Software Reuse

CHARLES W. KRUEGER

School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213

Software reuse is the process of creating software systems from existing software rather than building software systems from scratch. This simple yet powerful vision was introduced in 1968. Software reuse has, however, failed to become a standard software engineering practice. In an attempt to understand why, researchers have renewed their interest in software reuse and in the obstacles to implementing it.

This paper surveys the different approaches to software reuse found in the research literature. It uses a taxonomy to describe and compare the different approaches and make generalizations about the field of software reuse. The taxonomy characterizes each reuse approach in terms of its reusable *artifacts* and the way these artifacts are *abstracted*, *selected*, *specialized*, and *integrated*.

Abstraction plays a central role in software reuse. Concise and expressive abstractions are essential if software artifacts are to be effectively reused. The effectiveness of a reuse technique can be evaluated in terms of *cognitive distance*—an intuitive gauge of the intellectual effort required to use the technique. Cognitive distance is reduced in two ways: (1) Higher level abstractions in a reuse technique reduce the effort required to go from the initial concept of a software system to representations in the reuse technique, and (2) automation reduces the effort required to go from abstractions in a reuse technique to an executable implementation.

This survey will help answer the following questions: What is software reuse? Why reuse software? What are the different approaches to reusing software? How effective are the different approaches? What is required to implement a software reuse technology? Why is software reuse difficult? What are the open areas for research in software reuse?

Categories and Subject Descriptors: D.1.0 [**Programming Techniques**]: General; D.2.1 [**Software Engineering**]: Requirements/ Specifications—*languages*, *methodologies*, *tools*; D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*, *programmer workbench*, *software libraries*; D.2.m [**Software Engineering**]: Miscellaneous—*reusable software*; D.3.2 [**Programming Languages**]: Language Classifications—*specialized application languages*, *very high-level languages*; D.3.4 [**Programming Languages**]: Processors; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*abstracting methods*, *indexing methods*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*query formulation*, *retrieval models*, *search process*, *selection process*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*, *program transformation*.

This work was supported in part by the Hillman Fellowship for Software Engineering, in part by ZTI-SOF of Siemens Corporation, Munich, Germany, and in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, OH 45433-6543 under contract F33615-90-C-1465, ARPA Order 7597.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
©ACM 0360-0300/92/0600-0131 $01.50

General Terms: Design, Economics, Languages

Additional Key Words and Phrases: Abstraction, cognitive distance, software reuse

CONTENTS

INTRODUCTION
1. ABSTRACTION
   1 1 Abstraction in Software Development
   1 2 Abstraction in Software Reuse
   1 3 Cognitive Distance
2. ORGANIZATION OF THE SURVEY
3. HIGH-LEVEL LANGUAGES
   3.1 Abstraction in High-Level Languages
   3.2 Selection
   3.3 Specialization
   3.4 Integration
   3 5 Appraisal of Reuse Techniques in High-Level Languages
4. DESIGN AND CODE SCAVENGING
   4 1 Abstraction in Design and Code Scavenging
   4.2 Selection
   4.3 Specialization
   4.4 Integration
   4.5 Appraisal of Reuse Techniques in Design and Code Scavenging
5. SOURCE CODE COMPONENTS
   5 1 Abstraction in Source Code Components
   5 2 Selection
   5.3 Specialization
   5.4 Integration
   5.5 Object-Oriented Variants to Component Reuse
   5.6 Appraisal of Reuse Techniques in Source Code Components
6 SOFTWARE SCHEMAS
   6 1 Abstraction in Software Schemas
   6 2 Selection
   6 3 Specialization
   6.4 Integration
   6.5 Appraisal of Reuse Techniques in Software Schemas
7. APPLICATION GENERATORS
   7 1 Abstraction in Application Generators
   7.2 Selection
   7.3 Specialization
   7.4 Integration
   7.5 Software Life Cycle with Application Generators
   7 6 Appraisal of Reuse Techniques in Application Generators
8. VERY HIGH-LEVEL LANGUAGES
   8.1 Abstraction in Very High-Level Languages
   8.2 Selection
   8.3 Specialization
   8.4 Integration
   8.5 Appraisal of Reuse Techniques in Very High-level Languages
9. TRANSFORMATIONAL SYSTEMS
   9.1 Abstraction in Transformational Systems
   9.2 Selection
   9.3 Specialization
   9.4 Integration
   9.5 Appraisal of Reuse Techniques in Transformational Systems
10. SOFTWARE ARCHITECTURES
   10.1 Abstraction in Software Architectures
   10.2 Selection
   10.3 Specialization
   10.4 Integration
   10.5 Appraisal of Reuse Techniques in Software Architectures
11. SUMMARY
   11.1 Categories and Taxonomy
   11.2 Cognitive Distance
   11 3 General Conclusions
ACKNOWLEDGMENTS
REFERENCES

INTRODUCTION

The 1968 NATO Software Engineering Conference is generally considered the birthplace of the software engineering field [Naur and Randell 1968]. The conference focused on the *software crisis*—the problem of building large, reliable software systems in a controled, cost-effective way (it was at this conference that the term *software crisis* originated). From the beginning, *software reuse* has been touted as a means for overcoming the software crisis. The seminal paper on software reuse was an invited paper at the conference: *Mass Produced Software Components* by McIlroy [1968]. McIlroy proposed a library of reusable components and automated techniques for customizing components to different degrees of precision and robustness. McIlroy felt that component libraries could be effectively used for numerical computation, I/O conversion, text processing, and dynamic storage allocation.

Twenty-three years later, many computer scientists still see software reuse as *potentially* a powerful means of im-

proving the practice of software engineering [Boehm 1987; Brooks 1987; Standish 1984]. The advantage of amortizing software development efforts through reuse continues to be widely acknowledged, even though the tools, methods, languages, and overall understanding of software engineering have changed significantly since 1968 [Biggerstaff and Richter 1989].

In spite of its promise, software reuse has failed to become standard practice for software construction. In light of this failure, the computer science community has renewed its interest in understanding how and where reuse can be effective and why it has proven so difficult to bring the seemingly simple idea of software reuse to the forefront of software development technologies [Biggerstaff and Perlis 1989a, 1989b; Freeman 1987b; Tracz 1988].

Simply stated, software reuse is using existing software artifacts during the construction of a new software system. The types of artifacts that can be reused are not limited to source code fragments but rather may include design structures, module-level implementation structures, specifications, documentation, transformations, and so on [Freeman 1983].

There is great diversity in the software engineering technologies that involve some form of software reuse. However, there is a commonality among the techniques used. For example, software component libraries, application generators, source code compilers, and generic software templates all involve *abstracting*, *selecting*, *specializing*, and *integrating* software artifacts [Biggerstaff and Richter 1989]. In this survey, software engineering technologies are analyzed and contrasted in terms of their idiomatic reuse techniques, particularly along the four mentioned dimensions:

Abstraction. All approaches to software reuse use some form of abstraction for software artifacts. Abstraction is *the* essential feature in any reuse technique. Without abstractions, software developers would be forced to sift through a collection of reusable artifacts trying to figure out what each artifact did, when it could be reused, and how to reuse it.

Selection. Most reuse approaches help software developers locate, compare, and select reusable software artifacts. For example, classification and cataloging schemes can be used to organize a library of reusable artifacts and to guide software developers as they search for artifacts in the library [Horowitz and Munson 1989].

Specialization. With many reuse technologies, similar artifacts are merged into a single *generalized* (or *generic*) artifact. After selecting a generalized artifact for reuse, the software developer *specializes* it through parameters, transformations, constraints, or some other form of refinement. For example, a reusable stack implementation might be parameterized for the maximum stack depth. A programmer using this generalized stack would specialize it by providing a value for this parameter.

Integration. Reuse technologies typically have an *integration* framework. A software developer uses this framework to combine a collection of selected and specialized artifacts into a complete software system. A *module interconnection language* is an example of an integration framework [DeRemer and Kron 1976; Prieto-Diaz and Neighbors 1986]. With a module interconnection language, functions are *exported* from modules that implement them and *imported* into modules that use them. Modules are assembled into a system by interconnecting modules with the appropriate exports and imports.

From the software engineering perspective, software reuse pertains solely to the process of constructing software systems. Using existing sine routine source code during the *construction* of a program is considered an example of software reuse, but repeatedly invoking

that sine routine during the *execution* of the program is not. Also excluded are repeated program executions and verbatim duplication of programs for the purpose of distribution.

The goal of this paper is to introduce *research* efforts in software reuse; it is primarily a survey of the software reuse literature. To provide a coherent perspective of the diverse literature, a uniform taxonomy is developed. Different reuse approaches are then characterized using this taxonomy. Some simple comparative analysis identifies the relative merits and drawbacks of the different reuse techniques and forms the basis for generalizations about the field of software reuse.

As a survey of research literature, this paper is not intended to be a guide for selecting or implementing a reuse technology in practice. Many of the systems described are research prototypes that have not been scaled up or validated in practical use.

1. ABSTRACTION

Because abstraction is such an important part of software reuse, it is used as the unifying theme in this survey. This choice also reflects the view that successful application of a reuse technique to a software engineering technology is inexorably tied to raising the level of abstraction for that technology. Since raising abstraction levels for software engineering technologies has proven to be quite difficult, the relation between abstraction and reuse provides us with the first clue to why there are so few successful reuse systems.

Others have noted the relationship between software reuse and abstraction [Booch 1987; Neighbors 1984; Parnas et al. 1989; Standish 1984; Wegner 1983]. According to Wegner [1983, p. 30], for example, "abstraction and reusability are two sides of the same coin." He states that every abstraction describes a related collection of reusable entities and that every related collection of reusable entities determines an abstraction.

1.1 Abstraction in Software Development

Computer scientists often use abstraction to help manage the intellectual complexity of software [Shaw 1984]. An abstraction for a software artifact is a succinct description that suppresses the details that are unimportant to a software developer and emphasizes the information that is important. For example, the abstraction provided by a high-level programming language allows a programmer to construct algorithms without having to worry about the details of hardware register allocation.

Software typically consists of several layers of abstraction built on top of raw hardware. The lowest-level software abstraction is object code, or machine code. Assembly language is a layer of abstraction above object code. A programming language (e.g., Ada[1]) is a layer of abstraction above the assembly language. In modular languages such as Ada, the module specification can serve as a layer of abstraction above the implementation details in the module body.

These examples demonstrate that every software abstraction has two levels. The higher of the two levels is referred to as the abstraction *specification*. The lower, more detailed level is called the abstraction *realization*.[2] When abstractions are layered, the abstraction specification at one layer is the abstraction realization at the next higher layer. Figure 1 shows a hierarchy with two abstractions, L and M. Rep 1, Rep 2, and Rep 3 are three representations of the same software artifact, where Rep 1 is the most detailed (lowest level) representation. For abstraction L, Rep 2 is the abstraction specification, and Rep 1 is the abstraction realization. From the point of view

[1] Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

[2] *Implementation* is also common terminology for the lowest level of detail in an abstraction. However, implementation is often associated with executable source code, which is not necessarily the case for all abstractions. For this reason, realization is used in this survey.

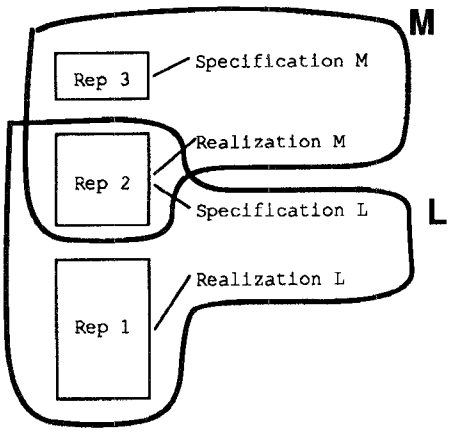

Figure 1. Two-level abstraction hierarchy.

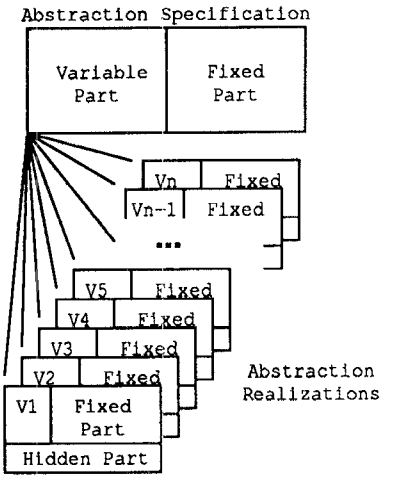

Figure 2. Mapping from a variable abstraction specification.

of abstraction *M*, Rep 3 is the abstraction specification, and Rep 2 is the abstraction realization.

An abstraction has a hidden part, a variable part, and a fixed part. The *hidden* part consists of the details in the abstraction realization that are not visible in the abstraction specification. The *variable* and *fixed* parts are visible in the specification. The variable part represents the variant characteristics in the abstraction realization, whereas the fixed part represents invariant characteristics in the abstraction realization. Therefore, as illustrated in Figure 2, an abstraction specification with a variable part corresponds to a collection of alternate realizations. The variable part of an abstraction specification maps into the collection of possible realizations.

For example, in an abstraction for stacks, the fixed part of the abstraction expresses the invariant characteristics for all stack realizations, such as the last-in-first-out (LIFO) semantics. The invariant stack behavior does not depend on the type of elements stored in the stack, so the element type can be in the variable part of the abstraction. Then each different element type corresponds to a different stack realization.

The partitioning of an abstraction into variable, fixed, and hidden parts is not an innate property of the abstraction but rather an arbitrary decision made by the creator of the abstraction. The creator decides what information will be useful to users of the abstraction and puts it in the abstraction specification. The creator also decides which properties of the abstraction the user might want to vary and places them in the variable part of the abstraction specification. Continuing with the stack example, the value for the maximum stack depth can be placed in either the variable, fixed, or hidden part of the stack abstraction. If it is placed in the variable part, the user has the ability to choose the maximum stack depth (e.g., 10, 1000, *unbounded*). If the maximum stack depth is placed in the fixed part, the user knows the predefined value of maximum stack depth but cannot change it. If placed in the hidden part, the stack depth is totally removed from the concerns of the user.

Abstraction specifications and realizations can take on many forms. They can be formal or informal, explicit or implicit. Consider the stack example written as a generic Ada package. The abstraction realization corresponds to an instantiation of the generic package with a particular stack element type. The abstraction specification, on the other hand, must be a combination of different descriptions

because of Ada's limited expressiveness. The generic package can provide the *syntactic* specification for operations of the stack abstraction, but the *semantic* specification must be expressed outside of the Ada language. One possibility is to use a formal notation such as Hoare axioms. Another is to use an informal description such as English text.

Semantic specifications are rarely derived from first principles. Abstraction creators implicitly assume that a certain amount of the abstraction specification is common knowledge among the users. As an extreme example, it might be sufficient simply to give "stack" as a semantic specification and assume that all users understand the stack abstraction without an explicit formal description.

In summary, an abstraction expresses a high-level, succinct, natural, and useful *specification* that corresponds to a less perspicuous *realization* level of representation. The abstraction specification typically describes "what" the abstraction does, whereas the abstraction realization describes "how" it is done. For an abstraction to be effective, its specification must express all of the information that is needed by the person who uses it [Shaw 1984]. This may include space/time characteristics, precision statistics, scalability limits, and other information not normally associated with specification techniques [McIlroy 1968].

1.2 Abstraction in Software Reuse

Abstraction plays a central and oftentimes limiting role in each of the other facets of software reuse:

Selection. Reusable artifacts must have concise abstractions so users can efficiently locate, understand, compare, and select the appropriate artifacts from a collection.

Specialization. A generalized reusable artifact is in fact an abstraction with a variable part. Specialization of a generalized artifact corresponds to choosing an abstraction realization from the variable part of an abstraction specification.

Integration. To integrate a reusable artifact into a software system effectively, the user must clearly understand the artifact's *interface* (i.e., those properties of the artifact that interact with other artifacts or the integration framework). An artifact interface is an abstraction in which the internal details of the artifact are suppressed.

1.3 Cognitive Distance

The effectiveness of abstractions in a software reuse technique can be evaluated in terms of the intellectual effort required to use them. Better abstractions mean that less effort is required from the user. To aid in evaluation, *cognitive distance* is introduced as an intuitive gauge for comparing abstractions.

Cognitive distance is defined as the amount of intellectual effort that must be expended by software developers in order to take a software system from one stage of development to another. From this definition, it should be clear that cognitive distance is not a formal measurement that can be expressed with numbers and units. Rather, it is an informal notion that relies on intuition about the relative effort required to accomplish various software development tasks.

For the creator of a software reuse technique, the goal is to minimize cognitive distance by (1) using fixed and variable abstractions that are both succinct and expressive, (2) maximizing the hidden part of the abstractions, and (3) using automated mappings from abstraction specification to abstraction realization (e.g., compilers). This can be summarized in an important truism about software reuse:

> For a software reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation.

This truism, along with others that arise later in the survey, are obvious and seemingly simple requirements on software reuse techniques that have proven difficult to satisfy in practice.

2. ORGANIZATION OF THE SURVEY

We partition the different approaches to software reuse into eight categories: high-level languages, design and code scavenging, source code components, software schemas, application generators, very high-level languages, transformational systems, and software architectures. They are presented in Sections 3 through 10, respectively, beginning with reuse techniques that rely on low-level abstractions such as assembly language patterns and progressing through higher level abstractions.

Each of the eight software reuse categories are discussed according to the following taxonomy:

Abstraction. What type of software artifacts are reused and what abstractions are used to describe the artifacts?

Selection. How are reusable artifacts selected for reuse?

Specialization. How are generalized artifacts specialized for reuse?

Integration. How are reusable artifacts integrated to create a complete software system?

Each section ends with a summary of the key taxonomic features for the category and a statement about its pros and cons. These summaries provide a convenient point of reference for comparing the different approaches to software reuse.

Although the categories and the taxonomy illustrate different approaches and issues in software reuse, they are not precise. The real software engineering technologies we will examine often have features that fit into several categories. For example, different features of the Ada language have relevance in four categories: High-Level Languages, Design and Code Scavenging, Source Code Components, and Program Schemas. Likewise, the relative importance of the four facets in the taxonomy differs among the different reuse categories. For example, application generators emphasize the specialization of a single highly abstracted artifact but typically do not require selecting or integrating artifacts. These imprecisions, however, should not detract from our goal, which is to explore the interesting issues in software reuse.

3. HIGH-LEVEL LANGUAGES

High-level languages such as C, Ada, Lisp, Smalltalk, and ML have not been treated in the literature as examples of software reuse. From the perspective of software developers prior to the existence of these languages, however, the goals and achievements for high-level languages have strong parallels to the current-day aspirations of software reuse researchers. Given their marked level of acceptance and success (e.g., a factor of 5 speedup in writing code [Brooks 1975]), it is interesting to examine high-level language technology in terms of software reuse.

3.1 Abstraction in High-Level Languages

Before high-level languages were developed, software development was done mainly with assembly languages. As developers discovered the inherent complexity of building large systems in this model, they began to search for more effective ways to express computation. The common implementation patterns used in assembly language programming became the primitive constructs in the next generation, high-level languages. Examples include iteration, branching, arithmetic expressions, relational expressions, data declarations, and assignment.

The reusable artifacts in a high-level language are assembly language patterns. A high-level language is at a level of abstraction above the assembly language artifacts. That is, high-level language constructs are *abstraction specifications*, whereas the corresponding assembly language artifacts are *abstraction realizations*.

For example, consider a conditional statement in a high-level language:

```
if (⟨expression⟩) then
  ⟨statements1⟩
else
  ⟨statements2⟩
endif
```

This reusable template is an abstraction specification that directly maps to an abstraction realization—the reusable assembly language pattern. Referring back to Figure 2, the *variable parts* in the abstraction specification are the ⟨*expression*⟩ and two ⟨*statements*⟩ slots. The *fixed part* of the abstraction specification corresponds to the semantic description of the conditional statement, given in the language reference manual:

> The *expression* is evaluated. If **true**, *execute* $statements_1$, otherwise $statements_2$.

The *hidden part* of the abstraction includes all of the assembly language details such as temporary data in the evaluation of the expression.

Programmers use only the fixed and variable parts of the abstraction specification. They never see the actual assembly language artifacts that are reused because the mapping from specification to realization is fully automated by the compiler.

3.2 Selection

Since there is a relatively small number of reusable artifacts (i.e., language constructs) in a high-level language, it is relatively simple for a programmer to *select* among them. The language reference manual and tutorial examples help a novice make the appropriate selections in a particular language. A programmer can typically master a high-level language in a matter of days or weeks.

3.3 Specialization

Most high-level language constructs are generalized constructs with parameterized slots. For example, conditional statements typically have parameterized slots for Boolean expressions and statements. Programmers *specialize* the generalized language constructs by recursively filling the parameterized slots with other language constructs of the appropriate type.

3.4 Integration

Programmers *integrate* statement-level constructs by recursive specialization as described in the previous paragraph. The larger, encapsulating constructs such as procedures, packages, or modules are integrated with a module interconnection language or with scope rules. At both the statement and module levels, the integration framework of a high-level language defines how individual constructs are composed to form a complete software system. That is, every language construct in a fully integrated program has a well-defined operational semantics that describes its effect on the run-time data and the run-time control flow [Loeckx and Sieber 1984]. (Module interconnection languages are addressed in more detail in Section 5.4.)

3.5 Appraisal of Reuse Techniques in High-Level Languages

From the perspective of today's software development technology, an analysis of high-level languages is almost trite. These languages are often the lowest level of abstraction used by software developers. The accomplishments, strengths, and weaknesses of high-level languages are well known. However, it is not widely recognized that high-level languages are examples of software reuse. Nor is it recognized that, in many ways, high-level language technology is a paragon of software reuse that researchers currently can only hope to emulate. For example, discovery of a new reuse technology that routinely offered a factor of 5 speedup in software development would be among the most significant software engineering achievements of the decade.

High-level language technology provides a good example of how abstraction impacts the effectiveness of software reuse techniques. Compared to using assembly languages, high-level languages are considerably more succinct and have a more natural form for expressing and reasoning about programs. In addition, programmers are largely unaware of the reuse that takes place. They use only the fixed and variable parts of the abstraction specification, while the mapping

Table 1. Reuse in High-Level Languages

Abstraction	The reusable artifacts in a high-level language are assembly language patterns. High-level language constructs serve as abstraction specifications for low-level assembly language patterns.
Selection	The semantics for each of a small number of constructs is described in a language manual. Experienced programmers easily commit the collection of constructs to memory, which facilitates selection.
Specialization	Language constructs are typically generalized with parameterized slots. Constructs are specialized by recursively filling the parameterized slots with constructs of the appropriate type.
Integration	Language constructs are integrated by recursive specialization, module interconnection rules, or scope rules. The operational semantics of the language defines the effect of integrating constructs on the run-time data and control flow.
Pros	Reusable assembly language patterns provide a factor of 5 speedup in development time compared to programming directly in assembly language because (1) high-level language constructs are succinct and natural, (2) compilers fully automate the mapping from abstraction specification to abstraction realization, and (3) programmers are completely isolated from compiler and assembly language details.
Cons	Due to the need for system design prior to coding, there is still a large cognitive distance between the informal requirements for a software system and its implementation. High-level languages are at a relatively low level of abstraction.

from specification to realization is fully automated by the compiler. Programmers do not have to examine or understand the internals of the compiler or the assembly language output, which completely isolates them from the details of the hidden and realization parts of the reuse abstraction.

The primary limitation of high-level languages as a reuse technology is the large amount of system design effort required prior to coding—high-level language programming comes late in the software development life cycle. Thus, there is a large cognitive distance between the informal requirements for a software system and its implementation in a high-level language. Table 1 summarizes the high-level language approach to software reuse.

4. DESIGN AND CODE SCAVENGING

Many programmers adopt an ad hoc, although effective, approach to reusing software system designs and source code. They scavenge fragments from existing software systems and use them as part of new software development. Experienced software developers are known to gain great leverage from reusing previous designs [Biggerstaff and Richter 1989].

The goal of design and code scavenging is to reduce the cognitive effort, the number of keystrokes, and therefore the amount of time required to design, implement, and debug a new software system. Scavengers copy as much as possible from analogous systems that have already been designed, implemented, and debugged.

4.1 Abstraction in Design and Code Scavenging

In design and code scavenging, the abstractions used by a software developer are mostly informal, and they often exist only in the mind of the developer. The abstractions are concepts about design and programming that a software developer has learned from experience. Formal education, where students are taught practical design concepts, provides another source of abstraction for scavengers.

The reusable artifacts in scavenging are source code fragments. In *code scavenging*, a contiguous block of source code is copied from an existing system. In *design scavenging*, a large block of code is copied, but many of the internal details are deleted while the global template of the design is retained. There is, of course,

a continuum between code scavenging with dense code blocks and design scavenging with sparse design templates.

A software developer creates an abstraction for an existing design or code fragment by remembering an abbreviated description. For example, the developer might remember the semantics of a stack and the location of some software that implements a stack. Then, if the developer needs a stack in a new software system, the existing implementation can be scavenged. In this case, the *abstraction realization* is the existing stack source code. The *abstraction specification* is the abbreviated description in the memory of the software developer.[3]

In code scavenging, the software developer is ultimately involved with all parts of the abstraction—the abstraction specification, the abstraction realization, and the mapping from specification to realization. Therefore, there is no *hidden part* of the abstraction. Initially the developer recalls an existing artifact from a mental abstraction specification. The mapping from specification to realization corresponds to the developer manually locating the existing artifact and modifying it for reuse. Modification and integration are performed at the realization level of the abstraction, which forces the developer to become intimately involved with details in the realization.

4.2 Selection

The abstractions a software developer has in his or her head can be thought of as a *library* of reusable designs and source code. While designing a new software system, the developer may notice similarities between the new system and abstractions in the *library*. The developer can directly reuse these abstractions in the new design, or the abstractions can be used as an *index* to locate existing source code for scavenging.

[3] I am being informal and taking great liberties with the notion of abstraction in this section.

4.3 Specialization

A programmer specializes a scavenged code fragment by manually editing it. The fragment is edited to resolve differences between the original context from where it was scavenged and the new context where it will be reused. For example, a scavenged stack implementation might push and pop integer elements, but a new implementation might need string elements. The programmer must thoroughly understand the lowest-level details in the code fragment to adapt it to its new context correctly.

4.4 Integration

Integrating a scavenged code fragment into a new context means that the programmer has to modify the fragment, the context, or both. For example, variable names in a scavenged code fragment may collide with existing variable names in the new system. This conflict can be resolved by either modifying the context or the fragment. Modifying the fragment corresponds to specialization, as described in the previous paragraph. The issues for modifying the context are essentially the same.

4.5 Appraisal of Reuse Techniques in Design and Code Scavenging

In ideal cases of scavenging, the software developer is able to find large fragments of high-quality source code quickly that can be reused without significant modification. In these cases, the payoff is high. The developer goes directly from an informal abstraction of a design to a fully implemented source code fragment. That is, the cognitive distance between the initial concept of a design and its final executable implementation is small.

In practice, the overall effectiveness of code scavenging is severely restricted by its informality. A programmer can only scavenge those code fragments he or she remembers or knows how to find. There is no systematic way to share fragments among many different programmers. Specialization and integration are ineffi-

Table 2. Reuse in Design and Code Scavenging

Abstraction	The reusable artifacts in scavenging are source code fragments. The abstractions for these artifacts are informal concepts that a software developer has learned from design and programming experience.
Selection	When a programmer recognizes that part of a new application is similar to one previously written, a search for existing code may lead to code fragments that can be scavenged.
Specialization	A programmer specializes a scavenged code fragment by manually editing it. The programmer must thoroughly understand the lowest level details of the code fragment in order to adapt it correctly to its new context.
Integration	Integrating a scavenged code fragment into a new context means that the programmer has to modify the fragment, the context, or both.
Pros	In ideal cases of scavenging, a software developer is able to adapt large fragments of source code without significant modification. In these cases, cognitive distance is small.
Cons	In the worst cases, a software developer spends more time locating, understanding, modifying, and debugging a scavenged code fragment than the time required to develop the equivalent software from scratch.

cient because they require a thorough understanding and direct editing at the realization level (source code). The programmer may introduce errors while modifying the fragment, so the validation, testing, and debugging must be repeated each time a code fragment is scavenged for a new context.

These limitations lead to the second truism of software reuse:

> For a software reuse technique to be effective, it must be easier to reuse the artifacts than it is to develop the software from scratch.

Efforts to extend the scope of code scavenging, especially across multiple programmers, often violate this simple constraint. For example, trying to scavenge unfamiliar code from a friend may result in spending more time locating, understanding, modifying, and debugging a code fragment than the time required to develop the equivalent software from scratch. Scavenging is inherently limited by the fact that developers are not isolated from the details in the abstraction realization level. They frequently work at the same level of abstraction while scavenging as they do when building software from scratch. Table 2 summarizes the design- and code-scavenging approach to software reuse.

5. SOURCE CODE COMPONENTS

McIlroy's [1968] *Mass Produced Software Components* introduced the notion of software reuse by proposing an industry of off-the-shelf source code components. These components were to serve as building blocks in the construction of larger systems. Given a large enough collection of these components, software developers could ask the question "What mechanism shall we *use*?" rather than "What mechanism shall we *build*?"

The nature of a reusable component technology strongly depends on the language in which it is implemented. The components proposed by McIlroy in 1968 were assembly language or Fortran subroutines and therefore emphasized reusable *functions*. Examples of collections of reusable functions include statistics libraries such as *SPSS* and numerical analysis libraries such as *IMSL*. Modern languages have a richer collection of program units, such as modules, packages, subsystems, and classes. The type of reusable components that can be written in these languages is not limited to functions but can also include data-centered artifacts such as *abstract data types* [Deutsch 1989; EVB Software 1985; Ichbiah 1983; Parnas et al. 1989]. That is, these languages make a clear distinction between control abstraction and data abstraction [Goguen 1986, 1989]. As an example of reusable data-centered components, Booch [1987] defines reusable Ada packages for 11 abstract data types: stacks, lists, strings, queues, deques, rings, maps, sets, bags, trees, and graphs.

He also gives reusable function-centered components for control abstractions such as sorting and searching.

The goal of off-the-shelf components is to reduce the cognitive effort, the number of keystrokes, and therefore the amount of time required to design, implement, and debug a new software system. The one-time cost of creating a reusable component is further amortized each time it is reused. Reusable components have proven fruitful in a few narrow areas such as numerical analysis, but there has been less success with collections of general-purpose components.

5.1 Abstraction in Source Code Components

Component reuse is analogous to code scavenging in that programmers copy existing source code artifacts into new software systems. However, reusable component technologies typically use more systematic techniques such as catalogs and libraries of components. In addition, reusable components are created and stored specifically for the purpose of reuse, whereas with scavenging the reused artifacts are extracted from software that was not written with reuse in mind.

A major challenge for researchers trying to implement large libraries of reusable components is to find concise abstractions for the components. Without abstractions, the user of a component library must examine the source code to determine what each component does. Although the source code is appropriate for the *abstraction realization* level, it is unreasonable to expect a user to search through a large library of source code to find an appropriate component. The library implementor must provide *abstraction specifications* that succinctly describe component behavior. The library implementor must also provide classification and retrieval schemes so users can efficiently search for specific components.

It is interesting to note that the best-known successes with component reuse have been in application domains with application-specific, "one-word" abstractions to describe components. These abstractions are universally understood by all programmers in the application domain. For example, in numerical analysis libraries, reusable components for *sine* and *matrix multiply* do not require detailed abstraction specifications. The names alone serve as complete abstractions for the users based on their countless hours studying high school and college mathematics. Another example is Booch's abstract data types (*stacks*, *lists*, *strings*, *queues*, *deques*, *rings*, *maps*, *sets*, *bags*, *trees*, and *graphs*), where the names serve as well-defined abstractions for most computer scientists. In both of these examples, the source code components are often accompanied by informal natural language descriptions, which provide more detailed abstraction specifications for the components.

As with code scavenging, the software developer reusing source code components is often involved with all parts of abstraction. Initially the developer uses the abstraction specification to locate an appropriate component. The mapping from specification to realization corresponds to the developer locating the component in the library and (if necessary) modifying it for reuse. Modification, if needed, and integration are performed at the realization level of the abstraction, which might force the developer to become involved with details in the component implementation.

5.2 Selection

The third truism of software reuse emphasizes a characteristic problem with reusable components:

> To select an artifact for reuse, you must know what it does.

When faced with a library of source code components, the user needs a level of abstraction that emphasizes *what* the components do as opposed to the source code that describes *how* it is done. Anna is an example of an Ada language extension for annotating components (i.e., Ada packages) with semantic descriptions

[Luckham and von Henke 1984]. Although Ada maintains a clear separation between the package declaration and the package implementation, the package declaration provides only syntactic information, which is not sufficient for describing its behavior and intended use. With Anna, it is possible to "understand how to use a package without inspecting its implementation in the private part and body" [Luckham and von Henke 1984, p. 123].

Anna, which stands for Annotated Ada, is a language for *formal annotations*. Annotations are predicates (i.e., Boolean-valued expressions) that express constraints on Ada language constructs such as data objects, types, subtypes, subprograms, and exceptions. For example, the following is an Ada subtype declaration for the even integers. The first line is the Ada declaration, and the second is an Anna annotation; it is impossible to express this subtype semantics in pure Ada [Luckham and von Henke 1984]:

```
subtype EVEN is INTEGER;
  --|where X : EVEN  = > X mod 2 = 0;
```

The next example is a simple Anna specification for the behavior of a *counter* package (also from [Luckham and von Henke 1984]). The two Anna annotations, which begin with --|, formally state that the value of a counter just after initialization will be 0 and that the value of the counter just after an increment will always be 1 more than its value just before the increment:

```
package COUNTERS is
  type COUNTER is limited private
  function VALUE (C : COUNTER) return NAT-
  URAL;
  procedure INITIALIZE (C : in out COUNTER);
  --| where out (VALUE (C) = 0);
  procedure INCREMENT (C : in out
  COUNTER);
  --| where out (VALUE(C) = VALUE(in C) + 1);
private
:
end COUNTERS;
```

In addition to component descriptions, a component library must provide software developers with techniques to locate components efficiently. Without such techniques, a developer must search through a large, monolithic collection of reusable components. This is analogous to searching for a book on a particular subject "in a public library that has no card catalog but has all books arranged alphabetically" [Embley and Woodfield 1987, p. 360]. This leads to the fourth and final truism of software reuse (which is a special case of the second):

> To reuse a software artifact effectively, you must be able to find it faster than you could build it

To address this problem, implementors of a component library can organize components with a classification scheme and then provide manual or automated retrieval techniques. The scheme for classification and retrieval is another level of abstraction over all of the components in the library. For example, the IMSL [1987] math library has a three-volume manual with components hierarchically classified by abstract computational or analytical capabilities. In addition, three different indexes serve as independent abstract interfaces to the library: a keyword index (KWIC), an ACM math software classification index, and an alphabetical index. The keyword and ACM classification indexes are both mappings from abstract descriptions (i.e., *abstraction specifications*) to reusable components in the library (i.e., *abstraction realizations*). About 1200 pages are used to describe and classify approximately 900 routines.

Automated tools for component selection may be necessary for component collections significantly larger than IMSL. An interesting small-scale example is the *netlib* system for automatically distributing mathematical software components via electronic mail [Dongarra and Grosse 1987]. Users' queries and requests are drafted according to a simple syntax and sent to an Internet address. A server at that address parses incoming messages and returns responses to queries or requested software components.

The netlib components are all public domain, and the netlib service is pro-

vided free of charge. In December, 1988, the netlib source code collection consisted of 51 different software packages (including Linpack, published ACM algorithms, Minpack, fft packages, and eigenvalue packages) and occupied about 75MB of storage. Although this system is limited to mathematical software and is relatively unsophisticated in terms of software classification and retrieval, it elicits images of an international distribution system for a wide range of reusable software artifacts.

5.3 Specialization

As with code scavenging, programmers can specialize reusable components by directly editing the source code. This approach, however, has the same drawbacks that were noted with code scavenging:

- Editing source code forces the software developer to work at a low level of abstraction. The effort required to understand and modify the low-level details of a component offsets a significant amount of the effort saved in reusing the component.
- Editing source code may invalidate the correctness of the original component. This eliminates the ability to amortize validation and verification costs over the life of a reusable component.

Implementors of reusable components can offer a more efficient approach to specialization with generalized components and construction-time parameters. Software developers specialize these components by setting parameters rather than directly editing source code. Well-known examples of this approach include parameterized macro expansions and Ada generics [Ichbiah 1983]. Parameterized components are a form of abstraction. For example, the Ada generic package specification represents a set of possible package realizations, or instantiations. The generic parameters correspond to the variable part of the abstraction specification. Mendal [1986] gives an in-depth example of a generic Ada package for sorting in which parameterization determines the type of elements that are sorted, the indexing scheme into a collection of elements, the partial ordering among elements, and so on.

5.4 Integration

Module interconnection languages provide a framework for integrating reusable components [DeRemer and Kron 1976]. Integrating reusable components into a software system is essentially the same as integrating components built from scratch. Module interconnection languages are an integral part of many programming languages such as Modula-3, Standard ML, and Ada, which makes these languages particularly good candidates for implementing reusable components. Readers not familiar with module interconnection languages might be interested in a survey by Prieto-Diaz and Neighbors [1986].

Naming and binding conflicts can present problems for the software developer integrating a reusable component into the context of a new system. Names imported into and exported from the component may clash or be incorrectly bound in the new system.

Ada provides mechanisms to overcome some of these naming problems [Ichbiah 1983]. The dot notation can be used to disambiguate name clashes. For example, if package P and Q both define functions named F, they can be uniquely identified as $P.F.$ and $Q.F.$ *Overloading* allows name clashes to be automatically disambiguated by the compiler on the basis of differences in the type signatures. And *synonyms* can be used in situations in which dot notation is too cumbersome or overloading becomes confusing.

The UNIX[4] *pipe* is another example of an integration framework for components [Kernighan 1984]. The reusable components in this paradigm are complete programs. Programs are integrated

[4] UNIX is a trademark of AT & T.

by sequentially "piping" the output from one program into the input of another program. For example, the UNIX command **who** is a program that outputs the list of persons logged onto a UNIX system, one per line. The UNIX command **lc** counts the number of linefeeds in an input string. Piping the output of **who** into **lc** produces a new program that counts the number of persons logged into a UNIX system:

```
who | lc
```

where the | designates a pipe.

5.5 Object-Oriented Variants to Component Reuse

Most of the abstraction, selection, specialization, and integration issues outlined earlier in this section also apply to object-oriented languages. The notion of *inheritance* from the object-oriented paradigm, however, offers a unique contribution to component reuse [Meyer 1989].

5.5.1 Inheritance and Subclasses

Class definitions in an object-oriented language such as Smalltalk are primarily a data abstraction mechanism. Inheritance and subclassing enhance the data abstraction mechanism by establishing a hierarchical relationship among classes and by allowing reuse to occur within the class hierarchy [Deutsch 1989; Liskov 1987]. For example, an *indexed collection* of items might be defined as a class (abstract data type) with a hidden storage representation and visible operations to *store* and *fetch* the ith item in the collection and to get the size of the collection. Subclasses of the index collection can reuse the storage representation and the fetch operation, while extending the type with new data and operations. For example, a programmer can define a *sequence* as a subclass of the indexed collection superclass, with an additional operator for concatenation. The programmer only has to write the concatenation operator for the sequence class definition since the data representation and fetch operation defined in indexed collection are reused.

Subclassing, therefore, corresponds to *specialization* of a superclass. That is, the superclass is a template that can be extended by a programmer to create a subclass. Everything in the superclass template is reused in the subclass [Lieberherr and Riel 1988]. The effort required to produce a subclass is proportional to the dissimilarity between the subclass and the superclass [Deutsch 1983].

Liskov [1987] discusses different forms of inheritance and the effect that each form has on the encapsulation of classes. In cases in which encapsulation is compromised for a class, reusability is likewise compromised. When the implementation of a class depends on implementation details in a superclass, the external dependency makes it difficult to understand and reuse the class.

Organizing classes into a subtype hierarchy helps the software developer locate and select reusable classes. Similar classes in a hierarchy are grouped close together. The path from the root of a class hierarchy down to a particular class is a path from a very general abstraction for the class down to more specific abstractions for the class. A developer looking for a reusable class navigates through the hierarchy until the closest match is located [Liskov 1987].

5.5.2 Multiple Inheritance

Some languages offer *multiple inheritance* as an extension to the inheritance mechanism just described. Multiple inheritance allows a class to have two superclasses, inheriting the data representations and operations from both [Stefik and Bobrow 1986]. Multiple inheritance is therefore a mechanism for merging multiple abstract data types into a single type.

There are several possible forms of multiple inheritance. The simplest is a disjoint union of two or more superclasses, where the data representations and the operations from the superclasses

do not interact in any way. In this case the new subclass reuses the properties of all of its parents without modifying their semantics. For example, the Smalltalk *read-write-stream* subclass is composed of multiple inheritance from the *read-stream* superclass and the *write-stream* superclass.

Another form of multiple inheritance is an intersecting union. In this case some of the data or operations from superclasses interact. For example, a *stack_set* class can be defined by multiple inheritance from a *set* superclass and a *stack* superclass. The *stack_set* maintains a stack and a set as (conceptually) a single data type. The semantics are defined so that the set always contains all objects in the stack, and as usual the set is an unordered collection containing no duplicates. Adding and removing items on the *stack_set* is allowed only through the *push* and *pop* operations. The *insert* and *delete* operations for *set* are disabled. The *push* and *pop* operations must be modified to update the set portion of the *stack_set* in addition to the original operations on the stack [Habermann et al. 1988].

In this example, modifying the *push* and *pop* operations in the *stack_set* detracts from the reusability of those operations since the software developer must manually modify them. There has been some research using declarative specifications that allow the software developer to merge operations from multiple superclasses almost as easily as merging the data [Kaiser and Garlan 1987, 1989].

5.6 Appraisal of Reuse Techniques in Source Code Components

Compared to code scavenging, reusable component libraries can be considerably more effective since components are written, collected, and organized specifically for the purpose of reuse. The most successful reusable component systems, such as the IMSL math library, rely on concise abstractions from a particular application domain. One-word abstraction specifications such as *sine* often allow a software developer to go directly from an informal requirement to a fully implemented and tested source code component. Thus, the cognitive distance between the informal concept and its final executable implementation is very small.

For components that do not have simple abstractions, more general specification techniques are required. Unfortunately most programming languages do not come with good abstraction mechanisms for describing component behavior and for doing classification and retrieval. And although some progress has been made using programming language extensions such as Anna, these formal specification languages are not without problems. The primary drawback is that nontrivial specifications are difficult to create and understand—specifications can often be as opaque as source code [Horowitz and Munson 1989]. This offsets the benefits of using them as abstraction specifications for reusable components; that is, it makes the cognitive distance relatively high.

Reusable components can be specialized either by editing the source code directly or with mechanisms such as the Ada generic. Generics provide a level of abstraction that isolates the software developer from many implementation details. Generics can also reduce the risk of inadvertently introducing errors when the component is reused. Using a formal specification language such as Anna, an Ada generic package can be verified to be correct for all legal instantiations of that package. Thus the user does not have to reverify the reused component.

Ada generics can only be parameterized with language constructs such as data types, data objects, and functions. In Section 6 we will see how greater degrees of specialization are possible by parameterizing components with higher level abstractions, such as space/time performance tradeoffs.

Creating a relatively complete and practical library of reusable components is a formidable challenge. Library implementors must have the theory, foresight, and means to produce a collection of com-

ponents from which software developers can select, specialize, and integrate to satisfy all possible software development requirements. This is currently possible to a limited degree for specific application domains that have a rich and thorough theoretical foundation, such as statistical analysis. General-purpose libraries, however, remain elusive for at least two reasons:

- The implementation characteristics and tradeoffs for data structures and computations are widely variable.
- Library size grows rapidly with respect to general-purpose component size.

Computer science provides general-purpose building blocks such as stacks and lists, but these do not always represent the dominant portion of the computation needed in large software systems [Booch 1987]. Yet even to construct an admittedly modest library of variations on 11 abstract data types (*stacks*, *lists*, *strings*, *queues*, *deques*, *rings*, *maps*, *sets*, *bags*, *trees*, and *graphs*) Booch [1987] had to create 501 parameterized components. In order to capture to a reasonable degree of tradeoffs in precision, granularity, range, and robustness, McIlroy [1968] envisioned a library with 300 different variations of the "lowly sine routine." Projecting from these examples, it appears that a library providing a versatile foundation for building software systems must be populated with a very large number of components.

With general-purpose components, bigger is better. A software developer who constructs a software system by selecting, specializing, and integrating 5 1000-line components is typically going be more effective than a developer using 1000 5-line components. Larger components are, however, more specific, and therefore, more of them are needed to populate a general-purpose library [Biggerstaff and Richter 1989].

For example, consider a programming language with 10 different types of statements. Ten different 1-line components can be built in this language, but using these 1-line components is equivalent to and as equally cost-effective as programming directly in the source language. A library of components of length 6 might be six times more cost effective to use, but there are 1,000,000 (10^6) possible components of length 6. In general there are 10^N candidate components for populating a library of N-line components in this model. It is therefore necessary for a library implementor to identify a small subset of large granularity components that can serve as building blocks for software systems. Table 3 summarizes the source code component approach to software reuse.

6. SOFTWARE SCHEMAS

Software schemas are a formal extension to reusable software components. Reusable components often rely on ad hoc extensions to programming languages to implement reuse techniques such as specification, parameterization, classification, and verification. With software schemas, however, these mechanisms are an integral part of the technology.

Although the intent of software schemas is similar to that of reusable components, the emphasis is on reusing abstract algorithms and data structures rather than reusing source code. To the software developer using reusable schemas, the source code representation will be much less prominent than the abstract notation for describing what the schema is intended to do, under what conditions it can be reused, and how to go about reusing it.

The PARIS system is a representative schema technology [Katz et al. 1989]. The reusable schema, or program templates, in the PARIS library are *instantiated* to produce source code. Each schema has a *specification* that includes the following:

- A formal semantics description of the schema (e.g., what the schema does),
- Assertions for correctly instantiating the schema (e.g., constraints on the variable parts of the schema), and

Table 3. Reuse in Source Code Components

Abstraction	Currently, the best abstractions for reusable components are domain-specific concepts, such as those found in the numerical analysis and statistical analysis packages. Specification languages such as Anna are general-purpose mechanisms for writing abstraction specifications for components.
Selection	Selection is easier in domain-specific libraries because components can be classified, organized, and retrieved using well defined properties of the domain. In a general-purpose component library, ease of component selection is dependent on the quality of the abstraction, classification, and retrieval schemes. Describing component behavior with a specification language can help software developers select among a small number of alternatives, but this approach is not suitable for a manual search in a large library.
Specialization	Generalized components with construction-time parameters represent the most effective approach to component specialization. Directly editing component source code is a less-controlled and less-efficient means of specialization. In some object-oriented languages, inheritance offers another means of reuse with superclass specialization.
Integration	Module interconnection languages typically provide the framework for integrating components. Naming and name binding are important module interconnection issues in component reuse since reusable components are constructed independently of any particular context.
Pros	Components are written, tested, and stored specifically for the purpose of reuse. Some application domains, such as numerical analysis, are particularly well suited to reusable component techniques since they have well-defined component abstractions. In these cases, cognitive distance is small.
Cons	Components without well-defined abstractions must be described with natural language or specification language descriptions. These descriptions can often be as difficult to understand as source code, thereby increasing the cognitive distance. Also, general-purpose component libraries will have to be very large, making them difficult to build and use.

- Assertions for the valid use of instantiated schema (e.g., preconditions and postconditions for the source code).

A software developer begins an interaction with PARIS by giving a *problem statement*—a formal set of computational requirements. PARIS then assists in finding a schema that can be instantiated to satisfy the problem statement provably.

6.1 Abstraction in Software Schemas

Knuth's [1973] book, *The Art of Computer Programming*, provides a library of abstract descriptions for many basic computer science algorithms and data structures. Programmers can informally use these abstractions when writing source code. The goal of schema researchers is to capture abstractions *formally* at this level for their schema libraries [Standish 1984].

In the software schema approach, the algorithms and data structures captured by the schemas are the reusable artifacts. The *abstraction specification* for a schema is a formal exposition of the algorithm or data structure, whereas the *abstraction realization* corresponds to the source code produced when the schema is instantiated. The *fixed part* of the abstraction specification formally describes the invariant computation or data structure of the schema, for example, sorting with respect to a particular partial ordering or a queue of an unspecified element type. The *variable part* of the abstraction specification describes the range of options over which a schema can be instantiated, such as the overflow length for a queue.

With schemas, the variable parts of the abstractions can be more general, or higher level, than is possible with parameterized components such as the Ada generic. Parameters in generics are limited to a subset of programming language constructs, whereas in schemas they can range over other software properties such as space/time efficiency tradeoffs. An even more complex type of parameterization is possible with nested

schemas, where parameters in a schema can be instantiated with other schemas.

With software schemas, software developers select, specialize, and integrate schemas at the abstraction specification level. In addition, the mapping from the specification to the source code realization (i.e., schema instantiation) can be automated, in which case developers are isolated from the details in the abstraction realization level. This is analogous to high-level languages and in contrast to code scavenging and reusable components.

As with reusable components, a major challenge to implementors of software schemas is to find suitable abstractions and abstraction representations for the schemas. Rich and Waters [1989, p. 313] suggest that the limited success in component and schema reuse "stems not from any resistance to the idea, nor from any lack of trying, but rather from the difficulty of choosing an appropriate formalism for representing components." Different algorithms and data structures have diverse properties, and this diversity must be expressible in schema representations. For example, *matrix add* is best defined by a step-by-step *operational* semantics description of *how* the computation is accomplished, whereas a *deadlock-free scheduler* is best described by a *declarative* set of constraints on *what* the computation must accomplish [Rich and Waters 1989].

In an attempt to provide a diverse base for describing reusable artifacts, the Plan Calculus[5] in the Programmer's Apprentice combines ideas from flowcharts, data abstraction, logical formalisms, and program transformations [Waters 1985]. A flowchart representation is used to describe algorithmic aspects of a schema such as control flow and data flow. Flowcharts can be annotated with logical preconditions and postconditions to capture the declarative aspects of the schemas. Empty boxes within a flowchart represent the variable part of the schema abstraction. A software developer instantiates such a schema by filling in the empty boxes with nested schemas.

PARIS uses temporal logic assertions and quantified logic predicates in schema preconditions, postconditions, and invariants [Katz et al. 1989]. The following example is a PARIS description for an *insertion module* that inserts an item into an ordered list. The APPLICABILITY CONDITIONS, or preconditions, state that the list is ordered before the insert. The RESULT ASSERTIONS, or postconditions, state that the list is ordered after the insertion and that the new item exists in the list.

```
:
APPLICABILITY CONDITIONS (PRE)
leq: D1 × D1 → boolean              --less than or
                                    --equal to
forall n | n element_of node :
  n → next != NULL = >
  leq(n → data, n → next → data)

RESULT ASSERTIONS (POST)
leq: D1 × D1 → boolean              --less than or
                                    --equal to
equal: D1 × D1 → boolean
forall n | n element_of node :
  n → next != NULL = >
  leq(n → data, n → next → data)

exists n | n element node :
  equal(n → data, newdata)
:
```

In Goguen's [1986] Library Interconnection Language (LIL), axiomatic *theories* with varying degrees of formality are attached to schema parameters, and *views* describe how schema instantiation satisfies the theories. The following example is a theory for partially ordered sets (POSET):

```
theory POSET is
  types ELT --elements in the set
  functions leq --less than or equal to
  vars E1 E2 E3 : ELT
    axioms
    (E1 leq E1)
    (E1 leq E3 if E1 leq E2 and E2 leq E3)
    (E1 = E2 if E1 leq E2 and E2 leq E1)
end POSET
```

[5] Using the terminology of Rich and Waters [1989], *plan* corresponds to the notion of software schema in this paper.

A schema implementor can attach this theory to any schema that is a partially ordered set. The implementor uses a view to bind the type ELT and the function LEQ in the theory to the corresponding constructs in the schema.

Other examples of schema formalisms include Volpano and Kieburtz's [1985, 1989] Software Templates with an ML-like functional notation and Embley and Woodfield's [1987] abstract data type schemas with a regular expression notation.

6.2 Selection

Reusable hardware components, such as TTL integrated circuits, are widely used in hardware design. When lamenting about the limited successes of software reuse, software engineerings often draw the analogy between software artifacts and hardware components. The analogy is particularly relevant when comparing the techniques for selecting TTL components to the techniques for selecting software components and schemas.

In Texas Instruments [1985] *The TTL Data Book*, approximately 1000 MSI/LSI TTL components are classified in a four-level hierarchy. The selection branching at each level in the hierarchy varies between 2 and 12 (the average being 5.6). Following is an example classification path for a frequency divider component (SN74LS57), with the hierarchy level preceding the selection made at that level; moving to deeper levels in the hierarchy reduces the number of candidate components until only one remains:

(1) Counters
(2) Frequency dividers/multipliers
(3) 60-to-1 frequency divider
(4) Low-power Schottky technology

If the user finds the simple abstractions presented at the branches insufficient for making a choice, more detailed abstractions are available for the current candidates. For every component in the data book, there is a detailed *exposition* presenting different properties of the device at different levels of abstraction. Techniques for describing components include natural language descriptions, input/output tables to describe the device function, logic-level schematics (gates, inverters, flip-flops, etc.), transistor-level schematics, timing diagrams to describe temporal relationships, plots of linear characteristics such as delay time versus device temperature, and the range of normal operating conditions. The more detailed abstractions are typically relevant as the selection process proceeds to deeper levels in the hierarchy.

Analogous to *The TTL Data Book*, practical techniques for selecting reusable software components or schemas can use hierarchical classification and multiple forms of exposition. For example, the IMSL mathematical library is a small-scale software library that has successfully used these techniques. It is interesting to note that the IMSL library and *The TTL Data Book* both contain approximately 1000 components.

General-purpose software libraries might contain several orders of magnitude more candidates than the special-purpose IMSL library. In addition, software libraries will probably evolve rapidly, and therefore it may not be practical to maintain them with books or manuals [Biggerstaff and Richter 1989]. Clearly, reuse technologies with a large number of reusable artifacts must provide users with powerful techniques for selecting artifacts. These selection techniques must address three important subproblems (analogous to *The TTL Data Book*):

- *Classification* of the artifacts in the library
- *Retrieval* of artifacts from the library
- *Exposition* of information necessary to understand, compare, choose, and use the artifacts

The following sections address these issues in detail.

6.2.1 Classification

Prieto-Diaz describes a classification scheme for software artifacts [Prieto-Diaz

1989; Prieto-Diaz and Freeman 1987]. In this scheme, artifacts are classified by two top-level *descriptors* and three lower-level *facets* within each descriptor:

Functionality. Describes the function the component is intended to perform. The three lower-level facets of functionality, ordered by decreasing relevance to reusability, are

(1) **Function**. Operation performed.
(2) **Object**. Type of data object on which the operation is performed
(3) **Medium**. Larger data structure in which the data object is located.

Environment. Describes the context for which the component was designed. The three lower-level facets of environment, ordered by decreasing relevance to reusability, are

(1) **System type**. Type of subsystem for which the component was designed.
(2) **Functional area**. Application dependent activities.
(3) **Setting**. Application domain.

The following is an example of a classification for a file compression routine that was designed for a database management system:

Function:	compress
Object:	files
Medium:	disk
System Type:	file handler
Functional Area:	DB management
Setting:	catalog sales

The classification scheme allows only predefined values in the facets. The predefined values for a particular facet are related by a *conceptual closeness graph*. An automated *evaluation system* can show the user a collection of conceptually similar artifacts.

There are practical limitations to this particular classification scheme. For example, it uses *function* as the primary classification facet, which emphasizes the reuse of functional components and ignores reuse of abstract data types. With this approach, the insert operation for a queue would be classified close to the insert operation for a set and not encapsulated in the queue data type. This classification scheme does, however, demonstrate how classification techniques can be applied to software components, or schemas, analogous to hardware components.

6.2.2 Retrieval

PARIS is notable for its sophisticated assistance in locating software schemas [Katz et al. 1989]. When presented with a *problem statement*, PARIS attempts to find a schema that satisfies the problem statement. So that PARIS can match schemas to problem statements, each schema in the library includes the following information:

Parameterization for the schema which is a list of *abstract entities* that must be supplied in order to use the schema.

A schema *specification* consisting of

- Assertions about the substitutions for the abstract entities,
- Preconditions for execution of an instantiation of the schema,
- Invariants for execution of an instantiation of the schema,
- Postconditions for execution of an instantiation of the schema,
- Temporal relations for the execution of an instantiation of the schema.

Proof of correctness for the generic schema specification.

This information is a detailed *abstraction specification* for a schema. Compared to the syntactic specification of an Ada generic, the semantic assertions that accompany a PARIS schema provide a more precise definition of how to reuse an artifact correctly. That is, a PARIS schema has a more narrow interface.

A *problem statement* is a set of computational requirements. Its form is similar to a schema specification in that there are preconditions and postconditions, but it contains no parameterized sections as do schemas. When presented with a problem statement, PARIS first constructs a list of preliminary candidate

schemas by finding all schemas in the library such that

- The preconditions of the problem statement are a superset of the schema preconditions
- The postconditions of the problem statement are a subset of the schema postconditions

The user then selects schemas from the candidates and attempts to find instantiations that will satisfy the schema assertions. Optionally, PARIS can attempt to instantiate a candidate schema automatically. A theorem prover is used to verify that a particular instantiation satisfies the problem statement and the assertions given for the schema.

With PARIS, users perform schema searches at the abstraction specification level and are therefore isolated from the implementation details at the abstraction realization level. PARIS does not, however, take advantage of the higher level abstractions understood by software developers. For example, a user cannot simply specify that a *queue* implementation is needed but rather must give an elaborate logic specification of the queue semantics. This monolithic, low-level abstraction representation is necessary for the theorem prover but is not ideal for human users.

Also, the state of the art for theorem provers is not sufficiently advanced for practical use. Finding a schema that satisfies a problem statement requires proof that the schema is logically consistent with the problem statement, which in general is well beyond the scope of current-day theorem-proving technology [Rich and Waters 1988].

6.2.3 Exposition

In *The TTL Data Book*, a diverse collection of device abstractions are presented to help users understand and select candidate components. Biggerstaff believes that finding an analogous, diverse collection of expositions for reusable software artifacts is the "fundamental operational problem that must be solved in the development of any reuse system" [Biggerstaff and Richter 1989, p. 6]. These abstractions must be useful to both the user and the tools for automated retrieval.

Following is a list of exposition techniques from *The TTL Data Book* (in boldface) and some suggestions for analogous software counterparts:

Natural language descriptions. This is certainly useful for informal schema descriptions and search techniques such as keyword lookup. But machine understanding of natural language is beyond current capabilities, which limits automated searching techniques [Rich and Waters 1988].

Input / output tables to describe the device function. Logic assertions can be used to express the semantic function of a schema (e.g., preconditions and postconditions). They are not particularly useful for large, complex schemas due to the complexity of the corresponding logic expressions.

Logic-level schematics. Data flow and control flow diagrams such as those used in Plan Calculus resemble the graphical abstractions found in hardware logic-level schematics. These diagrams have isomorphisms to logic formalisms that are appropriate for machine manipulation.

Transistor-level schematics. This corresponds to the source code level of a software schema. Although this is at too low a level for most of the reuse process, it is occasionally useful in the final stages of selection, specialization, and integration. For example, with TTL chips, transistor-level schematics can be consulted to determine whether or not unused pins require external connections. Knuth's WEB, a system for making source code more comprehensible, is one possible approach for presenting the schema implementation level [Bentley 1986].

Timing diagrams to describe temporal relationships. Temporal logic is suitable for expressing some of the time-dependent characteristics of software.

Plots of characteristics such as delay time versus device temperature. The characteristics of computations within a schema can be expressed in terms of the upper and lower bounds on time/space requirements as a function of input size.

Biggerstaff argues that hypertext systems can be used to manage a large collection of interrelated component or schema abstractions [Biggerstaff 1987; Biggerstaff and Richter 1989]. So-called "webs of information" allow the user to move easily between different abstract views of reusable artifacts.

Seer is an example of the hypertext approach [Latour and Johnson 1988]. It uses a graphical representation of the Booch [1987] taxonomy for reusable components. Each component in the library has a graphical *information web* that provides access to all of the available information on the component. This information includes formal, informal, syntactic, semantic, graphical, and textual descriptions in three different categories:

Specification information. What the component does.

Usage information. How to use the component correctly.

Implementation information. How the component is implemented.

All three types of information are useful in selecting schema. In addition, usage information applies to schema specialization and integration, and implementation information is occasionally useful for detailed integration decisions.

6.3 Specialization

Schemas are typically specialized either by *substituting* language constructs, code fragments, specifications, or nested schemas into parameterized parts of a schema or by *choosing* from a predefined enumeration of options.

It can be argued that in either case, specialization is a continuation of the selection process. Both selection and specialization are convergent steps toward a single, complete instantiation. In both cases, the software developer may use a formal specification to make the convergent step. In the case in which the software developer specializes by substituting a nested schema for a schema parameter, then specialization is exactly the same as selection since the nested schema must be *selected* from the library before it can be inserted.

PARIS uses both parameter substitution and enumerated choices for specializing the variable parts in the schema abstractions. The variable part of the abstraction is separated into two sections:

- *Nonprogram entities*, such as abstract data types, subtypes, ranges, functions, and variables.
- *Program sections*, which can be handwritten source code fragments or nested schema instantiations.

The values substituted for abstract nonprogram entities must come from the concrete nonprogram entities listed in the user's problem statement. The possible values may be constrained by the applicability conditions of the schema. Therefore, specialization of nonprogram entities in a schema corresponds to matching actual entities in the problem statement to abstract nonprogram entity parameters in the schema. It is truly a continuation of the selection process.

The instantiation of the *program sections* in a PARIS schema is best described as a substitution. Arbitrarily complex source code fragments are substituted into the program sections, restricted only by the assertions in the section conditions for the schema. If the source code fragment is derived by nested selection of another schema, however, it is a continuation of the selection process.

In LIL [Goguen 1986] and in Embley and Woodfield's [1987] reusable abstract data types, a schema can have multiple implementations. A schema implementor can create radically different implementations that satisfy the same semantic interface but that exhibit different performance characteristics. The perfor-

mance profile for the schema then becomes a parameter in the variable part of the schema abstraction. A software developer specializes such a schema by selecting from the enumerated performance options.

For example, consider a schema for an indexed sequential list. This schema maintains an ordered list of *elements*, so that (1) all elements of the list can be accessed in sequential order, and (2) elements can also be accessed individually with an index key. Sequential access is implemented with a linked list. The schema has two different implementations of the indexed access so a software developer reusing the schema can choose between two different space/time performance profiles. The first implementation provides faster access at the expense of storage space by maintaining an explicit index structure with pointers into the linked list. The second implementation exhibits slower access time and requires less space by omitting the index structure and doing indexed access with a linear search through the list. With this implementation, insertion and deletion are faster since there is no index to update when items are added to or removed from the list.

This example illustrates how schemas with multiple implementations address an important problem in software reuse: the tradeoff between generality and performance. The implementor of a reusable artifact typically tries to generalize the artifact, to make it suitable for a broad range of applications. Increased generality often translates into increased functionality, and increased functionality often translates into performance compromises in the implementation. A schema can, however, offer multiple implementations with a range of functionality and performance options. A software developer can choose the implementation that most closely matches the required functionality and performance. The schema is conceptually one large, generalized artifact, but in reality there can be many specialized implementations that do not pay a performance penalty for unnecessary functionality.

In some cases, schema implementations may perform significantly worse than very specialized code. If necessary, performance "hot spots" can be identified with conventional monitoring techniques during the execution of the final system, and then the hot spots can be tuned, replaced, or hand coded [Booch 1987].

6.4 Integration

The simplest approach to schema integration is to use the module interconnection language of the schema implementation language. For example, if schema instantiation produces Ada package code, an instantiated schema can be treated as a conventional Ada package. Integration would be based on the syntactic integration rules of Ada.

More sophisticated schema integration techniques rely on semantic specifications. In addition to the syntactic constraints on composition provided by conventional module interconnection languages, formal or informal semantic specifications serve to narrow the component interfaces to include only correct syntactic and semantic composition. For schemas with complex interfaces, it is particularly important to have a clear description of both the syntactic *and* the semantic interface [Latour and Johnson 1988].

LIL (introduced in Section 6.1) is an example of a schema technology that uses semantic specifications to compose schemas [Goguen 1986]. Parameters in Ada generics are extended with semantic constraints that guarantee an instantiation to be semantically correct. For example, consider a generic *Sort* package, parameterized by the type of elements it sorts. In LIL, the implementor of this package could put a semantic constraint on the parameter so the element type satisfies the *Partial Ordered Set* theory. This would guarantee that an instantiated *Sort* package would always be able to make a semantically meaningful "greater than or equal to" comparison on the elements.

In LIL, a distinction is made between *horizontal* and *vertical* schema composi-

tion. In vertical composition, higher-level abstractions are created. For example, a package implementing a low-level *List* abstraction might be used to implement a higher-level abstraction such as a *Sort* algorithm. In horizontal composition, schemas are instantiated by specialization. For example, a generic *Sort* schema instantiated with a partially ordered set of *Labels* results in a package that is still at the *Sort* level of abstraction. Thus, horizontal composition corresponds to schema composition using nested schemas.

6.5 Appraisal of Reuse Techniques in Software Schemas

Compared to reusable source code components, reusable schemas place a greater emphasis on the abstract specification of algorithms and data structures and place less emphasis on the source code implementation. This shift in emphasis helps reduce the cognitive distance between the informal requirements of a system and its executable implementation by isolating the software developer from the source-code-level detail. In cases in which the mapping from the schema abstraction specification to the source code realization is fully automated, the cognitive distance is reduced by one level in the software development life cycle—up from the level that describes *how* an algorithm or data structure is implemented to the level that specifies the semantics of *what* is implemented.

With TTL components, low abstraction-level workhorses such as the AND gates and INVERTERS offer leverage to hardware developers. By analogy, reusable software schemas for stacks, lists, trees, and so on offer leverage to software developers. It is the higher level TTL device abstractions, such as counters, multiplexers, and even CPUs, however, that give hardware designers the greatest leverage. These devices typically have a large ratio of internal gate counts to external pin count (i.e., large function-to-interface ratio). Similarly, software schemas with high-level abstractions and narrow interfaces allow system designers to select, specialize, and integrate powerful, high functionality artifacts with relative ease.

Unfortunately, with software systems we do not have many universal abstractions above the stack, list, tree, and so on level. Therefore, the semantics of higher level schema abstractions are often expressed with logic formalisms and specification languages. In a sense these formalisms become the programming language for software developers using the schemas. Thus, the challenge for implementors of a schema technology is to find abstraction formalisms that are natural, succinct, and expressive. Members of the PARIS project describe this challenge:

> Our experience in building the PARIS prototype indicates that transforming abstract logic expressions into precise implementation definitions is quite difficult. It is a nontrivial task to construct a system that is able to provide a friendly interface to the user and at the same time produce correct and efficient input to the theorem-proving mechanism [Katz et al. 1989].

Table 4 summarizes the software schema approach to software reuse.

7. APPLICATION GENERATORS

Application generators operate like programming language compilers—input specifications are automatically translated into executable programs [Cleaveland 1988]. Application generators differ from traditional compilers in that the input specifications are typically very high-level, special-purpose abstractions from a very narrow application domain.

By focusing on a narrow domain, the code expansion (ratio of input size to output size) in application generators can be one or more orders of magnitude greater than code expansion in programming language compilers. Levy [1986] reports that application generators have been used to create production quality commercial software at the equivalent rate of 2000 lines of source code per day.

Whereas software schemas reuse algorithm and data structure designs, application generators go one step further and reuse complete software system designs. In application generators, algorithms and

Table 4. Reuse in Software Schemas

Abstraction	The schema approach emphasizes the reuse of algorithms, abstract data types, and higher level abstractions. Formal semantic specifications are typically used to express schema abstractions. Examples include data flow, control flow, pre, post, and invariant logic expressions, regular expressions, temporal logic assertions, and axiomatic theories.
Selection	Classification and search schemes and multiple abstraction exposition forms, analogous to *The TTL Data Book*, offer promise as practical selection techniques for reusable software schemas. The issue of scale will make large schema libraries more difficult to use than the hardware analog. Automated assistance such as hypertext databases and theorem provers can help for schema selection.
Specialization	Schemas are typically specialized by filling in parameterized slots. Examples include insertion of arbitrary language constructs, choosing from an enumerated set of abstractions, or instantiating parameters with nested schemas. Schema specialization may involve characteristics not typically associated with parameterized software, such as explicit choices among run time performance alternatives.
Integration	Schema composition often involves both the semantic and syntactic schema interface. This results in a narrower interface when compared to syntactically typed languages such as Ada, so more semantic errors can be detected at design time.
Pros	When schemas are based on formal specifications, automated tools can be used for selection, specialization, and integration. This, in combination with the fact that software developers work at a higher level of abstraction, can reduce the cognitive distance between the requirements and the implementations of algorithms and data structures.
Cons	Formal specifications for schema abstractions can be large and complex. Even with automated tools it can be difficult for software developers to locate, understand, and use schemas. This complexity serves to increase the cognitive distance, thereby offsetting some of the advantages of using higher level abstractions.

data structures are automatically selected so the software developer can concentrate on *what* the system should do rather than *how* it is done. That is, application generators clearly separate the system specification from its implementation [Cleaveland 1988]. At this level of abstraction, it is possible for even nonprogrammers familiar with concepts in an application domain to create software systems [Horowitz et al. 1985].

Application generators are appropriate in application domains where

- Many similar software systems are written,
- One software system is modified or rewritten many times during its lifetime, or
- Many prototypes of a system are necessary to converge on a usable product.

In all of these cases, significant duplication and overlap results if the software systems are built from scratch. Application generators generalize and embody the commonalities, so they are implemented once when the application generator is built and then reused each time a software system is built using the generator.

7.1 Abstraction in Application Generators

The abstractions presented to the user of an application generator typically come directly from the corresponding application domain. The way these abstractions are presented is dependent on the application domain [Cleaveland 1988]. Different applications need different models of data and computation in the generated programs, and these different models are amenable to different exposition techniques [Levy 1986]. Examples include

- Textual specification languages (application-oriented languages, fourth-generation languages, etc.)
- Templates

- Graphical diagrams
- Interactive menu-driven dialogs
- Structure-oriented interfaces

The reusable artifacts in an application generator are the global system architecture, major subsystems within this global architecture, and very specific data structures and algorithms. The *abstraction specification* for an application generator describes all parts of the application domain that can be modeled in a generated system. The *abstraction realization* level corresponds to the executable systems produced by the generator. The *fixed part* of the abstraction specification corresponds to those parts of the application the user cannot modify during the generation process; the *variable part* corresponds to those parts of the application the user can customize.

For example, with a parser generator the user specifies (in the variable part of the abstraction specification) the grammar of the language to be parsed. The generator might, however, have LR(1) as a fixed part of the abstraction specification, in which case the user cannot change the parsing algorithm. The combination of the fixed part and the range of the variable part for an application generator determines the range of systems that can be generated; it is referred to as the *domain width* or *domain coverage* of the generator [Cleaveland 1988].

Abstraction is used effectively in application generators. The software developer works exclusively at a high-abstraction level. Typically the *hidden part* of the abstraction covers *all* of the implementation details in the generator, and the user does not view, understand, or modify the generator output. This is analogous to conventional high-level language compilers where the user treats the input specification as the lowest level of abstraction for software development. The workings of the compiler and the object code produced are black boxes.

It is interesting to compare application generators to software schemas. Both approaches use diversely and highly parameterized reusable artifacts. With application generators, the product is a complete, executable system; with software schemas a smaller unit, such as a module or a subsystem, is produced. The primary distinction is that schemas are abstractions for software implementation artifacts such as algorithms and data structures, whereas application generators are abstractions for the semantics of a particular application domain. In comparing the effectiveness of the two approaches, an important question to ask is whether or not it is easier for the user to deal with application domain abstractions rather than computational abstractions. Successful reuse applications, such as the IMSL mathematical library, provide some evidence that users are comfortable dealing with domain abstractions.

7.2 Selection

Selecting an application generator to solve a particular problem means choosing an application generator whose *domain coverage* includes the requirements of the problem statement. For situations in which one or more application generators are known to exist, the selection process may be easy. Currently, however, there are only a small number of application generators in a few limited domains, and their availability is not always well known [Cleaveland 1988]. Also, application generators are often highly specialized, limiting their domain coverage [Biggerstaff and Richter 1989]. As a result, application generators currently provide a very small coverage for software development in general.

As more and more application generators are built, we might expect to see libraries of application generators. Such a library would be similar to, but at a much higher level of abstraction than, the libraries of reusable components or schemas described in Section 6.2. Ideally, an application generator library would cover a broad enough range of applications to be considered a source of general computation for common applications. Classification, search, and exposition in

this library could be done in terms of domain characteristics rather than algorithmic or data requirements. Initial steps in the direction of an application generator library are described in Section 10.

7.3 Specialization

Specialization is the primary task of a software developer using an application generator. A software developer specializes an application generator by providing an input specification. Implementors of application generators have used a wide variety of techniques for specialization. To demonstrate this diversity, the following sections describe several application generators and the abstractions used for specialization.

7.3.1 "Conventional" Application Generators

Application generators are sometimes associated with only a narrow collection of business applications. For example, Horowitz et al. [1985] in "A Survey of Application Generators" discuss only generators for business-oriented, data-intensive applications. Application generators from this domain often cover high-level report generation, data processing, and data display techniques. The abstractions presented in these systems include

Database management based on a particular data model, such as hierarchical or relational. Specialization consists of specifying the database schema (types, fields, relations, etc.) for an application.

Textual report generation, which is often described by a nonprocedural (declarative) language and is based on associative data retrieval.

Graphical report generation, which uses a similar input specification as textual report generation but produces graphs, histograms, bar charts, scatter plots, pie charts, and so on.

Database manipulation for batch or interactive modification, processing, and restructuring of the database contents. Specifications can include constraints on particular data items, constraints between data items, access control constraints, and the way of viewing the contents of the database.

The following example is a simple input specification to a report generator and a sample output from the generated system [Horowitz et al. 1985]:

```
report
file SALES
title 'UNITS SOLD PER CUSTOMER'
  list
    by CUSTOMER
    across YEAR
    sum (UNITS)
    rowtotal
    columntotal
  where YEAR in 1980 . . . 1982
end report
```

	UNITS SOLD PER CUSTOMER			
CUSTOMER	1980 UNITS	1981 UNITS	1982 UNITS	TOTAL
A	2344	3455	1234	7033
B	234	1111	3221	4566
C	412	555	1212	2179
TOTAL	2990	5121	5667	13778

7.3.2 Expert System Generators

Expert systems are software systems that embody and apply expert knowledge to solve problems in a particular domain. Whereas conventional software system development codifies algorithmic problem-solving-knowledge, expert system development codifies the subjective problem-solving knowledge of experts [Hayes-Roth 1983]. By analogy, if application generators for conventional software systems reuse common algorithms and data structures, then application generators for expert systems can reuse common expert problem-solving methods [Krueger 1987]. This idea of generating expert systems is advocated by expert system development tools such as MOLE,

SALT, and SEAR [Marcus 1988; McDermott 1986].

Experts in different areas may use similar problem-solving techniques even though their actual knowledge is quite different. For example, an auto mechanic and a medical doctor may use a similar diagnostic method. Expert system generators reuse implementations of these generic problem-solving methods. The fixed part of an expert system generator's abstraction has a particular problem-solving method, whereas the variable part (by which it is specialized) consists of the expert knowledge from a particular area of expertise. The input specification is typically derived through an interactive dialog between the expert system generator and a domain expert. This process is referred to as *knowledge acquisition.*

For example, MOLE is an expert system generator that can be used for a wide variety of diagnostic tasks, including medical diagnosis, car repair, and production line troubleshooting [Eshelman 1988]. The fixed problem-solving method reused in MOLE consists of the following steps:

(1) Determine all *hypotheses* that explain the *symptoms.*
(2) If there is more than one hypothesis that explains a symptom, then identify what *data* are necessary to differentiate the hypotheses. Data for differentiation can serve the following roles:
    - Rule out causality between a hypothesis and a symptom (i.e., hypothesis implies the symptom in some cases, but not this case).
    - Rule out validity of the hypothesis (i.e., hypothesis implies the symptom, but hypothesis is not true in this case).
    - Rate the relative likelihood of the hypotheses causing the symptom (i.e., certain hypotheses are more likely to cause symptoms).
    - Rate the relative likelihood of the hypotheses (i.e., certain hypotheses are more likely to be true).
(3) Collect the data identified in step 2 and differentiate the hypotheses.
(4) If step 3 uncovers new *symptoms* go to step 1.

The input specification given to MOLE by a diagnostic expert is expressed in terms of domain-specific symptoms, hypotheses, and data. Each piece of knowledge in the input specification fits into one of the first two steps listed above. From the input specification, MOLE produces an executable expert system to diagnose problems in the expert's domain.

SEAR provides a more general framework for creating and reusing problem-solving methods [McDermott 1988]. The eventual goal is to have a library of commonly used problem-solving methods that can be selected and integrated to produce a wide range of different expert system generators.

7.3.3 Parser and Compiler Generators

Parser generators and *compiler–compilers* are probably the best-known examples of application generators. These systems have grown out of the well-developed compiler theory and the ubiquitous need for compilers in computer science.

The primary abstractions used in parser generators are regular expressions for generating lexical analyzers and context-free grammars for generating parsers. For example, Lex [Lesk and Schmidt 1979] and Yacc [Johnson 1979] are a pair of generators for producing parsers. Lex deals with lexical analysis; Yacc addresses parsing.

Parser generators and compiler–compilers reuse the knowledge gained through years of research and design experience in the compiler domain, including the general theories behind lexical analysis, parsing, syntactic and semantic error analysis, and code generation. This knowledge is embedded in the design and implementation of a generic compiler framework that is reused each time a compiler is generated. The framework includes such things as the generic parsing

algorithm and the parse tree data structures.

The specification language used as input to Lex is a regular expression language. An input specification describes the legal tokens (identifiers, keywords, constants, etc.) in the language to be compiled. For example, the following regular expression describes all tokens that have a letter for the first character, followed by zero or more letters, digits, or underscores:

```
[a – zA – Z][a – zA – Z0 – 9_]*
```

The specification language used as input to Yacc is a context-free grammar. The input specification to Yacc defines the legal syntactic structure of the language to be compiled. *Actions* can be associated with productions in the grammar. Actions can do static semantic checking, code generation, and so on. Whenever a construct in the grammar is recognized by the Yacc-generated parser, the associated actions are executed. Actions are expressed in C and are incorporated into the generated parser.

The following is an example Yacc grammar rule for a conditional statement in an imperative language:

```
stmt : IF '(' cond ')' stmt
     | IF '(' cond ')' stmt ELSE stmt
     ;
```

7.3.4 Structure-Oriented Editor Generators

Structure-oriented editors are a cross between text editors and compilers. A structure-oriented editor has knowledge of the syntax and semantics of a programming language and interactively assists users to construct programs in that language. Typically, structure-oriented editors store and manipulate programs in the form of *abstract syntax trees* (which are similar to parse trees) rather than text strings. This allows the editor to detect syntactic and semantics errors incrementally while a user is editing a program.

Structure-oriented editors for different languages have many similarities, including the abstract syntax tree management and the user interface. The primary differences between different structure-oriented editors are the syntax and semantics of languages. Structure editor *generators* capitalize on this by capturing the commonalities in the fixed part of an abstraction specification while allowing the language syntax and semantics to be expressed in a variable part of the abstraction specification. Gandalf [Staudt et al. 1986], Synthesizer Generator [Reps and Teitelbaum 1984], and Mentor [Donzeau-Gouge et al. 1984] are examples of structure editor generators.

Typically, four things are specified to generate an editor:

(1) **Tokens in the language**. Regular expressions are typically used to define tokens in the language, analogous to those shown for compiler–compilers.

(2) **Abstract syntax of the language**. Context-free grammars are typically used to describe the constructs in a language. The grammars specify constraints on the abstract syntax trees. The following is an example of two rules in an abstract syntax specification:

```
IF :: expression statements statements
⋮
statements :: STATEMENT_BLOCK | WHILE |
              CASE | IF | ...
```

(3) **Concrete syntax of the language**. The concrete syntax of a language describes how to map an abstract syntax tree into a textual representation for the user interface. This is referred to as prettyprinting or unparsing.

(4) **Semantics of the language**. The techniques for specifying language semantics vary among the different editor generator systems. The Cornell Synthesizer Generator uses *attribute grammars* to declare constraints among different constructs in the abstract syntax tree [Reps and Teitelbaum 1984]. Gandalf uses daemons, written in an imperative

database manipulation language. Daemons are associated with items in the abstract syntax and are triggered by different editing activities [Staudt et al. 1986]. ACL is a semantics specification language that combines the strengths of attribute grammars and daemons [Kaiser 1989].

7.3.5 Others

Cleaveland reports on application generators for user interfaces, switching software in telephone systems, test drivers, hardware interprocess communication, CAD design process descriptions, and printed circuit board layouts [Cleaveland 1988]. Levy describes application generators for designing telephone company "loop plants" and for allocating and managing construction and engineering work [Levy 1986]. Horowitz notes that spreadsheet generators like Visicalc are successful because they model a common type of application with a high-level, domain-specific abstraction [Horowitz and Munson 1989].

7.4 Integration

In cases in which an application generator can produce a complete executable system from one specification, integration is not an issue. Some application generators, however, produce subsystems that in turn require composition. For example, Lex and Yacc are used in combination to produce parsers.

Software developers using an application generator typically think in terms of the abstractions at the input specification level not in terms of the software produced as output. Therefore, it may be inconvenient or impractical for the software developer to use a conventional module interconnection language to integrate the output from an application generator. It is better if composition is expressed in terms of the higher-level abstractions visible at the input specification level. For example, with Lex and Yacc the integration done is through the abstract concept of tokens, which are emitted by the Lex lexer and consumed by the Yacc parser.

7.5 Software Life Cycle with Application Generators

The software development paradigm with application generators is significantly different from the conventional software development life cycle. This section examines the software life cycle for application generators and its impact on cognitive distance.

7.5.1 Using Application Generators

The leverage offered by application generators can be shown by comparing conventional software development to software development with application generators [Horowitz et al. 1985]. A simple version of the conventional software development life cycle might consist of the following steps:

System (or requirements) specification. The desired system functionality is expressed in terms of concepts (objects and operations) in the application domain. This phase is focused on *what* the system should do and avoids the issues of *how* the functionality is implemented.

Architectural design. The high-level system organization is expressed in terms of the implementation technology. This includes subsystem decomposition into modules, module interconnection structure (such as control and data flow), and correspondence to the system specification. This phase identifies *what* the modules should do, not *how* they are implemented.

Detailed design. Details are specified for the implementation of modules and their interconnection based on the architectural design. This phase begins to address *how* modules are implemented.

Coding. Source code is written according to the detailed design. This phase expands in full detail *how* each module operates.

Testing. The system is tested to verify

that the "right system is built" and that the "system is built right" [Zave 1984]. That is, testing serves two purposes: (1) validate the correspondence between the system specifications and the intent of the design, and (2) verify that the code satisfies the system specifications.

Modifications made in any phase are propagated down through the other phases. This is true both of changes made during the initial system development and of changes made during the evolution (i.e., maintenance) of the system.

With application generators, much of the conventional life cycle is automated. Software developers construct or modify only the *system specifications*. The *architectural design* and *detailed design* are embodied as reusable artifacts in the application generator. *Coding* is automated. An application generator takes the system specifications as input and produces executable code as output. Software developers using an application generator do not have to test that the generated code satisfies the system specification, because (theoretically) the implementors of the application generator verify that the generator always produces code that is consistent with the input specifications. The software developers need only validate the match between the input specifications and the intentions of their design. With application generators, evolution of a system is carried out by modifying the system specifications. The application generator automatically propagates new specifications through the conventional life cycle phases in exactly the same way the original system is generated.

Thus, the powerful leverage afforded by application generators comes from specifying systems directly in terms of domain concepts, thereby avoiding the low-level details of implementing source code. The cognitive distance is very small between the informal requirements of a system and the specification of that system in terms of application-specific abstractions. Compared to conventional software development, application generators reduce the cognitive distance for the software developer by moving much of the development effort into the hidden and automated parts of the generator.

7.5.2 Building Application Generators

Even in situations in which an application generator is not available, it may be advantageous to build an application generator to produce one software system [Cleaveland 1988; Levy 1986]. It is not intuitively obvious how building an application generator to generate one software system can be economically advantageous. As described earlier, however, constructing an application generator is appropriate when many similar software systems are written, when one software system is modified or rewritten many times during its lifetime, or when many prototypes of a system are necessary to converge on a usable product. The last two cases suggest situations in which application generators could be created as part of the development of a single software product.

The primary economic justifications for the "up-front" expense of building an application generator are (1) the relative ease of using domain-specific specifications and (2) ability to do rapid iterative prototyping. Taking these two factors into account, the cost of building an application generator is rapidly amortized during its life.

Levy [1986] presents economic models for determining the optimal distribution of effort in building a software system with an application generator. He shows that it is often most effective to spend most of the effort up-front constructing a powerful, highly productive application generator, thus requiring less time prototyping with the generator.

Levy [1986] refers to the process of building an application generator as *metaprogramming*. Metaprogramming requires expertise in both the application domain and in software system construction. Cleaveland [1988] describes the process as follows:

Recognizing an appropriate domain. Recognizing when the domain of inter-

est is suitable for application generator technology. Domains that are amenable to generator technology will have implementations with recognizable patterns at the source code level, have natural control and data abstractions, contain automatable transformations, and have dispersed information that can be consolidated. Domains with a well-understood theory, such as compilers, are easiest for application generator construction.

Defining domain boundaries. Deciding what aspects of the domain should be included in the generator (i.e., setting the domain coverage). Increasing the domain coverage allows the generator to handle more problems but typically makes the generator less efficient and harder to use.

Defining underlying abstraction. Identifying the abstraction presented to the user of the application generator. Common abstractions include sets, directed graphs, trees, formal logic systems, and computational models such as finite-state machines and spreadsheets. The 80/20 rule, where not everything in a domain falls into a clean computational model, sometimes applies [Levy 1986]. In this case, *escapes* must be provided so that the 20% can be programmed directly into the implementation technology.

Defining variant and invariant parts. Deciding what aspects of the abstraction are under control of the user and which parts are fixed.

Defining specification input. Defining the way in which the user specifies each instance of a generated program. As mentioned earlier, options include textual specification languages (application-oriented languages, fourth-generation languages, etc.), templates, graphical diagrams, interactive menu-driven dialog, and structure-oriented editing.

Defining products. Implementing the generator ("the easy part" [Cleaveland 1988]). Creating the set of programs that will generate the final code from a concise input specification.

Stage [Cleaveland 1988], SSAGS [Payton et al. 1982], and Draco [Neighbors 1989] are examples of systems that assist in building application generators. They rely on generator technology to build generators and are therefore application generator generators. Details of this approach are presented in Section 10, which deals with the reuse of system-level architectures.

7.6 Appraisal of Reuse Techniques in Application Generators

Application generators are practical and attractive when high-level abstractions from an application domain can be automatically mapped into executable software systems. This capitalizes on the users' understanding of semantics from the application domain rather than requiring users to derive domain semantics repeatedly from first principles in conventional software development technology. By eliminating many of the design and implementation steps found in the conventional software life cycle, application generators significantly reduce cognitive distance.

Standards can be captured in the fixed part of the abstraction for an application generator. For example, users may prefer a standard user interface for all generated systems rather than having to learn a new interface for each system built. Also, in application domains where there are established standards, it is easier to implement the standard in a single application generator than to have to interpret and implement the standard repeatedly [Cleaveland 1988].

A difficult challenge for implementors of application generators is defining the optimal domain coverage and the optimal distribution of domain concepts into the fixed and variable parts of the abstraction. Implementors must attempt to predict in advance the functionality and variability that software developers using the application generator will want.

A limited number of application generators are available, many of which have narrow domain coverage. As a result, application generators do not exist for most software development problems.

Table 5. Reuse in Application Generators

Abstraction	Abstractions come directly from the application domain. These high-level abstractions are mapped directly into executable source code by the generator.
Selection	Application generator libraries have not received much attention in the literature. The parallel between software schemas and application generators suggests, however, that library techniques could be used to select among a collection of application generators.
Specialization	Application generators are specialized by writing an input specification for the generator. Due to the diversity in application domain abstractions, the techniques used for specialization are also widely varied. Examples include grammars, regular expressions, finite-state machines, graphical languages, templates, interactive dialog, problem-solving methods, and constraints.
Integration	Application generators do not require integration techniques when a single executable system is generated. In cases in which a collection of generators produce a collection of subsystems, composition is best done in terms of domain abstractions.
Pros	Since high-level abstractions from an application domain are automatically mapped into executable software systems, most of the conventional software development life cycle is automated. This significantly reduces cognitive distance.
Cons	Because of the limited availability of application generators, many of which have narrow domain coverage, it is often difficult or impossible to find an application generator for a particular software development problem. It is difficult to build an application generator with appropriate functionality and performance for a broad range of applications.

Application generators offer both advantages and disadvantages with respect to the efficient execution of generated systems. One advantage is that complex optimizations can be designed and built once in the generator and incorporated into all generated systems. For major efficiency problems in an application domain, a significant optimization effort in the application generator can be amortized over all generated programs. Also, application generators can offer a range of functionality and performance options, similar to schemas with multiple implementations. On the negative side, increasing the generality of an application generator often requires compromises in time or space efficiency. Table 5 summarizes the application generator approach to software reuse.

8. VERY HIGH-LEVEL LANGUAGES

As the name suggests, *very high-level languages* (VHLLs) are an attempt at improving on the successes of conventional high-level languages (HLLs). High-level languages have constructs such as iteration and relational expressions for their abstraction specifications, and these specifications are compiled into assembly language realizations. Similarly, very high-level languages allow software developers to create executable systems using constructs that are considered high-level specifications relative to HLL languages. Because of this, VHLLs are also known as *executable specification languages*.

Developing software with VHLLs is very much like developing software with HLLs. Both VHLLs and HLLs provide a syntax and a semantics for expressing general-purpose computation. Both approaches use compilers that accept source code programs as input, make tests for syntactic and static semantic correctness, and create executable programs if a certain threshold of correctness is met. HLL source code may be an order of magnitude more succinct than the corresponding assembly language implementation, and likewise VHLL source code may be an order of magnitude more succinct than corresponding HLL implementations.

Very high-level languages resemble application generators in that high-level abstraction specifications are automatically translated into executable systems.

Application generators, however, use application-specific abstractions, whereas VHLLs use application-independent, general-purpose abstractions. With application generators, generality is sacrificed in order to capture much higher level abstractions in the specification language. Therefore, VHLLs have the advantage of being generally applicable to software development, whereas application generators have the advantage of being more succinct and powerful in the relatively small number of cases in which they are applicable.

This distinction between VHLLs and application generators exemplifies the tradeoff between *generality* and *leverage* in software reuse technologies [Biggerstaff and Richter 1989]. Typically, the more general a reuse technology is, the more effort is required to implement systems with it. For example, the following technologies are listed by increasing leverage and decreasing generality: assembly language, HLLs, software schemas, VHLLs, and application generators. The goal of VHLL research is to maximize the leverage offered by higher levels of specification without sacrificing computational generality.

The primary concern in VHLLs is not efficiency in program execution but rather efficiency in implementing and modifying programs. The SETL language designers use the slogan "slow is beautiful" to emphasize this point [Kruchten et al. 1984]. Similarly, in the early days of their development, HLLs were often considered too slow to be of practical value [Naur and Randell 1968]. But the increased software development productivity offered by HLLs, in addition to compiler optimizations and faster hardware, quickly made HLLs the norm for software development. It is possible that VHLLs, which are considered too inefficient by today's standards, will follow a similar course of evolution.

Looking ahead to Section 9, *transformational* systems use VHLLs as the basis for rapid prototyping and validation of software systems, then a series of human-guided transformations is applied to produce an efficiently executable system. This approach can be seen as an interim measure until it is possible to automate the transformation fully (i.e., compilation) into efficiently executable code.

8.1 Abstraction in Very High-Level Languages

VHLLs diverge from a trend we have seen thus far in the survey. The reuse approaches described in earlier sections all used abstractions that lie in the conventional software development life cycle. Higher level abstractions allow lower level artifacts to be reused. For example, HLL constructs are abstractions for assembly language code fragments; code scavenging and reusable components use abstractions for HLL system designs and code fragments; schemas are abstractions for algorithms and data structures, and application generators use abstractions from the application domain being modeled. Contrary to this trend, VHLLs typically have mathematical abstractions that are not widely used in the conventional software development life cycle. Implementors of a VHLL try to find a mathematical model that is both executable and effective for doing software development.

The reusable artifacts in VHLLs are analogous to those in HLLs. In HLLs, the reusable artifacts are embodied in the compiler mappings from language constructs to assembly language patterns. Similarly, VHLLs capture reusable HLL implementation patterns in their compilers [Cheng et al. 1984]. The VHLL constructs serve as *abstraction specifications* for reusable artifacts, whereas the output from the compiler corresponds to the *abstraction realization* level.

VHLL constructs have *fixed* and *variable parts*, similar to HLL constructs. For example, in the functional expression F(X), both F and X are variable parts of the function abstraction specification. Different functions can be substituted for F, and different expressions can be substituted for X. The fixed part of the function abstraction is given by the language

semantics for functional evaluation: Apply function F to expression X. Although the programmer must understand the semantics, they are fixed and cannot be modified.

As with HLL compilers and application generators, VHLLs effectively use abstraction for software reuse. Software developers use only the fixed and variable parts of the abstraction specification. The mapping from specification to realization is completely automated by the compiler, which isolates software developers from the hidden and realization parts of the reuse abstraction. The following three examples demonstrate the type of abstractions that have been used in VHLLs.

8.1.1 SETL

SETL is a VHLL based on set theory [Doberkat et al. 1983; Dubinsky et al. 1989; Kruchten et al. 1984]. Therefore, sets are the most important abstraction, defining the character and semantics of the language. With a small number of primitives, all mathematical operations can be succinctly expressed in set-theoretic terms. The data types in the language are either atomic types (such as integers, reals, and Boolean values) or composite types from set theory. The two composite data types are sets and tuples:

Sets. An *unordered* finite collection of distinct SETL objects (atoms, tuples, and sets).

Tuples. An *ordered* finite sequence of SETL objects.

Sets and tuples can be nested heterogeneously and to any depth.

Sets and tuples are formed either by enumeration of their *members*, as in

```
{1, 2, 3},
```

or with *set-formers*, which have the general form

```
{⟨exp⟩ : x1, ..., xn | ⟨cond⟩}.
```

The ⟨**exp**⟩ designates the values in the set, as $\mathbf{x_1}$ through $\mathbf{x_n}$ vary over all values satisfying the condition ⟨**cond**⟩. For example,

```
{2 * x : x | 0 < x < 50}
```

is the set of even numbers between 0 and 100. The set-former is an example of the power of reuse in SETL. Through reusable code *stubs*, the SETL compiler or interpreter can generate arbitrarily complex data objects from a succinct set-former specification.

The usual set operations are defined in SETL: union, intersection, difference, and power set. Higher level operations such as set iterators are also provided in the language. For example, the following denotes the subset of S whose members satisfy the predicate C(x):

```
{x in S | C(x)}
```

Maps in SETL correspond to functions in set theory. A map is a set of length-2 tuples, where the first element in each tuple is in the domain of the function and the second element is in the range. There are several high-level retrieval operations that can be applied to maps. For example, if F is a map, then F(X) yields the second component of the unique pair in F whose first component is *X*. Likewise, if F contains more than one tuple whose first component is X (this is legal), then F(X) is undefined, but F{X} yields the set of all second components whose first component is X. The notation F[S], where S is a set, returns the set of all elements in the combined ranges of every element in S.

Many common abstract data types can be implemented trivially in SETL. For example, SETL operations for insertion and deletion from the ends of tuples allow direct implementations of stacks and queues. The concatenation operation allows tuples to be used as lists.

8.1.2 PAISLey

In PAISLey, abstractions from set-based functional programming and communicating asynchronous processes are combined in a single VHLL [Zave and Schell 1986]. The set-based aspects of the lan-

guage are similar to SETL and will not be discussed here.

With PAISLey, software developers can define a collection of asynchronous parallel processes. The processes communicate via *exchange functions*. An *exchange function* consists of a pair of function calls, one in each of two different processes. During execution, when the two processes both reach the evaluation of their half of the exchange function, the argument of one function is passed as the return value to the other and vise versa (i.e., they exchange their arguments). This simple extension to the functional semantics allows parallelism to be directly modeled in the language.

Real-time constraints are another high-level abstraction in PAISLey. Timing constraints allow the static and dynamic execution of a system to be evaluated. Software developers can attach timing constraints to functions, expressed as lower bounds, upper bounds, distributions, or random values. Static inconsistencies and dynamic failures are detected and reported.

A powerful feature of the PAISLey execution environment allows incomplete programs to be executed. If an undeclared function is called during execution, the user is prompted for a value to use in place of the evaluation. Declared but undefined functions can be handled in several ways:

- In default mode, a distinguished value will always be returned, depending on the type of the return value declared. For example, 0 is returned by default for integers.
- In random mode, a pseudorandom value is returned.
- In interactive mode, the user is prompted for the return value.

8.1.3 MODEL

MODEL is a declarative constraint language, whose abstract model of computation is based on the simultaneous solution to a collection of constraint equations [Cheng et al. 1984; Prywes and Lock 1989]. Data flow and control flow are automatically determined by the compiler, not explicitly specified as in imperative HLLs.

A MODEL program contains data declarations and consistency equations. The MODEL compiler does data flow analysis to determine if the consistency equations can be uniquely satisfied, and to find the order of computations that will assign consistent values to all of the data objects.

For example, the first of the following equations specifies that the final value of an account balance must be equal to the initial value of balance minus the debit total plus the credit total for that account. The equation for credits specifies equality with the old value for A, and debits is equal to the old value of B minus 10% of credits:

```
new.bal = old.bal − debits + credits;
credits = old.A
debits = old.B − (.1*credits)
```

The compiler detects that values for the debit total and the credit total must be computed before the new balance can be calculated and that credits must be computed before debits. The appropriate code is generated.

The MODEL compiler checks for *completeness* and *consistency*. Completeness requires that all variables used are also defined. Consistency requires that no circularities exist in the constraints and that there are no type mismatches.

8.2 Selection

With VHLLs, selection can take place on two different levels:

- A software developer selects a VHLL that is appropriate for a particular application. This is analogous to selecting an application generator.
- A software developer selects among the VHLL constructs during application programming. This is analogous to selecting constructs while programming with a high-level language.

The former is guided by the latter. That is, a software developer selecting a VHLL for a particular application makes the decision by comparing how easy it is to program the application with the different VHLLs. The language will have a very strong influence on how a problem is solved. For example, SETL and PAISLey encourage software developers to think about problems in terms of sets of objects and functions on those sets, whereas MODEL encourages thinking in terms of data objects with static interobject constraints. The best language for a particular application provides the smallest cognitive distance from initial concept to executable implementation.

8.3 Specialization

VHLLs have parameterized language constructs, or templates, that are specialized by recursively substituting other language constructs. For example, the functional template F(⟨x⟩) can be specialized by substituting another functional expression for ⟨x⟩:

```
F(G(2))
```

A less conventional type of specialization is found in a class of VHLLs known as *wide-spectrum languages*. Wide-spectrum languages contain both VHLL constructs and HLL constructs. The low-level constructs are present to accommodate efficiency concerns. Since low-level specifications are more difficult to read and modify, software developers should program at this level only as a refinement, or specialization, of stable high-level designs [Kruchten et al. 1984; Liu and Paige 1979].

SETL is an example of a wide-spectrum language. The following high-level SETL code can be specialized into a more efficient low-level form. This example creates the subset of positive integers and the subset of negative integers from a set S [Liu and Paige 1979]:

```
POS := {X element S | X > 0},
NEG := {X element S | X < 0},
```

Executing this code, however, requires two scans over set S. The following low-level SETL code is more difficult to read, but it requires only a single scan over set S during execution:

```
POS := nl;
NEG := nl;
(forall X element S)
  if X > 0 then
    POS with X,
  elseif X < 0 then
    NEG with X;
end forall X;
```

SETL also allows software developers to specialize the way language constructs are implemented at run time (for the sake of efficiency). For example, the default run-time implementation of SETL sets is a hash table. For sets that are only subject to union and intersection operations, however, a bit string implementation is more efficient. Ideally, the compiler or interpreter would automatically make the optimal implementation choice, but in general this is beyond the scope of current technology. These types of specializations must be guided by software developers. Human-guided transformations will be discussed in more detail in Section 9.

8.4 Integration

PAISLey and other side-effect-free functional languages simplify function integration because of *encapsulated computation*: Computation within a function is only influenced by its input parameters and return values from the functions it calls, and the only influence a function has is through its return value. Control flow and data flow are unified, which makes it easier for a software developer to reason about function integration.

MODEL and other declarative languages simplify integration even more by completely eliminating the issue of control and data flow at the programming level. In MODEL, the order in which the constraint equations are written is irrelevant since the compiler performs data flow analysis to determine a correct (and close to optimal) sequencing of operations [Tseng et al. 1986].

Just as source code components written with HLLs can be reused, software developers can also reuse VHLL components. Declarative languages like MODEL simplify the integration of reusable VHLL components. Two or more components can be merged without concern for the run-time ordering of computations since this is done automatically by the compiler. To merge components, a software developer only needs to specify the relationships (constraints) between data objects in the different components [Kaiser and Garlan 1987; Tseng et al. 1986].

8.5 Appraisal of Reuse Techniques in Very High-Level Languages

VHLLs use high-level mathematical abstractions suitable for general-purpose software development. The goal of VHLL implementors is to find abstractions that are more natural and expressive than the abstractions in HLLs. As a result, VHLL programs can be an order of magnitude more succinct than corresponding HLL programs. VHLLs are not, however, as powerful as application generators since application generators use domain-specific abstractions, which can be at a much higher level of abstraction.

Like HLLs and application generators, VHLLs have compilers to map abstraction specifications directly into executable realizations. This isolates the software developer from the hidden and realization parts of the abstraction, so the software developer is largely unaware of the reuse that takes place. The lowest level of detail that software developers work with are the high-level abstractions in the VHLL, so the cognitive distance between the initial concept of a system and the executable implementation is relatively small.

The first validated Ada compiler was built using the VHLL SETL [Kruchten et al. 1984]. The experiences of that project demonstrate the strengths and limitations of current-day VHLL technology. Although it was possible to prototype the compiler rapidly, it was far too inefficient for production use. Also, the SETL Ada compiler was built by the SETL language design group, which does not answer the question of whether or not the set theoretic foundation of SETL is a suitable abstraction for the general population of software developers, building a wide range of applications.

Zave notes that our notions of *specification* and *implementation* are floating rather than fixed [Zave and Schell 1986]. Although these two levels of program abstraction will always exist (with the specification level always at a higher level of abstraction than the implementation level), both levels have been gradually and simultaneously moving higher. For example, the implementation level of today resembles the specification level of 25 years ago. The goal of VHLL research is to continue this trend by making executable specification languages a viable alternative for implementing software systems. Table 6 summarizes the very high-level language approach to software reuse.

9. TRANSFORMATIONAL SYSTEMS

With transformational systems, software is developed in two phases:

(1) Software developers describe the semantic behavior of a software system using a high-level specification language.
(2) Software developers then apply *transformations* to the high-level specifications. The transformations are meant to enhance the efficiency of execution without changing the semantic behavior.

The two phases make a clear distinction between specifying *what* a software system does and the implementation issues of *how* it will be done [Zave 1984].

The first phase is equivalent to using a VHLL. Software developers create an executable system in a language that has a relatively small cognitive distance from the developer's informal requirements for the system [Balzer 1989]. The second phase, however, is not present in the

Table 6. Reuse in Very High-Level Languages

Abstraction	The abstractions used in VHLLs can be viewed as specifications when compared to HLLs. For this reason VHLLs are sometimes referred to as executable specification languages. VHLLs typically have mathematical abstractions, such as set theory or constraint equations.
Selection	Selection can take place at two levels: (1) selecting a VHLL that is most appropriate for a particular application and (2) selecting the language constructs that best represent the application.
Specialization	VHLLs have parameterized language constructs that are specialized by recursively substituting other language constructs. Another form of specialization is found in wide-spectrum VHLLs, where the run-time representation of language constructs can be specialized to provide better performance.
Integration	Function integration in languages such as PAISLey is simplified by side-effect-free encapsulated computation. Control flow and data flow are unified and localized to function composition, function arguments, and return values. Declarative languages such as MODEL simplify integration further with order-independent specifications and compiler-generated control flow and data flow.
Pros	High-level mathematical abstractions are automatically mapped into executable systems. VHLLs are suitable for general-purpose software development. The VHLL abstractions can be more natural and expressive than HLL abstractions, and VHLL programs can be an order of magnitude more succinct than HLL programs. Thus the cognitive distance for application development can be significantly less using VHLLs rather than HLLs.
Cons	The run-time performance of systems written with VHLLs is typically poor. In wide-spectrum languages, performance can be improved by using low-level constructs, but this increases cognitive distance. It is also not clear whether high-level mathematical abstractions are suitable for software development by the average programmer.

pure VHLL paradigm. The goal in this phase is to produce an executable system that satisfies the high-level specification and that also exhibits performance comparable to an implementation in a conventional HLL. The transformation phase can be thought of as an interactive, human-guided compilation. Human intervention is necessary because issues such as automatic algorithm and data structure selection are beyond current compiler technology.

A transformation is a mapping from programs to programs [Partsch and Steinbruggen 1983]. Application of a transformation T to a program P can be expressed as

$$P \times T \rightarrow P',$$

where P' is the result of the transformation and is semantically equivalent to but more efficient than P [Cheatham 1989]. A transformational *development history* is the sequence of transformations applied during the creation of a software system. A development history can be expressed as:

$$P_0 \times T_0 \rightarrow P_1$$
$$P_1 \times T_1 \rightarrow P_2$$
$$\vdots$$
$$P_N \times T_N \rightarrow P_{N+1}$$
or
$$((\ldots(P_0 \times T_0) \times T_1) \ldots \times T_N) \rightarrow P_{N+1},$$

where each P_{i+1} is a more efficient but semantically equivalent implementation of P_i.

9.1 Abstraction in Transformational Systems

Transformational system programming languages often have higher level abstractions than VHLLs. The execution efficiency of these higher level abstractions is often too poor to be directly executed in VHLLs. Transformational systems can, however, afford to use them because software developers apply transformations that enhance their efficiency.

Since the first phase of programming with transformational systems is similar to programming with VHLLs, this sec-

tion concentrates primarily on the second phase—the transformations. The following three sections describe three different kinds of reusable artifacts in the transformation phase: prototypes, development histories, and transformations.

9.1.1 Prototypes

Prototyping and its associated benefits are not unique to transformational systems. Transformational systems are, however, unique in the way prototypes are reused. In conventional software development, a prototype serves to validate the requirements of a system and to identify the critical implementation issues, but typically the prototype is a throwaway system. In this case software developers reuse the informal experience gained in designing and implementing the prototype. With transformational systems, however, rather than throwing the prototype away it can be reused for two other purposes: (1) a formal specification of the system and (2) as the basis for applying a sequence of transformations that transform the formal specification directly into an efficiently executable implementation [Cheatham 1989].

9.1.2 Development Histories

One of the key benefits of the transformational approach is that software developers perform maintenance and evolution on the specification of the system, not on the implementation. This is advantageous since information in a specification is often localized and independent, whereas information at the implementation level is often widely dispersed and tightly interdependent for the sake of efficiency. It is much easier to understand and modify the information before it has been dispersed throughout an implementation [Balzer 1989].

When a software developer makes modifications at the specification level, transformations have to be applied again to produce a new efficient implementation. When modifications to the specification are small, significant portions of the previous development history can often be reused. PADDLE is a transformational system that stores the development history as a sequence of applied transformations [Balzer 1989; Fickas 1985]. All or part of a previous development history can be *replayed* after a specification is modified.

A problem with PADDLE is that it does not encode the rationale behind the sequence of transformations. A PADDLE development history is essentially a program that defines a sequence of transformations, but the design decisions that went into writing the program are not saved. As a result, PADDLE development histories are difficult to understand and modify.

Glitter is a transformational system that addresses this problem. The design decisions that a software developer makes while applying a sequence of transformations are explicitly encoded in *rules* [Fickas 1985]. These rules provide a level of abstraction for the development history. The rules are the *abstraction specification*, and the sequence of transfor-mations correspond to the *abstractionrealization*. Rules make development histories easier to read, understand, reuse, and modify. Glitter can automatically map design decisions into an appropriate sequence of transformations. Thus, a software developer can specify *what* needs to be accomplished by the transformations, and Glitter will figure out *how* to do it. Therefore, Glitter rules isolate the software developer from the hidden and realization parts of the design history abstraction. Glitter rules are described in more detail later.

9.1.3 Transformations

A transformation is a mapping from syntactic patterns of code into functionally equivalent, more efficient patterns of code. A transformation has a *match pattern*, *applicability conditions*, and a *substitution pattern*. The match pattern is a template of code that defines what to search for in the initial program, and the

substitution pattern defines a pattern of code that replaces the constructs in the transformed program. The applicability conditions define semantic constraints on when the transformation should be applied. For example, the following template provides an implementation for inserting an element in a queue [Cheatham 1989]; the first line is the match pattern, and the remainder of the template is the substitution pattern (there are no applicability conditions in this example):

```
Insert $element Into $queue == >
  begin
    $queue.count := $queue.count + 1;
    $queue.list[$queue count] := $element;
  end
```

$element and $queue are *pattern variables* that represent arbitrary syntactic code fragments. At transformation time, these code fragments from the initial program are inserted verbatim in the corresponding pattern variables in the substitution pattern.

Many transformations are reusable. For example, the following are common identity transformations on predicate specifications:

```
NOT NOT $predicate == > $predicate
$predicate AND $predicate == >$predicate
```

9.2 Selection

Glitter uses expert system technology to help software developers select from a collection of reusable transformations. As described earlier, Glitter *rules* are abstractions that explicitly describe what a transformation or sequence of transformations will accomplish. More specifically, rules embody expert knowledge about how to accomplish goals during program transformation. Goals can be accomplished by either applying other rules or directly applying transformations.

Glitter rules have three parts:

Goal. The transformation issue that is addressed by the rule.

Strategies. A list of alternative ways to accomplish the goal. Each strategy may involve nested subgoals or the application of transformations.

Selection rationale. Guidelines on how to select among the different strategies.

For example, one *goal* in transforming a specification into an efficient implementation is to replace all references to past execution states with references to information in the current state. The *strategies* for doing this include

- Caching historical information until it is needed (eager evaluation)
- Computing the historical information from information in the current state (lazy evaluation)

The *selection rationale* for choosing among these alternatives include

- Space/time tradeoffs for caching versus derivation
- Whether or not it is possible to rederive historical references from the current context

Glitter automates as much of the transformation process as possible. In situations in which strategies are incomplete or selection rationale fails, however, the system must ask the software developer for guidance. Fickas [1985] reports that Glitter can automate anywhere from 65% to 90% of the transformation selections and applications. Note that if Glitter could automate 100%, it would be a compiler for the specification language.

9.3 Specialization

After being selected, transformations are typically applied without the software developer having to modify or specialize them. Pattern variables in a transformation are a generalization, but the pattern variables are automatically specialized with existing code fragments in the program when a transformation is applied. Therefore, from the software developer's point of view, specialization is in the hidden part of the transformation abstraction.

9.4 Integration

The integration of transformations is analogous to functional composition. For example, the application of two transformations to a program can be expressed as either the sequence of two mappings or the application of a composed mapping; that is,

$$P_0 \times T_0 \rightarrow P_1$$
$$P_1 \times T_1 \rightarrow P_{final}$$

or

$$P_0 \times (T_1 \circ T_0) \rightarrow P_{final}$$

The only time software developers need to be concerned about the composition of transformations is when the the order of application is significant. For example, when P_A is not identical to P_B in

$$P_0 \times (T_1 \circ T_0) \rightarrow P_A$$
$$P_0 \times (T_0 \circ T_1) \rightarrow P_B.$$

P_A and P_B will be semantically equivalent since transformations always preserve semantics, but they may differ in terms of implementation characteristics.

9.5 Appraisal of Reuse Techniques in Transformational Systems

The VHLL used in the first phase of a transformational system can have higher level abstractions than pure VHLLs such as SETL, PAISLey, and MODEL. The run-time performance of these higher level abstractions is very poor with the current VHLL compiler technology, but the human-guided transformations of the transformational approach produce more efficient implementations. Since transformational systems can use higher level abstractions than pure VHLLs, they exhibit a smaller cognitive distance between informal requirements of a software system and its written specification [Balzer 1989; Feather 1989].

The second phase in the transformational approach is essentially a human-guided compilation. Compared to VHLLs that use fully automated compilation, software developers expend additional effort to produce an efficiently executable system. By involving the software developer in the compilation process, transformational systems increase the cognitive distance in order to achieve better run-time performance.

Expert system technology has proven effective for reusing transformations. The rules in Glitter represent expert knowledge about applying transformations to achieve particular goals during the transformation phase. Glitter can be viewed as an interface to a library of reusable transformations. Software developers present high-level goals to Glitter, and Glitter helps locate the reusable transformations that satisfy those goals. The primary challenge for the transformational approach is to identify the types of optimizations that software developers use to implement efficient systems and to codify these optimizations into transformations and rules. Table 7 summarizes the transformational system approach to software reuse.

10. SOFTWARE ARCHITECTURES

Reusable *software architectures* are large-grain software frameworks and subsystems that capture the global structure of a software system design. This large-scale global structure represents a significant design and implementation effort that is reused as a whole. Reuse at this scale offers significant leverage in the development of software.

Examples of reusable software architectures include database subsystems that are specialized and reused in different applications; the framework for a compiler into which different lexical analyzers, parsers, and code generators can be inserted; rule-based and blackboard architectures for expert systems; adaptable user interface architectures; and even design frameworks for oscilloscopes [Garlan 1990; Krueger 1992; Lane 1990; Shaw 1991].

Software architectures are analogous to very large-scale software schemas. Software architectures, however, focus on subsystems and their interaction rather than data structures and algorithms [Shaw 1989]. Software architectures are

Table 7. Reuse in Transformational Systems

Abstraction	Transformations are reusable artifacts that describe how to map source code fragments into semantically equivalent, but more efficient, source code fragments. Rules are abstractions for expert knowledge about how to apply transformations to achieve higher level goals.
Selection	Reusable transformations can be stored in a library. Selection of transformations from this library can be enhanced by rule-based analysis that identifies sequences of transformations that satisfy high-level transformation goals.
Specialization	Transformations are typically applied without modification. Therefore, specialization is typically not an issue for transformation reuse.
Integration	The integration of transformations is implicit in the order in which they are applied. The issues are similar to functional composition.
Pros	Transformational systems use general-purpose, high-level programming abstractions. These abstractions can be at a higher level than with VHLLs because the human-guided transformations produce more efficient implementations than is possible with VHLL compilers. The higher level abstractions reduce cognitive distance.
Cons	Applying transformations requires time and effort from the software developer, thereby increasing the cognitive distance. As transformation systems improve, however, more of this effort is automated.

also analogous to application generators in that large-scale system designs are reused. Application generators, however, are typically standalone systems with implicit architectures, whereas software architectures can often be explicitly specialized and integrated with other architectures to create many different composite architectures.

It is interesting to note that some of the more powerful reuse techniques discussed earlier implicitly capitalize on reuse at the software architecture level. For example, experienced software engineers scavenge previous designs with great leverage. Application generators reuse significant portions of system designs and implementations within each generated system. Transformational systems reuse the design of efficient implementations for high-level specifications. Prototyping and experience provide software developers with positive and negative forms of design reuse (i.e., designs that work versus those that do not).

10.1 Abstraction in Software Architectures

Draco is an example software architecture technology [Freeman 1987a; Neighbors 1984, 1989]. With Draco, each software architecture is encapsulated in its own application generator. The output from these "architecture generators" can be used as building blocks for higher level architecture generators. Therefore, Draco is an architecture generator *generator*.

In Draco, each software architecture has a *domain language* and a set of *components* that implement the domain language. The domain language corresponds to the *abstraction specification* for an architecture. It captures the relevant abstractions (i.e., objects and operations) for a software architecture in a particular domain. The design of domain languages is identical to the design of the abstraction specifications for conventional application generators, as described in Section 7.5.2:

- Recognizing an appropriate domain
- Defining domain boundaries (domain coverage)
- Defining the underlying abstractions
- Defining the variant and invariant parts of the abstraction
- Defining the form of the language

The art of *domain analysis* and domain language design has become an independent area of research [Prieto-Diaz and Arango 1991]. Example domains range from relatively small areas, such as sets and stacks, to arbitrarily large application domains, such as navigation, process control, compilation, and data processing.

Draco *components* are written to implement each *object* and *operation* in a domain language. Components therefore correspond to the *abstraction realization* level for a software architecture. These components define the execution semantics of a domain language, making it executable.

Draco provides software developers with a recursive model of reuse. The components that implement a new domain language are written using other domain languages (possibly including the base language Lisp). For example, the implementation of a parser architecture might use an existing domain language for tree-structured data. A reusable domain language built on several recursive layers of domain languages represents a significant amount of reuse.

The software developer only deals with the top layer domain language in such a recursive reuse structure. The nested domain languages used to implement the top layer language are in the *hidden part* of the top layer abstraction.

10.2 Selection

Once the domain language for a Draco software architecture is written and the components implemented, it is available for use (reuse) in building applications or new software architectures. As with libraries of other types of software artifacts, a large collection of software architectures must be accompanied by tools and techniques for selection (classification, searching, and exposition). In particular, a library system must help software developers find software architectures with the smallest cognitive distance between the architecture and the application requirements.

Analogous to application generators, when an appropriate software architecture does not exist for an application, it may still be advantageous to create a new reusable architecture. This up-front cost is justified in cases in which application will be modified many times in its life or when many prototypes are needed to converge on a usable system. In addition, the new software architecture becomes available for reuse in subsequent application and architecture designs.

10.3 Specialization

Once a system is built using a Draco software architecture, it can be specialized for more efficient execution in two ways: source-to-source transformations and component refinement.

Source-to-source transformations, like those described in Section 9, are optimizations applied to programs written in a domain language. These transformations produce semantically equivalent but more efficient specifications in the same domain language. For example, in an algebraic domain, the following simple transformation converts exponentiation to multiplication for the special case of squares [Neighbors 1984]:

$$\$X^2 == \rangle \$X * \$X$$

Component refinements are alternative implementations of the same domain language construct that have different performance characteristics. This is exactly like the multiple schema implementations in Section 6.3. For example, in an algebraic domain, the exponential operator in the domain language could be implemented by both the binary shift method and the Taylor expansion method. The software developer using this domain language would choose the component refinement with the best execution profile for a particular application.

10.4 Integration

The primary difference between Draco and conventional application generators is that Draco allows an arbitrary collection of software architectures to be integrated into a single application, whereas conventional application generators are typically standalone. This flexibility adds considerable power to the Draco model. A reusable architecture can be used as a subsystem in many larger architectures rather than being limited to a single application generator.

Table 8. Reuse in Software Architectures

Abstraction	Architectural abstractions come directly from application domains. High-level domain abstractions are captured in the "domain language" for an architecture and are automatically mapped into executable implementations
Selection	The analogy between software schemas and software architectures suggests that library techniques for doing classification, search, and exposition could be used to select among a collection of reusable software architectures.
Specialization	Software developers can specialize software architectures to produce implementations that match the performance requirements of an application. Specialization may be *horizontal* with source-to-source transformations or *vertical* with component refinements.
Integration	Integrating different domain languages into a single application requires a special-purpose module interconnection language. For example, the Draco module interconnection language checks for type representation consistency and for domain-specific semantic consistency in the architecture interfaces.
Pros	Application developers use high-level abstractions from an application domain to instantiate and compose software architectures. Therefore, like application generators, the cognitive distance is small. In addition, reusable architectures can be used either stand-alone to create end-user applications or as building blocks to create higher level software architectures.
Cons	Creating reusable software architectures is difficult, and many such architectures will be needed to populate a general-purpose library of architectures.

Special care must be taken in the Draco module interconnection language since multiple software architectures with a variety of domain languages may be integrated into a single application. For example, the interconnection language must accommodate different domain languages and even different component refinements with different representations of the same data object. When objects are passed in and out of functions, type representation consistency must be maintained. In addition to syntactic consistency, semantic consistency can be maintained by attaching preconditions and postconditions to the interface of a component refinement.

10.5 Appraisal of Reuse Techniques in Software Architectures

As with application generators, the leverage offered by software architectures comes from the small cognitive distance between informal concepts in an application domain and executable implementations. The software developer using a software architecture typically works at the abstraction specification level. The mapping from abstraction specification to abstraction realization is mostly automated (although in systems like Draco the software developer can apply transformations and choose among different component refinements to increase efficiency). This automation isolates the software developer from the hidden and realization parts of the abstraction.

Software architectures can be used as either standalone application generators to create end-user applications or as building blocks for creating higher level architectures. Therefore, in cases in which a reusable software architecture does not exist for a particular application, lower level software architectures can be composed to create the new architecture.

Similar to other reuse techniques, it will be difficult to build a large, general-purpose library of high-quality reusable software architectures [Neighbors 1989]. This is due to both the challenge of identifying and building appropriate software architectures and the difficulty of identifying and building effective library tools. Table 8 summarizes the software architecture approach to software reuse.

11. SUMMARY

In this survey, a broad range of software reuse techniques from the research literature was described and compared. The

different approaches to software reuse were partitioned into categories then compared using a uniform taxonomy.

11.1 Categories and Taxonomy

For this survey, the field of software reuse was partitioned into eight categories, each of which illustrates a characteristic set of issues and techniques in software reuse:

- High-level languages
- Design and code scavenging
- Source code components
- Software schemas
- Application generators
- Very high-level languages
- Transformational systems
- Software architectures

The following taxonomy for the categories allowed us to compare and contrast different reuse techniques as well as illustrate some of the fundamental issues in software reuse:

- Abstraction
- Selection
  - Classification
  - Retrieval
  - Exposition
- Specialization
- Integration

11.2 Cognitive Distance

The notion of cognitive distance was introduced as an intuitive gauge to compare the effectiveness of reuse techniques. Cognitive distance is informally defined as the amount of intellectual effort that must be expended by software developers in order to take a software system from one stage of development to another.

Software reuse techniques can reduce cognitive distance in two ways:

(1) Reduce the amount of intellectual effort required to go from the initial conceptualization of a system to a specification of the system in abstractions of the reuse technique.

(2) Reduce the amount of intellectual effort required to produce an executable system once the software developer has produced a specification in terms of abstractions of the reuse technique.

To reduce cognitive distance with the first method, implementors of a reuse technique must move the abstraction specifications in the technique closer to the abstractions used to reason about applications informally. For example, application generators use domain-specific abstractions for their programming model.

Following is an approximate ranking of the eight reuse categories, rated on how well they minimize cognitive distance by the first method (where one is best):

(1) Application generators
(2) Software architectures
(3) Transformational systems
(4) Very high-level languages
(5) Software schemas
(6) Source code components
(7) Code scavenging
(8) High-level languages

To reduce cognitive distance with the second method, implementors of a reuse technique must partially or fully automate the mapping from abstraction specifications to executable abstraction realizations. Cognitive distance is reduced or eliminated by minimizing the need for software developers to see and understand the hidden and realization parts of the abstractions. For example, very high-level languages use compilers or interpreters to map abstract specifications in the language directly into an executable form.

Following is an approximate ranking of the eight reuse categories, rated on how well they minimize cognitive distance by the second method (where one is best):

(1) High-level languages
(2) Very high-level languages

(3) Application generators
(4) Software architectures
(5) Transformational systems
(6) Software schemas
(7) Source code components
(8) Code scavenging

An ideal software reuse technology would use both approaches to reduce cognitive distance. That is, a software developer using this technology would quickly be able to select, specialize, and integrate abstraction specifications that satisfied a particular set of informal requirements, and the abstraction specifications would be automatically translated into an executable system. In addition, this ideal technology could be used in all application domains.

Of the eight reuse technologies examined in this report, the reusable software architectures probably come closest to this ideal description. A "perfected" form of this technology would have a complete library of useful software architectures, ranging from small-grained domains such as sets and stacks to large-grained application domains such as airline navigation control systems. The library system would allow software developers to locate the most appropriate reusable artifacts for their application easily. The software architectures would be specialized and integrated in terms of concise high-level abstraction specifications from the application domain. Executable systems would be automatically produced from these high-level specifications.

11.3 General Conclusions

- What is software reuse? Software reuse is the process of using existing software artifacts rather than building them from scratch. Typically, reuse involves the selection, specialization, and integration of artifacts, although different reuse techniques may emphasize or deemphasize certain of these.
- Why reuse software artifacts? The primary motivation to reuse software artifacts is to reduce the time and effort required to *build* software systems. The quality of software systems is enhanced by reusing quality software artifacts, which also reduces the time and effort required to *maintain* software systems.
- What is required to implement a software reuse technology? Creating a complex executable system with fewer key strokes and less cognitive burden on software developers clearly implies a higher level of abstraction. For a software developer to select, specialize, and integrate reusable artifacts successfully, the reuse technology must provide natural, succinct, high-level abstractions that describe artifacts in terms of "what" they do rather than "how" they do it. The ability of a software developer to practice software reuse is limited primarily by the ability to reason in terms of these abstractions. In other words, there must be a small cognitive distance between informal reasoning and the abstract concepts defined by the reuse technology.

  In areas of research that strive to raise the level of abstraction in software development, much of the leverage has come from software reuse. In fact, much of the software reuse literature is not about new techniques developed from first principles of reuse but rather about ongoing research projects that have discovered that reuse is responsible for the advantages offered by high-level abstractions in their systems.
- Why is software reuse difficult? Useful abstractions for large, complex, reusable software artifacts will typically be complex. In order to use these artifacts, software developers must either be familiar with the abstractions a priori or must take time to study and understand the abstractions. The latter case can defeat some or all of the gains in reusing an artifact. The former case is where we have seen significant successes in reuse. Examples include (1) math libraries used by developers who are familiar with the mathematical concepts, (2) abstract

data type schemas such as stacks and queues used by developers who are familiar with the semantics, and (3) domain-specific application generators and domain languages used by developers who are familiar with concepts in a particular application domain.

The following truisms were included earlier in the survey. They are simple statements on reuse that have been difficult to satisfy in practice.

For a software reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation.

For a software reuse technique to be effective, it must be easier to reuse the artifacts than it is to develop the software from scratch.

To select an artifact for reuse, you must know what it does.

To reuse a software artifact effectively, you must be able to "find it" faster than you could "build it."

What are the open areas of research in software reuse? There is clearly much work that remains to be done. In general, the search for high-level abstractions for software artifacts is probably the most crucial, but this type of research is not new or unique to software reuse.

Library issues are also critical to many reuse technologies. These issues, however, lead right back to abstraction since classification, search, and exposition are heavily dependent on high-level abstractions for software artifacts in a library.

ACKNOWLEDGMENTS

Nico Habermann contributed greatly to this survey through his patient reading, insightful comments, and thoughtful discussion. John McDermott, Barbara Staudt Lerner, Peter Feiler, John Ockerbloom, and the anonymous reviewers all provided valuable feedback and detailed comments. Ellen Douglas gave many suggestions on improving the technical writing style (although I admittedly have not done justice to her efforts). My initial interest in the subject of software reuse was sparked in a discussion group headed by Mary Shaw at the Software Engineering Institute.

REFERENCES

BALZER, R. 1985. A 15 year perspective on automatic programming. *IEEE Trans. Softw. Eng. SE-11*, 11 (Nov.), 1257–1268.

BALZER, R. 1989. A 15 year perspective on automatic programming. In *Frontier Series: Software Reusability: Volume II—Applications and Experience*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 289–311, Chap. 14. Originally Balzer [1985].

BENTLEY, J. 1986. Programming pearls. *Commun. ACM 29*, 5 (May), 364–369.

BIGGERSTAFF, T. 1987. Hypermedia as a tool to aid large scale reuse. In *Proceedings of the Workshop on Software Reuse* (Oct.). Rocky Mountain Institute of Software Engineering, Boulder, Colo.

BIGGERSTAFF, T. J., AND PERLIS, A. J., EDS. 1989a. *Frontier Series: Software Reusability: Volume I —Concepts and Models*. ACM press, New York.

BIGGERSTAFF, T. J., AND PERLIS, A. J., EDS. 1989b. *Frontier Series: Software Reusability: Volume II—Applications and Experience*. ACM Press, New York.

BIGGERSTAFF, T., AND RICHTER, C. 1987. Reusability framework, assessment, and directions. *IEEE Softw. 4*, 2 (Mar.), 41–49. Also in Tracz [1988] and Biggerstaff and Richter [1989].

BIGGERSTAFF, T., AND RICHTER, C. 1989. Reusability framework, assessment, and directions. In *Frontier Series: Software Reusability: Volume I–Concepts and Models*. Biggerstaff, T. J., and Perlis A. J., Eds. ACM Press, New York, pp. 1–17, Chap. 1. Originally Biggerstaff and Richter [1987].

BOEHM, B. W. 1987. Improving software productivity. *IEEE Comput. 20*, 9 (Sept.), 43–57.

BOOCH, G. 1987. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin/Cummings Publishing Company, Inc., Menlo Park, Calif.

BROOKS, F. P. JR. 1975. *The Mythical Man-Month*. Addison-Wesley, Reading, Mass.

BROOKS, F. P. 1987. No silver bullet: Essence and accidents of software engineering. *IEEE Comput. 20*, 4 (Apr.), 10–19.

CHEATHAM, T. E. JR. 1983. Reusability through program transformations. In *Workshop on Reusability in Programming* (Newport, RI, Sept.). ITT Programming, Stratford, Conn., pp. 122–128. Also in Cheatham [1984, 1989].

CHEATHAM, T. E. JR. 1984. Reusability through program transformations. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept.), 589–594. Originally Cheatham [1983].

CHEATHAM, T. E. JR. 1989. Reusability through program transformations. In *Frontier Series: Software Reusability: Volume I—Concepts and Models*. Biggerstaff, T. J., and Perlis, A. J., Eds., ACM Press, New York, pp. 321–335, Chap. 13. Originally Cheatham [1983].

CHENG, T. T., LOCK, E. D., AND PRYWES, N. S. 1984. Use of very high level languages and program generation by management professionals. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept.), 552–563.

CLEAVELAND, J. C. 1988. Building application generators. *IEEE Softw. 5*, 4 (July), 25–33. Also in Prieto-Diaz and Arango [1991].

DEREMER, F., AND KRON, H. H. 1976. Programming-in-the-large versus programming-in-the-small. *IEEE Trans. Softw. Eng. SE-2*, 2 (June), 80–86.

DEUTSCH, P. L. 1983. Reusability in the Smalltalk-80 programming system. In *Workshop on Reusability in Programming* (Newport, R. I. Sept.). ITT Programming, Stratford, Conn., pp. 72–76. Also in Freeman [1987b].

DEUTSCH, L. P. 1989. Design reuse and frameworks in the Smalltalk-80 system. In *Frontier Series: Software Reusability: Volume II—Applications and Experience*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 57–71, Chap. 3.

DOBERKAT E., DUBINSKY, E., AND SCHWARTZ, J. T. 1983. Reusability of design for complex programs: An experiment with the SETL optimizer. In *Workshop on Reusability in Programming* (Newport, R. I., Sept.) ITT Programming, Stratford, Conn., pp. 106–108.

DONGARRA, J. J., AND GROSSE, E. 1987. Distribution of mathematical software via electronic mail. *Commun. ACM 30*, 5 (May), 403–407.

DONZEAU-GOUGE, V., KAHN, G., LANG, B., AND MELESE, B. 1984. Document structure and modularity in Mentor. In *Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Penn. Apr.). ACM SIGSOFT/SIGPLAN, New York, pp. 141–148.

DUBINSKY, E., FREUDENBERGER, S., SCHONBERG, E., AND SCHWARTZ, J. T. 1989. Reusability of design for large software systems: An experiment with the SETL optimizer. In *Frontier Series: Software Reusability: Volume I—Concepts and Models*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 275–293, Chap. 11.

EMBLEY, D. W., AND WOODFIELD, S. N. 1987. A knowledge structure for reusing abstract data types. In *9th International Conference on Software Engineering*, (Monterey, Calif., Mar.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 360–365. Also in Tracz [1988].

ESHELMAN, L. 1988. MOLE: A knowledge-acquisition tool for cover-and-differentiate systems. In *Kluwer International Series in Engineering and Computer Science. Automatic Knowledge Acquisition for Expert Systems*. Kluwer Academic Publishers, Boston, Mass , pp. 37–80, Chap. 3.

EVB SOFTWARE 1985. *An Object Oriented Design Handbook for Ada Software*. EVB Software Engineering, Inc., Fredrick, Maryland.

FEATHER, M. S. 1983. Reuse in the context of a transformation based methodology. In *Workshop on Reusability in Programming* (Newport, R.I., Sept.). ITT Programming, Stratford, Conn., pp. 50–58. Also in Freeman [1987b] and Feather [1989].

FEATHER, M. S. 1989. Reuse in the context of a transformation based methodology. In *Frontier Series: Software Reusability: Volume I—Concepts and Models*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 337–359, Chap. 14. Originally Feather [1983].

FICKAS, S. F 1985. Automating the transformational development of software. *IEEE Trans. Softw. Eng. SE-11*, 11 (Nov.), 1268–1277.

FREEMAN, P. 1983. Reusable software engineering: Concepts and research directions. In *Workshop on Reusability in Programming* (Newport, R.I., Sept.). ITT Programming, Stratford, Conn., pp. 2–16. Also in Freeman [1987b]

FREEMAN, P. 1987a. A conceptual analysis of the Draco approach to constructing software systems. *IEEE Trans. Softw. Eng. SE-13*, 7 (July), 830–844. Also in Freeman [1987b].

FREEMAN, P., ED. 1987b. *Tutorial: Software Reusability*. IEEE Computer Society Press, Washington, D.C.

GARLAN, D. 1990. The role of formal reusable frameworks. In *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development* (Napa, Calif , May). ACM Press, New York, pp. 42–44.

GOGUEN, J. A. 1986. Reusing and interconnecting software components. *IEEE Comput. 19*, 2 (Feb.), 16–28. Also in Freeman [1987b] and Prieto-Diaz and Arango [1991].

GOGUEN, J. A. 1989. Principles of parameterized programming. In *Frontier Series: Software Reusability: Volume I—Concepts and Models*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 159–225, Chap. 7.

HABERMANN, A. N., KRUEGER, C., PIERCE, B., STAUDT, B., AND WENN, J. 1988. Programming with views. Tech. Rep. CMU-CS-87-177, Carnegie-Mellon University, Pittsburgh, Penn.

HAYES-ROTH, F., WATERMAN, D., AND LENENT, D., EDS. 1983. *Teknowledge Series in Knowledge Engineering*. Vol. 1, *Building Expert Systems*. Addison-Wesley, Reading, Mass.

HOROWITZ, E., AND MUNSON, J. B. 1983. An expansive view of reusable software. In *Workshop on Reusability in Programming* (Newport, R.I., Sept.). ITT Programming, Stratford, Conn., pp. 250–262. Also in Horowitz and Munson [1984], Freeman [1987b], and Horowitz and Munson [1989].

HOROWITZ, E., AND MUNSON, J. B. 1984. An expansive view of reusable software. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept.), 477–487. Originally Horowitz and Munson [1983].

HOROWITZ, E., AND MUNSON, J. B. 1989. An expansive view of reusable software. In *Frontier Series: Software Reusability: Volume I—Concepts and Models*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 19–41, Chap. 2.

HOROWITZ, E., KEMPER, A., AND NARASIMHAN, B. 1985. A survey of application generators. *IEEE Softw. 2*, 1 (Jan.), 40–54.

ICHBIAH, J. D. 1983. On the design of Ada. In *Information Processing 83*. Mason, R.E.A., Ed. IFIP, Elsevier Science Pub., New York, pp. 1–10.

IMSL 1987. *IMSL Math/Library User's Manual*. 1.0 Edition, Houston, Tex.

JOHNSON, S. C. 1979. *Yacc: Yet Another Compiler-Compiler in the UNIX Programmer's Manual —Supplementary Documents*. 7th ed. AT & T Bell Laboratories, Indianapolis, Ind.

KAISER, G. E. 1985. Semantics for structure editing environments. Ph.D. dissertation, Carnegie-Mellon Univ.

KAISER, G. E. 1989. Incremental dynamic semantics for language-based programming environments. *ACM Trans. Program. Lang. Syst. 11*, 2 (Apr.), 169–193. See also Kaiser [1985].

KAISER, G. E., AND GARLAN, D. 1987. Melding software systems from reusable building blocks. *IEEE Softw. 4*, 4 (July), 17–24. Also in Tracz [1988].

KAISER, G. E., AND GARLAN, D. 1989. Synthesizing programming environments from reusable features. In *Frontier Series: Software Reusability: Volume II—Applications and Experience*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 35–55, Chap. 2.

KATZ, S., RICHTER, C. A., AND THE, K. 1987. PARIS: A system for reusing partially interpreted schemas. In *9th International Conference on Software Engineering* (Monterey, Calif., Mar.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 377–385. Also in Tracz [1988] and Katz et al. [1989].

KATZ, S., RICHTER, C. A., AND THE, K. 1989. PARIS: A system for reusing partially interpreted schemas. In *Frontier Series: Software Reusability: Volume I—Concepts and Models*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 257–273, Chap. 10. Originally Katz et al. [1987].

KERNIGHAN, B. W. 1983. The Unix system and software reusability. In *Workshop on Reusability in Programming* (Newport, R.I., Sept.). ITT Programming, Stratford, Conn., pp. 235–239. Also in Kernighan [1984] and Freeman [1987b].

KERNIGHAN, B. W. 1984. The Unix system and software reusability. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept.), 513–518. Originally Kernighan [1983].

KNUTH, D., E. 1973. *The Art of Computer Programming*. Addison-Wesley, Reading, Mass.

KRUCHTEN, P., SCHONBERG, E., AND SCHWARTZ, J. 1984. Software prototyping using the SETL programming language. *IEEE Softw. 1*, 4 (Oct.), 66–75.

KRUEGER, C. W. 1987. Expert system engineering and its relation to conventional software engineering. Ph.D. area qualifier paper. Carnegie-Mellon Univ., Pittsburgh, Penn. Available from author on request.

KRUEGER, C. W. 1992. Application-specific object management architectures. Ph.D. dissertation, Carnegie-Mellon Univ.

LANE, T. G. 1990. User interface software structures. Ph.D. dissertation, Carnegie-Mellon Univ.

LATOUR, L., AND JOHNSON, E. 1988. Seer: A graphical retrieval system for reusable Ada software modules. In *The 3rd International IEEE Conference on Ada Applications and Environments* (Manchester, N.H., May). IEEE Computer Society Press, Los Alamitos, Calif., pp. 105–113.

LESK, M. E., AND SCHMIDT, E. 1979. *Lex: A Lexical Analyzer Generator in the UNIX Programmer's Manual—Supplementary Documents*, 7th ed. AT & T Bell Laboratories, Indianapolis, Ind.

LEVY, L. S. 1986. A metaprogramming method and its economic justification. *IEEE Trans. Softw. Eng. SE-12*, 2 (Feb.), 272–277.

LIEBERHERR, K. J., AND RIEL, A. J. 1988. Demeter: A CASE study of software growth through parameterized classes. In *The 10th International Conference on Software Engineering* (Singapore, Apr.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 254–264.

LISKOV, B. 1987. Data abstraction and hierarchy. In *OOPSLA '87, Addendum to the Proceedings* (Orlando, Fla., Oct.). ACM Sigplan, New York, pp. 17–34.

LIU, S., AND PAIGE, R. 1979. Data structure choice/formal differentiation. Tech. Rep. NSO-15, Courant Institute of Mathematical Sciences, New York.

LOECKX, J., AND SIEBER, K. 1984. *Wiley and Teubner Series in Computer Science: The Foundations of Program Verification*. John Wiley, New York.

LUCKHAM, D. C., AND VON HENKE, F. W. 1984. An overview of Anna, a specification language for Ada. In *1984 Conference on Ada Applications and Environments* (St. Paul, Minn., Oct.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 116–127.

MARCUS, S., ED. 1988. *Kluwer International Series in Engineering and Computer Science: Automatic Knowledge Acquisition for Expert Systems*. Kluwer Academic Publishers, Boston, Mass.

MCDERMOTT, J. 1986. Making expert systems explicit. In *Information Processing 86*. IFIP, Elsevier Science Pub., New York, pp. 539–544.

MCDERMOTT, J. 1988. Preliminary steps toward a taxonomy of problem-solving methods. In *Kluwer International Series in Engineering and Computer Science. Automatic Knowledge Acquisition for Expert Systems*. Marcus, S., Ed., Kluwer Academic Publishers, Boston, Mass., pp. 225–256, Chap. 8.

MCILROY, M. D., 1968. Mass produced software components. In *Software Engineering; Report on a conference by the NATO Science Committee* (Garmisch, Germany, Oct.). Naur, P., and Randell, B., Eds. NATO Scientific Affairs Division, Brussels, Belgium, pp. 138–150.

MENDAL, G. O., 1986. Designing for Ada reuse: A case study. In *2nd International Conference on Ada Applications and Environments* (Miami Beach, Fla., Apr.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 33–42.

MEYER, B. 1987. Reusability: The case for object-oriented design. *IEEE Softw. 4*, 2 (Mar.), 50–63. Also in Tracz [1988] and Meyer [1989].

MEYER, B. 1989. Reusability: The case for object-oriented design. In *Frontier Series: Software Reusability: Volume II—Applications and Experience*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 1–33, Chap.1. Originally Meyer [1987].

NAUER, P., AND RANDELL, B., EDS. 1968. *Software Engineering; Report on a Conference by the NATO Science Committee*. NATO Scientific Affairs Division, Brussels, Belgium.

NEIGHBORS, J. M. 1983. The Draco approach to constructing software from reusable components. In *Workshop on Reusability in Programming* (Newport, R.I., Sept.). ITT Programming, Stratford, Conn., pp. 167–178. Also in Neighbors [1984] and Freeman [1987b].

NEIGHBORS, J. M. 1984. The Draco approach to constructing software from reusable components. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept.), 564–574. Originally Neighbors [1983].

NEIGHBORS, J. M. 1989. Draco: A method for engineering reusable software systems. In *Frontier Series: Software Reusability: Volume I—Concepts and Models*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 295–319, Chap. 12. Also in Prieto-Diaz and Arango [1991].

PARNAS, D. L., CLEMENTS, P. C., AND WEISS, D. M. 1983. Enhancing reusability with information hiding. In *Workshop on Reusability in Programming* (Newport, R.I., Sept.). ITT Programming, Stratford, Conn., pp. 240–247. Also in Freeman [1987b] and Parnas et al. [1989].

PARNAS, D. L., CLEMENTS, P. C., AND WEISS, D. M. 1989. Enhancing reusability with information hiding. In *Frontier Series: Software Reusability: Volume I—Concepts and Models*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 141–157, Chap. 6. Originally Parnas and Clements [1983].

PARTSCH, H., AND STEINBRUGGEN, R. 1983. Program transformation systems. *ACM Comput. Surv. 15*, 3 (Sept.), 199–236.

PAYTON, T., KELLER, S., PERKINS, J., ROWLAN, S., AND MARDINLY, S. 1982. SSAGS: A syntax and semantics analysis and generation system. In *6th International Computer Software and Applications Conference (COMPSAC82)* (Chicago, Ill., Nov.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 424–432.

PRIETO-DIAZ, R. 1989. Classification of reusable modules. In *Frontier Series: Software Reusability: Volume I—Concepts and Models*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 99–123, Chap. 4.

PRIETO-DIAZ, R., AND ARANGO, G., EDS. 1991. *Domain Analysis and Software System Modeling*. IEEE Computer Society Press, Los Alamitos, Calif.

PRIETO-DIAZ, R., AND FREEMAN, P. 1987. Classifying software for reusability *IEEE Softw. 4*, 1 (Jan.), 6–16. Also in Freeman [1987b].

PRIETO-DIAZ, R., AND NEIGHBORS, J. M. 1986. Module interconnection languages. *J. Syst. Softw. 6*, 4 (Nov.), 307–334. Also in Freeman [1987b].

PRYWES, N. S., AND LOCK, E. D. 1989. Use of the model equational language and program generator by management professionals. In *Frontier Series: Software Reusability: Volume II—Applications and Experience*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 103–129, Chap. 5.

REPS, T., AND TEITELBAUM, T. 1984. The synthesizer generator. In *Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Penn., Apr.). ACM SIGSOFT/SIGPLAN, New York, pp. 42–48.

RICH, C., AND WATERS, R. 1983. Formalizing reusable software components. In *Workshop on Reusability in Programming* (Newport, R.I., Sept.). ITT Programming, ITT, Stratford, Conn., pp. 152–159.

RICH, C., AND WATERS, R. C. 1988. Automatic programming: Myths and prospects. *IEEE Comput 21*, 8 (Aug.), 40–51.

RICH, C., AND WATERS, R. C. 1989. Formalizing reusable software components in the programmer's apprentice. In *Frontier Series: Software Reusability: Volume II—Applications and Experience*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 313–343, Chap. 15. Expanded version of Rich and Waters [1983].

SHAW, M. 1984. Abstraction techniques in modern programming languages. *IEEE Softw. 1*, 4 (Oct.), 10–26.

SHAW, M. 1989. Larger scale systems require higher-level abstractions. In *Proceedings of the 5th International Workshop on Software Specification and Design* (May). IEEE Computer Society Press, Los Alamitos, Calif., pp. 143–146.

SHAW, M. 1991. Heterogeneous design idioms for software architectures. In *Proceedings of the 6th International Workshop on Software Specification and Design* (Como, Italy, Oct.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 1–8.

STANDISH, T. A. 1984. An essay on software reuse. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept.), 494–497.

STAUDT, B. J., KRUEGER, C. W., HABERMANN, A. N., AND AMBRIOLA, V. 1986. The GANDALF system reference manuals. Tech. Rep. CMU-CS-86130, Carnegie-Mellon Univ., Pittsburgh, Penn.

STEFIK, M., AND BOBROW, D. G. 1986. Object-ori-

ented programming: Themes and variations. *The AI Mag. 16*, 4, 40–62.

TEXAS INSTRUMENTS 1985. *The TTL Data Book*. Vol. 2. Texas Instruments, Dallas, Tex.

TRACZ, W., ED. 1988. *Tutorial: Software Reuse: Emerging Technology*. IEEE Computer Society Press, Los Alamitos, Calif.

TSENG, J. S., SZYMANSKI, B., SHI, Y., AND PRYWES, N. S. 1986. Real-time software life cycle with the model system. *IEEE Trans. Softw. Eng. SE-12*, 2 (Feb.), 358–373.

VOLPANO, D. M., AND KIEBURTZ, R. B. 1985. Software templates. In *8th International Conference on Software Engineering* (London, UK, Aug.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 55–60.

VALPANO, D. M., AND KIEBURTZ, R. B. 1989. The templates approach to software reuse. In *Frontier Series: Software Reusability: Volume I—Concepts and Models*. Biggerstaff, T. J., and Perlis, A. J., Eds. ACM Press, New York, pp. 247–255, Chap. 9.

WATERS, R. C. 1985. The programmer's apprentice: A session with KBEmacs. *IEEE Trans. Softw. Eng. SE-11*, 11 (Nov.), 1296–1320.

WEGNER, P. 1983. Varieties of reusability. In *Workshop on Reusability in Programming* (Newport, R.I., Sept.). ITT Programming, Stratford, Conn., pp. 30–44. Also in Freeman [1987b].

ZAVE, P. 1984. The operational versus the conventional approach to software development. *Commun. ACM 27*, 2 (Feb.), 104–118.

ZAVE, P., AND SCHELL, W. 1986. Salient features of an executable specification language and its environment. *IEEE Trans. Softw. Eng. SE-12*, 2 (Feb.), 312–325.

Received February 1990; final revision accepted October 1991.