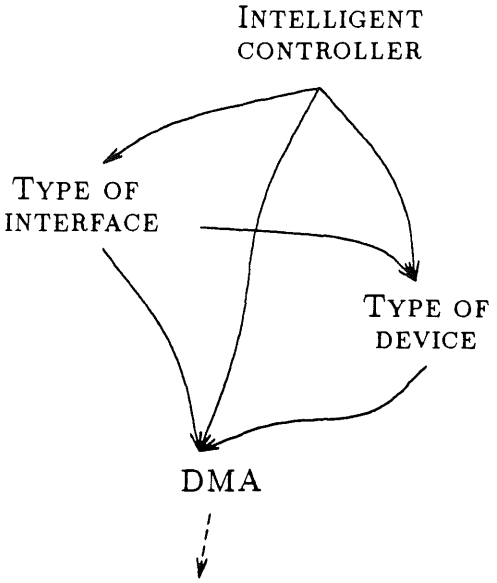


Figure 7.11: A feature dominance graph for disk drivers

more interesting the attribute. Produce a partial ordering according to their perceived effect.

In the case shown in Figure 7.11, the expert consulted judged that attributes Intelligence in the controller, Type of interface, Type of device and DMA were the ones whose values had a stronger impact on the structure of a disk driver. For instance, the effort required to implement a driver using a "non-intelligent" controller (as opposed to an "intelligent" controller) was estimated as 3:1 by our expert; the effort required to implement a driver if Direct Memory Access capability is not available was estimated as 5:1. Still the impact of intelligent/stup was considered much greater than that of DMA/not DMA.

Not all known attributes participate in a FD graph. For example, the graph in Figure 7.11 does not include attributes such as NSPT, or the number of R/W heads in the disk. Further, experts sometimes cannot establish dominance relations between *all* attributes in the graph. In those cases we assume that the differences are not relevant.

Heuristic. Assign weights to the attributes in the FD graph that are consistent with the relations in the graph.

Assigning weights to attributes is one approach to encoding the experts perceptions in a form that can be utilized by the clustering algorithm that produces the

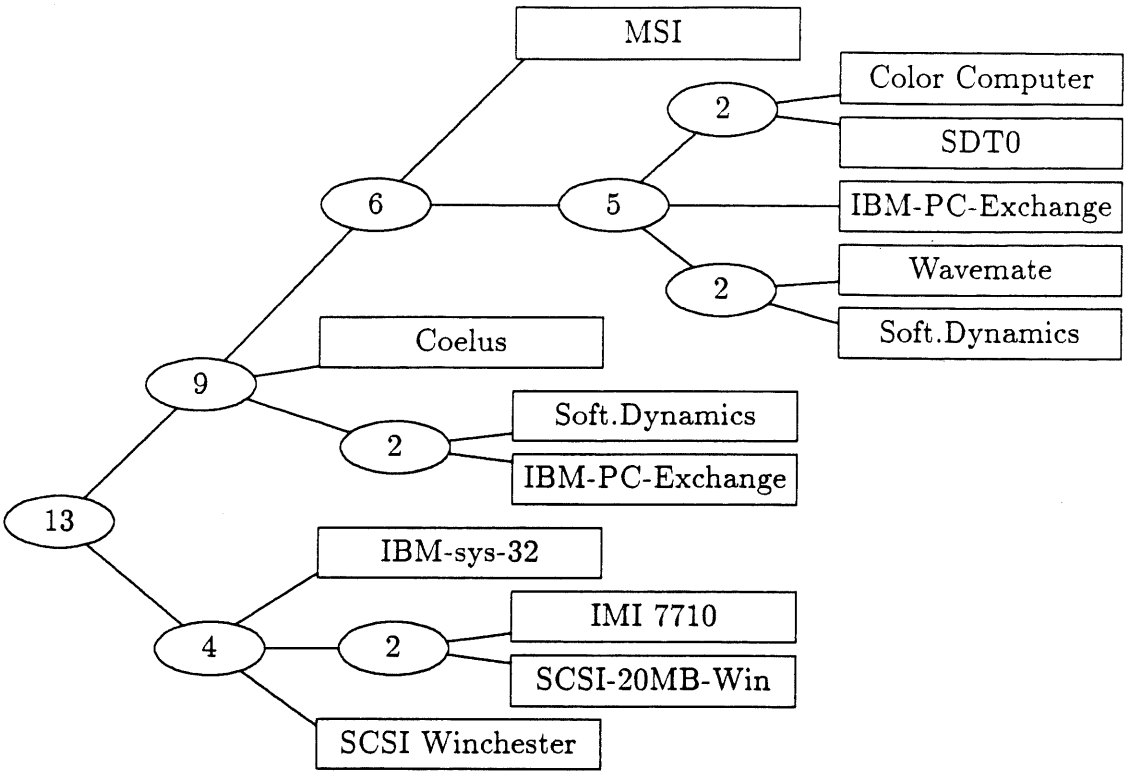


Figure 7.12: Disk drive descriptors clustered using a weighted CU

classification trees. For instance, in the case of Figure 7.11, the value 4 was assigned to the Intelligent Controller attribute, 3 to the DMA attribute, etc. It must be noticed that by assigning numeric weights we are introducing more constraints than those offered by the expert—a difference in numeric weights always exists even though an arrow may not be present in the graph.

> **Heuristic.** Scale the attribute weights to tune their effect on the clustering evaluation function.

In the case of CU a scaling factor equal to the number of attributes in the specification descriptor ensures the complete dominance of the attribute. For instance, if in the case of the FD graph in Figure 7.11, the weights 4, 3, 2 are scaled to 36, 24, 12, we obtain the classification in Figure 7.12.

If we compare the trees in Figures 7.12 and 7.10, the classification bias is apparent. For instance, the lowest node (with three children) in Figure 7.10 represents a class of drives that share the following values: floppy, two R/W heads, double density, 80 cylinders per unit, sector size 256B, numbering of tracks starts with 1, and numbering of sectors on second side resets to 0. From the point of view of package selection that clustering is not interesting if we consider the feature dominance

relations in Figure 7.11. On the other hand, the clustering presented in Figure 7.12 partitions the set of instances with respect ot the "intelligence" in the controller into two categories with nine and four members. These categories are in turn partitioned according to whether DMA capabilities are available. Thus, the clusters labelled 4 and 9 in the first level, suggest possible packages that capture the notions of "intelligence" in controllers while also maintaining a CU-based distribution of the remaining un-biased values.

7.3.5 Classification trees as an aid to knowledge elicitation

We use Cobweb as a "proposer" for classification trees. Depending on the characteristics of the domain, the attributes used to describe specifications and the weights assigned to them, the classification trees produced may or may not reflect the perceptions of the expert with respect to an ideal distribution of packages in the domain.

This is the reason for including a revision cycle. The classification tree is used to drive a process of knowledge elicitation. Each new classification tree submitted to the expert provides an opportunity to examine his/her past experience. From those recollections, valuable information on the impact of specification features on implementations is uncovered.

The expert must decide whether the clustering proposed matches his/her own experience by analyzing the distribution of values over partitions or, levels in the classification tree and between levels. Questions such as:

> "Would you build a generic package that implements this feature over this range of values?"
>
> "Would you rather build a package Intelligent-controller-driver with parameter DMA/not DMA, or a DMA-controller with parameter Intelligent/non-intelligent controller?"
>
> Or, if the expert is not happy with a partition at a given level in a tree, "Which values are distributed in a 'nonsensical' way and why would you do it differently?"

In summary, the purpose of the cycle of tree proposal and revision is to make systematic the process of identifying classification attributes and ranges of values and for making explicit the implementation issues that justify those choices. As the propose-revise cycle converges, each one of possible packages identified, is a candidate to be the root of what Parnas called a "program family" [Par72] [Par76].

7.4 Package selection

Summary. Given a pattern of reuse represented as a classification tree,

How can we identify a collection of packages to form an appropriate Reuse Working Set?

The motivation for developing a Reuse Working Set is economic, to improve the efficiency of the reuser. Hence, implementation and reuse costs are the key paramenter in the selection procedure. We introduce some working definitions for the cost of implementing packages and the cost of reusing them, and define an approach to package selection based on a cost-benefits analysis.

Given a pattern of reuse as a classification of events in $Log_R(T)$, we now demonstrate how an explicit representation for patterns of reuse can be used to answer packaging questions:

- Which packages should we build to attain a desired Average Adjustment Cost? and
- Given a budget for implementing a Reuse Working Set, which packages should we build?

In preparation for answering these questions we propose some working definitions for costs. These definitions can be refined and extended in a number of ways. Our purpose here is only to illustrate the kinds of analyses that are possible by using explicit descriptions of patterns of reuse. A detailed example of can be found in [TA88].

7.4.1 Packaging costs—working definitions

Given a specification Q, represented as a set of attributes-value pairs $[a_1 = v1, \ldots, a_k = v_k]$, the cost of adjusting a package P to fit the needs of a specification Q is defined as the cost of adapting the implementation of Π so that it provides each of the services $a_i = v_i$ required by the customer. Adjusting a package can be a straightforward process—e.g., the instantiation of parameters; it may require changes to a package implementation operators, or in some cases, it may require the construction from scratch of some subcomponent not foreseen at domain analysis time. For example, in the domain of disk drivers, as new controller or drive technologies are introduced in the market, the parts in a drivers' implementation that communicate with those interfaces must be redesigned.

We define the *package adjustment cost* for a package P relative to a customer specification Q and a reuser R as the sum of the cost of ensuring that each feature in Q is implemented by P.

$$AdjCost_R(Q, P) = \sum_k Ensure - implementation(Q(a_k), P)$$

For the sake of simplicity in the presentation we will ignore the impact of implementation dependencies on cost. If a collection of packages is available, the reuse cost for a specification Q is computed as the minimum over the members of the collection. The package with minimum adjustment cost becomes the *best candidate* for reuse. We define the *reuse cost for a specification* Q relative to a reuser R and a collection of packages $\{P_j\}$ as:

$$AvgReuseCost(Q, \{P_j\}, R) = min_{\{P_j\}} AdjCost_R(Q, P_j)$$

The *average adjustment* cost for a collection of n customer specifications $\{Q_i\}$, relative to a reuser R and a collection of packages $\{P_j\}$ is defined as:

$$ARC(\{Q_i\}, \{P_j\}, R) = \frac{\sum_i AvgReuseCost(Q_i, P_j, R)}{n}$$

Given a class of specifications $\{Q_i\}$, represented as attribute-value pairs, the cost of implementing a package, $P = \Pi(Q_i)$, depends on two thresholds: an implementation threshold which selects attributes and a genericity threshold which selects attribute values.

The *implementation threshold, IT*, is a number in the range [0..1], that establishes which attribute defined by the set of specifications must be implemented. Only those attributes that have non-nil values, $a_i = v_{ij}$, with probability higher than IT are implemented.

The *genericity threshold, GT*, is a number in the range [0..1] that establishes which attribute values defined by the set of specifications must be implemented. Only those attributes that have values $a_i = v_{ij}$ with probability higher than GT are implemented. Other factors could be used to define genericity thresholds; for instance, the cost of implementing particular values for attributes in possible package descriptors.

$GT = 1$ corresponds to the conjunctive case: only those features shared by all specifications in the class are implemented in the package. The conjunctive case results in "skeletal" packages that must be filled-in to account for non-shared features. For example, a generic package for disk drivers would contain very little besides an abstract specification of entry points and skeletons for the major data structures. Packages for floppy disks would include code for performing "seek" operations on disks or "verify" operations after "write" that would not be present in conjunctive packages for Winchester-type disks.

$GT = 0$ corresponds to the disjunctive case: every attribute-value present in a specification descriptor is implemented by the package. This could lead to massive packages that are expensive to construct or reuse.

We illustrate the use of IT and GT with an example. If attribute a_2 in the description of specifications in $\{Q_i\}$ has a non-nil value 70 percent of the time, and the non-nil value distribution is:

$P(a_2 = v_{21}) = .05$
$P(a_2 = v_{22}) = .35$
$P(a_2 = v_{23}) = .30$
$P(a_2 = nil) = .30$

Then, if we adopt $IT = .5$, a package P would implement some form of the service described by a_2, given that the probability of non-nil values is, $(.05 + .35 + .30) = .7 \geq IT = .5$. Which forms of a_2 would be implemented depends on the genericity threshold. If, for example, $GT = .4$, values v_{22} and v_{23} will be supported, as the probability of $a_2 = v_{22}$ is $(.35/.7) = .5 \geq .4$ and the probability of $a_2 = v_{23}$ is $.428 \geq .4$

> **Heuristic.** If the genericity threshold excludes all the values, the probability with the highest value should be implemented.

Given a class of specifications $\{Q_i\}$, the cost of implementing a package to realize them, $P_i = \Pi(Q_i)$ may be approximated using two cost components: a *basic implementation cost*—for implementing those services selected by the implementation threshold IC—and an estimated *genericity cost*— the increase over the basic implementation cost due to the need to implement multiple values of some feature, as selected using the threshold GT.

7.4.2 Package selection based on average reuse cost

We formulate the problem as: Given a pattern of reuse and a desired average cost of reuse cost, which packages should be built?

We assume that past reuse history is a predictive of future patterns of reuse, that patterns of reuse are defined in terms of classification trees as described above. Thus, the set $\{Q_i\}$ is defined by the set of specifications in $Log_R(T)$.

An iterative procedure for determining which packages to implement starts by setting "reasonably high" values for IT, GT. High values as an initial condition results in package definitions that are overly restrictive in terms of the set of features and features values supported. This initial condition is based on the assumption that the richer the packages the more expensive it is to build them.

1. For each class C_i in the classification tree, compute the average reuse cost for specifications classified under the class with respect to the possible package, associated with the class, $\Pi_c(C_i)$.

2. Repeat:

   (a) Select a set of C_i covering all the classified specifications and such that $\Pi_c(C_i)$ has the lowest Reuse Cost.

(b) Compute a general Average Reuse Cost by weighting each class using the probability of the class.

$$\sum_i P(C_i)\ RC(Ci, \Pi_c(C_i))$$

3. If the general average reuse cost is greater than the desired average cost in all cases considered, restrict the desired coverage by lowering the thresholds IT, GT and go back to step 2. Otherwise, suggest the set of possible packages as possibles for implementation.

Intuitively, each time the thresholds IT, GT are lowered, the packages identified are more generic, or support more services. In this example, we ignore the cost of actually developing the packages; in the next Section we present another method based on the opposite assumption, there exists a fixed budget for the development of packages.

7.4.3 Package selection based on construction budget requirement

We formulate the problem as: Given a fixed budget for constructing a Reuse Working Set, which set of packages should be built to obtain a good average reuse cost? (We say good, and not best, because the data considered by the procedures is based on cost-estimates and expert perceptions which are not precise.)

The following heuristic is used to answer the question: avoid investing on constructing packages that satisfy "rare" combinations of requirements and focus on constructing packages to support the most common and frequent specifications can be realized. A method for determining which packages to implement follows:
Set the initial values of thresholds IT, GT to a "reasonably low" value. (This will result in an adequate but overly expensive Working Set)

1. For each class C_i in the classification tree, estimate the Implementation Cost of $\Pi_c(C_i)$,

2. Select a set of C_i which covers all the classified specifications and whose $\Pi_c(C_i)$ have the lowest Implementation Costs.

3. Add up the estimated Implementation Costs. If the total is greater than the desired average, increase the values of IT or GT, and go back to step 2. Otherwise, suggest the set of possible packages as candidates for implementation.

Intuitively, this procedures identifies in each cycle those cases for which there is low demand and high implementation costs. To keep package development costs under budget, support for the most rare and expensive cases must be dropped. For instance, in the disk drivers domain, a suggestion validated by real-life data is to drop support for diskettes having `attribute-9: uniform density = no` (see Figure 7.5)—these are a relatively rare case of double density diskettes with a single density first track.

Summary—towards an objective basis for package design

The motivation for the studies reported in this Chapter is to provide an objective basis for package selection.

In the case of Ada, Booch [Boo83] suggests that Ada packages be used to collect named collections of declarations, groups of related program units, abstract data types and abstract state machines. In the Ada Reuse Guidelines of the Alvey Eclipse project, we read: "It is useful to introduce the notion of a 'component library'. Our view of an Ada component library is a collection of packages and subprograms, each with a distinct name. ... Some components will be closely related, in terms of the abstraction modelled or functionality provided; the organization of the library should support and reflect such groupings" [BEG86].

In the absence of universal rules, the concepts and methods proposed provide a framework for decision making based on explicit treatment of environment-specific patterns of reuse and cost-benefits analyses.

We are aware of the limitations of weighted category utility as a measure of clustering quality. In principle, richer evaluation functions that could account for the probability of combined values across attributes would be desirable. In practice, weighted CU has served us well because an expert's intuitions about the difficulty of implementations are rather fuzzy and limit the accuracy of the results in a way that cannot be improved by finer evaluation functions. (In [AT88] we summarize the conclusions drawn from our use of the Cobweb function.)

7.4.4 Shifting patterns of reuse

Replacement strategies in the Reuse Working Set

From the viewpoint of the analogy between the problems of efficiency in first-order reuse and memory management, the methods discussed in this Chapter define the selection and replacement strategies for packages in a reuse working set.

We have applied the clustering approach to defining patterns of reuse both incrementally and non-incrementally. The incremental version is cumulative, in that the classification tree evolves as new specifications are submitted to the reuser, but old specifications are not "forgotten". Other clustering models have been demonstrated that have the capability of forgetting concepts that are not used for a period of time, e.g., [Kol83] [Leb83].

To fully capture shifts in patterns of reuse, a more strict implementation of working sets must be applied, the specifications would have to be taken from $Log_R(T)$ for a some time period T fixed, instead of using always the value of the log from the time the reuse infrastructure was put into use. A literal implementation of the Working Set concept requires that a new classification be made periodically, including all specifications submitted to the reuser over the past period of time T, or using the last n specifications, where n could be computed using some function

of T. This task can be better performed using the non-incremental version of the clustering algorithm.

Patterns of reuse as a basis for organizing library of components

Patterns of reuse can be derived from the analysis of the queries made to a library of of domain-specific components. A clustering system watching over the stream of queries to the library can mechanically produce (and evolve) descriptions of patterns in the environment of the library.

Different evaluation functions can be used in parallel to identify patterns that reflect different viewpoints. The resulting clusterings provide a basis for the rational design and optimization of indices and packages:

1. Indices to the library can be evolved match relevant patterns in the environment to facilitate access to package definitions that are most "predictive" of user needs, or that have the lowest expected reuse cost. This is a typical problem in self-organizing data-structures.

2. Explicitly represented changes in patterns of reuse provides an objective basis for a librarian to assess the adequacy of the contents and organization of the library. Together with costs-benefits analyses they allow to evolve the library to provide more cost-effective services to customers. Using the memory management analogy, to design package replacement strategies that evolve a reuse working set.

We do not discuss here the use of patterns of reuse in the design of adaptable library structures.

7.5 Summary

In this Chapter we have discussed a how to improve the efficiency of reusers through the packaging of components. In particular, we have proposed

- a representation for describing patterns of reuse in an environment to support the identification and selection of reusable packages;

- a technique to elicit domain-specific heuristics for package design from experts based on the generation and revision of classification trees; and,

- methods for selecting packages based on cost-benefit analyses.

Chapter 8

Conclusions

The infrastructural aspect of reusability

Most research on reuse in software construction to date has been based on the assumption that reusable information exists in appropriate form. The focus of mainline research on reuse has been on the operational aspect of reuse, on how to make reuse practical.

There exists an infrastructural aspect: a key precondition to reuse is the availability of quality, reusable information.

The infrastructural problem involves: 1) the identification and acquisition of reusable information, 2) the modularization and organization of the information, and 3) the representation in a form to fit the technology of the target reusers.

We have argued, by analogy with software engineering, for a systematic, *domain engineering* attack on the problem of generating and evolving *reuse infrastructures*. In this dissertation we have focused on *domain analysis*, the front-end activity in the domain engineering process.

An engineering approach to domain analysis

We have discussed two views of domain analysis. "Pure" domain analysis falls in the category of a theory formation activity. From that viewpoint, only an (inspired!) expert in a problem domain can qualify as a domain analyst.

A similar viewpoint could be adopted on software construction, a process that is also complex and poorly understood. As a reaction, the *software engineering* strategy has been to decompose the task into a system of activities and workproducts that isolate and constrain the execution of the "creative" steps. While we are bound by Boehm's observation that variations between people account for the biggest difference in productivity—"Do everything you can to get the best people working on *your* project" [Boe87]—on the average the engineering approach to software construction can be credited with a rationalization of the process leading to improvements in quality and productivity.

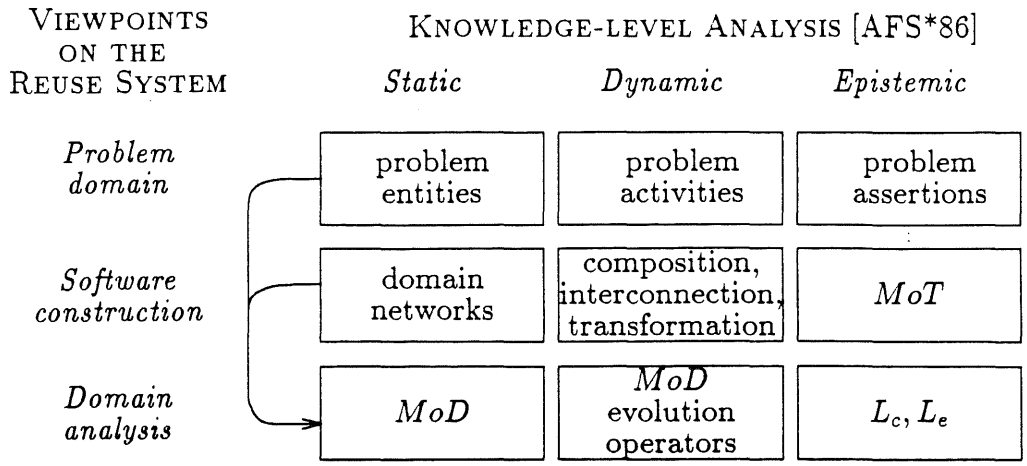


Figure 8.1: A map of the conceptual structures proposed

This is the practical effect that we need to re-create in the process of building reuse infrastructures. Towards that end, we have adopted an engineering perspective on the acquisition of reusable information. It does not make economic sense to re-create the expert's hard-won insights and understanding of a problem domain. Rather, we must take advantage of existing knowledge, acquire it and represent it in a form that can be communicated, evolved, and cast into different languages to be integrated into actual reuse technologies.

Results

The dissertation formulates a framework for the acquisition and evolution of domain-specific information (Figure 8.1). The results of this work include:

- A domain engineering strategy for the generation of reuse infrastructures (Section 3.4).
- An operational definition of practical domain analysis as a process of enhancing the competence and efficiency of a reuse system (Section 3.5).
- An analysis of reusers from a systems viewpoint (Sections 3.1 and 3.2). A classification of reuse systems based on the form of reuse they realize (Section 4.2), and its implications for domains analysis (Section 4.4).
- A language, MoDL for representing domain-specific information of the first-order (i.e., the kind of information needed to develop software components) (Chapter 5).

- A method for enhancing the competence of a reuse system through the evolution of a model of a first-order (Chapter 6). The method includes techniques for information acquisition, analysis and representation based on the MoD structure.

- Methods for the explicit modelling of patterns of reuse in the environment of a reuser. Patterns of reuse provide an objective basis for designing packages of reusable components and for organizing libraries of domain-specific components (Chapter 7).

The methods proposed meet the requirements outlined in Section 3.6. They are systematic, support the incremental evolution of the reuse infrastructures, and are independent of the technology of the target reuser. The scope of the methods is defined by the MoT and the MoD representation language adopted.

Future Work

When outlining the organization of the dissertation (Figure 1.4) we distinguished between three levels: the domain engineering framework, the domain analysis methods, and the level of reuse in software construction.

At the Domain Engineering level, there is a need for a comprehensive *theory of reuse systems* to provide a foundation to the development of reuse infrastructures. In particular, we must categorize reuse mechanisms from the point of view of their information-processing structure and the types and roles of the information being reused.

Rigorous approaches to characterizing the semantics of problem domains and domain interconnections are needed, together with empirical studies of the architectural and representation properties of domain networks to support domain engineering-in-the-large.

At the domain analysis level, a major thrust in domain analysis research must be towards *acquiring empirical data* on the technical and economic aspects of reuse infrastructures.

Analyses on real-life domains must be performed to gain a better understanding of the nature of the process, and the difficulties in scaling-up the application of the methods for domain analysis.

Costing data must be collected to model the economics of infrastructure development and evolution. Data on the performance aspects of the domain analysis process itself is needed to develop tactics appropriate to specific environments and stages in the process.

Methods for information acquisition and verification customized to various models of the reuse task need be explored. We have proposed a generic method for first-order reusers. The general domain engineering framework should now be applied to higher-order, or more constrained forms of reuse. It may be of practical interest

to developed specialized, "stronger" versions of the methods that exploit aspects of particular reuser architectures.

Designing libraries of components is a difficult problem. Domain analysis methods together with the identification of patterns of reuse provide one possible attack on the library design problem. Research on the *characterization of patterns of reuse* must be pursued; in particular, we need to acquire experience with different measures of clustering quality. Our experiences with category utility is only an initial step in this direction.

Finally, *a substantial amount of tool-support is needed* to assist the domain analyst. Our experience shows that the volume and the complexity of interdependencies between items of information are major obstacles to applying these methods to "bigger" domains. Models of domains must be *grown* in an environment that facilitates the management of information. An implementation of a MoDL object-base is the next natural step toward making the domain analysis methods practical. Software to assist in the following areas is needed:

1. An object-base for persistent storage of information and associative retrieval of object definitions,
2. maintenance of dependencies and identification of paths of (potential) propagation of change,
3. (syntactic-level) verification of MoD integrity constraints, and
4. assistance in the classification of concept definitions.

Major themes

The research discussed in this document embodies five major themes. First, given our limited knowledge of the nature of reuse processes, domain-specific approaches appear to be the most promising avenue to reusability in the short-term.

Second, reuse systems *are* systems. We know little about the conceptual structure and the performance aspects of reuse systems. This should be a focus of theoretical research in reusability.

Third, the acquisition of reusable resources should be considered within the larger context of a domain engineering framework analogous to the framework developed by the software engineering community to support the systematic development and evolution of software.

Fourth, domain analysis should be viewed as an engineering activity. Practical domain analysis methods should be designed to support the systematic approximation to performance goals under explicit economic constraints.

Finally, reusability is a ubiquitous process. Instances of the domain analysis problem surface in many fields besides software construction—requirements analysis in software engineering, knowledge acquisition for expert systems, conceptual

modelling in database design. A discipline of domain analysis should be open to contributions from all fields.

Bibliography

[Abb87] R. Abbot. Knowledge Abstraction. *Communications of ACM*, 30(8):664–671, August 1987.

[ABFP85] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon. Maintenance and Porting of Software by Design Recovery. In *Proc. Conference on Software Maintenance*, pages 42–49. IEEE Computer Society Press, November 1985.

[ABFP86] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon. TMM: Software Maintenance by Transformation. *IEEE Software*, 3(3):17–39, May 1986.

[AFS+86] J. Alexander, M. Freiling, S. Shulman, J. Staley, S. Rehfuss, and S. Messick. Knowledge Level Engineering: Ontological Analysis. In *Proc. AAAI - 86, II*, pages 963–968, 1986.

[AGB+86] A. Ambler, D. Good, J. Browne, W. Biurger, R. Cohen, C. Hoch, and R. Wells. Gypsy: A language for specification and implementation of verifiable programs. In *Software Specification Techniques*, pages 421–440. Addison-Wesley, 1986.

[Ame87] Scientific American. Technology: Faces, Couches, Cats..., November 1987.

[And77] J. Anderson. Induction of Augmented Transition Networks. *Cognitive Science*, 1:125–157, 1977.

[App83] D. Appelt. Telegram: A Grammar Formalism for Language Planning. In *8th. Internatinal Joint Conference on Artificial Intelligence*, pages 594–599. IJCAI, August 1983.

[Ara88a] G. Arango. Evaluation of a Reuse-based Software Construction Technology. In *Proc. Second IEE/BCS Conference on Software Engineering 88.* The British Computer Society, July 1988.

[Ara88b] G. Arango. Notes from an Analysis of the SDOS Disk Drivers. Technical Report ASE-RTM-84, ASE, ICS, University of California, Irvine, July 1988.

[AS85] B. Adelson and E. Soloway. The Role of Domain Experience in Software Design. *IEEE Transactions on Software Engineering*, SE-11(11):1351–1360, November 1985.

[AT81] Ashton-Tate. Dbase II Users Manual. Technical report, Ashton-Tate, 1981.

[AT88] G. Arango and E. Teratsuji. Notes on the Application of the COBWEB Clustering Function to the Identification of Patterns of Reuse. Technical Report ASE-RTM-87, ASE, ICS, University of California, Irvine, July 1988.

[B+86] B. Burton et al. A Practical Approach to Ada Reusability. Technical report, Intermerics Inc., Aerospace Systems Group, Huntington Beach, CA, 1986.

[Bar85] D. Barstow. Domain-specific Automatic Programming. *IEEE Transactions on Software Engineering*, SE-11(11):1321–1336, November 1985.

[Bar88] D. Barstow. Automatic Programming for Streams II. In *Proc. 10th. International Conference on Software Engineering*, pages 439–447. IEEE Computer Society Press, April 1988.

[Bax87a] I. Baxter. Derivation Histories. Technical Report ASE-RTP-077, Advanced Software Engineering Project, ICS, U. of California, Irvine, March 1987.

[Bax87b] I. Baxter. Domain Connection Discovery. Technical Report ASE-RTP-67, ICS, University of California, Irvine, March 1987.

[Bax87c] I. Baxter. PCL: A Production Control Language. Technical Report ASE-RTP-80, Advanced Software Engineering Project, ICS, U. of California, Irvine, November 1987.

[Bax87d] I. Baxter. Propagation of Change in Transformational Systems. Technical Report ASE-RTP-76, Advanced Software Engineering Project, ICS, U. of California, Irvine, 1987.

[Bax87e] I. Baxter. Reusing Design Histories via Transformational Systems. Technical Report ASE-RTP-79, Advanced Software Engineering Project, ICS, U. of California, Irvine, November 1987.

[BD77] R. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BDGP87] B. Barnes, T. Durek, J. Gaffney, and A. Pyster. A framework and economic foundation for software reuse. Technical Report SPC-TN-87-011, Software Productivity Consortium, Reston, VA, June 1987.

[BEG86] M. Bott, A. Elliott, and R. Gautier. Ada reuse guidelines – report. Technical Report ECLIPSE/REUSE/DST/ADA-GUIDE/RP, Alvey Eclipse Programme, February 1986.

[Ben84] J. Bennet. ROGET: Acquiring the Conceptual Structure of a Diagnostic Expert System. In *IEEE Proc. Workshop on Principles of Knowledge Based Systems*, pages 75–82. Denver, Co, December 1984.

[Ben86] J. Bentley. Little Languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[Ber87] E. Berard. Software Reusability Cannot Be Considered In A Vacuum. In *Proc. Compcon 1987*, pages 390–391. IEEE Computer Society Press, 1987.

[Big88] T. Biggerstaff. *Reusable Software.* Addison Wesley, Reading, MA, 1988.

[BMS84] M. Brodie, J. Mylopoulos, and J. Schmidt. *On Conceptual Modelling.* Springer Verlag, New York, 1984.

[BMW84] A. Borgida, J. Mylopoulos, and H. Wong. Generalization/specialization as a basis for software specification. In *On Conceptual Modelling*, pages 87–117. Springer Verlag, New York, 1984.

[Boe87] B. Boehm. Industrial software metrics top 10 list. *IEEE Software*, 4(5):84–85, September 1987.

[Boo83] G. Booch. *Software Engineering with Ada.* Benjamin/Cummings Publ. Co., Menlo Park, CA, 1983.

[Bor85] A. Borgida. Features of Languages for the Development of Information Systems at the Conceptual Level. *IEEE Software*, 2(1):63–73, January 1985.

[BR87] T. Biggerstaff and C. Richter. Reusability—Framework, Assessment, and Directions. *IEEE Software*, 4(2):41–49, March 1987.

[Cam84] Camp. Common Ada Missile Packages. Technical Report BR-14646-1, Raytheon Company, Missile Systems Division, 1984.

[Cam87] Camp. Common Ada Missile Packages. Technical Report —Product presentation literature, Mc. Donnell Douglas Corp., 1987.

[Car83] J. Carbonell. Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In *Machine Learning, Vol. 1*, pages 137–161. Morgan Kaufmann, Los Altos, CA, 1983.

[CF82] P. Cohen and E. Feigenbaum. *The Handbook of Artificial Intelligence, Vol. III.* W. Kaufmann, Los Altos, CA, 1982.

[CG82] A. Cardenas and W. Grafton. Challenges and Requirements for New Application Generators. In *AFIPS Conference Proceedings, 1982 National Computer Conference*, pages 341–349. AFIPS Press, June 1982.

[Cha83] B. Chandrasekaran. Towards a Taxonomy of Problem Solving Types. *AI Magazine*, 4(1):9–17, Winter/Spring 1983.

[Cle87] J. Cleaveland. Building Application Generators. Presentation at the Dept. of Computer Science, U. of California, Irvine, April 1987.

[Cle88] J. Cleaveland. Building Application Generators. *IEEE Software*, 5(6):25–33, July 1988.

[CLF88] CLF. CLF Manual. Technical Report CLF Project, USC/Information Sciences Institute, Marina del Rey, CA, January 1988.

[Cox86] B. Cox. *Object Oriented Programming.* Addison Wesley, 1986.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Dal76] P. Dale. *Language Development—Structure and Function.* Holt Rinehart and Winston, New York, 1976.

[Dat82] C. Date. *An Introduction to Database Systems, Volume 1.* Addison Wesley, Reading, MA, 1982.

[DB] T. Dietterich and J. Bennet. Improving Machine Learning Terminology: Operationality. Dept. of Computer Science, Oregon State University, Corvallis, Oregon, and Tecknowledge, Inc., Palo Alto, CA., 1987.

[Den68] P. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11:323–333, May 1968.

[Der83] N. Dershowitz. *The Evolution of Programs.* Birkhauser, Boston, MA, 1983.

[DGKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Mélèse. Document Structure and Modularity in Mentor. In *Proc. Software Engineering Symposium on Practical Software Development Environments*, pages 141–148. ACM SIGPLAN, Vol. 19, No. 5, May 1984.

[Dil81] D. Dilworth. *Scientific Progress.* D. Reidel Publishing Co., Boston, 1981.

[dJ80] S. Peter de Jong. The System for Business Automation (SBA): A Unified Application Development System. In *Information Processing 80*, pages 469–474. North Holland Publishing Company, October 1980.

[DK76] F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, SE-2(1):80–86, June 1976.

[dK84] J. de Kleer. Choices Without Backtracking. In *Proc. AAAI National Conference on AI*, pages 79–85. AAAI, 1984.

[DM86] G. DeJong and R. Mooney. Explanation Based Learning. A Differentiating View. Technical report, Computer Science Laboratory, U. of Illinois, 1986.

[DoD86] DoD. Reusability Guidebook, (draft). In *STARS - 4th. Applications Workshop, San Diego*, September 1986.

[Doy79] J. Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12:231–272, 1979.

[DS87] S. Dietzen and W. Scherlis. Analogy in Pogram Development. In *The Role of Language in Problem Soving 2*, pages 95–118. North Holland, 1987.

[EM86] L. Eshelman and J. McDermott. MOLE: A Knowledge Aquisition Tool That Uses its Head. In *Proc. AAAI-86, Vol. Engineering*, pages 950–955, 1986.

[FAM+85] M. Freiling, J. Alexander, S. Messick, S. Rehfuss, and S. Shulman. Starting a Knowledge Engineering Project: A Step-by-Step Approach. *The AI Magazine*, 5(3):150–164, Fall 1985.

[Fei77] E. Feigenbaum. The art of artificial intelligence: Themes and case studies of knowledge engineering. In *IJCAI-5*, pages 1014–1029, 1977.

[Fic85] S. Fickas. A Knowledge Based Approach to Specification Acquisition and Construction. Technical Report CIS-TR-85-13, U. of Oregon, 1985.

[Fic87a] S. Fickas. Automating the Specification Process. Technical Report CIS-TR-87-05, U. of Oregon, December 1987.

[Fic87b] S. Fickas. Problem Acquisition in Software Analysis: A Preliminary Study. Technical Report CIS-TR-87-04, U. of Oregon, August 1987.

[Fis87a] D. Fisher. Knowledge Acquisition Via Incremental Conceptual Clustering. Technical Report PhD Thesis, U. of California, Irvine, 1987.

[Fis87b] D. Fisher. Knowledge Acquisition Via Incremental Conceptual Clustering. *Machine Leraning*, 2:139–172, 1987.

[Fre77] G. Frege. Compound Thoughts. In *Logical Investigations*, pages 55–78. Basil Blackwell, Oxford, 1977.

[Fre80] P. Freeman. Reusable Software Engineering: A statement of long-range research objectives. Technical Report UCI-ICS-TR159, Dept. of Information and Computer Science, U. of California, Irvine, November 1980.

[Fre83] Peter Freeman. Reusable Software Engineering: Concepts and Research Directions. In *Proceedings of the Workshop on Reusability in Programming*, pages 129–137. ITT, September 1983.

[Fre84] P. Freeman. A Conceptual Analysis of the Draco Approach to Constructing Software Systems. *Trans. on Software Engineering*, SE-13(7):830–844, July 1984.

[Fre87a] P. Freeman. *Software Perspectives.* Addison Wesley, Reading, MA, 1987.

[Fre87b] P. Freeman. *Software Reusability.* IEEE-Computer Society, March, 1987.

[Gau86a] B. Gautier. Ada Reuse Guidelines. Technical Report ECLIPSE /REUSE /DST /ADA-GUIDE /RP, Computer Science Dept., University College of Wales, Penglais, Aberystwyth, UK, February 1986.

[Gau86b] R. Gautier. A language for describing ada software components. Technical report, Computer Science Dept., University College of Wales, Penglais, Aberystwyth, UK, 1986.

[GC85] M. Gluck and J. Corter. Information, uncertainty and the utility of categories. In *Proc. 7th. Annual Conf. of the Cognitive Science Society*, pages 283–287. U. of California, Irvine, August 1985.

[Ger83] S. Gerhart. Reusability Lessons from Verification Technology. In *ITT Proceedings of the Workshop on Reusability in Programming*, pages 110–121, Stratford, CT, 1983. ITT.

[GKM85] S. Nowlan G. Kahn and J. McDermott. Strategies for knowledge acquisition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(5):511–522, September 1985.

[Gog86] J. Goguen. Reusing and Interconnecting Software Components. *IEEE Computer*, 19(2):16–28, February 1986.

[Gol67] E. Gold. Language Identification in the Limit. *Information and Control*, 10:447–474, 1967.

[Gol83] A. Goldberg. The Influence of an Object-Oriented Language on the Programming Environment. In *Proc. of 1983 ACM Computer Science Conference*, pages 35–54. ACM, February 1983.

[Goo82] D. Good. Reusable problem domain theories. Technical Report 31, Institute for Computing Science, UT Austin, September 1982.

[Gre84] S. Greenspan. Requirements Modelling: A Knowledge Representation Approach to Software Requirements Definition. Technical Report CSRG-155, Dept. of Computer Science, U. of Toronto, March 1984.

[Gro84] CIP Language Group. *The Munich Project CIP*. Springer Verlag, New York, 1984.

[Had45] J. Hadamard. *The Psychology of Invention in the Mathematical Field.* Dover Publications, New York, 1945.

[HKN85] E. Horowitz, A. Kemper, and B. Narasimhan. A Survey of Application Generators. *IEEE Software*, 2(1):40–54, January 1985.

[Hof87] R. Hoffman. The problem of extracting the knowledge of experts from the perspective of experimental psychology. *The AI Magazine*, 8(2):53–67, Summer 1987.

[Hyp88] Hypertext. Special Issue: Hypertext. *Communications of the ACM*, 31(7):816–895, July 1988.

[Ich83] J. Ichbiah. On the Design of Ada. In *IFIP - Information Processing 83*, pages 1–10. R. E. A. Mason, Publ., 1983.

[iee87] TOOLS, Making Reusability a Reality. *IEEE Software*, 4(4), July 1987.

[ITT83] ITT. *Proceedings of the Workshop on Reusability in Programming.* ITT Programming, Stratford, CT, 1983.

[Jac83] M. Jackson. *System Development.* Prentice/Hall International, Englewood Cliffs, NJ, 1983.

[Jon83] G. Jones. Identifying basic categories. *Psychological Bulletin, American Psychological Association*, 94(3):423–428, 1983.

[JvEB82] T. Janssen and P. van Emde Boas. Some Observations on Compositional Semantics. In *Logic of Programs*, pages 137–149. Springer Verlag, 1982.

[Kah84] G. Kahn. On when diagnostic systems want to do without causal knowledge. In T. O'Shea, editor, *ECAI-84: Advances in Artificial Intelligence*, pages 21–30. Elsevier Science Publishers, New York, 1984.

[KK76] B. Knobe and K Knobe. A Method for Inferring Contex-free Grammars. *Information and Control*, 31:129–146, 1976.

[Kli85] G. Klir. *Architecture of Systems Problem Solving.* Plenum Press, New York, 1985.

[Kol83] J. Kolodner. Reconstructive memory: A computer model. *Cognitive Science*, 7:281–328, 1983.

[KP81] D. Knuth and Michael Plass. Breaking Paragraphs into Lines. *Software Practice and Experience*, 11:1119–1184, 1981.

[Kuh70] T. Kuhn. *The structure of scientific revolutions.* The University of Chicago Press, 1970.

[LC87] P. Langley and J. Carbonell. Language Acquisition and Machine Learning. In *Mechanisms of Language Acquisition*, pages 115–155. Lawrence Erlbaum Associates, 1987.

[Leb83] M. Lebowitz. Concept learning in a rich input domain. In *Machine Learning*, pages 193–214. Tioga, Palo Alto, CA, 1983.

[Leh80] M. Lehman. On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle. *The Journal of Systems and Software*, 1(3):213–221, 1980.

[Leh81] M. Lehman. Programs, Life-cycles and Laws of Software Evolution. In *Infotech State of the Art Report, Series 9, No. 6*, pages 263–291. Pergamon Infotech Ltd., Maidenhead, Berkshire, UK, 1981.

[Lei87] J. C. Leite. A Survey on Requirements Analysis. Technical Report ASE-RTP-072, ICS, University of California, Irvine, June 1987.

[LF87] D. Lenat and E. Feigenbaum. On The Thresholds of Knowledge. In *Proc. IJCAI 87, Vol. II*, pages 1173–1182, 1987.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Software Development.* The MIT Press, 1986.

[LPS86] D. Lenat, M. Prakash, and M. Shepherd. CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks. *The AI Magazine*, VI(4):65–83, Winter 1986.

[LST84] M. Lehman, V. Stenning, and W. Turski. Another look at software design methodology. *ACM* SIGSOFT *Soft. Eng. Notes*, 9(2):38–53, April 1984.

[Lub86] M. Lubars. A knowledge-based design aid for the construction of software systems. Technical Report UIUCDCS-R-86-1304, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, IL, 1986.

[Mat82] Mathematica. RAMIS 2: User Manual. Technical report, Mathematica Products Group, Inc., Princeton, N.J., 1982.

[Mat87] Y. Matsumoto. A Software Factory: An Overall Approach to Software Production. In *Software Reusability*, pages 155–176. IEEE Computer Society Press, 1987.

[McC80] D. McCracken. A Guide to Nomad for Application Development. Technical report, National CSS, 1980.

[McC86] R. McCain. A software development methodology for reusable components. IBM Federal Systems Division, Houston, Texas, 1986.

[McG87] F. McGarry. Software Reuse – Report 10th. Minnowbrook Workshop. *DACS Newsletter*, VI(3):1, September 1987.

[McI76] M. McIlroy. Mass Produced Software Components, from 1969 NATO Conference in Software Engineering. In *Software Engineering Concepts and Techniques*, pages 88–98. Petrocelli/Charter, Brussels, 1976.

[Mic82] D. Michie. High-road and Low-road Programs. *AI Magazine*, 3(1):21–22, Winter 1982.

[Mic88] J. Micallef. Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages. *J. of Object-Oriented Programming*, 1(1):12–36, April 1988.

[MKKC86] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-Based Generalization—A Unifying View. In *Machine Learning I*, pages 47–80. Kluwer Academic Publishers, 1986.

[MM86] S. Marcus and J. McDermott. SALT: A Knowledge Acquisition Tool for Propose-and-Rrevise Systems. Technical Report CMU-CS-86-170, Dept. of Computer Science, Carnegie-Mellon University, 1986.

[MM88] David A. Marca and Clement L. McGowan. *SADT: Structured Analysis and Design Technique.* McGraw-Hill, New York, 1988.

[MMW85] S. Marcus, J. McDermott, and T. Wang. Knowledge Acquisition for Constructive Systems. In *Proc. International Joint Conference on Artificial Intelligence 85*, pages 637–639. IJCAI, 1985.

[MO87] R. T. Mittermeir and M. Oppitz. Software bases for the flexible composition of application systems. *IEEE Transactions on Software Engineering*, SE-13(4):440–460, April 1987.

[Mos81] J. Mostow. *Mechanical Transformation of Task Heuristics into Operational Procedures.* PhD. Dissertation, Department of Computer Science, Carnegie Mellon University, 1981.

[MS83] R. Michalski and R. Stepp. Learning from Observation: Conceptual Clustering. In *Machine Learning, Vol. 1*, pages 331–364. Morgan Kaufmann, Los Altos, CA, 1983.

[MT84] T. Maibaum and W. Turski. On what exactly is going on when software is developed step-by-step. In *IEEE Procs. 7th. Intl. Conf. on Software Engineering*, pages 528–533. IEEE Computer Society, 1984.

[Nei81] J. Neighbors. *Software Construction Using Components.* PhD. Thesis, Dept. of Information and Computer Science, U. of California, Irvine, 1981.

[Nei84] J. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Trans. on Software Engineering*, SE-10:564–573, September 1984.

[Nei88a] J. Neighbors. Draco: A Method for Engineering Reusable Software Systems. In *Reusable Software.* Addison Wesley, 1988. See [Big88].

[Nei88b] J. Neighbors. Why Do Domain Analysis? In *Proc. Reuse Workshop.* Rocky Mountain Institute of Software Engineering, October 1988.

[NSM85] R. Neches, W. Swartout, and J. Moore. Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of their Development. *IEEE Transactions on Software Engineering*, SE-11(11):1337–1350, November 1985.

[NvD78] W. Newman and A. van Dam. Recent Efforts Toward Graphics Standardization. *ACM Computing Surveys*, 10(4):365–380, December 1978.

[P+82] T. Payton et al. SSAGS: A Syntax and Semantics Analysis and Generation System. In *Proc. Computer Software and Applications Conference*, pages 424–433. CS Press, 1982.

[Par72] D. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1054–1058, December 1972.

[Par76] D. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[Par83] H. Partsch. Abstract data types as a tool for requirements engineering. In *Lecture Notes in Computer Science No. 74. Requirements Engineering*, pages 139–158. Springer-Verlag, 1983.

[Par86] H. Partsch. Transformational Program Development in a Particular Domain. *Science of Computing Programming*, 7(2):99–241, September 1986.

[Pat72] W. Patton. *Numerical Control.* Reston Publishing Co., Reston, VA, 1972.

[PCW83] D. Parnas, P. Clements, and D. Weiss. Enhancing Reusability Trough Information Hidding. In *ITT Proceedings of the Workshop on Reusability in Programming*, pages 240–247. ITT, Stratford, CT, 1983.

[PD85] R. Prieto-Diaz. *A Software Classification Scheme.* PhD. Thesis, U. of California, Irvine, CA, 1985.

[PD87a] R. Prieto-Diaz. A Software Classification Scheme for Reusability. *IEEE Software*, 4(1):6–16, January 1987.

[PD87b] R. Prieto-Diaz. Breathing New Life into Old Software. *GTE Journal*, 1(1):23–31, 1987.

[PD87c] R. Prieto-Diaz. Domain Analysis for Reusability. In *Proc. Compsac-87*. Tokyo, Japan, October 1987.

[Per87a] D. Perry. Software Interconnection Models. In *9th. Int. Converence on Software Engineering*, pages 61–69. IEEE Computer Society Press, March 1987.

[Per87b] D. Perry. Version Control in the Inscape Environment. In *9th. Int. Converence on Software Engineering*, pages 142–149. IEEE Computer Society Press, March 1987.

[Pet87] C. Petrie. Revised Dependency-Directed Bactracking for Default Reasoning. In *AAAI 87 Sixth Natinal Conference on Artificial Intelligence*, pages 167–172. Morgan-Kaufmann Publishers, July 1987.

[PFAB86] C. Potts, A. Finkelstein, M. Aslett, and J. Booth. Formal Requirements Specification—FOREST. Technical Report FOREST Deliverable Report 2, Alvey Project (UK), November 1986.

[Poi68] H. Poincaré. *La Science et L'Hypothèse*. E. Flammarion, Paris, 1968.

[Pol81] G. Polya. *Mathematical Discovery - On understanding, learning and teaching problem solving*. John Wiley and Sons, New York, 1981.

[Pre87] D. Prerau. Knowledge Acquisition in the Development of a Large Expert System. *The AI Magazine*, 8(2):43–51, Summer 1987.

[PS83] H. Partsch and R. Steinbrüggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.

[RB60] E. Madden R. Blake, C. Ducasse. *Theories of the Scientific Method*. University of Washington Press, Seattle, 1960.

[Reb81] R. Reboh. Knowledge engineering techniques and tools in the PROSPECTOR environment. Technical Report SRI Project 5821,6415 and 8172/Technical Note 243, Artificiall Intelligence Center, SRI International, June 1981.

[Ric81] C. Rich. A Formal Representation for Plans in the Programmer's Apprentice. In *Proc. of the 7th. International Joint Conference on Artificial Intelligence*, pages 1044–1052. IJCAI, Vacouver, Canada, 1981.

[RL79] B. Poynton R. Lanergan. Reusable Code: The Application Development Technique of the Future. In *Proc. of the IBM SHARE/GUIDE Software Symposium.* IBM, Monterey, CA., October 1979.

[rmi87] *RMISE 87*, Boulder, Colorado, October 1987. Rocky Mountain Institute on Software Engineering.

[RN84] J. Reggia and D. Nau. An Abductive Non-Monotonic Logic. In *Nonmonotonic Reasoning Workshop*, pages 385–395. AAAI, October 1984.

[RNY83] J. Reggia, D. Nau, and P. Young. Diagnostic expert systems based on a set covering model. *Int. J. Man-Machine Studies*, 19:437–460, 1983.

[Rob86] G. Robins. The NIKL Manual. Technical report, Information Sciences Institute, Marina del Rey, CA, April 1986.

[RPNP85] J Reggia, B. Perricone, D. Nau, and Y. Peng. Answer Justification in Diagnostic Expert Systems—Part I: Abductive Inference and its Justification. *IEEE Transactions on Biomedical Engineering*, BME-32(4):263–267, April 1985.

[SA88] D. Steier and P. Anderson. *Comparing Algorithm Syntheses.* Springer-Verlag, in press, New York, 1988.

[SE84] E. Soloway and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Trans. on Software Engineering*, SE-10(5):595–609, September 1984.

[Sep87] V. Seppanen. Reusability in Software Engineering. In *Software Reusability*, pages 286–297. IEEE Computer Society Press, 1987.

[Sha77] D. Shapere. Scientific Theories and Their Domains. In *The Structure of Scientific Theories, (Ed.) F. Suppe*, pages 519–565. U. of Illinois Press, 1977.

[Sha84a] M. Shaw. Abstraction Techniques in Modern Programming Languages. *IEEE Software*, 1(4):10–26, October 1984.

[Sha84b] M. Shaw. The Impact of Modelling and Abstraction Concerns on Modern Programming Languages. In *On Conceptual Modelling*, pages 49–83. Springer-Verlag, New York, 1984.

[Sim83] H. Simon. Why Should Machines Learn? In *Machine Learning*, pages 25–38. Tioga, Palo Alto, CA, 1983.

[Sin87] M. Sintzoff. Expressing Program Developments in a Design Calculus. In *Logic of Programming and Calculi of Discrete Design*, pages 343–365. NATO ASI Series, Vol. F36, Springer-Verlag, Berlin, 1987.

[SM86] R. Stepp and R. Michalski. Conceptual clustering – inventing goal-oriented classifications of structured objects. In *Machine Learning II, (Eds.) R. Michalski, J. Carbonell and T. Mitchell*, pages 471–498. Morgan Kaufmann Publishers. Inc., Los Altos, CA, 1986.

[SS83] W. Scherlis and D. Scott. First Steps Towards Inferential Programming. *Procs. IFIP 9th. World Computer Congress*, pages 199–212, 1983.

[Sta73] T. Standish. Observations and Hypotheses About Program Synthesis Mechanisms. Technical Report TR 2780-AP Memo 9, Bolt, Beranek and Newman, Inc., December 1973.

[Sta75] T. Standish. Extensibility in Programming Language Design. In *Proc. National Computer Conference*. AFIPS Press, 1975.

[Ste81] M. Stefik. Planning and Meta-planning (MOLGEN: Part 2). *Artificial Intelligence*, 16:141–169, 1981.

[Ste87] R. Stepp. Concepts in Conceptual Clustering. In *Proc. IJCAI-87, Vol. 1*, pages 211–213. Morgan Kaufumann Publ. Inc., Los Altos, CA, 1987.

[Sto77] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, MA, 1977.

[Sup77] F. Suppe. *The Structure of Scientific Theories*. U. of Illinois Press, 1977.

[SW86] I. Sommerville and M. Wood. A Knowledge-based Software Components Catalogue. Technical Report Research Report CS/ST/5/86, U. of Strathclyde, Glasgow, Scotland, 1986.

[Swa83] W. Swartout. XPLAIN: A System for Creating and Explaining Expert Consulting Programs. Technical Report ISI/RS-83-4, USC/Information Sciences Institute, Marina del Rey, CA, July 1983.

[TA88] E. Teratsuji and G. Arango. Notes from a Study of User Interface Initialization in Unix. Technical Report ASE-RTM-85, ASE, ICS, University of California, Irvine, July 1988.

[Tra87a] W. Tracz. Software Reuse: Motivators and Inhibitors. In *Proc. of COMPCON'87*, pages 358–363, February 1987.

[Tra87b] W. Tracz. Software Reuse Myths, Program Analysis and Verification Group, Dept. of Electrical Engineering, Stanford University, 1987.

[Tur86] W. Turski. And no philosopher's stone either. In *Information Processing 86*, pages 1077–1080. North Holland, September 1986.

[VB19] K. Vanlehn and W. Ball. A Version Space Approach to Learning Context-free Grammars. *Machine Learning*, 2(1):39–74, 19.

[VD83] N. Vitalari and G. Dickson. Problem Solving for Effective Systems Analysis: An Experimental Exploration. *Communications of the ACM*, 26(11):948–956, November 1983.

[WA86] D. Wile and D. Allard. Worlds: An Organizing Structure for Object Bases. In *Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 16–26. ACM SIGPLAN Notices, Vol. 22, N.1, 1986.

[WC83] K. Wexler and P. Culicover. *Formal Principles of Language Acquisition.* The MIT Press, Cambridge, MA, 1983.

[Weg84] P. Wegner. Capital Intensive Software Technology. *IEEE Software*, 1(3):7–45, July 1984.

[Wil83] D. Wile. Program Developments: Formal Explanations of Implementations. *Communications ACM*, 26(11):902–911, November 1983.

[Wil86] D. Wile. Local Formalisms: Widening the Spectrum of Wide-Spectrum Languages. In *Working Conference on Program Specifications and Transformations.* IFIP WG 2.1, 1986.

[Win79] T. Winograd. Beyond Programming Languages. *Communiations of the ACM*, 22(7):391–401, July 1979.

[WLJ85] Elliot Soloway W. Lewis Johnson. PROUST: Knowledge-Based Program Understanding. *IEEE Trans. on Software Engineering*, SE-11(3):267–275, March 1985.

[Woo70] W. Woods. Transition Network Grammars for Natural Language Analysis. *Communications of the ACM*, 13(10):591–606, October 1970.

[Yue87] K. Yue. What Does It Mean To Say That A Specification Is Complete. In *Proc. 4th. Int. Workshop on Software Specification and Design*, pages 42–49. IEEE Computer Society Press, April 1987.

[ZdJ77] M. Zloof and S. de Jong. The System for Business Automation (SBA): Programming Language. *Communications of the ACM*, 20(6):385–396, June 1977.

Appendix A

Glossary

Adjustment Cost of a Package (Section 7.4.1)
the adjustment cost of a package P relative to a customer specification Q and a reuser R the cost of adapting the services offered by P to match the services required by the specification Q, computed as the sum of the cost of $AdjCost_R(Q, P) = \sum_k CompleteCost(Q(a_k), P)$

Competence (Section 3.2.1)
Given two states of a reuse system, the difference in competence between them is defined by the cardinality of difference between their covered sets. Intuitively, δ_c, is the number of specifications that can be implemented by one and only one of the systems.

Component (Section 4.2)
The term component is used to refer to named modules of reusable informations that can be manipulated by a target reuser. For example, first-order components are composed or interconnected to create larger systems, second-order components are composed into implementation plans that must be executed at reuse time to generate the desired systems.

Component Basis (Section 7.1)
Given a reuser R and a class of specifications S, a family of reusable software components I is a component basis for R and S if: 1) the set of specifications that can be implemented using basic elements and the the operators available to R is a superset of I, 2) the members of the basis cannot be generated from other members of of the basis by using the operators available to R.

Component Cache (Section 7.2.2)
We define a *reuse cache* for the environment of a reuser R, as a collection H of implemented packages, such that there exists a high probability that queries in $Log_R(T)$ can be implemented by reusing members of H *as they are.*

Compositional reuse (Section 4.3.1)
A compositional approach to reuse in software specification and construction is based

on the principle that: "the meaning of a construct is specified in terms of the meaning of its syntactic subcomponents". The composition process is context-free.

Concept Definition (Section 5.2.1)
Using the MoD representation language, concepts in a problem domain, are defined as collections of property-value pairs (also called MoD objects), where the properties define relations between the subject and other objects, their values being other objects in the MoD.

Covered set of specifications (Section 3.2)
If the system P, in state $\mathcal{S}$, can produce satisfactory implementations for a set of specifications S we say that P *covers* S, $S = C(\mathcal{S})$. The size of S is used as a measure of coverage.

Definition Neighborhood of a Concept Description (Section 5.2.1)
The definition neighborhood of a concept description is the set of concepts descriptions that are values of the subject's attributes. The descriptions in the neighborhood of a concept provide an definition for the semantics of the subject.

Design Recovery (Section 6.12)
Given an implemented software system, the problem of design recovery consists in producing a specification for that system and an explanation of why the existing implementation is in fact an implementation of the specification.
A method of design recovery has been proposed (Section 6.13) based on three steps: 1) identification of implementation components, 2) design hypothesis generation and 3) hypothesis selection.

Domain, Problem Domain
(Section 2.5) In a given community, items of real-world information come to be associated as bodies of information or problem domains having the following characteristics: 1) deep or comprehensive relationships among the items of information are suspected or postulated with respect to some class of problems; 2) the problems are perceived as significant by the members of the community; and 3) there exists the knowledge to produce solutions to the problems.
(Section 4.1) A problem domain D is characterized extensionally by collection of pairs of specifications of software system and their implementations, $D = \{ (S, I_S) \}$. Each pair $\{ (S, I_S) \}$ is called an exemplar of the domain.

Domain Analysis
(Section 2.4.2)"A domain analysis is an attempt to identify the objects, operations and relationships between what domain experts perceive to be important about the domain" [Nei88b].
(Section 3.2.3 The purpose of practical domain analysis is to evolve the state of the MoD—the knowledge of the problem domain available to the reuse system—so that the reuse system attains a specified level of performance within resource constraints.

Domain engineering process (Section 1.3.1)
a domain engineering process for software reuse consists of the following activities:

1. *Domain analysis*: the identification and acquisition of information in a problem domain to be reused in software specification and construction.
2. *Infrastructure specification*: the modularization and organization of reusable information as required by a reuse plan; for example, subprograms, objects, transforms, database schemas, packages.
3. *Infrastructure implementation*: the design and encoding of the pieces resulting from the specification process using particular representations required by the technology of the reusers.

Domain Engineering-in-the-large (Section 6.2.3)
We use the expression domain engineering-in-the-large to refer to the activity of designing new domain networks out of reusable MoDs, in direct analogy with programming-in-the-large.

Domain Network (Section 2.2.3, 6.2.3)
A domain network is a family of MoDs interconnected using "escape" and "implementation" relations. Domain networks capture the results of the process of domain analysis by representing in the MoDs problem-specific information needed to specify systems in the domain, and plans for the implementation of those specifications.

Domain-oriented Reuse (Section 4.1.1)
A reuse system $P = \langle MoT, T_R, RI, \{S, I_S\} \rangle$ is said to be domain-oriented if $MoT_{T_R}(S, RI) = I_S$, RI is non empty, and the items of information in RI are non-trivial, and necessary for producing I_S.

Domain-Specific Reuse (Section 4.1.1)
The size of the class S that can be implemented by a domain-oriented reuse system measures of degree of *domain-specificity* of the system.

Efficiency of a Reuser (Section 3.2.1)
The efficiency of a reuse system is measured by a costing function f defined in the environment of the system that computes the average cost of parsing and implementing specifications for systems in the domain. The definition of f depends on the technology of the reuser.

Evolution Trigger (Section 6.1.1) Evolution triggers are events that suggest the need to change the state of the MoD. Evolution triggers can be the result of the analysis of the MoD or of a reuse infrastructure derived from the MoD. In Section 6.4.1 evolution triggers are represented using unbound MoD-evolution operators.

First-order Reuser (Section 4.2.1)
A first-order reuser operates on a first-order infrastructure. A first-order infrastructure contains reusable information that is *not* evaluated at software construction time. Such is the case of libraries of Cobol programs, Ada generics, abstract data types, spreadsheet templates, Smalltalk objects.

Genericity Cost of a Package (Section 7.4.1)
The genericity cost of a package is the increase over the basic implementation cost

of a instance of the package due to the need to implement multiple value of some feature.

Goal Dependency Network (Section 6.2.1)
A goal dependency network includes assertions as nodes, and captures the relations between external goals and constraints in the problem domain and objects in the MoD. Its primary purpose is to provide a justification to the semantics of the concepts defined in the MoD. The assertions in the GDN also provide guidelines for the classification and generalization of MoD definitions, and for the selection of explanations in design recovery.

Hypothetical Reuse (Section 6.5.2)
Hypothetical reuse is the informal execution, by the domain analyst, of the reuse procedures prescribed by the model of the task, using the information in the MoD state directly.

Implementation Closure of a MoD object. (Section 5.4)
The implementation closure of an S-object s describes how is s implemented in terms of the subsidiary domain network. Using the MoD-graph structure, the implementation closure of s, is represented as a subgraph of the I-graph rooted at the s and related by **implementor** and **plan** relations to the objects in subsidiary domains that are used to implement it.

Implementation Cost of a Package (Section 7.4.1)
Given a class of specifications $\{Q_i\}$, the cost of implementing a package to implement them, $P_i = \Pi(Q_i)$ may be approximated using two cost components: a *basic implementation cost*—for implementing those services selected by the implementation threshold IC—and an estimated *genericity cost*— the increase over the basic implementation cost due to the need to implement multiple values of some feature, as selected using the threshold GT.

Learning Component of a Reuse System (Section 3.2.3)
The purpose of the learning component in a reuse system is the (goal-directed) generation of new values for the goal-related variables of P to increase the performance of the reuse system. In our study we have isolated the MoD as *the* goal-related variable in a reuse system. The learning component is defined by five traits, 1) a model of the problem domain, 2) methods for evolving the model, 3) a *Reuse Log*, 4) a source of domain expertise, and 5) a set of support technologies T_L.

Model of the Domain, MoD (Chapter 5)
The MoD structure is a data structure for representing reusable information about the problem domain. The state of the MoD is defined by the information captured in the structure at a given time. A MoD state for a first-order, compositional reuser captures an abstract representation of the parts and composition operators that the reuser may employ in constructing software in the domain. Chapter 5 discusses a MoD representation language, MoDL. (To maintain a consistent notation we use italics, MoD, to refer to a MoD as a variable of the reuse system, and roman font to refer to the abstract structure.)

Model of the reuse Task, MoT (Section 3.1.1)
A Model of the reuse Task is a rigorous characterization of how reuse is performed. This includes a definition of the form of the reusable objects, of operators for manipulating those objects and their applicability conditions, and of a control strategy for achieving the goal of implementing a specification through reuse.
(To keep a consistent notation we use italics, MoT, to refer to a MoT as a variable of the reuse system, and roman font to refer to the general concept of model of the reuse task.)

Orders of Reuse (Section 4.2)
Orders of reuse is a criterion for categorizing reuse systems. The order of a reuser is an indication of the "depth" of the implementation knowledge explicitly available to the reuse system in some form of RI. First-order information is captured in solution components and operators over solution components. Second-order information corresponds to plan-components for the generation of solutions and operators over plans, and so on.

Operational (Section 1.4 2.4.2)
A goal is operational for an agent in a particular environment if that agent, by performing some computation using the available information, can determine whether the goal is achieved [DB].

Performance Component of a Reuse System (Section 3.1.1)
The performance component of a reuse system realizes the actual process of reuse. It is characterized by the following state variables: a model of the reuse task, MoT; a pool of technologies to support reuse, T_r; a reuse infrastructure, RI; a class of specifications, S.

Reuse Environment (Section 3.1.2)
The factors that determine the values of the input variables of the reuse system constitute its environment. We characterize the environment of a reuse system in terms of four features: the reuse task, the problem domain, the available technologies to support reuse, and cost-benefits models for evaluating when it is cost-effective to practice reuse or invest in the evolution of the existing infrastructures.

Reuse Event (Section 7.2.2) For a compositional reuser, a reuse event is an attempt to implement a specification reusing the available infrastructure. A reuse event is triggered by a request, or *query*, made by an agent that we call a customer. A query is a possibly incomplete statement in a domain-specific language. The reuser must decompose the specification, retrieve implementation parts, and produce a composite implementation. A reuse event may be successful or not.

Reuse Infrastructure (Section 3.1.1)
A *reuse infrastructure* is defined as a collection of reusable abstractions that can be instantiated to specify and/or implement systems in a particular problem domain. The reusable information must be such that, 1) it is of the type specified by the MoT of the system, and 2) it is encoded in a form that allows for manipulation by the technology of the reuser.

Reuse Log (Section 7.2.2)
A reuse log for a reuser R, $Log_R(T)$, is a record of reuse events over a period of time T.

Reuse system (Section 3)
Reuse systems are viewed as composed of two parts, a performance component (or, actual reuser) and a learning component. The performance component executes the actual process of reuse, aided by a reuse infrastructure generated by the learning component.

Reuse working set (Section 7.2.2)
We define a *reuse working set* for the environment of a reuser R, as a collection W of partially implemented components, or packages, such that *most* queries in $Log_R(T)$ can be implemented by completing members of W.

Appendix B

MoD Representation Language

The grammar for the MoD representation language, MoDL, is based on RML's [Gre84]. It extends the RML language by incorporating model definition as a language clause, new types of built-in objects and property categories, and by extending the form of assertions. As a concession to practicality, it also introduces natural language assertions for the representation of problem goals, design rationales, and other forms of justification information.

> Notation: meta-symbols ⟨ ⟩ ::= { } [ ] | “”
> Symbols in boldface, italics and all-caps are terminals.
> Symbols consisting of a string in angle brackets are nonterminals.
> [⟨ symbol ⟩] zero or one occurrence of ⟨ symbol ⟩
> {⟨ symbol ⟩} zero or more occurrences of ⟨ symbol ⟩
> “⟨ symbol ⟩” atomic element ⟨ symbol ⟩
> {⟨ symbol ⟩}+ one or more occurrences of ⟨ symbol ⟩

⟨MoD⟩ ::= { ⟨ model-definition ⟩ } +
⟨model-definition⟩ ::= ⟨ m-subject ⟩ ({ ⟨ factual-properties ⟩})
in {⟨ m-classifier ⟩}+
isa {⟨ m-superclass ⟩}+
with {⟨ m-def-properties ⟩}
⟨ m-subject ⟩ ::= ⟨ user-defined-model ⟩
⟨ m-classifier ⟩ ::= ⟨ model ⟩
⟨ m-superclass ⟩ ::= ⟨ model ⟩
⟨ m-def-properties⟩ ::= ⟨ m-property-category⟩{⟨ m-def-property⟩}
⟨ m-def-property⟩ ::= ⟨attribute⟩ : ⟨ m-property-value⟩
⟨ m-property-category ⟩ ::= **id**| **part** | **constraint** | **supported-by**
⟨ m-property-value⟩ ::= { “[” ⟨object ⟩ “]” }⟨ object ⟩ [⟨bindings ⟩]

| ⟨ model ⟩ [⟨bindings ⟩]
⟨ user-defined-model ⟩ ::= ⟨ upper-case-identifier ⟩

⟨ model ⟩ ::= { ⟨ object-definition ⟩ } +
⟨object-definition⟩ ::= ⟨ o-subject ⟩ ({ ⟨ factual-properties ⟩})
in {⟨ o-classifier ⟩}+
isa {⟨ o-superclass ⟩}+
with {⟨ definitional-properties ⟩}
⟨ o-subject ⟩ ::= ⟨ user-defined-object ⟩
⟨ o-classifier ⟩ ::= ⟨ object ⟩
⟨ o-superclass ⟩ ::= ⟨ object ⟩

⟨factual-properties ⟩ ::= ⟨ fact ⟩ {, ⟨ fact ⟩}
⟨fact ⟩ ::= ⟨ attribute ⟩ : ⟨ attribute-expression ⟩
⟨ attribute-expression ⟩ ::= ⟨ object ⟩ | ⟨ attribute⟩
| ⟨ built-in-term ⟩ | ⟨ attribute-expression ⟩ @ ⟨ attribute ⟩
⟨ attribute ⟩ ::= ⟨ user-defined-attribute ⟩ | *self* | *now*

⟨definitional-properties ⟩ ::= ⟨d-def-properties⟩ | ⟨ s-def-properties⟩
| ⟨ j-def-properties ⟩

⟨d-def-properties⟩ ::= ⟨d-property-category⟩{⟨def-property⟩}
⟨j-def-properties⟩ ::= ⟨j-assertion-pcat⟩{⟨def-property⟩}
⟨s-def-properties⟩ ::= ⟨s-property-category⟩{⟨def-property⟩}

⟨def-property⟩ ::= ⟨attribute⟩ :⟨property-value⟩
⟨ property-value ⟩ ::= { "[" ⟨object⟩ "]" }⟨ object ⟩ [⟨bindings ⟩]
⟨bindings ⟩ ::= ⟨fact ⟩ {, ⟨ fact ⟩}

⟨ d-property-category ⟩ ::= ⟨ d-plan-pcat ⟩ | ⟨d-design-pcat⟩
⟨ s-property-category⟩ ::= ⟨ s-entity-pcat ⟩ | ⟨s-activity-pcat⟩ |
| ⟨ s-assertion-pcat ⟩

⟨ j-assertion-pcat ⟩ ::= **id** | **part** | **supports** | **neg-supports**
| **supported-by** | **neg-supported-by**
⟨ s-entity-pcat ⟩ ::= **id** | **part** | **association** | **necessary** | **supported-by**
| **invariant** | **initcond** | **finalcond** | **producer** | **constraint**
| **consumer** | **modifier** | **implementor** | **supported-by**
⟨ s-activity-pcat ⟩ ::= **id** | **input** | **control** | **output** | **precond**
| **postcond** | **trigger** | **stopcond** | **part** | **constraint**
| **modifies** | **implementor**
⟨ s-assertion-pcat ⟩ ::= **id** | **argument** | **part** | **constraint**

⟨ d-plan-pcat ⟩ ::= **id** | **precondition** | **part** | **technology**
| **satisfies** | **obligation** | **supported-by**
⟨ d-design-pcat ⟩ ::= **id** | **design-goals** | **plan** | **constraint** | **decision**
| **supported-by**

⟨object ⟩ ::= ⟨ user-defined-object ⟩ | ⟨ built-in-object ⟩
| ⟨ constructed-object ⟩
⟨ built-in-object⟩ ::=
⟨ OBJECT ⟩ | ⟨ TOKEN ⟩ | ⟨ CLASS ⟩ | ⟨ META-CLASS ⟩
| ⟨ S-ENTITY ⟩ | ⟨S-ACTIVITY⟩ | ⟨ S-ASSERTION ⟩
| ⟨PLAN ⟩ | ⟨ DESIGN ⟩ | ⟨ J-ASSERTION ⟩
| ⟨S-ENTITY-CLASS⟩ | ⟨S-ACTIVITY-CLASS⟩
| ⟨S-ASSERTION-CLASS ⟩ | *nothing*
| ⟨PLAN-CLASS⟩ | ⟨DESIGN-CLASS⟩ | ⟨J-ASSERTION-CLASS ⟩
| ⟨S-ENTITY-METACLASS⟩ | ⟨S-ACTIVITY-METACLASS ⟩
| ⟨S-ASSERTION-METACLASS⟩ | ⟨ J-ASSERTION-METACLASS ⟩
| ⟨PLAN-METACLASS⟩ | ⟨DESIGN-METACLASS ⟩
⟨ constructed-object ⟩ ::= *class-of*⟨ user-defined-object ⟩

⟨ assertion ⟩ ::= ⟨predicate ⟩ (⟨bindings ⟩)
| *forall* ⟨ identifier ⟩ *in* ⟨ object ⟩ ⟨assertion ⟩
| *exists* ⟨ identifier ⟩ *in* ⟨ object ⟩ [*suchthat* | *with*] ⟨ assertion ⟩
| (⟨assertion ⟩) | "{" ⟨assertion ⟩ "}" | "[" ⟨ assertion ⟩ "]"
| *not* ⟨assertion ⟩
| ⟨ assertion ⟩ ⟨connective ⟩ ⟨ assertion ⟩
| ⟨ arithmetic-expression ⟩ | "⟨ natural-language-expression ⟩"
⟨ connective ⟩ ::= *and* | *or* | *implies* | *iff*
⟨ predicate ⟩ ::= ⟨ user-defined-object ⟩ | ⟨built-in-predicate ⟩
⟨ built-in-predicate ⟩ ::= IN | ISA | = | ≠ | DURING
| NEXT | BEFORE | SAMEBEGIN
⟨ built-in-term ⟩ ::= RANGE(⟨bindings ⟩) | PVAL(⟨bindings ⟩)
| START (⟨bindings⟩) | END(⟨bindings⟩)

⟨ user-defined-object ⟩ ::= ⟨ upper-case-identifier ⟩ | ⟨ assertion ⟩
⟨user-defined-attribute ⟩ ::= ⟨ lower-case-identifier⟩