7-30-1993

A Cognitively Motivated System for Software Component Reuse

Michael Joseph Mateas
Portland State University

AN ABSTRACT OF THE THESIS OF Michael Joseph Mateas for the Master Of Science in Computer Science presented July 30, 1993.

Title: A Cognitively Motivated System for Software Component Reuse

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:

[redacted]

Jean Scholtz, Chair

[redacted]

James Hein

[redacted]

Beatrice Oshika

Software reuse via component libraries suffers from the twin problems of code location and comprehension. The Intelligent Code Object Planner (ICOP) is a cognitively motivated system that facilitates code reuse by answering queries about how to produce an effect with the library. It can plan for effects which are not primitive with respect to the library by building a plan that incorporates multiple components. The primary subsystems of ICOP are a knowledge base which describes the ontology of the library, a natural language interface which translates user queries into a formal effect language (predicates), a planner which accepts the effect and produces a plan utilizing the library components, and an explanation generator which accepts the plan and produces example code illustrating the plan. ICOP is currently implemented in Prolog and supports a subset of the Windows 3.0 API.

A COGNITIVELY MOTIVATED SYSTEM FOR SOFTWARE COMPONENT REUSE

by

MICHAEL JOSEPH MATEAS

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
1993©

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Michael Joseph Mateas presented July 30, 1993.

[redacted]

Jean Scholtz, Chair

[redacted]

James Hein

[redacted]

Beatrice Oshika

APPROVED:

[redacted]

Leonard Shapiro, Chair, Department of Computer Science

[redacted]

Roy W. Koch, Vice Provost for Graduate Studies and Research

DEDICATION

Dedicated to Anne Siegel, without whose loving support this work would not have been accomplished.

ACKNOWLEDGEMENTS

I would like to acknowledge the guidance and friendship of my advisor, Dr. Jean Scholtz.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

PROBLEM STATEMENT

INTRODUCTION

The software crisis has been recognized since the late 1960's [Sommerville, 1989]. Software systems tend to be delivered late, cost more than originally predicted, and difficult to maintain. With demand for code exceeding supply, it does not make sense to keep reimplementing the same functionality. Yet a 1983 study indicates that of all code written in that year, less than 15 percent was unique and specific to a particular application [Jones, 1984]. The field of software reuse is concerned with standardizing the remaining 85 percent of this code and providing tools to reuse it.

OVERVIEW OF SOFTWARE REUSE

Software reuse is not a new idea. The first compilers supported reuse of common machine language patterns such as looping and branching. Now these constructs could be reused by writing some short but understandable symbol sequences (high level language) rather than rewriting the much longer machine language pattern. The first operating systems provided commonly needed services such as i/o. The blocks of code necessary to perform these services could now be reused by making operating system calls rather than having to rewrite these blocks for every program. Biggerstaff and Perlis' [1984] conceptual map for looking at the software reuse field shows that these two approaches are still extensively used for delivering reusable functionality. Work in the software reuse field can be divided into two conceptual classes: the reuse of patterns (generation) and the reuse of building blocks (composition).

Pattern Reuse

Pattern reuse is based on the idea of generation. The idea here is to build a system which accepts some sort of terse yet easy to specify input and produces a program as output. For example, imagine some system that could take the natural language utterance "Give me a program that sorts strings" and produces as output an executable program to perform such sorting. The particulars of sorting strings (algorithms and data structures) are being reused, but these particulars, rather than being stored as some atomic building block (a sort routine) are stored as a potential pattern of activation of the generating system. A compiler for a high level language can be thought of as a system for reusing blocks of assembly code. A "while" loop, which is terse and easy to read in the source, is turned into a much longer, standard block of assembly code. Pattern reuse approaches can be divided into roughly three major categories: language based generators, application generators and transformation systems.

Language Based Generator. A language based generator extends the idea of a high level language. A very high level language (VHLL) provides a small set of semantically neutral components which are more abstract than those provided by a standard high level language. Such an abstract language can be used across a wide set of domains. Problem oriented languages (POL) provide a rich set of semantics aimed at a particular application domain. By providing constructs directly supporting a particular domain, the programmer's job of mapping a domain problem into the language is much easier. The programmer is reusing a domain analysis.

Application Generator. Application generators also encode domain specific information, but store it in the generator rather than the language. The input to an application generator is usually quite simple and does not really qualify as a full fledged language. A 4GL which lets one construct databases by describing the database fields and input screens, is an example of an application generator.

Transformation System.Transformation systems are similar to language based generators. They take some input written in a terse, easy to understand, but highly inefficient notation and, by successive transformations, turn it into an efficient but more difficult to read executable form. The transformation system itself is simple and fairly generic. The rules of transformation are stored as separate declarative knowledge (data) outside of the transformation system. It is these rules which are being reused.

Building Block Reuse

The reuse of building blocks is probably what comes to most people's minds when they think of code reuse. The idea is to compose preexisting atomic components to produce a desired effect. An example of a typical atomic component is a subroutine from a subroutine library. Modern object-oriented approaches are also a building block methodology. Here, many separate pieces of reusable code can be localized in a single object.

Building block reuse is not without problems [Horowitz and Munson, 1984]. First, it is difficult to determine which pieces of functionality are generally useful and can be described by a parameterized piece of code. This is a domain analysis problem. Secondly, once a library of useful components exists, how do you describe them in a manner that the user of the library will find comprehensible? This is a code comprehension problem. Thirdly, should the components be implemented in a standard programming language and be made available in an object library, thus becoming language and machine dependent, or should they be described by some sort of high level design language? This is an implementation problem. Finally, how does the user locate a particular component within a potentially large library? This is a code location problem. Additional problems are raised if the user wants to modify a reusable component to perform a slightly different task. Object oriented languages, with their support for inheritance, are one approach to this problem.

SUPPORTING BUILDING BLOCK REUSE

Library Components Interrelated

This thesis tackles the problems of code location and comprehension in building block reuse. The libraries provided by today's increasingly complex graphical environments present an interesting problem. Environments such as the Macintosh, Windows or X-Windows provide a huge number of routines, data types, macros and constants for use by the programmer. All of the components of these libraries are richly intertwined. Generally, a programmer's problem is not solved by locating a single routine. Rather, a pattern of routines and other library components (data types, etc.) must be used to accomplish the task. So a programmer must not only locate a single routine out of a large library, but must also locate an interwoven set of routines and other software objects. A programmer must not only comprehend a single routine, but must also comprehend the interactions between a set of routines and other software objects. A tool that supports use of these libraries must support the location and comprehension of patterns of use of these components.

Knowledge Transfer

These graphical environments have another property of interest. At a conceptual level, they are quite similar. They all have concepts of windows, controls, messages, etc. So someone with expertise in using one of these libraries, say, the Macintosh Toolbox, should be able to use their mental model of the library to transfer their skills to another, say the Windows API. Yet at a lower level of detail, these libraries are dissimilar. The names of routines are different, the routines are organized in a different manner, etc. While the knowledge of the conceptual level of one of these libraries will aid the programmer in transfering to another, the need to focus on low level details of the new library might hinder this positive transfer effect. Also, in places where two libraries may differ at a conceptual level, the programmer's mental model of the first library may actually result in negative transfer, hindering their ability to properly use the second library. A tool to support use of these libraries should support transfer between libraries of similar functionality, providing retrieval of collections of code objects from conceptual descriptions in places where the libraries are conceptually similar, and pointing out differences in places where the libraries are conceptually different. With this basic idea of what capabilities a reuse support system should provide, the task is now to design such a system.

MOTIVATED DESIGN

How should one go about designing a system to support reusability? Traditional system design tends to be technologically motivated. This technological motivation, and its place within the broad spectrum of problems, is captured nicely by Lehman's S-P-E program classification scheme [Lehman, 1991]. S and E programs are the extremes at the ends of the spectrum with P programs in the middle. S programs are those which need to satisfy some pre-stated specification. The specification is complete; it is the sole determinate of program correctness. The writer of an S program has no concern for where the specification came from. Their design task consists purely of making maneuvers within a technologically determined space (as conditioned by the concepts made available in the language used). Formal approaches to software engineering concern themselves with S type programs. An E program attempts to solve a problem in a real world domain. All consequences of the program's use, including its effectiveness in communicating with human users, and the manner in which the use of the program changes the very domain for which the program was written, determine the acceptability of the program as a solution. These systems always escape full formalizability. Berry [1992] argues that it is necessary to consider nontechnical issues such as management, psychology and sociology when developing E programs. Fischer laments that the traditional overemphasis on technology has lead to systems which do not effectively solve real world problems, cannot be adapted to changing conditions, and impose unnecessary constraints on users [Fischer, 1987].

A system to support reusability will be used in the complex, real-world domain of software development. Such a system is an example of an E program. It should take account of the fact that fundamentally, programming is a cognitive act engaged in by human beings. In Chapter II, a cognitive model of programming is developed which serves as a framework to motivate tool design. At the end of Chapter II, a cognitively motivated design for a system which facilitates reuse is given. Chapters III-VI describe the components of this system, Chapter VII summarizes the results of this thesis, and Chapter VIII describes prospects for future work.

CHAPTER II

COGNITIVE MODEL OF PROGRAMMING

INTRODUCTION

A cognitive model of human behavior starts with the premise that the complexity of this behavior is a result of a simple architecture responding to a complex environment, where the environment includes both sensory data and memory [Simon, 1981]. Simon uses the analogy of an ant making its way across a wind and wave tossed beach. The ant's trail is irregular, with many twists and turns, though there is a general large scale direction to its movements. If its path were drawn on a piece of paper, the path would be quite complex and difficult to describe. But this complexity resides in the environment (the twists and turns of the sand) rather than in the ant. A simple behavioral repertoire can generate complex behavior in a complex environment. To the extent that human behavior is analogous to that of an ant, human behavior can be modeled with a simple cognitive architecture interacting with a complex environment. This environment includes the internal environment of memory. Starting with this model, the cognitive structure of programming can be described in two stages: the invariant and relatively simple human processing architecture, and the structure of the internal (programming knowledge) and external (programming) environment.

ARCHITECTURE OF COGNITION

Memory

Short Term Memory. There are two types of memory, short term and long term. Short term memory (STM) can hold approximately 7 +- 2 chunks of information [Miller, 1956]. These chunks tend to decay rather rapidly. Short term memory stores the immediate context of cognition. The term "chunk" is somewhat ambiguous. It is any piece of information that can be manipulated as a unit. Chunking is a generic learning strategy. This is the process by which smaller pieces of information which were at one time manipulated as independent entities, become combined into a larger structure. Chunking can increase the effective capacity of short term memory.

Long Term Memory. Long term memory (LTM) has a virtually unlimited capacity. The information in LTM is often characterized as stored in a highly elaborate network (semantic net). This net provides a large number of associations through which stored data can be recalled. The unit of LTM is called the schema [Curtis, 1989]. This is a knowledge structure that bundles together the information necessary to manipulate a concept. The construction of these schema is facilitated by the chunking process in STM.

Processor

The human processor is responsible for a complex interplay between STM and LTM [Card, Moran and Newell, 1983] called the recognize-act cycle. The chunks in STM associatively trigger chunks in LTM. These LTM chunks are loaded into STM, where they prime the next cycle. The recognize-act cycle is the fundamental unit of cognition. More complex acts such as planning are built out of organized sequences of these cycles.

GOAL-DIRECTED MODEL OF COGNITION

The next level of specificity in cognitive modeling is a goal-directed model of cognition which Simon calls design [1981]. Programming is a special case of this cognitive activity. The basic recognize-act cycle has no constraints on the type of associations that take place. The associations between STM and LTM could result in a crazy chaotic jumble of chunks becoming present in STM, or, at the other extreme, in a few chunks repeatedly firing each other ad infinitum. In normal human behavior, however, this rarely happens. Instead, thinking results in a relatively ordered movement towards a goal. This general process of goal-directed thinking Simon calls design, where design is construed most generally (eg. design of buildings, actions, mathematical solutions, etc.). In design, search is in general necessary because many chunks in LTM are associated with any given chunk in STM. Since a limited number of these associations can actually be loaded into STM, they will need to be explored serially. The search process is driven by two types of domain knowledge, declarative knowledge (facts) about the domain, and procedural knowledge that describes how to manipulate the facts in an orderly manner. A domain expert is distinguished both by the number of chunks of knowledge they have and the structure of the association web linking these objects.

This general structure implies that a model of programming should account for two types of knowledge: declarative and procedural. Declarative knowledge schemas represent the "facts" of programming and procedural design knowledge is used to guide the process of programming. The programming and design models below are presented in increasing order of complexity and specificity.

PROGRAMMING KNOWLEDGE

Syntactic and Semantic Knowledge

Shneiderman and Mayer [1979] divided the declarative knowledge of programming into two categories, syntactic and semantic. Syntactic knowledge is knowledge of the mechanics of programming languages. It is acquired essentially by rote. New knowledge will often interfere with previous knowledge since new syntactic knowledge tends to be additive rather than integrative. For example, knowledge of the syntax of the assignment operators in C and Pascal ("=" and ":=") can't be integrated; the syntactic difference must be memorized.

Semantic knowledge consists of language independent programming concepts. This knowledge exists at several levels of abstraction. At the low abstraction end are concepts like the "actions" of assignment and conditional statements. At an intermediate level are concepts like looking for a maximum value in an array or swapping the contents of two variables. At a high level of abstraction are concepts like searching and sorting methods.

Plans

Soloway et al [1984a, 1984c] present a refined model of the declarative knowledge. They argue that this knowledge is structured in a hierarchy of plan types. Strategic plans describe a global decomposition strategy for an algorithm, such as a process/read looping strategy. Tactical plans describe more concrete operations, such as a running total loop. Finally, implementation plans specify the manner in which the more abstract plans are implemented in the code of some particular programming language. For example, a while loop in Pascal that is used to repeatedly read in numbers and add them to a running total until a stop number is entered is using both the strategic plan read/process (abstract structure of the loop) and the tactical plan running total (particular processing to be performed by loop). The strategic and tactical plans are language independent; the implementation plans are language dependent. In addition to the various levels of plan knowledge, line level semantic and syntactic knowledge of the language used are also necessary in order to carry out the low level implementation of the plans.

There are interactions between the implementation plans and the more abstract plans. Different languages make different abstract plans more difficult to implement than others. Proper use of a language involves learning the abstract plans which are most elegantly implemented in the language. However, since the abstract plans are stored as separate chunks of knowledge from the implementation plans, the abstract plans are not automatically adjusted by switching to a new language with new implementation plans [Scholtz, in press]. Scholtz showed that Pascal programmers writing code in Ada and Icon tended to use Pascal-like plans in both languages, even though Pascal-like abstract plans are not the ones most readily expressed in Icon (a string processing language). To the extent that the same abstract plans are readily supported across two languages, positive transfer will be seen. To the extent that the new language facilitates a different set of abstract plans, negative transfer will be seen. Since plans are built by a process of chunking, it makes sense that the lower level semantic features of a language, which serve as the building blocks, will influence the structure of the abstract plans.

Code Level Plan Structure

Rist [in press] argues that the deep structure of a program is its plan structure. The particular ordering of code statements in any given program is its surface (or shallow) structure. The surface structure is generated by applying an organizational scheme to the deep structure. The three organizational schemas he presents are procedural, functional and object schemas. His definition of plan structure is different from Soloway's. For Rist, the plan structure is the control and data flow graph of the program. The nodes of this graph are individual lines of code. A plan is a branch of this plan structure. What Soloway calls a plan, Rist calls a plan schema. Plan schemas are known solutions to common programming problems. A plan (branch of the graph) may or may not be a plan schema. A tool called PARE has been written by Rist to extract the plan structure automatically from the program code. Functional, procedural and object-oriented programming paradigms can be explained in terms of an organizational schema applied to a deep structure. A problem with this approach is that it is knowledge poor. By looking just at the program code, deep structure contains only lines of program code connected by control and data flow links. The plan schema captured by this representation will only be at the implementation level. As Scholtz [in press] showed, more abstract plan schemas do influence implementation plans across programming paradigms. Further, different abstract plans are more appropriate in different languages. The deep structure contains none of this more abstract knowledge. So while this view of plan structure is certainly well-defined and useful for talking about implementation level plans, it does not appear to suffice as a cognitive model of programming knowledge.

DESIGN KNOWLEDGE

The hierarchy of plan types and the syntactic/semantic knowledge of a programming language serves as a model of declarative programming knowledge. This section presents a model of the declarative and procedural knowledge of the design process and the relationship between the application domain (where the problem that needs to be solved resides) and the programming domain (where the solution artifact resides). Most of the models below describe some aspect of the global design process. As with the programming knowledge models above, they are presented in increasing order of complexity and specificity. The first model presented, however, describes the local composition of plans rather than the global design process.

Rules of Discourse

At a low level of design control, there are rules which determine how plans are composed together. Soloway and Ehrlich [1984b] call this knowledge the rules of programming discourse. These rules are shown in Figure 1. Programs that follow these rules are called planlike and programs that break these rules un-planlike.

Soloway and Ehrlich performed two experiments to test the rules of discourse hypothesis. In the first, a group of novices and a group of experts were given both planlike and un-planlike programs which had a missing line. With the planlike programs, experts were able to correctly supply the missing line much more often then novices. With the un-planlike programs, experts and novices performed about the same. In the second experiment, experts were given both planlike and un-planlike programs to study. They were then asked to reproduce the programs they had studied. Their recall was

1) Variable names should reflect function.

2) Do not include code that will not be used.

2a) If there is a test for a condition, then the condition must have the potential of being true.

3) A variable that is initialized via an assignment statement should be updated via an assignment statement.

4) Do not do double duty with code in a non-obvious way.

5) An IF should be used when a statement body is guaranteed to be executed only once, and a WHILE used when a statement body may need to be repeatedly executed.

Figure 1. The rules of programming discourse.

significantly higher with the planlike programs. Both of these experiments support the hypothesis that rules of discourse are a part of an experienced programmer's knowledge.

These discourse rules are low level. This design knowledge is at a level of abstraction similar to tactical plans. There must be more abstract knowledge to drive the design process.

Funnelling Control Strategy

Shneiderman and Mayer [1979] describe the control strategy of design as a "funnelling" from the abstract to the concrete. The programmer's internal representation of the program starts out general and becomes progressively more concrete until specific code details are worked out. This working out of a detailed representation of the code can proceed in a top-down or bottom-up manner. Top-down design requires that the more general aspects of the internal representation are worked out before the more particular aspects.

Bottom-up design starts with language statements and builds up more abstract structures. Shneiderman and Mayer have no model of when top-down or bottom-up design will occur. They mention that perhaps some types of problems are suited more to one technique than the other.

Design Schema

Jeffries, Turner, Polson, and Atwood [1981] describe a design schema which controls the recursive decomposition of a problem. A problem is broken down into smaller and smaller components until recognized solution components (plans) are found. They provide a list of 11 abstract productions which characterize the design schema. A list of unsolved subproblems is maintained and stored in sorted order of priority. A highest priority subproblem is pulled off of the list, and evaluated. Either it is a problem for which a solution is readily available, or it is not. If not, a solution model is created for the problem. A search is done for a solution matching the solution model. If none is found, the solution model is decomposed into subproblems and the subproblems are placed on the list.

This model of the design process adds specificity to the funnelling model. However, there are some questions which still need to be answered. When will top-down and bottom-up design processes be seen? During the decomposition process, some internal representation must be used before the program (or pieces of the program) has reached the level of specificity of the strategic plan. What is this representation? How does knowledge of the application domain (real world) and specific knowledge in CS (such as space/time efficiency) interact with the decomposition process? The next three models provide answers for these questions.

Unified Model of Top-Down and Bottom-Up Design

Rist [1989, in press] provides a unified model of top-down and bottom-up processes at the level of tactical plans. Every plan has a goal and a focus. The goal is what the plan is supposed to achieve. The focus is the primary line of code which achieves the goal. The rest of the plan is support for the focus. For example, a counting plan has the goal of counting the number of times something occurs. The focus line is "count := count + 1." The rest of the plan (a loop plus an initialization of count to 0), supports the focus. Top-down development takes place when the plan is already known. The goal is used to retrieve the plan. If a plan to accomplish the goal is not known, however, the plan must be constructed. This starts from the focus line since this line is most directly related to the goal. Development then proceeds outwards from the focus in a bottom-up manner. The newly created plan is stored, and, with repeated use, becomes automatic.

Design Executive

Adelson and Soloway [1985] provide a model of the design process in which six behaviors occur: mental model formation, systematic mental model expansion, mental model simulation, constraint representation, plan label retrieval, and note making. The mental model is a representation of the design in progress. This mental model must support simulation of the design. Simulation is used to compare the problem statement with the design. The problem statement is generally given as a desired behavior for an artifact. The design model is not a behavior; it is a partially designed mechanism. By simulating the design model, the resulting behavior can be compared with the desired behavior and steps taken to reduce the differences. As time progresses, the design model becomes more specific. This expansion takes place systematically across the design. Systematic expansion is required in order to maintain a simulateable model. In order to produce a simulation, the i/o of the components of a model must be at the same level of description. The constraint representation of a design is an alternative representation which supports property assertion and the deduction of implications. A runable mental model cannot be constructed until the design has reached a certain level of specificity. The constraint model is used to constrain the allowable simulation models and thus aid in achieving the requisite specificity. Constraint activity is seen most often in the early part of a design. Plan labels are retrieved when elements of the design already have solutions stored in memory. They serve as place holders; later they will be used as an index to look up the plan. Note making behavior is seen when concerns are raised that are not at the current level of detail of the mental model. When the model has been expanded to a level matching that of the concerns, the notes are used to remind the designer of things to consider at that level.

Adelson and Soloway's design model provides an internal representation used during the decomposition process. Decomposition is driven by a simulateable model of the design. This model serves as the representation at levels of detail more abstract than that of plans. Domain knowledge still needs to be incorporated into the process. Adelson and Soloway found that some design behaviors change for a designer in an unfamiliar domain. In particular, global models are not built and consequently global simulation is not performed. This leads to errors in systematic expansion. Thus domain knowledge certainly seems important in design.

Domain and Algorithm Spaces

Kant [1985] describes a model which explicitly takes account of the domain knowledge. Design is seen to take place in two main search spaces and two auxiliary search spaces. The main search spaces are the domain and algorithm spaces. The algorithm space contains knowledge of implementation issues and the domain space contains knowledge of domain issues. The design model in her paper is motivated by protocols of experts developing a convex hull algorithm. In this case the domain space contains geometry and visual reasoning knowledge. The algorithm execution space and the example generation space are extensions of the algorithm space and domain space respectively. The algorithm execution space is where design simulation takes place. The example generation space is used to generate standard, degenerative, and counter-examples. These examples are used to test the algorithm in the execution space. Design is accomplished by performing searches in the various spaces. For example, if a component needs to be expanded and it is supposed to give a known output, construct an example of this output study and its properties. The example would be generated in the example space, and studied in the domain space. Properties in the domain space might trigger plans in the algorithm space. The design would be expanded and then executed (to test it) in the execution space. Kant's model directly accounts for the role of domain knowledge in design.

Unification of Design and Domain Knowledge

The most complete model of design to be presented here is Guindon's [1990]. Seven knowledge categories are used to guide design: domain knowledge, inferred and added constraints, external design notation, design methodology, design schema, problem solving and design heuristics, and preferred evaluation criteria. The domain knowledge is similar to Kant's domain space, with the exception that simulation can take place here as well. Domain simulations lead to discovery not only of solution knowledge but also of problem goals and evaluation criteria. Inferred and added constraints are used to constrain and disambiguate the problem specification. Inferred constraints are not given directly in the requirements but can be deduced from the requirements and domain knowledge. Added constraints are chosen by a designer to limit the range of possible solutions. For example, the protocol study performed by Guindon involved the design of an elevator control system. One designer chose early on to use a distributed control scheme to avoid having a single point that results in global breakdown. Though such a constraint appears nowhere in the specification, it serves to limit the allowed designs. External notations are used to support design simulation, which would otherwise be too cognitively taxing to perform. They also provide a set of operators for design expansion. Search during design can include searching for a good notation. Design methods provide a set of operators and their applicability tests for transforming the requirements into a solution. An example of a design method is the Jackson System Development Method. Design schemas are problem decompositions that are applicable to a set of problem types. For example, one of the designers recognized the elevator problem as a special case of resource allocation systems. Drawing on memories of a film controller design he'd done, he quickly sketched out a high-level solution decomposition including alternative solutions for sub-systems and evaluation criteria for these alternatives. Design heuristics guide the search process. Two example heuristics are 1) divide the system into nearly independent subsystems, and 2) solve a simplified version of a problem and expand the solution to encompass more complex situations. Finally, preferred criteria are used to limit the size of the search space. These criteria go hand in hand with the added constraints above. For example, the criterion of high reliability goes with the added constraint that there should be no single point of failure.

This last design model explicitly accounts for the role of domain knowledge as well as the effect of design methodologies and generic design heuristics, while retaining the basic pattern of design as a movement from the abstract to the concrete.

The cognitive model of programming builds on the generic architecture of the human information processor by detailing the declarative and procedural knowledge used during software design. Now this cognitive model can be used as a framework within which to study cognitively motivated systems for software reuse.

COGNITIVELY MOTIVATED SYSTEMS

Other research has focused on building systems based on cognitive principles to support software reuse. This section reviews some of these systems. After describing general cognitive concerns of software reuse, several systems are described which attempt to alleviate these problems. The next section describes the system developed in this thesis. This system builds on the ideas found in the cognitively motivated systems described below.

Cognitive Concerns of Software Reuse

Curtis [1989] points out that designers are already reusing knowledge when they design. Everything from the low level implementation plans to high level design schema can be view as reusable components. This implies that one approach to software reuse is to externalize the plans and schemas in a development environment. This would accomplish two goals. Novices could progress more rapidly towards expert performance by having the expert knowledge directly available for study and use. Experts could design more rapidly by having the knowledge structures they use directly available in a machine executable format, thus bypassing the need to physically translate the mental structures into machine structures. Curtis also mentions the importance of indexing the components in a manner that matches the programmer's model of the domain. When the structure of a component library matches the programmer's domain model, they can more easily switch between the domain space in which high level goals might be stated and the application space in which components exist. Fischer et al [1991] have listed six problems that programmers have in reusing software: they do not have well-formed goals or plans, do not know of the existence of components, do not know how to access components, do not know when to use components, do not understand the results produced by components, and do not know how to combine, adapt, and modify components. Several cognitively motivated systems have been built which attempt to alleviate some or all of these problems.

Programmer's Apprentice

Rich and Waters' Programmers Apprentice [1989, 1990] contains a knowledge base of reusable implementation plans called cliches. The plans are stored using a language independent knowledge representation scheme called the plan calculus. The plans are accessed using an an extension of the

EMACS editor called KBEmacs. By using a special plan editing and description language, a relatively short description in the plan language is turned into a program in the target language. The Programmer's Apprentice can be viewed as a very high level language as described in the first section. It provides leverage by taking a small set of high level descriptions (plan labels) and turning them into a detailed program. In addition, it supports the creation of plans using the target language with some machine readable annotations. These are converted into the language neutral plan calculus. Rich and Waters do not explicitly motivate their system from cognitive grounds. Yet from the general cognitive model of software design, we can see that the Programmer's Apprentice works by reifying implementation plans.

Bridge

Bonar and Liffick [1991] describe an alternative method for reusing plans in their Pascal tutor called Bridge. They also use a high level programming approach, but for them a high level language means the vague, heuristic sort of specifications used by humans while talking to each other rather than the more algebraic formalisms demanded by automatic programming systems. The Bridge tutor leads a novice through a problem in three steps. First the user specifies high level informal plans in natural language. The system then leads the user through specifying more detailed plans using an iconic language. Finally, the iconic plans are implemented in Pascal code. At each level of abstraction, Bridge can detect buggy plans. There are facilities for the user to add new plans by means of an iconic (visual) programming language. Besides making available a library of plans for reuse, Bridge guides the user through the design stages from abstract, vague specification to detailed code.

Fischer's Systems

Architecture. Fischer [1987] describes a generic architecture for intelligent design environments. Such environments are intended to support incremental, evolutionary reuse and redesign. The 8 major components of this architecture are the visualization system, design kits, documentation system, analysis system, critics, help system, instructional system, and explanation system. The visualization system provides a graphic representation of a design, a design kit provides a set of components useful within a domain, the documentation system supports design rational and argumentation among multiple designers, the analysis system runs design simulations, critics are knowledge based agents which comment on a growing design, the help system provides online help about the design system and its domain, the instructional system provides tutoring about the design system and its domain, and the explanation system aids in comprehension of the components available in the design kit. The functionality provided by the design kit and explanation system are described in more detail below, since these two systems most closely provide the code location and comprehension facilities being focused on in this thesis.

Design Kits. Design kits support what Fischer calls human problem-domain communication [Fischer and Lemke, 1988]. By communicating with a computer tool in the language of the problem domain, the user is relieved of the burden of translating goals and operators in the problem domain into the system domain. This reduces (or eliminates) the difference between the domain space and application space, thus simplifying the design process. The construction kits Fischer describes generally involve direct manipulation of a palette of tools. Design environments provide further aid via knowledge based systems such as critics, which offer advice as a design progresses [Fischer et al, 1992]. Reusable component comprehension is aided via the explanation system, which presents annotated examples of components in use [Fischer et al, 1991]. A design kit can include a query system to locate reusable components. This becomes necessary as the number of reusable components increases; direct manipulation of component palettes becomes unwieldy. The Codefinder system [Fischer et al, 1991] supports query by reformulation. A semantic network is used to describe the components. Spreading activation is used to locate components near the first set of query keywords. In response to a query, a list of components sorted in order of activation strength and a list of related (activated) keywords is presented. Keywords and components can be added to the query. By iterating in this manner, the system helps the user narrow an initially vague query. The spreading activation helps alleviate problems with indexing inconsistency.

Explanation System. Fischer's explanation system makes use of examples to facilitate component comprehension. Other workers have found this to be an effective technique [Neal, 1990, Rosson and Carroll, in press]. Neal created a base of Pascal examples. The programmers in her study used the examples both for code reuse (at a plan level) and to understand Macintosh specific language features and procedures. Rosson and Carroll seeded a Smalltalk environment with application examples. The subjects in their study not only reused the components they found in the example applications, they also reused the patterns of component use. That is, code from the example applications served as templates for their own application. This again shows reuse of plan level knowledge embedded in the example as well as reuse of the components used in the example.

Motivated Redesign of the Smalltalk Browser

Another approach to the cognitively motivated design of software environments is to analyze pre-existing environments and use the analysis to suggest improvements. Bellamy [in press] takes this approach by applying strategy analysis to the Smalltalk environment. Strategy analysis is an extension of claims analysis. In claims analysis, one attempts to articulate the psychological theory implicit in a tool. This psychological theory is making claims about the way people work; by examining these claims, the tool can be redesigned to embody a more truthful psychological theory. Strategy analysis extends claims analysis by offering a specific methodology for producing analyses by looking at strategies of tool use. The primary problem Bellamy found was the difficulty of locating reusable components in the Smalltalk hierarchy. The browsing approach supports serendipitous discovery of new classes, but distracts the user from the original task, sometimes to the point that the user loses track of the original task. The class names are not always a good indicator of function. Examining a working application suggests reuse possibilities, but it can be difficult to map application behavior into specific classes and methods. Tracing code in the debugger places a heavy load on short term memory, as the user has to maintain the context of many classes and methods spread throughout the hierarchy. By providing an enhanced browser which supports multiple views of the hierarchy, and a project organizer which maintains a context for all the Smalltalk tools (including the browser) within a particular application, Bellamy was able to mitigate many of these problems.

Motivated Redesign of a Smalltalk Tutor

Singley, Carroll, and Alpert [1991] take a similar approach with the design of a Smalltalk tutor motivated through claims analysis. They found that users had trouble managing goals, finding classes, and comprehending code. The goal management problems are related to the serendipitous browsing supported by the environment. Both low and high level goals can be forgotten during this browsing process. The Goalposter maintains a list of goals which the system has inferred from user activities. It is acting as an externalization of part of short term memory. The Adaptive Index adds a query mechanism to the browser. Only those classes and methods related to the query are shown. The comprehension process is aided by a commentator, which provides hypertext help on selected pieces of code. Finally, the need for a Guru is hypothesized. The Guru performs a post-mortem analysis of the user's project, both clarifying the design process and offering suggestions for improvement.

The Cognitive Browser

Green et al [in press] have designed a cognitive browser for object oriented programming systems. They analyze object oriented programming in terms of 5 cognitive dimensions: viscosity, hidden dependencies, premature commitment, perceptual cueing, and role expressiveness. Viscosity is resistance to change. It disturbs working memory by requiring the user to manually manage some complex change operation. Hidden dependencies are links between entities which are not readily apparent. These dependencies include links which are important to the user, not just those deemed important by the environment designer (usually the dependencies which are technically easy to show). The larger the number of dependencies (hidden or not), the greater the viscosity of the system. Premature commitment occurs when the user must make decisions too early in the design process. This occurs in object oriented systems which force the design of class structures in a top-down manner. If the system has high viscosity, premature commitment is a big problem, since decisions made early in the design process are going to be difficult to change. Perceptual cueing is the redundant coding of important attributes in a notational system. An example is the coding of functional grouping in the layout of an electronics schematic. Role expressiveness is the ease with which a user can comprehend meaningful structure. Unlike the other dimensions, which are structural, this is a mentalistic dimension. Role expressiveness is a function of how easily the user can translate between the notational system provided by the environment and some internal mental representation. For example, if programming plans are taken to be an internal mental structure (as described above), then the systems which reify plan structures in their notational system (such as the Bridge tutor) should exhibit high role expressiveness.

In terms of these cognitive dimensions, Green et al find that Smalltalk-like environments exhibit extensive premature commitment, high viscosity, and poor role-expressiveness. This leads them to the following requirements for a cognitive browser: code location support, code comprehension support, design rational, and support for modification of entire class structures. The basis for providing this support is a description level. They note that program code does not express all of the programmer's knowledge about the program.

The description level is a place to store this knowledge. This description can be used to support location, changes and comprehension.

The description level supports code location by allowing three query styles: attribute searching, query by effect, and query by analogy. Attribute searching is the most straightforward. The user can invent attributes and relations and place them in the description level. Queries take the form "Show me all the components with attributes A, B, C." Effectively, the description level supports the user in creating dynamic indexing schemes.

Search by effect allows the user to specify a desired effect for a piece of code. The browser searches for code which satisfies the effect. The problem is how to specify the desired effect. One approach is to use a formal, declarative specification language. Green et al provide an example of such a specification for a stack. This formal approach, however, has several problems. For large chunks, such as a text editing window, the formal specification becomes unwieldy. Further, the specification may not match the user's domain model. The user probably does not think of a stack as a set of preconditions, postconditions and operators, but rather as an entity labeled by the term "stack." This label can of course be expanded, but this requires a cognitive effort that may be equivalent to writing the code. In plan terms, a formal specification of a stack is a description of the plan rather than a plan label. As Adelson and Soloway [1985] showed, designers retrieve plans by label.

The third search modality, search by analogy, uses some kind of similarity measure to answer queries of the form "Find something similar to A." Computational models of analogical reasoning can be used as a basis for such functionality.

THE INTELLIGENT CODE OBJECT PLANNER

The system developed in this thesis, the Intelligent Code Object Planner (ICOP), supports code location and comprehension in complex libraries. To make this problem concrete, ICOP has been instantiated to support a subset of the Windows 3.0 Application Program Interface (API).

System Components

ICOP consists of four main components: a limited natural language interface, a knowledge base, a planning system and an example generation system. The natural language interface accepts English queries regarding a library and translates them into a semantic formalism. The knowledge base uses a frame language to describe the components of the library. The planning system accepts the semantic representation of the query returned by the natural language interface and attempts to find a set of library objects (routines, etc.) which satisfy the query. Finally, the example generation system constructs example code out of the plan returned from the planning system. Each of these components is described in greater detail in the next chapter.

Design Motivation

Programmers working with a complex library want to create specific effects. For example, with a windowing library, a programmer may want to create a window. This requires the use of more than one atomic library component. A plan utilizing multiple library components is needed. Since programmers move from goal, to plan focus, to plan generation, ICOP facilitates library reuse by accepting a focal goal and returning the detailed plan. ICOP effectively supports the recall of plans by label, where the label expresses the goal of the plan. The limited natural language interface allows the programmer to use their natural model of the domain. In the case of a windowing system, common domain objects are windows, controls and messages. ICOP allows the programmer to talk directly in terms of these words rather than having to learn some formal specification language. If experimental protocols reveal that programmers wish to refer to certain effects in a manner different than that supported by the current interface, it is easy to add synonyms to the interface by expanding the vocabulary. This decoupling of the the domain model from the library details supports transfer between libraries in the same domain. The knowledge base can be expanded to describe the new library while queries are still made using the same vocabulary. Once ICOP has understood a natural language domain query and found a library plan which performs the desired effect, it must communicate this plan to the programmer. Examples are an effective way to communicate to programmers. By expressing the plan directly in example code, the programmer does not have to understand some intermediate formalism and then translate this into program code.

Relationship to Other Cognitively Motivated Systems

ICOP builds on ideas found in the cognitively motivated systems described in the previous section. It uses the concept of plans, which is found in The Programmer's Apprentice [Rich and Waters, 1989, 1990] and Bridge [Bonar and Liffick, 1991]. However, rather than having a plan base of explicitly stored plans, ICOP builds plans from the atomic components represented in its knowledge base. The example systems developed by Neal [1990], Rosson and Carroll [in press] and Fischer, Henninger and Redmiles [1991] all use examples to facilitate both component level and plan level comprehension.

ICOP also uses examples to facilitate comprehension, but rather than storing these examples explicitly in an example base, the examples are constructed dynamically using knowledge about the general form that examples should take. The automatic construction of plans and examples frees the knowledge engineer from having to explicitly represent examples for all possible user queries. ICOP allows the user to query by effect as suggested in the Cognitive Browser project [Green et al, in press]. Rather than using a formal effect language, however, the user can communicate the desired effect in the natural language of the domain.

CHAPTER III

THE KNOWLEDGE BASE

FRAME SYSTEM

Generic Frame System

Frames and Slots. A frame is a collection of fields, called slots, which describe some entity in the world. The slots hold values. In addition, the slots can have an internal structure of subfields (called facets) which describe properties of the slot such as default value or the type of values allowed. When a slot holds a value which is the name of another frame, this represents some relationship between the two frames. The primary actions that one performs on slots are getting and setting values. Two relationships are considered quite important: isA and instanceOf. IsA relates a frame which represents a set or class to its superset. InstanceOf relates a frame which represents some non-set entity to the set to which it belongs. Both of these relationships have their inverses which can be explicitly represented as slots. A form of default reasoning called inheritance takes place along the isA and instanceOf slots. When a value is requested from a slot which has none, a frame system will check the corresponding slot on the frame pointed to by the isA or instanceOf slot. In this way, default values for slots can be stored in frames representing general things.

Attachments and Methods. Procedural knowledge can be attached to frames in the form of slot attachments and methods. Slots attachments are pieces of procedural code which are activated when the slot is accessed in particular ways. Two common attachments are the get-attachment and the put-attachment. These pieces of code fire when a get or put is performed on the value of a slot, respectively. A get-attachment can be useful for calculating a value which depends on the values of multiple slots.

A method is a named piece of procedural code which is attached to a frame, but not to any particular slot. It corresponds to the idea of method in object-oriented programming. Both methods and procedural attachments can be inherited.

Figure 2 illustrates some of these ideas. The Person frame is a subset (isA) of Mammal and an instanceOf Class. Class is a frame which stores general information about frames which represent classes. This shows that a frame can simultaneously represent both a set (in this case the set of people) and a thing (in this case the thing that is the set of people). The Age slot contains a default age for people. This slot demonstrates the idea of facets. Age has two facets, a value and a type. The type can be used by the frame system to insure that objects of the right type are used to fill the value of the slot. The Computer-Scientist frame is also a set and a thing. It is a subset of the set of people and a member of the set of professions. The Michael-Mateas frame describes a particular person. This person belongs to two sets (Adult-Male and Computer-Scientist). The isA and instanceOf relationships do not have to be filled by only a single frame. If these relationships are limited to single frames, the system is single inheritance; the frames are organized in an inheritance hierarchy. If multiple frames are allowed to fill these slots (as in this example), the system exhibits multiple inheritance. The frames are arranged in an inheritance graph (directed acyclic graph, or DAG). The Age

```
Person
        isA: Mammal
        instanceOf:Class
        Age:{ value:40, type:integer }

Adult-Male
        isA:Person
        sex:Male

Computer-Scientist
        isA:Person
        instanceOf:Profession
        personalStyle:nerd

Michael-Mateas
        instanceOf:{Computer-Scientist, Adult-Male}
        Age:{ value:27, type }
        personalStyle:wired
```

Figure 2. Example of generic frame system.

slot has a value which overrides the value of 40 which would normally be inherited all the way from Person. The type of the age slot, however, is inherited from Person. The personalStyle slot overrides the value that would be inherited from Computer-Scientist.

Reifying Slots. The basic frame representation can be extended by representing slots as frames. This extends the idea of giving slots an internal structure with facets. Where the facets of a slot must be repeated every time the slot is used, a frame describing a slot allows one to create a single localized description of the slot. It makes the slot into a thing. The kinds of slots one might find on a frame describing a slot are domain (the kind of frames this slot can legally be attached to), range (the kinds of values that can fill the slot), and of course isA and instanceOf. By including a slot transfersThrough which is filled by a list of slots through which this slot can acquire default values by inheritance, one effectively places a frame containing this slot within multiple DAGs. Now any arbitrary relation can have the power of isA as an organizing relation for a set of frames.

Frame System for ICOP

The frame system used in this thesis is written in Prolog. The core of this system was written by David Novick at the Oregon Graduate Institute. It has been enhanced with type checking capabilities.

Note on Terminology. In the rest of this thesis, any symbolic names which refer to any piece of the Windows library or some part of ICOP appear in italics. In addition, predicates and functors are distinguished by following them with a "/" and the number of arguments. Thus, an arity two predicate *my_predicate* is written as *my_predicate/*2. Generally, the number of arguments is not important; the "/" notation is used to distinguish predicate and functor names from other names in the text.

Frames. A frame is represented by the arity four functor shown in Figure 3.

```
frame(name:<frame_name>,
parent:<parent_name>,
children:[<child_1>, <child_n>],
slots:[<slot_1>:facets([<facet_1>, <facet_n>]),
       <slot_n>:facets([<facet_1>, <facet_n>])]).
```

Figure 3. Frame functor.

Any name in angle brackets is a metavariable. In a real frame, these names will be replaced by names chosen by the user. The ":" has been defined as a Prolog infix operator. The first argument in the frame functor is the name of the frame. The second argument is the name of the parent frame. Inheritance flows through this argument. Only one parent is allowed, making this a single inheritance system. The third argument is a list of children frames. These frames lie below the given frame in the inheritance hierarchy. Finally, the last argument is a list of slots belonging to this frame. Each slot has list of facets. By convention, one of these facets should be called *value*. This facet actually holds the value which fills the slot. The rest of these facets describe or modify the value in some way.

Operations on Frames.The primary functions supported by the frame system are getting the value of a slot facet and setting the value of a slot facet. While the names chosen for facets are arbitrary, the frame system provides special semantics for two facet names. The name *value* is the value of the slot. If one gets or sets a slot without specifying a particular facet, the facet accessed is the *value* facet. The *default* facet provides the value for a slot in the event that the *value* facet is empty.

Procedural Attachments.The system also supports procedural attachments and methods. The procedural attachments are attached to a slot as a special facet named either *get_value* or *set_value*. The value of this facet is a conjunction of Prolog goals. When a get or set is performed on the *value* facet of a slot, the appropriate attachment is called after performing the get or set. In addition to the conjunction of goals, an attachment facet also stores an unbound variable (by convention named *Caller*) which is bound to the name of the frame the get or set was performed upon at call time. The predicates in the conjunction of goals can refer to the *Caller* variable. Since an attachment might be stored on a frame high in the inheritance hierarchy, the *Caller* variable allows the attached code to properly determine the context of the call when a frame lower in the hierarchy is manipulated.

Methods. Methods are stored as slots named *method* having a special structure, rather than the normal list of facets. This structure is built using the ":" and ":-" operators and includes the name of the method, an unbound variable to hold the call context (in the same manner as described above for attachments), and a conjunction of Prolog goals. If there are several methods attached to a frame, there will be several slots named *method.* Normally, one would not want to put multiple slots having the same name on a frame. But the method slot has been given special semantics so that methods are distinguished by their names, which are part of the internal structure of the slot. The frame system provides a special predicate to allow the user of the system to call a method on a frame. In the case of a call to either an attachment or method, the frame system attempts to satisfy the goals constituting the attachment or method through the use of the built in control predicate *call/1*.

Properties of the Frame System. There are several interesting things to note about this frame system. First it is single inheritance. Each frame sits somewhere in an inheritance tree rather than an inheritance DAG. Second, inheritance only flows along the parent link. In general, inheritance links can be view as just another slot. The *isA* and *instanceOf* slots just happen to be two which are widely used. Here, the parent and children links have been distinguished from the rest of the slots. So not only is the system single inheritance, each frame sits in a single tree, rather than in multiple trees coordinated by multiple slots. In the example in Figure 2 (page 35), a *Computer-Scientist isA Person* (indicating a superset relationship) and is an *instanceOf Profession* (indicating a membership relationship). Inheritance flows along both of these links. Such a structure is not possible in the frame system used in this thesis. An ambiguity that then arises is whether the *Parent* relationship indicates superset or membership. This is a problem that will be discused in the section describing the knowledge base. Finally, the slots are structured by facets, but have not been reified as independent frames.

Type Checking. To help maintain knowledge base coherence, a type system was added to the core frame system described above. Every slot can have an additional facet named *type*. The value of this facet describes the type of the filler that the *value* facet can take. The types currently recognized by the type system are *frame, atom, integer, string, none, list_of(<type>)* (where *<type>* is any of the recognized types), and a name that is the name of some frame. A value is checked for conformance with the *atom, integer,* and *string* types through the use of built-in predicates. Any value conforms to the type *none*. A value conforms to the *frame* type if this value is the name of some frame. A value conforms to *list_of(<type>)* if the value is a list and each member of the list conforms to *<type>* (which is checked by recursively calling the type checker on each member of the list). A value conforms to a type which is the name of some frame if the value is the name of a frame which is subsumed by the type in the inheritance hierarchy. For example, if *Computer-Scientist* is a child of *Profession, Computer-Scientist* can legally fill the value facet of a slot whose type is *Profession*. A value facet can also be filled with a name that takes arguments (a predicate or functor). The predicates and functors used to describe relationships in the knowledge base are themselves represented as frames. These frames describe the number and type of objects that can be taken as arguments, and, in the case of functors, the type of object returned. If the value of a slot is a predicate, it conforms to the type (which is the name of some frame) if the predicate name is subsumed by the type, and each argument of the predicate is subsumed by the corresponding type found in the list of valid arguments in the predicate frame. If the value of a slot is a functor, it conforms to the type if the return object of the functor is subsumed by the type and each argument of the functor is subsumed by the corresponding type found in the list of valid arguments in the functor frame. Whenever an attempt is made to set the value of a slot, type checking is performed. The set will fail if the type check fails. The type check is defined to succeed for slots which do not have a *type* facet or for slots whose *type* facet is empty.

Consistency Checking

In building a knowledge base of even this small size, it becomes important to automate some consistency checks, especially since frames are richly related, causing a change in one part of the knowledge base to propagate to other parts of the knowledge base. The consistency checker checks that the frames are syntactically well formed, every frame is mentioned in the child list of its parent, every frame in a child list exists, every slot has a value, every slot has a type, and that the value of every slot conforms to its type. A frame is considered well formed if it has the form *frame(name:_, parent:_, children:_, slots:_)* where the "_" matches any string. The next two checks make sure that the parent-child links are consistent. The value check is more of a warning than an error. Many frames may have slots without values (because they represent an abstract entity), but it is a good idea to look at a list of these slots to make sure that some slot which should have a value has not been overlooked. A slot without a type is an error; every slot should have a type. The type check is performed in the same manner as described above. The consistency checker calls predicates in the frame system to perform this check. In addition, the method *check_type* (described below in the section on routine frames) is called on frames for which the other checks fail. The iterative construction of ICOP's knowledge base was greatly simplified by utilizing this semantic consistency checker.

GLOBAL ONTOLOGY

ICOP's knowledge base currently describes a subset of the Windows 3.0 API. The description of the knowledge base is divided into two parts. First, the ontology of the knowledge base is described. This gives a global map of its structure. Then, a detailed description of the frame type used to describe routines is given. Routine frames are the most complicated frames in the knowledge base. Since most other entities in the knowledge base support the expression of routines, a detailed description of routine frames should provide a good understanding of other parts of the knowledge base as well.

An ontology for a domain describes the existent entities for that domain. The decision to make something a "thing" is arbitrary. The final test of an ontology is adequacy; is the ontology's way of chopping up the domain adequate to the task at hand? Each domain entity is described by a frame in the knowledge base. ICOP's ontology for Windows 3.0 is motivated both by the need to support code location and comprehension, and by the desire to construct a deep model of the domain. It is hoped that this deep model will allow ICOP to be more easily extended to handle other domains (both other windowing systems and other types of libraries entirely) as well as support an intelligent tutoring system. Thus, there may be parts of the knowledge base that seem deeper than strictly necessary to support a code location system.

Some frames have no slot structure. For the kind of planning currently supported by ICOP, the mere existence of certain named frames suffices. To support an intelligent tutoring and documentation system, these frames would be given a more complex structure. For example, frames describing memory resources do not currently have any slots. These frames could be enriched by adding slots which describe the actual memory structure of the resource. The idea is that the global ontology is deep enough to support these more complex reasoning processes. The structureless frames provide locations ("hooks") where more detailed knowledge can be entered.

Routines

The primary entities in the knowledge base are routines. After all, ICOP supports the reuse of code objects, and in procedural libraries routines are the primary objects available for reuse. These routines are organized in their own hierarchy. The principal of organization of this hierarchy is that routines with similar effects are siblings in the hierarchy. Their parent will be a generic routine frame which does not actually describe a routine in the library, but rather describes the similarities that the children share. This will become clearer when the details of the structure of the routine frame are given and the planning algorithm is described. The effect of each routine is described by a slot filled with a list of predicates. These predicates are the fundamental domain operators recognized by ICOP.

Predicates and Functors

The predicates and functors used within the knowledge base are themselves described explicitly by frames. Predicate frames describe the predicate's arity and the legal types of its arguments. Functor frames include this same information and, in addition, describe a return type for the functor. Predicates are mappings from domain objects (in the knowledge base) to truth values. A functor is a mapping from domain objects (in the knowledge base) to another domain object.

Predicates. The predicate types found in the predicate hierarchy are *operator, constraint, state* and *source.* Operators are the fundamental actions recognized by ICOP. For example, *add_object(<container>, <memory_object>)* expresses that it is true that some *object* has been placed in some *container.* Containers and objects are other members of the ontology which will be described later. The effects of routines are described in terms of these operators. The user's query regarding a domain action is also translated into this operator language.

Constraints indicate that some constraining relationship exists between the arguments. For example, *assoc_type(<atom>, <frame_name>)* indicates that the type indicated by the *<frame_name>* (meaning that frame and all of its children) should be associated with an *atom,* where an *atom* is an arbitrary sequence of characters. In other words, some name has been given a type. The primary difference between constraints and operators is one of interpretation. Operators are intended to represent domain actions, while constraints represent static relationships.

State predicates indicate some relationship regarding the state of the system. For example, *received(<message>)* means that the message indicated by the argument has been received by a window.

Source predicates indicate the default source for a routine parameter. For example, *user_source(<user_source_object>)* expresses that a parameter will be provided by the user and that this parameter should be a

<user_source_object>. If the planning system had added some routine to its current plan, and *user_source* was indicated for one of its parameters, the planner would look no further to satisfy the parameter.

Functors. Just as in the predicate calculus, functors are used to describe composed objects. Some things in the world are best described by naming their parts. A functor is a function which takes as its arguments the parts of something and returns (constructs) the thing. An example is *icon_func(<icon>, <x>, <y>)* which takes an *icon* (meaning the actual piece of memory describing a bitmap which is an icon), a screen x coordinate and a screen y coordinate and returns a *drawable_icon* (mean the actual pattern of light which is placed on the screen). One can clearly see that the *icon* (piece of memory) and *drawable_icon* (screen object) are different objects by looking at their attributes. It makes no sense to talk about the screen location, height, and width of a piece of memory, while these attributes are valid for a pattern on the screen. Conversely, the pattern on the screen does not occupy a given number of bytes, while the memory pattern certainly does. Functors allow one to point to an object in the domain (in this case a pattern on the screen) by pointing to other objects (in this case a bitmap and a coordinate pair) out of which the first object can be built.

Objects

The predicates and functors express relationships between objects. These objects are themselves represented within a hierarchy of the knowledge base. The first level of objects is *container*, *memory_object*, *screen_object*, and *user_source_object*.

Containers. Containers are things that can contain other objects. *Memory* and *file* are containers. *System_memory* and *user_memory* are recognized as two subtypes of *memory*, and *executable_file* is recognized as a subtype of file. Currently, containers have no slots. These frames could be made richer by including such information as capacity, the type of objects stored in system memory vs. user memory, the fact the files have names, etc.

Memory Objects. Memory objects are objects which can occupy space in containers. The main types of memory objects are referenceable objects, declareable objects, state objects and values. Referenceable objects are objects which cannot be directly examined by a programmer. In the Windows domain, for example, a window is a memory object which contains all of the system information about a screen window. The contents and structure of this memory object cannot be directly examined. Instead, this object is passed to system routines indirectly by means of a handle. The referenceable object frames do not have any slots. These frames could be made richer by including information about the actual memory structure.

Declareable objects are typed memory objects which the user can manipulate in a manner similar to language built-in types. They can be declared, assigned to, examined, etc. Under *declareable_object* are *fundamental_type* and *library_type*. Fundamental types are the C built in types. Library types are defined by the library (in this case Windows). Subtypes of the library types are library defined structures, handles, pointers and simple types (simple typedefs of built-in types).

State objects contain some system state. For example, a device context contains information about the current pen, brush, background, color, clipping rectangle, etc. for a graphic device. State objects are similar to referenceable objects in that they cannot be directly examined by a programmer. Unlike referenceable objects, they cannot be pointed at indirectly by a handle. To gain state information, a programmer must usually make special calls which copy some aspect of a state object into a declareable object where it can be examined.

Value objects are values which declareable objects can take on. Many value objects are integer constants which have been *#defined* by Windows. These constant names can be used as arguments to various routines. For example, the background mode constants *opaque* and *transparent* are passed to *SetBkMode* and returned by *GetBkMode,* Windows routines which get and set the drawing background mode. By representing the concept of a background mode constant as a separate frame, it allows the argument and return values of routines which manipulate the background mode to be expressed accurately. Without the concept of this constant, one would have to express the values used by *SetBkMode* and *GetBkMode* as *integer.* Yet is is not the case that these routines accept or return any integer. Only those specific integers which have been given background mode names by Windows are valid for these routines.

Screen Objects. Screen objects represent actual patterns of color on the display screen or objects which are intimately involved in the display process. The three kinds of screen objects are device coordinates, drawable objects, and window parts. Device coordinates are the various coordinate systems that can be used to locate points on the screen. Drawable objects are the screen pattern associated with a memory representation. As explained in the description for icon_func above, the memory representation of an object and the corresponding pattern on the screen should be represented as distinct things because they have distinct sets of properties. Window parts are entities like scroll bar, title bar, and client area (the drawable area of a window). In addition to being patterns of color, a user can interact with a window part using the mouse or keyboard.

User Source Objects. User source objects are values that a user can supply to a routine. For example, *x_coord* represents an x coordinate that a user should supply. *Memory_object_name* represents a string which names some other memory object (eg. a resource) which the user should supply. These objects are the arguments to *user_source/1*.

Distinction Between Sets and Members

The *isA* and *instanceOf* slots express the distinction between sets and members. Though one could put slots with these names on frames in ICOP's knowledge base, the frame system does not give them their usual semantics. In ICOP, this distinction is made by location in the hierarchy. Leaves are individual things, anything else is implicitly a set (the set of all leaf frames which are below a given frame in the hierarchy). This is not a clean way to handle this distinction. For one thing, the distinction is made implicitly by the way the knowledge base is manipulated rather than in some explicit, declarative manner. Another problem is that it is not possible to express that something is both a set and a thing (like *Computer-Scientist* in the example). Finally, it is not possible to express that something is a set without also representing at least one member of the set (ie. placing at least one child in the child list). In future versions of ICOP, this representational deficiency should be remedied by giving inheritance semantics to slots named *isA* and *instanceOf*.

ROUTINE FRAMES

Root Routine

The root of the routine frame hierarchy is *root_routine.* This frame appears in Figure 4.

```
frame(name:root_routine,
        parent:[],
        children:[draw_icon,
                get_device_context,
                release_device_context,
                load_resource_abstract],
        slots:[ routine_name:facets([value,type:atom]),
                parameter_list:facets([value,type:list_of(declareable_object),
                        min:1]),
                default_source:facets([value, type:list_of(source), min:0]),
                return_normal:facets([value, type:declareable_object]),
                return_error:facets([value:null, type:value]),
                main_effect:facets([value, type:list_of(operator), min:1]),
                micro_effect:facets([value, type:list_of(operator), min:0]),
                constraint:facets([value, type:list_of(constraint), min:0]),
                preconditions:facets([value, type:list_of(state), min:0])]).
```

Figure 4. Root frame of the routine hierarchy.

The parent value is the empty list []. This indicates that the frame has no parent; it is the root of a hierarchy. The children slot contains a list of the immediate children frames. These frames should partition the set of effects that can be produced by routines in the library. Each of their children should partition the set of effects represented by their parent and so on, until one comes to a leaf frame which represents an actual library routine producing a specific effect. The slots describe the properties of a particular library routine or of some abstract routine (not actually available in the library) which represent a set of effects. These abstract frames are useful both for searching (as will be describe below in the section on the planner) and as a location to place slot values which should be inherited by multiple routine frames. Most slots in the *root_routine* have no value. There is no default value that makes sense at that level of abstraction. However, the type facet of every slot does have a value. All other routine frames will inherit these types.

The *routine_name* slot contains the actual library name of a routine. Its type is *atom*, meaning that the routine name can be any arbitrary sequence of characters. The *parameter_list* contains the list of parameter types which the routine takes. Its type is *list_of(declareable_object)*. If any atom is used in the *parameter_list* which is not the name of some frame in the *declareable_object* subhierarchy, a type error will be detected. The *default_source* is a list of sources for the parameters of the routine. This list is always the same length as *parameter_list*, with the sources in one-to-one correspondence with the parameters (that is, there should be one source for every parameter). If no default sources exist for a parameter (meaning that the planner should not assume that the value for a routine argument is going to come from some specific place), then the arity zero source predicate *no_default/0* is placed in the *default_source* list. The type of the *default_source* is *list_of(source)*, where *source* is a type of predicate. *Return_normal* and *return_error* describe the library routine return type in the event of normal termination and the return value in the event of error termination respectively. The *return_error* slot has a default value of null, as this is a common error return value for routines. The *main_effect* slot contains a list of operators which describe the main effect of the routine. Currently, the planner assumes that the main effect is described by a single operator. The *micro_effect* is a list of operators which describe in detail what the routine does. The planner does not currently access the *micro_effect*. It is intended to be used by the natural language generator of an intelligent tutoring and documentation system to describe the operation of a routine. The example given below of a leaf routine frame which describes an actual library routine should make the idea behind the *micro_effect* clearer. The *constraint* slot holds a list of constraints which pertain to objects mentioned in the main and micro effect slots. Currently, the only type of constraint used is *assoc_type/2*, which associates an atom with a type. If it does not prove useful to include any other type of constraint which pertains to the effect in this slot, the next version of the knowledge base could change the name of this slot to *associated_type* and change the type facet to *list_of(assoc_type)* rather than *list_of(constraint)*. The *preconditions* slot contains a list of state conditions which must hold in order for the routine to be called. For example, in the Windows API, the routine *BeginPaint* should only be called if the window to which *BeginPaint* refers has received a paint message. The *preconditions* slot allows this kind of information to be expressed.

Example Routine

The frame in Figure 5 describes the *DrawIcon* routine which draws an icon on the device associated with a device context (ie. screen, printer, etc.). The *routine_name* slot holds the library name of the *DrawIcon* routine. The *parameter_list* expresses that *DrawIcon* takes four parameters: a handle to a device context, two integers (which are coordinates) and a handle to an icon. The *default_source* expresses that the handle to the device context and the handle to the icon have no default source, while the two integers are x and y coordinates and should be supplied by the user. The *return_normal* and *return_error* slots express that *DrawIcon* normally returns a boolean (which an examination of the *micro_effect* reveals takes the value *true*) and that a

```
frame(name:draw_icon,
        parent:root_routine,
        children:[],
        slots:[  routine_name:facets([value:'DrawIcon', type]),
                 parameter_list:facets([
                          value:[hdevice_context, integer, integer, hicon],
                          type, min]),
                 default_source:facets([value:[no_source,
                          user_source(x_coord),
                          user_source(y_coord),
                          no_source],
                          type, min]),
                 return_normal:facets([value:boolean, type]),
                 return_error:facets([value:false, type]),
                 main_effect:facets([value:[
                          draw(device_1, icon_func(param_2, param_3, icon_1))
                          ], type, min]),
                 micro_effect:facets([value:
                          [dereference(param_1, system_memory_1, device_1),
                          dereference(param_4, user_memory_1, icon_1),
                          draw(device_1, icon_func(param_2, param_3, icon_1)),
                          return(true)],
                          type, min]),
                 constraint:facets([value:
                          [assoc_type(system_memory_1, system_memory),
                          assoc_type(device_1, device_context),
                          assoc_type(user_memory_1, user_memory),
                          assoc_type(icon_1, icon)], type, min]),
                 preconditions:facets([value:[
                          has_state(context_mapping_mode, mm_text,device_1)
                          ], type, min])])]).
```

Figure 5. Frame representing the routine *DrawIcon*.

value of false is returned if an error occurs. The main_effect expresses that the main effect of the *DrawIcon* routine is to draw on some device context the *drawable_icon* which is described by an *icon* and two coordinates, the coordinates being supplied by the second and third parameters.

The *micro_effect* expresses that the total operation of the routine involves dereferencing the first parameter (which is a handle to a device context) to retrieve the device context pointed to by the handle in system memory. The fourth parameter (which is a handle to an icon) is dereferenced to retrieve the icon in user memory. The icon is then drawn on the device context (this operator is the same one which occurs in the main effect) and the value *true* returned. Now it is clear what type of information is captured by the micro effect which is not captured in the main effect. Imagine a system which answers natural language queries regarding particular routines. A query such as "Which icon is drawn?" or "What is the first parameter used for?" is not answerable based only on the main effect. All one can say from the main effect is that some icon is drawn on some device context at the coordinates given by the second and third parameter. The *micro_effect* provides the richer knowledge which would be needed by a system which can answer such questions.

In the main and micro effects, arbitrary atoms are used as arguments to operators rather than the names of frames which describe objects. This will cause the predicate type checking procedure described above to fail. The *assoc_type/2* constraints which appear in the *constraint* slot solve this problem. Each of the atoms which appear in the effects slot is bound to some object which is described by a frame. Why not include the names of these objects directly in the effect operators rather than using this indirection? Direct use of object names would not work if more than one instance of an object is referred to in the micro effect. For example, an operation which copies some information between two device contexts will have to refer to two different device contexts in the micro effect. The indirection provided by *assoc_type/2* allows these two device contexts to be distinguished by choosing an arbitrary atom to name them. (eg. *source_context, destination_context*). The symbol *param_<number>* also appears as an argument to predicates and functors in the effect slots. This symbol has special semantics. It refers to the object which must be passed to the *<number>* parameter of the routine. The last slot, *preconditions*, expresses that the mapping mode of device context on which the icon is drawn must have the value *mm_text* (meaning that the x and y coordinates indicate the number of pixels in the x and y direction from the origin).

Example of Type Checking

Now that the meanings of the routine frame slots have been described and a particular routine frame has been examined, type checking on each slot can be stepped through, both to provide an example of the type checking algorithm described above, and to motivate some methods that are attached to the *root_routine* (and thus available to all routines).

The *routine_name* value of *DrawIcon* satisfies the type of *atom*, since Prolog considers any expression an atom if it consists of lowercase alphanumeric characters or appears in single quotes. The parameter value is the list [*hdevice_context*, *integer*, *integer*, *hicon*]. The type is *list_of(declareable_object)*, meaning that the *value* facet must take a list where each element of the list is a frame subsumed by *declareable_object* in the frame hierarchy. In this case, each of the four parameters is indeed a declareable object (an object which can be directly declared, modified, and inspected by the programmer). The value of the *default_source* is [*no_source*, *user_source(x_coord)*, *user_source(y_coord)*, *no_source*] and the type is *list_of(source)*. The *source* frame is the parent frame of predicates which express the relationship of some object serving as a default value for a routine parameter. The *no_source/0* predicate is a child of *source* and thus satisfies the type. The *user_source/1* predicate takes an argument which must be a *user_source_object* (this is expressed in the *user_source/1* frame). *X_coord* and *y_coord* are found to be the names of frames which are subsumed by the *user_source_object* frame. The arguments to *user_source/1* thus satisfy their type requirements, while *user_source/1* is found to be a child of source (and is thus subsumed by source), satisfying the *source* type requirement. Finally, the value of *default_source* is indeed a list, each element of which satisfies source. The *return_normal* and *return_error* slots are checked by simple frame subsumption. The remaining slots all contain lists of predicates and are checked in the same manner as the *default_source* slot. A point of interest is that the checking of the *main_effect* will descend through one more level of recursion due to the presence of the *icon_func/3* functor as the second argument to the *draw/2* predicate.

The effect slots do present a problem for type checking, however. The arguments present in these predicates are not the names of frames, but rather are arbitrary atoms which are bound to frame names by *assoc_type/2*. Two methods are attached to the root routine to bind the associated types to the atoms which appear in the effect predicates. The *check_type* method takes the name of a slot as an argument, though the only two slots for which it could possibly succeed are *main_effect* and *micro_effect*. This method gets the value of an effect slot, constructs a new effect list with the appropriate substitution of a type (frame name) for each predicate argument as described by the *assoc_type/2* predicates in the *constraint* slot, and then performs the type check on this newly constructed list. The *bind_main_effect* method constructs a new main effect with the appropriate types and returns this new main effect. This method is used by the planner. Both *check_type* and *bind_main_effect* will search up through the inheritance hierarchy to find type bindings for effect arguments if a binding cannot be found on the local frame. The situations in which this occurs can be illustrated with two examples. If the value of one effect slot is inherited and the other is not, then the local constraints slot will contain the *assoc_type/2* bindings for the non-inherited effect, while the bindings for the inherited effect will be found on the same frame as this effect. If the bindings of some inherited effect operators within an effect slot are changed while the other effect operator bindings stay the same, the bindings of the unchanged operators will be found on the frame on which they reside. In both of these cases, the basic idea is that when inherited type bindings need to be overridden, just those bindings which have changed are placed in the *constraint* slot. The unchanged bindings will be inherited even though the local *constraint* slot has a value.

Most type checks are performed by the frame system whenever a slot value is set. The type check on the main and micro effects, however, is not performed during a set for two reasons. First, the semantics for these slots are peculiar to this particular application. The frame system is designed to support generic operations useful for any knowledge base. Supporting explicitly defined predicates and functors is a general enough operation to support in the frame language. The binding of types to atoms in the routine frames, however, is particular to this application. Of course, the frame system could always attempt to call a *check_type* method on any slot for which all other checks failed, thus providing a hook for an application developer to implement application dependent semantics. In general, this is a good idea. The second problem here, however, is that the correctness of the value of an effect slot depends on the value of the *constraint* slot. Making changes to either one of these slots could potentially require making coordinated changes to both slots. If the constraints were changed first, it could make the effects become illegal. If the effects are changed first, this could introduce new atoms that are not yet mentioned in the *constraint* slot. For this reason, the type check is not performed during a set for the effect slots. This is not a problem for this particular application because the values of the slots on the routine frames are not changed while the system is running. These frames statically describe the routines available in a library. The consistency checker, which performs syntax checks and semantic consistency checks on the knowledge base, does call the *check_type* method on the effect slots.

CHAPTER IV

NATURAL LANGUAGE INTERFACE

The natural language interface accepts an English query and translates it into an operator describing a desired main effect. There are two main stages in this processing: syntactic processing in which an augmented transition network (ATN) is used to locate the main verb and the object of the verb, and semantic processing in which an attempt is made to build a valid operator from the verb and verb object.

SYNTAX

The syntactic subsystem accepts a sentence input by the user and produces a register structure representing the sentence. The primary components of the syntactic processor are an ATN grammar, the string preprocessor, an affix stripper, and an interpreter. Each of these components is described below.

The ATN

Transition Networks. A transition network consists of a set of states and a set of directed arcs connecting the states. The arcs are labeled by input symbols and input symbol categories. Some set of states is distinguished as start states and some set of states is distinguished as terminal states. As input symbols arrive, transitions can take place from one state to another along an outward arc whose label matches the input symbol. Such networks have the same generative power as regular grammars. Recursive transition networks (RTN) augment the basic transition network by allowing arcs to represent entire networks. Traversing such an arc in a network is legal only if the transition network associated with the arc can be traversed from a start state to a terminal state. RTNs have the same generative power as context free grammars. An augmented transition network (ATN) further augments RTNs by associating a register structure with the network. A register consists of features dimensions and roles, where feature dimensions can take a value from some primitive set of dimensions and roles can be filled by some other register. A word or set of words from the input sentence is associated with each register. The arcs of an ATN are labeled by conditions and actions in addition to symbols and symbol categories, where conditions can test the current state of a register and actions can modify a register. Basically, an ATN provides an RTN with a modifiable memory and rules for modifying the memory. ATNs have the generative power of context sensitive grammars.

The ATN for ICOP is expressed in Prolog as a definite clause grammar. Definite clause grammars are similar to context free grammars except that arbitrary Prolog expressions can appear on the right hand side in addition to grammar symbols. This provides the ability to perform tests and actions, the two necessary features for an ATN.

The Grammar. ICOP answers questions about how to achieve an effect using a library. This type of question takes the form of asking how to perform an action on an object ("How do I <verb> an <object>?"). In the future, ICOP may handle information requests as well ("Tell me about <object>?"). Though none of ICOP's other subsystems currently knows how to handle such a request, the syntactic processing stage can correctly parse this kind of sentence. The four types of sentences currently understood by the grammar are shown in Figure 6.

```
How <swallow> <verb> <noun phrase>?
        Eg. "How do I create a window?"
How is/are <noun phrase>(s) <verbed>?
        Eg. "How are windows created?"
Tell <swallow> about <noun phrase>s?
        Eg. "Tell me about windows?"
Tell <swallow> about verbing <noun phrase>(s)?
        Eg. "Tell me about creating a window?"
```

Figure 6. Four sentence types understood by ATN.

The *<swallow>* grammar symbol is special. It matches any number of any kind of word. Words are swallowed until a sentence that the grammar recognizes is found or the entire sentence is swallowed (and recognition fails). This allows the grammar to recognize sentences which have been embellished with words which do not add to the meaning of the sentence. For example, "How do I draw an icon?" and "How in the world is it possible to draw an icon?" are both reduced to "How draw an icon?" which is recognized by the grammar. This does have the side effect of allowing ungrammatical sentences such as "How draw an icon?" to be recognized. However, ICOP's grammar is not motivated by the desire to reject ungrammatical sentences but rather by the desire to assign meaning to as wide a range of sentence as possible within the domain.

ICOP's natural language interface currently handles an extremely limited subset of English, even though the ATN formalism is capable of representing a much richer subset. Using a formalism more powerful than strictly needed by the current interface provides a rich foundation on which ICOP's interface can grow.

The Register Structure. Given the simple form of the sentences handled, the registers have a simple structure. The two registers filled by the ATN are shown in Figure 7.

Sentence Register

Object -	filled by a noun phrase register
Action -	filled by the verb in infinitive form
Type -	filled by "plan" or "info"

Noun Phrase Register

Head -	filled by noun in singular form
Describers -	filled by list of adjectives and nouns in singular form

Figure 7. Registers filled by ATN.

No attributes other than the sentence type are included. Any attributes needed during parsing (verb form, number) are passed as arguments in the definite clause grammar. The slot structure is influenced by the semantic form of the queries. All queries are either requests for a plan or requests for information (this is stored in the sentence type slot). Plan requests are always requests for some action on an object. Information requests are always requests for information about some particular object. If an object can be described in one word (eg. menu), it will be in the head slot of the noun phrase. If the object takes several words to describe (eg. handle to an application instance), the base word (handle) will appear in the head and the other words (application, instance) will appear in the list of describers. Currently, only one noun phrase is ever created during the processing of a query. This noun phrase is always the object of the sentence. The code has been made general enough (register names are passed as arguments) so that if, in the future, multiple registers of the same type are needed, it will not require a major code revision. The registers are implemented using frames managed by the frame system.

The String Preprocessor

The string preprocessor takes the string typed by the user, which is represented as a list of characters in Prolog, and converts it into a list of atoms where each word becomes an atom. During this process, all characters are converted to lower case and extraneous punctuation is removed. The list of atoms is then processed by the ATN.

Affix Stripper

The action verb is stored in the register in its infinitive form and nouns are stored in their singular form. This simplifies later processing of the register structure by the semantics subsystem. The affix stripper converts plural nouns to singular nouns and past participle and present participle verb tenses to the infinitive tense. Regular plural nouns and verbs with regular present and past participles are constructed automatically by this component. Only irregular forms need to be explicitly represented in the lexicon. Since so many verbs take either -en (eg. given) or -ed (eg. dropped) for the past participle form, the system tries both forms while looking for the infinitive in the lexicon. This means that incorrect forms like "droppen" will be accepted by the parser. However, as mentioned above, the motivation of the parser is to try an extract meaning from sentences rather than to reject ungrammatical sentences.

The Interpreter

To facilitate debugging of the parser, a special Prolog interpreter that produces ATN specific trace messages was written. The grammar actually runs in this interpreter rather than directly in Prolog. With a debug switch on, the interpreter produces trace messages when processing particular predicates. Since the meta-language understood by the interpreter is precisely Prolog (the object language), the interpreter can be removed and the grammar run directly in Prolog when debugging is complete.

SEMANTICS

The semantics analyzer accepts the sentence register produced by the ATN and attempts to construct an operator that represents the desired effect expressed in the query. There are three phases in this process: matching the verb of the sentence to an operator, matching the head and describers of a sentence to objects in the knowledge base, and testing the arguments of the operator for semantic validity. In this last stage, any arguments needed by the operator which were not explicitly given by the user are filled in.

Determining the Operator

The main verb is a valid operator if there is an operator frame with the same name as the verb. For example, the query "How do I draw an icon?" contains the verb "draw". There is an operator frame with the name *draw*, therefore *draw/2* will be selected as the operator. If there is no frame whose name is the same as the verb, an attempt is made to map the verb to operator. Mappings from verbs to operators are represented by the predicate *map_action(<verb>, <operator>)*. A *map_action/2* fact expresses that some verb should be considered a synonym for some operator. For example, the query "How do I load an icon?" contains the verb "load." There is no operator frame with the name "load," so that condition for converting a verb to an operator fails. Next, an attempt is made to map "load" to an operator. It so happens that there is a *map_action/2* fact *map_action(load, add_object)* which expresses the fact the "load" should be considered a synonym for the operator *add_object/2*. If the verb "load" can have more than one meaning in the context of using a code library, these multiple meanings can be represented by multiple *map_action/2* facts containing "load" as the first argument. In the event that another attempt is made to map the verb to an operator (perhaps due to the objects of the verb not meeting the operator argument constraints), the next operator mapping will be tried.

Determining the Objects

The next stage in semantics processing is to convert the head and describers obtained from the ATN into an object or list of objects. Before attempting to map the head and describers to objects, the head is appended to the end of the describers. In the query "How do I get a handle to an application instance?", the head is "handle" and the describers are [application, instance]. The new list formed for object mapping is [application,instance,handle]. Of course, if the describers are empty (as in the query "How do I draw an icon?"), then the new list still contains one word.

The predicate *find_object/2* performs the mapping from a list of one or more words to a list of one or more objects. There are four ways for *find_object/2* to succeed. If there is only one word in the list (meaning there was only a head with no describers), then there is an object corresponding to this word if there is an object frame whose name is this word. If the frame has children, meaning that the frame actually describes a set of objects, then the actual object returned will be a leaf in the subtree whose root is this frame. This accommodates user queries which are at a more general level than the primitive objects represented in the system. For example, a user may ask the query "How do I load a resource?" Now it so happens that there is a frame named *resource* whose children include all of the resource objects known by ICOP. However, if the object mapping stage returned the object *resource*, the query that would be constructed is *add_object(user_memory, resource).* This query would fail because there is no particular routine described in the knowledge base that loads resources in general, though there are routines which load particular resources. However, instead of returning resource, the object mapper will find the *resource* frame in the knowledge base and begin searching the *resource* subtree for leaf frames. If the first leaf frame it found was *system_icon*, then the object mapper would return *system_icon*. Now there is a routine for loading a system icon; an example will successfully be constructed. If the user asks the system to generate another example, *find_object/2* will be tried again (after the example generator has tried to construct a different example for the plan to load a system icon, and the planner has tried to construct an alternate plan to load a system icon). This time, *find_object/2* might return the object *menu*. Now an example for loading a menu will be created. This mapping of non-primitive (set) objects to primitive objects is one way in which ICOP attempts to provide specific examples in response to general information needs.

The second way in which *find_object/2* succeeds is if there is only one word in the list and a *map_object/2* fact can be found for the word. The *map_object/2* facts are analogous to the *map_action/2* facts. They allow natural language words to serve as synonyms for some object. An attempt is still made to find a primitive object if the object indicated by *map_object/2* is non-primitive. This allows words to serve as synonyms for abstract (non-primitive) objects.

The third way in which *find_object/2* succeeds is by concatenating a multiple word list (the describers are not empty) into a single atom and recursively checking whether this single atom satisfies one of the first two cases. The single atom created through concatenation consists of each word in the list separated by an underbar ("_"). Thus the list [application, instance, handle] becomes application_instance_handle.

The final way in which *find_object/2* succeeds is by finding some partition of the list of words such that each piece of the partition recursively satisfies *find_object/2*. This is the only way in which a list of words can be mapped to a list of objects rather than a single object. The examples at the end of this chapter will make each of these four cases for matching words to objects more clear.

Checking Semantic Validity

After the head and describers have been converted into a list of objects which serve as potential arguments for the operator, these arguments are tested for semantic consistency with the operator. This processing deals with the problem that sentences which are syntactically valid may be semantically invalid. For example, the utterance "I want to drink a rock" has proper syntactic form, but is invalid because the object of the verb does not "fit" the verb (rock is not a legal argument of the drink operator). The frame describing each operator has a slot containing the legal arguments for that operator. Each object in the object list is compared against the legal arguments. If there is some argument which subsumes the object, the object is valid. This step also orders the objects. For example, if there are two objects in the object list, and the first object is subsumed by the second argument and the second object is subsumed by the first argument, then order of the objects will be switched. If no argument subsumes an object, a check is made to see if some functor can map the object to an object which is subsumed by an argument. If a functor is found with an argument that subsumes the object and whose return value is subsumed by an operator argument, the return value of the functor is placed in the ordered list of objects. After the object list has been ordered, any operator arguments which were not given explicitly in the query are filled with the appropriate value from the list of legal arguments. The next section gives some examples of query processing.

Examples

Example 1.Consider the query "How do I draw an icon?" The ATN grammar recognizes this sentence and produces the register shown in Figure 8.

```
Sentence Register
        Object:         Noun Phrase 1
        Action:         draw
        Type: plan

Noun Phrase 1
        Head:           icon
        Describers:
```

Figure 8. Registers produced during parse of the query "How do I draw an Icon?".

The first step in semantic processing is mapping the verb to an operator. An operator frame named *draw* does indeed exist, so the operator corresponding to the action "draw" is *draw/2*. The next step is mapping the noun phrase to a list of objects. The describers are empty, so the word list to be mapped is [icon]. A non-primitive frame named *icon* is found. The first primitive child located beneath *icon* is *defined_icon* (meaning an icon defined by the user, as opposed to a system icon), so "icon" maps to *defined_icon*. Now the test is performed to check if *defined_icon* is subsumed by an argument of *draw/2*. *Draw/2*'s legal arguments are [*device_context, gdi_drawable_object*]. Neither of these arguments subsumes *defined_icon*, so this test fails. However, there is a functor *icon_func/3* taking an icon as one of its arguments (subsuming *defined_icon*) which maps to the object *drawable_icon*. *Drawable_icon* is subsumed by the second argument of *draw/2* (*gdi_drawable_object*), so *drawable_icon* fills the second argument. Finally, the first argument, which was not filled by any object mentioned in the query, is filled by the first object in *draw/2*'s valid argument list (*device_context*). The final operator produced after semantic processing is *draw(device_context, drawable_icon)*.

Example 2. As a second example, consider the query "How is a bitmap added to user memory?" The register structure produced by the ATN is shown in Figure 9. There is no operator frame named "add," so the first case for mapping a verb to an object fails. However, there is a *map_action/2* rule mapping "add" to *add_object/2*. Now the head is appended to the end of the describers producing the list [user,memory,bitmap]. The first *find_object/2* case which handles lists containing multiple words attempts to mapuser_memory_bitmap to an object. This attempt fails. The final *find_object/2* clause partitions the list as [user], [memory,bitmap] and tries to

Sentence Register

Object:	Noun Phrase 1
Action:	add
Type: plan	

Noun Phrase 1

Head:	bitmap
Describers:	[user,memory]

Figure 9. Registers produced during parse of the query "How is a bitmap added to user memory?".

map each of these to a list of objects. The attempt to find an object corresponding to [user] fails, so the attempt to call *find_object/2* on [memory,object] is never even tried. The next partition tried is [user,memory], [bitmap]. The first list is mapped to the object *user_memory* by the second *find_object/2* clause. The second list maps to *bitmap,* because there is a primitive frame named *bitmap*. So the object list produced from the word list [user,memory,bitmap] is [*user_memory, bitmap*]. Finally, these objects are checked for semantic validity with *add_object/2*. The valid arguments of *add_object/2* are [*container, memory_object*]. *Container* subsumes *user_memory* and *memory_object* subsumes *bitmap,* so the test succeeds. The final operator representation of the query is *add_object(user_memory, bitmap)*.

Syntax vs. Semantics. The final clause of *find_object/2,* which attempts to find a partition of the list which can be mapped to objects, is necessary because the ATN represents multiple objects as single objects. In the example above, the noun phrase in "... add a bitmap to user memory" was described in the register structure as the single entity "bitmap" modified by the describers "user memory." Actually, these are two separate objects, where the preposition "to" relates an object to a location. The semantics processing had to break this back into two separate objects. However, the noun phrase in "... get a handle to an application instance" is correctly represented as a single object "handle" modified by "application instance"; this will not need to be broken into separate objects during semantic processing. This inconsistency is caused by the fact that the ATN does not currently distinguish between transitive and bitransitive verbs. "Get" is transitive and thus takes only one object. "Add" is bitransitive, therefore the object of the preposition becomes the second object of the verb rather than describers of the first object. Since the ATN does not make these distinctions, they have to be unraveled in the semantics.

CHAPTER V

THE PLANNER

The planner accepts an effect and produces a list of routines which have the desired effect. There are three major stages in the planning process: find a routine which produces the desired effect (the plan focus in Rist's terminology), satisfy any preconditions of this routine by recursively finding a focus routine for each precondition and satisfying the preconditions of this new focus routine, and satisfy any postconditions for all routines present after the second step. To aid in understanding the planning process, the general description of each processing step will be accompanied with an example of plan generation. Before understanding the processing, however, it is important to understand the plan representation.

PLAN REPRESENTATION

Routine Representation

Each routine in a plan is represented with the *routine/3* functor. This functor has the form *routine(<routine_frame_name>, return(Unbound), <list_of_sources>)*. The *<routine_frame_name>* is the name of the frame which describes the library routine. *Return(Unbound)* is a functor representing the return value of the routine. The unbound Prolog variable in the argument of *return/1* will be bound to a name during the example generation stage. The list of sources is a list of source predicates describing where each parameter of the routine should come from. The *routine/3* functor in Figure 10 describing the Windows *BeginPaint* procedure illustrates these ideas.

```
[routine(begin_paint, return(_B),
        [code_source(window_procedure,hwnd),
        user_source(user_declareable_object,_A)])
```

Figure 10. Routine functor for *BeginPaint*.

Begin_paint is the name of the frame describing the *BeginPaint* routine. The _B in the *return/1* functor is the unbound variable which will be bound during the example generation stage to the name of the variable which accepts *BeginPaint*'s return value. The list of two sources describes how *BeginPaint*'s two arguments will be satisfied. The *code_source(window_procedure, hwnd)* predicate expresses that the first argument is satisfied by the *hwnd* argument of the *window_procedure* code object. The *user_source(user_declareable_object, _A)* predicate indicates that the second argument is satisfied by a variable declared by the user. The unbound variable _A will be bound to the actual name of this user declared variable during the example generation stage.

Plan Representation

A plan consists of a list of *routine/3* functors. A plan to draw an icon is shown in Figure 11. This plan consists of the four Windows routines *BeginPaint, LoadIcon, DrawIcon* and *EndPaint.* Looking at an entire plan, the purpose of the unbound variables in the *return/1* functor and the source predicates becomes clear. They are used to express the dataflow dependencies of the plan. The unbound variables for values which should be shared between routines are unified. For example, the first argument to *DrawIcon*

```
[routine(begin_paint, return(_B),
        [code_source(window_procedure,hwnd),
        user_source(user_declareable_object,_A)]),
routine(load_icon_resource, return(_D),
        [code_source(main_entry,hInstance),
        user_source(memory_object_name,_C)]),
routine(draw_icon, return(_G),
        [routine_source(begin_paint,0,_B),
        user_source(x_coord,_E), user_source(y_coord,_F),
        routine_source(load_icon_resource,0,_D)]),
routine(end_paint, return(_H),
        [routine_source(begin_paint,1,hwnd),
        routine_source(begin_paint,2,_A)])]
```

Figure 11. Representation of plan to draw an icon.

should come from the return value of *BeginPaint*. This is expressed by the first source predicate in *draw_icon*'s source list and *begin_paint*'s *return/1* functor. The first source predicate for *DrawIcon* is *routine_source(begin_paint, 0, _B)*. This expresses that the first argument for *DrawIcon* should come from the zeroth argument of *BeginPaint*; the zeroth argument means the return value. The variable in the *return/1* functor for *BeginPaint* has been unified with the variable in the *routine_source/3* predicate, as is indicated by the shared name "_B." When the example generator first creates a name for a return value or an argument, the pattern of unification in the plan ensures that this name will be shared properly among the elements of the plan. With this understanding of the plan representation, it is now time to look at the stages of plan production.

SEARCHING FOR THE FOCUS

Discrimination Tree

The first stage in the planning process is finding the routine which produces the desired effect. This is done by performing a depth first search of the routine frame hierarchy. When a frame is found which has no children and whose *main_effect* subsumes the query effect, the routine represented by this frame satisfies the query. The *main_effect* slot of the routine frames is used by the planner to prune branches off the search tree. Subtrees are only searched if the main effect at the root of the subtree subsumes the query effect. The routine hierarchy can be seen as a discrimination tree, in which nodes on the same level describe effects of the same generality, and children describe less general effects.

Search Example

For example, the query that would have produced the plan shown above to draw an icon is *draw(device_context, drawable_icon).* Assume that the immediate children of the root routine are *draw_object*, *load_resource* and *get_device_context*. This first level of routine frames with their effect are shown in the Figure 12.

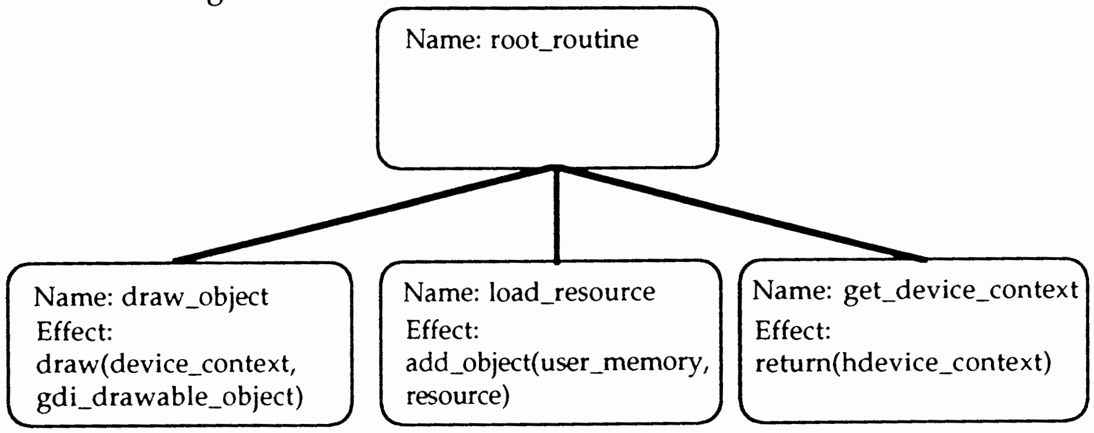

Figure 12. First level of discrimination tree.

The search to satisfy the query *draw(device_context, drawable_icon)* begins with the child list of the root routine. Since the search is performed left to right, the first frame examined is the *draw_object* frame. In this case, the main effect of *draw_object*, *draw(device_context, gdi_drawable_object)*, subsumes the query, since the predicates subsume each other (*draw/2* subsumes *draw/2*), the first arguments subsume each other (*device_context* subsumes *device_context*) and the second arguments subsume each other (*gdi_drawable_object* subsumes *drawable_icon*). The search would then proceed down into this subtree. Suppose, however, that *draw_object* was the rightmost frame. Then *load_resource* and *get_device_context* would be checked first. In each case, subsumption would fail, because *add_object/2* and *return/1* do not subsume *draw/2* (this is checked by looking at the relative positions of these predicates in the predicate hierarchy). The subtrees below *load_resource* and *get_device_context* have been pruned from the search. Once the search descends into the *draw_object* hierarchy, the subsumption check process is repeated until a routine frame is found which has no children and whose effect subsumes the query. Once a routine is found, a *routine/3* functor is built by placing the name of the located frame in the first argument, a *return/1* functor with a new unbound variable in the second argument, and an empty list (which will eventually become the list of sources) in the third argument. For this query, the *routine/3* functor produced is *routine(draw_icon, return(_G), [])*.

SATISFYING THE PRECONDITIONS

Satisfying Arguments, Not State

Once the focus of the plan has been found, an attempt is made to satisfy the preconditions of the focus. In order to use a routine, one must have the proper arguments to pass to the routine. Satisfying the preconditions of the focus means satisfying the arguments of the focus routine. Note that this has nothing to do with the preconditions slot on the routine frames. The preconditions slot describes states that must be true in the world in order to use a routine. Some of these states may not be under programmer control, such as whether a window has received a paint message. Other states are under programmer control, such as the coordinate mapping mode of the device context. Unlike many planners, ICOP's planner does not know the state of the world prior to plan execution; ICOP assumes that state preconditions have been met. The preconditions list was included on frames to be used, along with the micro effect, by a natural language generator in producing comments, tutorials, answers to questions, etc. The preconditions could also be used to produce alternate examples. One can envision a user making a query, getting a commented example in reply, then requesting a plan showing what to do if the preconditions are not met. It would not be complicated to add a new predicate to the planner which processes the preconditions list as well as the parameter list while planning. The current implementation of ICOP, however, does not do this. As will be seen in the description of the example generator, some preconditions do effect the example being produced.

Invoking Effect Rules

To satisfy the routine arguments, the default source list is examined. Any source which is not *no_source* is added to the source list in the *routine/3* functor with no change. When *no_source* is encountered, this means that no default is specified for the corresponding argument. The planner must find a way to satisfy this argument. The type of the unsatisfied argument is found in the parameter list. Then, effect rules are used to determine the effect of a routine which could satisfy these arguments. What is needed is a routine which returns the required type either directly as a return value or indirectly through an argument pointer. Since the main effect indexes the routines, the only way to search for a routine without a main effect would be to examine the parameters and return value of every routine with no children. In the event that a parameter was found which is a pointer to the correct type, the micro effect would need to be examined to determine whether this routine was actually altering the value of the object pointed at by the pointer. Altogether, this would be a highly inefficient way to proceed. One alternative would be to assume that the main effect of a routine which produces a value of type *<type>* is *return(<type>)*. In building the knowledge base, however, this alternative constrains the selection of the main effect. Other processing components of ICOP may find it more convenient to have a different effect hilighted as the main effect. For example, the *LoadIcon* routine loads an icon into memory and returns a handle to this icon. Should it's main effect be *return(hicon)* or *add_object(user_memory, icon)*? The effect rule allows maximum flexibility in choosing the main effect by providing a mapping between a parameter need and the main effect of the routine which satisfies such a parameter. The form of an effect rule is *effect(<type>, <effect>) if*

<condition> where the *<type>* is some argument type, the *<effect>* is some effect, and the *<condition>* is a conjunction of Prolog goals. The conjunction of Prolog goals allows an arbitrary condition to be tested before choosing an effect as the proper effect to look for to satisfy an argument of type *<type>*. In the example plan, the current plan focus is *draw_icon*. In examining the default source list, two *no_source* predicates are found, one for the first argument and one for the fourth argument. The type of the first argument is *hdevice_context* (handle to a device context). The type of the fourth argument is *hicon* (handle to an icon). The effect rules relevant in these two cases are shown in Figure 13.

```
effect(hdevice_context, return(hdevice_context)).
effect(Resource_handle, add_object(user_memory, Object)) :-
        subsumes(Resource_handle, handle),
        ask(Resource_handle, get-referenced_object:Object),
        subsumes(Object, resource).
```

Figure 13. Effect rules relevant during planning for drawing an icon.

The first effect rule is applicable to the *hdevice_context* argument. This rule states that if a handle to a device context is needed, the appropriate effect to look for is indeed the return of a handle to a device context. The second effect rule is applicable to the *hicon* argument. This rule states that if the required argument is a handle, and this handle references a resource, then the appropriate effect to look for is adding this resource to user memory. In the case of *hicon*, *hicon* is a type of handle, the *referenced_object* slot on the *hicon* frame is filled by *icon*, and *icon* is a type of *resource*. Therefore, the appropriate effect to look for is *add_object(user_memory, icon)*.

Recursively Satisfying Preconditions

Once an effect has been found by firing the effect rules, a focal routine for this effect is found and the preconditions of this new routine are satisfied. Routines are added to the plan to satisfy the preconditions of this new focus. As each routine is added to the plan, it becomes the focus; it's preconditions are recursively satisfied. This continues until no unsatisfied preconditions remain. All postconditions are deferred until the precondition processing phase has been completed. Note that this does not involve recursively calling the entire planner. If the entire planner were called, the focus would be found, preconditions satisfied, and postconditions satisfied for each routine as it was processed. This can create goal interaction problems in which satisfying the postcondition undoes the effect of a routine before the user of some data object produced by the routine has a chance to use it. An example of this problem will be described below in the section on postcondition processing.

Determining the Dataflow

As each argument is satisfied, the last routine in the satisfying subplan (this will be the routine which immediately satisfies the argument) is examined. A new *routine_source/3* functor is created with the name of the immediate satisfier in the first argument. The satisfier is then examined to determine which of its arguments produces the desired object. If its return type is the same type as the argument, it is assumed that the return value is the needed value; the second argument of *routine_source/3* is set to 0 and the third is an uninstantiated variable which is unified with the return variable of the satisfier. If the return value is not the same type as the argument, each of the satisfier's arguments is examined in turn. The first one which matches the desired type is assumed to be the source. The last argument of the matching source predicate is unified with the third argument of the *routine_source/3* predicate. The second argument of *routine_source/3* is set to the number of the satisfying argument. In this way, the dataflow through the plan is expressed.

Precondition Processing Example

In the example plan, the current focus is *DrawIcon*. The first *no_source* source encountered is for the *hdevice_context* argument (argument one). The effect rules are fired and the corresponding effect found is *return(hdevice_context)*. The search through the routine tree retrieves *BeginPaint*. Its source list is searched and no *no_source* sources are found. Since *BeginPaint* is the last (and only) routine in the subplan which satisfies *hdevice_context*, it is examined to determine how it produces an *hdevice_context*. Its return value has this type; the return value is unified with the third argument of *routine_source/3* and the second argument is set to 0 (return value is source). The second *no_source* source found in *DrawIcon* is for the fourth argument *hicon*. The effect rules are fired and the corresponding effect found is *add_object(user_memory, icon)*. The routine tree is searched and the routine *LoadIcon* is found. It's source list contains no *no_source* sources. Again the satisfying subplan consists of one routine. It is found that *LoadIcon* returns an *hicon*; the *routine_source/3* for the fourth argument of *DrawIcon* is set accordingly. There are no more unsatisfied preconditions. The plan at this stage is show in Figure 14.

```
[routine(begin_paint, return(_B),
        [code_source(window_procedure,hwnd),
        user_source(user_declareable_object,_A)]),
routine(load_icon_resource, return(_D),
        [code_source(main_entry,hInstance),
        user_source(memory_object_name,_C)]),
routine(draw_icon, return(_G),
        [routine_source(begin_paint,0,_B),
        user_source(x_coord,_E), user_source(y_coord,_F),
        routine_source(load_icon_resource,0,_D)])]
```

Figure 14. Plan to draw an icon after precondition processing.

POSTCONDITION PROCESSING

Concern Rules

The last stage of plan processing satisfies any postconditions. Postconditions "clean up" the plan. Determining the postconditions is done by firing concern rules. Concerns have the form *concern(<routine_frame_name>, <effect>) if <condition>* where the *<condition>* is a conjunction of Prolog goals. A concern is a way of saying that a certain effect should always occur at the end of a plan in which a certain routine occurs. A routine satisfying the effect is found and it is placed at the end of the plan. No attempt is made to satisfy the preconditions of a routine added because of a concern. It is assumed that such a routine will have default sources for all of it's arguments. After a routine is found, an attempt is made to bind any *routine_source/3* sources. Cleanup routines may refer to specific routines and arguments which they use to satisfy their own arguments. The plan prior to the concern is searched for any such explicitly mentioned routines, and the appropriate variables are unified. Every routine in the plan is given the chance to fire a concern.

Postcondition Processing Example

In the example, the only relevant concern is *concern(begin_paint, end_paint)*. Note that the concern rule is returning a routine frame name rather than an effect. If the search system of the planner is given a routine name rather than an effect, it immediately returns with the routine as the plan. In other words, the plan for a routine is the routine itself. This concern expresses that if a plan contains *begin_paint*, then *end_paint* should be placed on the end of the plan. *End_paint*'s default source list contains two *routine_source/3* sources: *routine_source(begin_paint, 1, _A)*, and *routine_source(begin_paint, 2, _B)*. *Begin_paint* is found in the plan, and _A and _B are unified with the appropriate variables in *Begin_paint*'s source list. No other concerns match for the plan. The final plan is shown in Figure 11 on page 72.

Avoiding Plan Interactions

In the section describing precondition processing, it was mentioned that postcondition processing should be deferred until all preconditions have been satisfied, rather than recursively calling the entire planner and thus satisfying postconditions as each new routine is added. The *draw_icon* example shows what happens if postconditions are not deferred. If *begin_paint*'s postconditions were processed immediately after *begin_paint* was added to the plan, *end_paint* would be added between *begin_paint* and *load_icon_resource*. This would invalidate the device context handle before *draw_icon* got a chance to use it. Deferring postconditions until the end of the plan are an attempt to avoid this type of negative interaction.

CHAPTER VI

EXAMPLE GENERATOR

The example generator accepts a plan produced by the planning component and produces example code illustrating the plan. For many reusable code libraries, the example code could be as simple as a linear ordering of the plan routines preceded by the appropriate variable declarations. However, in a message based windowing system such as Windows (the Macintosh toolbox and X windows have this same architecture), various pieces of user code are called asynchronously by the operating system in response to system activity such as the mouse being clicked or a window being opened. This means that the routines in the plan may be scattered nonlocally in the example. In order to accomplish this, the example generator uses a grammar of Windows examples to build up a syntax tree for the example. As each routine in the plan is encountered, the syntax tree is modified according to rules which take into account the current routine being added to the example and the current structure of the example. There are three steps involved in adding each routine to the example: placing the routine name in the appropriate location in the example, adding the routine parameters (and making the appropriate variable declarations), and adding the return value (and its variable declaration). Once the syntax tree for the example has been built, the actual text of the example is written by walking along the example tree.

UNIFICATION GRAMMAR

Functional Description

The grammar for Windows examples is expressed as a unification grammar [Mellish, 1990]. In unification grammar, a phrase is expressed by a functional description. A functional description states the attributes and values of a phrase. For example, the functional description for the phrase "it hit" might be expressed as shown in Figure 15 [Mellish, 1990].

```
[s,
subj=[person_number=(3+sing), text=[root=it]],
pred=[first=[
        mainverb=[root=hit],
        compls=[
                first=[np] ] ] ] ]
```

Figure 15. Example functional description for phrase "it hit".

The functional description in Figure 15 says that the phrase "it hit" is a sentence with the third person singular subject "it" and a predicate consisting of the main verb "hit" and a noun phrase.

Any expression of the form X=Y in the functional description indicates that attribute X takes the value Y, where Y is either itself a functional description or an atom. Single atoms (such as "s") indicate additional properties of the phrase. Whether a functional description is legal or not depends on the grammar. Given a partial functional description for a phrase, such as the one above, it is possible in general to match this description against the grammar in order to test grammaticallity and fill in additional attributes which can be computed from those explicitly given. It is also possible to match two functional descriptions to test whether they consistently describe different attributes of the the same phrase. In the process of matching these descriptions, attributes missing in one description but present in the other will be filled. It is this property of successively matching consistent functional descriptions to build an ever more complex phrase which is used in the example generator to build up a functional description of the entire example.

Grammar Specification

A specification of a unification grammar generally consists of three parts: descriptions of the categories of phrases and the attributes of these phrases, sharing rules between attributes which constrain attributes of some phrases to match attributes of other phrases, and finally computed properties which serve as abbreviations for combinations of attributes. The grammar of Windows examples only makes use of the first part of this grammar specification. The example grammar, with English explanations of each grammar specification, appears in Figures 16 and 17. The symbols on the left hand side of the grammar rules are the legal phrase types. The right hand sides indicate the attributes of a phrase type. The "**" symbol should be read as "and"; the list of properties on the right hand side are all of the properties of a given phrase. A property may be a simple atom or may be followed by a ":" and a phrase type. If a property is a simple atom, this property can be filled by anything; this is a primitive property. Properties with phrase types can only be filled with phrases of that type. The "list" phrases in Figure 17 always have two properties: first and rest. First will be filled by some phrase type (or may be primitive); rest must be filled by a list of the same type. A list phrase is composed of an arbitrary number of some other type of phrase.

Rule 1. program <--> forward:forward_list ** global:decl_list **
main:winmain ** proc:winproc

A program consists of some number of forward declarations, some number of global variable declarations, a main procedure and a window procedure.

Rule 2.decl <--> type ** name

A variable declaration consists of a type and a variable name.

Rule 3. winmain <--> var:decl_list ** create:create_window **
routines:routine_list **message

A main procedure consists of some number of variable declarations, a piece of code which creates a window, some number of routines, and a message loop.

Rule 4. create_window <--> register ** create ** show

The creation of a window consists of window registration, window creation and setting window visibility.

Rule 5. routine <--> return ** name ** parameters:param_list

A routine consists of a return variable, a routine name, and a parameter list.

Rule 6. winproc <--> var:decl_list ** case:case_list

A window procedure consists of some number of variable declarations and some number of message cases.

Rule 7. case <--> name ** routines:routine_list

A message case consists of a message name and some number of routines.

Figure 16. Unification grammar rules for Windows examples (excluding "list" rules).

```
forward_list <--> first ** rest:forward_list
decl_list <--> first:decl ** rest:decl_list
case_list <--> first:case ** rest:case_list
routine_list <--> first:routine** rest:routine_list
param_list <--> first ** rest:param_list
```

Figure 17. Unification grammar "list" rules.

Prolog Implementation

In Prolog, each of the phrase types is represented by a functor with the same number of arguments as the phrase has attributes. For example, a program phrase with no attributes specified is represented by *program(_A, _B, _C, _D)*. A program with nothing else specified except that it has some window procedure would look like *program(_A, _B, _C, winproc(_D, _E))*. The primary operation performed on functional descriptions is matching. The predicate to perform matching (called *matches/2*) was implement by Chris Mellish [1990] and is used with minor modifications in this thesis. Matching is used to build up complex phrases. For example, the call to *matches/2* in Figure 18 produces a phrase in which the window procedure has a variable declaration of type *HDC* and a case for the *WM_PAINT* message.

```
| ?- X matches [winproc=[var=+[type='HDC'],
case=+[name='WM_PAINT']]].

X = program(_G,_F,_E,winproc(decl_list(decl('HDC',_D),_C),
case_list(case('WM_PAINT',_B),_A))) ?

yes
| ?-
```

Figure 18. Building *WM_PAINT* case with *matches/2*.

The variable on the left hand side of *matches/2* is unified with the minimally instantiated phrase structure which satisfies the functional description on the right hand side. If one then wanted to give the name *hdc* to the variable of type *HDC* and add the routine *LoadIcon* to the *WM_PAINT* case, this X can be unified with an additional functional description. This is shown in Figure 19.

```
| ?- X matches [winproc=[var=+[type='HDC'],
case=+[name='WM_PAINT']]],
X matches [winproc=[var=+[type='HDC', name=hdc],
case=+[name='WM_PAINT', routine=+[name='LoadIcon']]]].

X = program(_H,_G,_F,winproc(decl_list(decl('HDC',hdc),_E),
case_list(case('WM_PAINT',
routine_list(routine(_D,'LoadIcon',_C),_B)),_A))) ?

yes
| ?-
```

Figure 19. Adding a routine to the *WM_PAINT* case with *matches/2*.

The two functional descriptions above make use of an additional operator "=+". This operator is used to add structures to a list of structures. The functional description on the right hand side of the "=+" is unified with the first phrase in the list which satisfies the description. If no existing phrase in the list satisfies a description, a new phrase which satisfies the description is added to the end of the list. The example in Figure 20 uses this operator to add a new case to the list of cases in the window procedure.

```
| ?- X matches [winproc=[var=+[type='HDC'],
case=+[name='WM_PAINT']]],
X matches [winproc=[case=+[name='WM_CREATE']]].

X = program(_H,_G,_F,winproc(decl_list(decl('HDC',_E),_D),
case_list(case('WM_PAINT',_C),
case_list(case('WM_CREATE',_B),_A)))) ?

yes
| ?-
```

Figure 20. Adding a new case with *matches/2*.

The functional description can be thought of as specifying a path through a syntax tree. Thus one can use functional descriptions to pinpoint specific places on the syntax tree where a new structure should be added. Of course, if the path specified by the functional description is illegal, the *matches/2* predicate fails. This occurs in Figure 21 in the attempt to declare a global variable in the window procedure.

```
| ?- X matches [winproc=[global=+[name=foo, type='FOO']]].

no
| ?-
```

Figure 21. *Matches/2* failing due to specifying an illegal syntax tree.

In the example generator, the *matches/2* predicate is used to build a syntax tree for the example.

BUILDING THE EXAMPLE

Adding a Routine

The example generator calls *transform_example/2* on each routine in the plan to build the example tree. *Transform_example/2* takes a *routine/3* functor and an example tree. Since the example tree is modified by unifying some uninstantiated variable in the tree with a structure, there is no need to include a third parameter to return the transformed tree. The right hand side of each *transform_example/2* clause consists of some number of tests of the properties of the routine and the structure of the current example tree and some calls to *matches/2* which transform the tree. As new cases involving routines and current tree structures are discovered, they can be handled by adding new clauses to the *transform_example/2* predicate.

Filling the Parameters

Fill_parameters/5 is called by *transform_example/2* to fill in the parameters of a routine. Filling the parameters can require declaring a local variable or even copying a value into a global variable in addition to placing a variable name in the parameter list of a routine. The five arguments to *fill_parameters/5* are the list of parameter types, the list of sources, a routine phrase which has been matched with the routine added by *transform_example/2*, a procedure phrase which has been matched with *winproc* or *winmain*, whichever was modified by *transform_example/2*, and the entire current example tree. Since the example contains the routine and procedure (*winproc* or *winmain*) being modified, it would appear there is no reason to pass the routine and procedure separately. They are passed separately to tell *fill_parameters/5* exactly which routine to fill the parameters of, and which procedure to add any variable declarations to. Without some indication of the procedure begin modified, *fill_parameters/5* would have to look in both *winmain* and *winproc* for the routine to modify. If both procedures happened to contain the routine in their routine list, *fill_parameters/5* would have to do more work to determine which should be modified. With the routine and procedure phrases, *fill_parameters/5* can refer directly to the appropriate structures with no search; since the phrases have been matched with the current example tree, any changes to these phrases will automatically occur in the example tree through the unified uninstantiated variables in the local phrases and example tree. Some clauses of *fill_parameters/5* make changes to other parts of the example besides the procedure in which the routine occurs and the parameter list of the current routine. For this reason, the entire example tree is passed as well. Like *transform_example/2*, new clauses can be added to *fill_parameters/5* to handle new combinations of parameter types and sources. Each clause of *fill_parameters/5* recursively calls itself on the rest of the parameter and source list.

As each parameter is processed, the uninstantiated variable in the corresponding source predicate is unified with the name of the variable added to the example. Because of the pattern of unification in the plan expressing the dataflow, this variable name will propagate to the appropriate consumers of the data object. For every combination of parameter type and source, there are usually two *fill_parameters/5* clauses; one for when the parameter being processed involves adding a new variable to the example, and another when the name of the variable has already been instantiated due to a variable being added earlier during example processing.

Filling the return value is quite simple. Since it is guaranteed that there is no previous source for the return, all that has to be done is making a declaration of the appropriate type, matching the new variable with the return attribute of the routine phrase in the syntax tree, and unifying the new variable with the *return/1* functor in the appropriate *routine/3* functor of the plan (to ensure proper dataflow).

BUILDING AN EXAMPLE FOR THE DRAW ICON PLAN

Now that the general stages of building an example have been seen, the details of this process are examined by stepping through the processing of the plan to draw an icon. The first routine processed is shown in Figure 22.

```
routine(begin_paint, return(_B),
        [code_source(window_procedure,hwnd),
        user_source(user_declareable_object,_A)]).
```

Figure 22. *Routine/3* functor for *BeginPaint*.

One of the *transform_example/2* clauses looks at the preconditions of a routine and checks if there is the precondition that the window must have received a message. This condition is satisfied for the *begin_paint* routine. The routine *BeginPaint* is placed in the *WM_PAINT* case of the window procedure. *Fill_parameters/5* is called to add the parameters for *BeginPaint* to the example. One of the *fill_parameters/5* clauses includes the condition that the source for the current parameter is an argument of the window procedure. This is true for the first parameter of *BeginPaint*. The atom *hwnd* is added to *BeginPaint*'s parameter list phrase. Another *fill_parameters/5* clause contains the condition that the source of a parameter is a user declared variable and that the type of the parameter is a pointer to some type. As for any *fill_parameters/5* condition which involves a source predicate with a potentially uninstantiated variable, there are two paired clauses, one where the variable is instantiated and one where it is not. In this case, the variable is not instantiated. A variable name is created and the declaration for this new variable is added to the declaration list of the window procedure. Since the type of the declared variable, *PAINTSTRUCT*, is in all capitals, the variable name created is *paintstruct* (all lower case). If this was the second *PAINTSTRUCT* variable declared in the example, the new variable would take the name *paintstruct1* and so on. Since *BeginPaint*'s second parameter is actually a pointer to a *PAINTSTRUCT* (indicated by the functor *pointer_func(<frame_name_of_type>)* in the parameters slot of the *begin_paint* frame), the string *&paintstruct* is added to the parameter list of *BeginPaint*. The last step in filling this parameter is unifying the variable in the *user_source/2* predicate with the name *paintstruct*. Incidentally, ICOP can express the difference between a parameter in which the pointer should point to a declared piece of memory and a pointer which should not point to a declared piece of memory. Suppose that *BeginPaint* allocates the space for the *PAINTSTRUCT*. Then one should pass an unallocated pointer. Both versions of *BeginPaint* would have *pointer_func(paint_structure)* in their parameter slot. But in the default source slot, the version which needs an allocated pointer (as is the case here) would have the source predicate *user_source(user_declareable_object, _A)*, while the other version of *BeginPaint* would have the source *user_source(user_pointer, _A)*.

Back in *transform_example/2*, the return value is filled by creating the name *hdc* (the type of the return value of BeginPaint is *HDC*), declaring the variable in the window procedure, adding the name to the routine phrase, and unifying *hdc* with the uninstantiated variable in the *return/1* functor.

The next routine processed is shown in Figure 23. The *transform_example/2* clause applying in this instance contains the condition that the main effect of a routine is adding an object into user memory. This clause adds the routine to the *WM_CREATE* case of the window procedure. This captures the rule that if the effect of a routine is to add something to

```
routine(load_icon_resource, return(_D),
        [code_source(main_entry,hInstance),
        user_source(memory_object_name,_C)]).
```

Figure 23. *Routine/3* functor for *LoadIcon.*

memory, then this should occur only once at some point before any possible use of the object. The *WM_CREATE* message is sent to a window when it is first created. In this case, the *LoadIcon* routine is added to the routine list of the *WM_CREATE* case. Now the parameters are added. The *fill_parameters/5* clause applying for the first parameter is the one which contains the condition that the source be a parameter of the program's main entry routine (*WinMain*) and that the use of this parameter takes place in the window procedure. In this case, a global variable is declared which holds a copy of the *hInstance* parameter, an assignment is added to the list of routines in the main procedure (functional description [name='=', param=+'hInstance', return='hInstance_copy']), and the global variable is added to the list of parameters of *LoadIcon.* The *fill_parameters/5* clause applying for the second parameter contains the condition that the source be *user_source(memory_object_name, _A).* The predicate filler *memory_object_name* indicates that the argument is satisfied by a string. This source is placed in the default source slot of routines which accept a string naming the object on which they operate. In this case, the string names some icon resource in an executable file. *Fill_parameters/5* creates the name *name*, places this in *LoadIcon*'s parameter list, and unifies this string with the variable in the *user_source* predicate. Back in *transform_example/2,* a return variable name is created (*hicon*), the variable is added to the declaration list (static *HICON hicon;*), matched with the return attribute of the routine phrase, and unified with the variable in the *return/1* functor. The only interesting aspect of this processing is the "static" declaration. This particular clause of *transform_example/2* declares the return value static. This captures the rule that if an object is loaded at window creation, the handle to this object must be static so that future invocations of the window procedure (called by the operating system in response to events) will still have a valid handle.

The third routine processed is shown in Figure 24.

```
routine(draw_icon, return(_G),
        [routine_source(begin_paint,0,_B),
        user_source(x_coord,_E),
        user_source(y_coord,_F),
        routine_source(load_icon_resource,0,_D)]).
```

Figure 24. *Routine/3* functor for *DrawIcon*.

The *transform_example/2* clause applying here contains the condition that the main effect of the routine is drawing some drawable object on a display context and that the example already contains a *WM_PAINT* case. This clause expresses the rule that if a *WM_PAINT* case is already present (perhaps because of an earlier routine with a paint message received precondition, as in this example), then any routine which draws should also be added to the *WM_PAINT* case. In this case, *DrawIcon* is added to the *WM_PAINT* case. Now the parameters for *DrawIcon* are added. For the first parameter, the *fill_parameters/5* clause which applies contains the condition that the source be a routine and that the variable in the *routine_source* predicate be bound. The variable name is added to the parameter list. This is the same variable name which accepts the return value from *BeginPaint*. The next two parameters are handled by a clause which tests whether the source is a coordinate value supplied by the user. An integer constant is added to the parameter list in each case. The final parameter is handled the same as the first one. The variable name which accepts the return value for *LoadIcon* is added to the parameter list. The return value for *DrawIcon* is a boolean which indicates whether the icon was drawn successfully or not. A boolean declaration is added to the declaration list of the window procedure, the newly declared variable is matched to the return attribute of the routine phrase, and the variable name is unified with the variable in the *return/1* functor.

The final routine processed is shown in Figure 25.

```
routine(end_paint, return(_H),
        [routine_source(begin_paint,1,_A),
        routine_source(begin_paint,2,_B)]).
```

Figure 25. *Routine/3* functor for *EndPaint*.

The preconditions slot on the *end_paint* frame contains the precondition that a paint message has been received. *End_paint* is added to the example by the same *transform_example/2* clause as *begin_paint*, resulting in *EndPaint* being added to the end of the *WM_PAINT* case. Both parameters of *EndPaint* are supplied by a *routine_source/3* source. Since the third argument of both sources contains a value (an atom in one case and an instantiated variable in the other), they are both handled by the *fill_parameters/5* clause which adds a preexisting value to a parameter list. *EndPaint* does not return a value (as indicated by a return type of void in the *return_normal* slot of the *end_paint* frame), so nothing is matched with the return attribute of the routine phrase. The example has now been built. The final example tree is shown in Figure 26.

```
program(_P,
        decl_list(decl('HANDLE',hInstance_copy),_O),
        main(_N,_M,
                routine_list(routine(hInstance_copy,=,
                        param_list(hInstance,_L)),_K),_J),
        winproc(
                decl_list(decl('PAINTSTRUCT',paintstruct),
                decl_list(decl('HDC',hdc),
                decl_list(decl('static  HICON',hicon),
                decl_list(decl('BOOL',bool),_I)))),
                case_list(case('WM_PAINT',
                        routine_list(routine(hdc,'BeginPaint',
                                param_list(hwnd,
                                param_list('&paintstruct',_H))),
                        routine_list(routine(bool,'DrawIcon',
                                param_list(hdc,param_list(10,
                                param_list(15,
                                param_list(hicon,_G)))),
                        routine_list(routine(_F,'EndPaint',
                                param_list(hwnd,
                                param_list(&paintstruct,_E))),_D)))),
                case_list(case('WM_CREATE',
                        routine_list(routine(hicon,'LoadIcon',
                        param_list(hInstance_copy,
                        param_list('"name"',_C))),_B)),_A))))
```

Figure 26. Syntax tree for *DrawIcon* example.

Write_tree/1 walks through this tree in a depth-first manner writing out the example program. Any uninstantiated variables in the tree are skipped. The output of *write_tree/1*, which is the output of the example generator, is show in Figure 27.

```
HANDLE hInstance_copy;

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance
    LPSTR lpszCmdLine, int nCmdShow)
{

    ...
    hInstance_copy = hInstance;
    ...

}

long FAR PASCAL WndProc (HWND hwnd, WORD message, WORD wParam,
    LONG lParam)
{
    PAINTSTRUCT paintstruct;
    HDC hdc;
    static HICON hicon;
    BOOL bool;

    switch (message)
    {
        WM_PAINT:
            hdc = BeginPaint (hwnd, &paintstruct);
            bool = DrawIcon (hdc, 10, 15, hicon);
            EndPaint (hwnd, &paintstruct);
            return(0);

        WM_CREATE:
            hicon = LoadIcon (hInstance_copy, "name");
            return(0);

        ...
    }
    return DefWindowProc( hwnd, message, wParam, lParam);
}
```

Figure 27. Output of example generator for *DrawIcon* plan.

CHAPTER VII

CONCLUSIONS

ICOP serves as a proof of concept model for a system which facilitates software reuse by providing support for code location and comprehension. Chapter II describes a psychological model of programming. Three aspects of this model are most important to consider in building a system to support location and comprehension. First, programmers think in terms of domain goals, moving from a domain space to an application (artifact) space. Second, plans are used as an internal representation scheme to store pieces of artifact which accomplish goals. Finally, examples are an effective means of communicating information to programmers. These three cognitive considerations lead to the design of ICOP. The system uses a limited natural language interface to accept queries expressed directly in the domain language, not forcing the programmer to translate their request into the language of the library. The planner then constructs a plan to satisfy the desired effect. This plan includes multiple components from the library, conveying both plan level knowledge (patterns of use) as well as detailed knowledge regarding the use of particular components (eg. parameters and return values). Finally, the plan is illustrated with example code, taking advantage of the programmer's ability to successfully extract information from an example.

ICOP builds on ideas found in other cognitively motivated systems. It uses the concept of plans, which is found in The Programmer's Apprentice

[Rich and Waters, 1989, 1990] and Bridge [Bonar and Liffick, 1991]. However, rather than having a plan base of explicitly stored plans, ICOP builds plans from the atomic components represented in its knowledge base. The example systems developed by Neal [1990], Rosson and Carroll [in press] and Fischer, Henninger and Redmiles [1991] all use examples to facilitate both component level and plan level comprehension. ICOP also uses examples to facilitate comprehension, but rather than storing these examples explicitly in an example base, the examples are constructed dynamically using knowledge about the general form that examples should take. The automatic construction of plans and examples frees the knowledge engineer from having to explicitly represent examples for all possible user queries. ICOP allows the user to query by effect as suggested in the Cognitive Browser project [Green et al, in press]. Rather than using a formal effect language, however, the user can communicate the desired effect in the natural language of the domain.

ICOP's design is intended to be extendible to other library domains. This is facilitated by the explicit representation of predicates and by the deep ontology. By representing predicates and functors explicitly in the knowledge base, the limited natural language interface and the planner become independent of the library domain. All references to domain specific predicate and functors are made by exploring the predicates and functors in the knowledge base rather than through explicit use in the procedural code. The explanation generator is domain dependent, since the structure of examples in a given domain does depend upon the domain. The deep ontology used in the knowledge base is intended to make transfers to other domains easier in two ways. First, some aspects of the ontology should be directly reusable. For example, the concept of containers such as memory and files and objects which take up space in these containers should be useful in many library domains. Second, the Windows specific pieces of the ontology should serve as an example for building deep ontologies of other domains.

CHAPTER VIII

FUTURE WORK

There are many research directions suggested by ICOP. The areas for future research work can be divided into four categories: empirical validation, extending the current functionality, designing new functionality, and exploring applicability of ICOP's design to other domains.

EMPIRICAL VALIDATION

Since the design of ICOP was strongly motivated by cognitive considerations of the programming process, it is essential that ICOP be empirically validated. The programmer populations of interest are programmers who do not have experience in Windows but have written applications for windowing systems with a similar architecture (eg, Macintosh, X Windows), intermediate to expert programmers who have not written programs for a windowing environment but have used a windowing environment at some point (so they know what a window and a mouse is), and programmers with experience in Windows development. The initial set of experiments would divide each population into two groups, one of which has access to printed material and the standard Windows on-line help, and the other which has access to this plus ICOP. Each group is given a small Windows program to write and asked to talk aloud while writing it. In the talk aloud protocols, problems in program development caused by unresolved information needs are of particular interest. The impact of ICOP on the program development process will be analyzed. The protocols can also suggest future functionality that ICOP should have (new kinds of queries, different ways of phrasing existing queries, etc.).

ENHANCING EXISTING FUNCTIONALITY

Expanding the Knowledge Base

The first area of existing functionality that must be expanded is the knowledge base. It currently represents a small subset of the Windows library. As new library routines are represented, it will be interesting to watch what happens to the size of the effect language (predicates). The ideal behavior is that the effect language grows at a much slower rate than the number of new routines. What has been seen so far is that adding a new routine can sometimes require additions to the operators, states, functors, and object hierarchy, with these additions then supporting many new routines.

Generating Code Comments

The micro effect slot on routine frames was included to support the generation of natural language describing the routine. The simplest way to incorporate natural language describing routines into the existing design is to generate comments for the example code.

Representing Plans

Currently, ICOP does not represent plans in its ontology, only individual routines. The knowledge representation should be extended to represent plans as well. Such plans could be used to hilight standard or preferred ways of achieving effects. Currently, all plans that ICOP's planner can construct for achieving an effect are considered of equal desirability. Plans could also be used as an alternate way to represent postconditions. The postcondition processing in the planner generates "clean up" code for the plan. For example, if the routine *BeginPaint* appears in a plan, then *EndPaint* should appear at the end of the plan. An alternate way of representing this dependency is with a noncontiguous plan that states that drawing is accomplished by calling *BeginPaint*, some number of routines, and *EndPaint*. This has the advantage that the dependency between *BeginPaint* and *EndPaint* is represented locally (in one frame) rather than implicitly in the planner. Plan languages such as the Programmer's Apprentice plan calculus [Rich & Waters, 1989, 1990] should be explored.

DESIGNING NEW FUNCTIONALITY

New functionality of interest includes improving the interface, supporting transfer across libraries, supporting programmer modifiability of the knowledge base, and providing intelligent tutoring.

Improving the Interface

Coupling ICOP with a Development Environment. The current interface for ICOP consists of a natural language interface with queries typed from within Prolog. ICOP should be more strongly coupled with a programming environment so that working examples can be directly copied from ICOP to an editor window. In addition, such strong coupling could support context sensitive queries, in which clicking on a routine or data object within the editor generates an example using the routine or object, thus providing an alternative query mechanism to the natural language interface.

Supporting Natural Language Queries. Additional interface elements should support the natural language interface. A thesaurus browser, which lets the user explore what types of terms and concepts are known by the knowledge base, would facilitate querying. Such a browser was found useful in the medical information retrieval system Coach [Kingsland, Harbourt, Syed, Schuyler, 1993].

Supporting Multiple Aspects of Queries. Finally, the interface should support exploring different aspects of a query. After the query "How do I draw an icon?" produces the example, the user might want to explore how an icon is represented in memory, the structure of the PAINTSTRUCT data object, or a plan which tests whether LoadIcon succeeded. One way of handling this would be to provide a menu after every query which contains common queries for additional information. A more integrated way of handling this is to produce a small hypertext in response to each query. The example for drawing an icon would have buttons for common additional queries (such as error testing) as well as links from every word in the example which denotes an object in the knowledge base to a screen describing that object. ICOP would become an intelligent documentation system. Instead of writing documentation for a library in the traditional way, a knowledge base rich enough to support natural language generation would be written for the library. This has the advantage over English prose that the knowledge base can be mechanically checked for semantic consistency. When the user types a query, a small custom hypertext answering the query is constructed by the system. Such a system provides intelligent access to information, alleviate the hypertext navigation problem. When the library is changed, those frames representing the changed library objects are updated. New answers to queries will now be automatically produced; the technical writers (who are now knowledge engineers) do not have to worry about explicitly updating examples and cross references in a text.

Programmer Modification of Knowledge Base

Ideally, ICOP should support reuse for custom libraries used internally by a company as well as large libraries sold commercially. Since libraries used internally may be constantly changing and not be budgeted for knowledge engineers to maintain a knowledge base, it is important that the programmers themselves be able to make changes to the knowledge base when they change the library. Issues involving interfaces which support knowledge updates by people who are not professional knowledge engineers should be explored.

Intelligent Tutoring System

When a library for an entirely new domain is first used, the programmer will not know the domain concepts well enough to articulate queries. In such a case, an intelligent tutoring system (ITS) can assist the programmer in gaining the new domain knowledge. An ITS must be able to to direct the presentation of knowledge when the user requests general information (such as "Tell me about programming in Windows"). An attempt has been made to make ICOP's knowledge base general enough to support the reasoning processes of an ITS.

SUPPORTING DIFFERENT DOMAINS

Libraries for New Domains

ICOP's knowledge base currently supports a subset of the Windows API. A windowing library was chosen because the rich intertwining between components of such a library makes for an interesting reuse problem. There are other domains, however, which also have complex libraries that must be reused. Since the planner and semantic processor only refer to domain concepts via the explicitly defined predicates and objects in the knowledge base, the knowledge base is the main component which would have to be changed to support a different domain. How easy will it be to develop a knowledge base for ICOP for another domain? Some parts of the ontology should be reusable. For example, the concepts of containers (memory), objects which can be placed in containers (various types of variables), and objects which indirectly refer to another object (handles), should be useful in many domains.

Object-Oriented Libraries

ICOP can be extended to object-oriented libraries as well. Methods would be represented in the same manner as routines, with a new kind of frame representing classes. This frame would have slots with lists of method frames and member frames. The really new aspect would be handling queries where there is currently no object with a method satisfying the query precisely. Now a subclassing algorithm would have to determine which class is closest to producing the desired effect. The plan would then consist of defining a subclass of this class and changing one or more of the inherited methods. A procedural library does not have this concept of plans which involve modifying some library object.

REFERENCES

Adelson, B., & Soloway, E. (1985). The role of domain experience in software design. IEEE Transactions on Software Engineering, 11, 1351-1360.

Bellamy, R. (in press). Redesigning Programming Strategies: An Approach and an Example. In R. Winder (Ed.), Proceedings of the NATO advanced research workshop: User centered requirements for software engineering environments. Springer-Verlag.

Berry, D. (1992). Academic legitimacy of the software engineering discipline (Tech. Rep. No. CMU/SEI-92-TR-34). Pittsburgh: Carnegie Mellon University.

Biggerstaff, T., & Perlis, A. (1984). Forward to the special issue on software reuse. IEEE Transactions on Software Engineering, 10, 474-476.

Bonar, J., & Liffick, B. (1991). Communicating with high level plans. In J. Sullivan & T. Sherman (Eds.), Intelligent user interfaces (pp. 129-156). New York, NY: ACM Press.

Card, S., Moran, T., & Newell, A. (1983). The human information-processor. In The Psychology of Human-Computer Interaction (pp. 23-97). Hillsdale, NJ: Erlbaum.

Curtis, B. (1989). Cognitive issues in reusing software artifacts. In T. Biggerstaff & A. Perlis (Eds.), Software reusability Vol. 2. Applications and experience (pp. 269-287). New York, NY: ACM Press.

Fischer, G. (1987). Cognitive view of reuse and redesign. IEEE Software, July, 67-70.

Fischer, G., & Lemke, A. (1988). Construction kits and design environments: Steps toward human problem-domain communication. Human-Computer Interaction, 3, 179-222.

Fischer, G., Henninger, S., & Redmiles, D. (1991). Cognitive tools for locating and comprehending software objects for reuse. In Proceedings of the 13th International Conference on Software Engineering, May, Austin TX.

Fischer, G., Grudin, J., Lemke, A., McCall, R., Ostwald, J., Reeves, B., & Shipman, F. (1992). Supporting indirect collaborative design with integrated knowledge-based design environments. Human-Computer Interaction. 7, 281-314.

Green, T. R. G., Gilmore, D. J., Blumenthal, B. B., Davies, S. & Winder, R. (in press). Towards a Cognitive Browser for OOPS. International Journal on Human-Computer Interaction.

Guindon, R. (1990). Knowledge exploited by experts during software system design. International Journal of Man-Machine Studies, 33, 279-304.

Horowitz, E., & Munson, J. (1984). An expansive view of reusable software. IEEE Transactions on Software Engineering, 10, 477-487.

Jeffries, R., Turner, A., Polson, P. & Atwood., M. (1981). The Processes involved in designing software. In J. R. Anderson (Ed.), Cognitive skills and their acquistion (pp. 255-283). Hillsdale, NJ: Erlbaum.

Jones, T. (1984). Reusability in programming: A survey of the state of the art. IEEE Transactions on Software Engineering, 10, 488-493.

Kant, E. (1985). Understanding and automating algorithm design. IEEE Transactions on Software Engineering, 11, 1361-1374.

Kingsland, L.C., Harbourt, A.M., Syed, E.J. & Schuyler, P.L. (1993). Coach: applying UMLS Knowledge Sources in an expert searcher environment. Bulletin of the Medical Library Association, 81, 178-183.

Lehman, M. (1991). Software engineering, the software process and their support. IEE Software Engineering Journal, 6, 243-257.

Mellish, C. (1990). Generating natural language explanations from plans. In L. S. Sterling (Ed.), The practice of Prolog (pp. 181-223). Cambridge MA: The MIT Press.

Miller, G. (1956). The magical number seven, plus or minus two. Psychological Review, 63, 81-97.

Neal, L. (1990). Support for software design, development, and reuse through an example-based environment. In Proceedings of the Fifth Annual RADC Knowledge-Based Software Assistant Conference (KB5A-5), Sept. 24-28, 1990. Syracuse, NY.

Rich, C., & Waters, R. (1989). Formalizing reusable software components in the programmer's apprentice. In T. Biggerstaff & A. Perlis (Eds.), Software reusability Vol. 2. Applications and experience (pp. 269-287). New York, NY: ACM Press.

Rich, C., & Waters, R. (1990). The programmer's apprentice. New York, NY: ACM Press.

Rist, R. (1989). Schema creation in programming. Cognitive Science, 13, 389-414.

Rist, R. (in press). Search through multiple representations. In R. Winder (Ed.), Proceedings of the NATO advanced research workshop: User centered requirements for software engineering environments. Springer-Verlag.

Rosson, M. & Carroll, J. (in press). Active programming strategies in reuse. In European Conference on Object-Oriented Programming. Springer-Verlag.

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences, 8, 219-238.

Scholtz, J. (in press). The effect of the mental representation of programming knowledge on transfer. In R. Winder (Ed.), Proceedings of the NATO advanced research workshop: User centered requirements for software engineering environments. Springer-Verlag.

Simon, H. (1981). The sciences of the artificial. Cambridge, MA: MIT Press.

Singley, M., Carroll, J., & Alpert, S. (1991). Psychological design rationale for an intelligent tutoring system for smalltalk. In J. Koenemann-Belliveau, T. Moher, & S. Roherton (Eds.), Empirical Studies of Programmers: Fourth Workshop (pp. 196-209). Norwood, NJ: Ablex.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. (1984a). What do novices know about programming?. In A. Badre & B. Shneiderman (Eds.), Directions in Human-Computer Interaction (pp. 27-54). Norwood, NJ: Ablex.

Soloway, E. & Ehrlich, K. (1984b). Empirical studies of programming knowledge. IEEE Transactions on Software Engineering, 10, 595-609.

Soloway, E., & Ehrlich, K. (1984c). An empirical investigation of the tacit plan knowledge in programming. In J. Thomas, & M. Schneider (Eds.), Human Factors in Computer Systems (pp. 113-133). Norwood, NJ: Ablex.

Sommerville, I. (1989). Software Engineering (p. 3). Addison-Wesley.

APPENDIX

The appendix contains sample frames from the knowledge base.

Functor mapping two *integers* and an *icon* to *drawable_icon*

```
frame(name:icon_func,
    parent:arity_3_functor,
    children:[],
    slots:[arity:facets([value, type]),
        arg_constraint:facets([value:[integer, integer, icon], type,
            number]),
        return:facets([value:drawable_icon, type])]).
```

Frame representing action (predicate) of drawing on a device context

```
frame(name:draw,
    parent:arity_2_operator,
    children:[],
    slots:[arity:facets([value,type]),
        arg_constraint:facets([value:[device_context, gdi_drawable_object],
            type, number])]).
```

Frame representing a memory container

```
frame(name:memory,
    parent:container,
    children:[user_memory,
        system_memory],
    slots:[]).
```

Frame representing a referenceable object (a type of memory object)

```
frame(name:referenceable_object,
    parent:memory_object,
    children:[gdi_object,
        region,
        resource,
        window,
        application_instance],
    slots:[]).
```

Frame representing a handle (a type of declarable object)

```
frame(name:handle,
    parent:windows_type,
    children:[hbitmap,
        hcursor,
        hdevice_context,
        hicon,
        hinst,
        hmenu,
        hwindow,
        pointer],
    slots:[library_name:facets([value:'HANDLE', type]),
        referenced_object:facets([value:referenceable_object,
            type:referenceable_object])]).
```

Frame representing a device context (a type of state object)

```
frame(name:device_context,
    parent:state_object,
    children:[],
    slots:[owner:facets([value:window, type]),
        attributes:facets([value:[
            attrib(context_mapping_mode, mm_text, mapping_mode),
            attrib(window_origin, point_func(0, 0), point),
            attrib(viewport_origin, point_func(0, 0), point),
            attrib(window_extents, point_func(1, 1), point),
            attrib(viewport_extents, point_func(1, 1), point),
            attrib(context_pen, pen_func(black_pen_const), pen),
            attrib(context_brush, brush_func(white_brush_const), brush),
            attrib(context_font, font_func(system_font_const), font),
            attrib(context_bitmap, no_val, bitmap),
            attrib(current_pen_pos, point_func(0, 0), point),
            attrib(context_background_mode, opaque, integer),
            attrib(background_color, rgb_func(255, 255, 255),
                color_specification),
            attrib(text_color, rgb_func(0, 0, 0), color_specification),
            attrib(context_drawing_mode, r2_copypen, integer),
            attrib(context_stretching_mode, black_on_white, integer),
            attrib(context_polygon_fill_mode, alternate, integer),
            attrib(context_intercharacter_spacing, int_func(0), integer),
            attrib(brush_origin,
                device_coords_func(point_func(0,0),screen_coord), point),
            attrib(context_clipping_region, no_val, region)], type, min])]).
```

Parent of stock brush hierarchy (integer constants)

```
frame(name:stock_brush_const,
    parent:integer_constant,
    children:[black_brush_const,
        dark_gray_brush_const,
        gray_brush_const,
        hollow_brush_const,
        light_gray_brush_const,
        null_brush_const,
        white_brush_const],
    slots:[library_name:facets([value, type]),
        object_type:facets([value, type])]).
```

Parent of drawable object hierarchy

```
frame(name:gdi_drawable_object,
    parent:screen_object,
    children:[drawable_arc,
        drawable_chord,
        drawable_ellipse,
        drawable_icon,
        drawable_line,
        drawable_pie,
        drawable_point,
        drawable_polygon,
        drawable_polyline,
        drawable_poly_polygon,
        drawable_rectangle,
        drawable_roundrect],
    slots:[]).
```

Parent of *user_source_objects*

```
frame(name:user_source_object,
    parent:object,
    children:[memory_object_name,
        user_declareable_object,
        coord],
    slots:[source_type:facets([value, type:declareable_object])]).
```

Frame representing *BeginPaint* routine (inherits some properties from *get_device_context*)

```
frame(name:begin_paint,
    parent:get_device_context,
    children:[],
    slots:[routine_name:facets([value:'BeginPaint', type]),
        parameter_list:facets([value:[hwindow, pointer_func(paint_struct)],
            type, min]),
        default_source:facets([value:[code_source(window_procedure, hwnd),
            user_source(user_declareable_object, Val1)], type, min]),
        return_normal:facets([value, type]),
        return_error:facets([value, type]),
        main_effect:facets([value, type, min]),
        micro_effect:facets([value:[
            dereference(param_1, memory_1, window_1),
            associated([window_1], device_context_1),
            associated([window_1], paint_info_1),
            get_attribute(invalid_rectangle, invalid_rect_1, paint_info_1),
            make_rect_region(invalid_rect_1, rect_region_1),
            set_attribute(context_clipping_region, rect_region_1,
                device_context_1),
            dereference(param_2, memory_1, paint_struct_1),
            fill_info(paint_info_1, paint_struct_1),
            make_reference(hdevice_context_1, memory_1,
                device_context_1),
            send_message(window_1, erase_background_message),
            return(hdevice_context_1)], type, min]),
        constraint:facets([value:[assoc_type(memory_1, memory),
            assoc_type(window_1, window),
            assoc_type(device_context_1, device_context),
            assoc_type(paint_struct_1, paint_struct),
            assoc_type(paint_info_1, paint_info),
            assoc_type(hdevice_context_1, hdevice_context),
            assoc_type(rect_region_1, region),
            assoc_type(invalid_rect_1, rectangle)], type, min]),
        preconditions:facets([value:[received(window_1, paint_message)],
            type, min])]).
```