Knowledge Intensive Natural Language Generation with Revision

by

Ben Eric Cline

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

in

Computer Science

Approved:

J. Terry Nutter, Chairman

John W. Roach

Edward A. Fox

Helen J. Crawford

F. Gail Gray

May 1994
Blacksburg, Virginia

Knowledge Intensive Natural Language Generation with Revision

by

Ben Eric Cline

Committee Chairman: J. Terry Nutter

Computer Science

(ABSTRACT)

Traditional natural language generation systems use a pipelined architecture. Two problems with this architecture are poor task decomposition and the lack of interaction between conceptual and stylistic decisions making. A revision architecture operating in a knowledge intensive environment is proposed as a means to deal with these two problems. In a revision system, text is produced and refined iteratively. A text production cycle consists of two steps. First, the text generators produce initial text. Second, this text is examined for defects by revisors. When defects are found, the revisors make suggestions for the regeneration of the text. The text generator/revision cycle continues to polish the text iteratively until no more defects can be found. Although previous research has focused on stylistic revisions only, this paper describes techniques for both stylistic and conceptual revisions.

Using revision to produce extended natural language text through a series of drafts provides three significant advantages over a traditional natural language generation system. First, it reduces complexity through task decomposition. Second, it promotes text polishing techniques that benefit from the ability to examine generated text in the context of the underlying knowledge from which it was generated. Third, it provides a mechanism for the integrated handling of conceptual and stylistic decisions.

For revision to operate intelligently and efficiently, the revision component must have access to both the surface text and the underlying knowledge from which it was generated. A knowledge intensive architecture with a uniform knowledge base allows the revision software to quickly locate referents, choices made in producing the defective text, alternatives to the decisions made at both the conceptual and stylistic levels, and the intent of the text. The revisors use this knowledge, along with facts about the topic at hand and knowledge about how text is produced, to select alternatives for improving the text.

The Kalos system was implemented to illustrate revision processing in a natural language generation system. It produces advanced draft quality text for a microprocessor users' guide from a knowledge base describing the microprocessor. It uses revision techniques in a knowledge intensive

environment to iteratively polish its initial generation. The system performs both conceptual and stylistic revisions. Example output from the system, showing both types of revision, is presented and discussed. Techniques for dealing with the computational problems caused by the system's uniform knowledge base are described.

Acknowledgements

As I approach the end of this long, winding road called "getting a Ph.D.," it's time to reflect on my journey and acknowledge those who have aided me in my travels.

My major advisor, J. Terry Nutter, contributed good advice, pertinent ideas, and lots of hard work. Her reading of countless article and dissertation drafts nearly qualifies as "beyond the call of duty." Her "nutterization" of my prose has taught me much about writing. Foremost, the direction she has given to this research is paramount to its success. Her sense of which fork in the road to take has served me well.

The advice from my other committee members, Helen J. Crawford, Edward A. Fox, F. Gail Gray, and John W. Roach, is greatly appreciated. Each also contributed their valuable time in reading my research proposal and my dissertation.

I thank the administration at Virginia Tech for providing me with the opportunity to take a year of educational leave from my job to work towards this degree. Without this time, the tasks of taking the Ph.D. qualifying exam and finding a research topic would never have been completed. I thank the management of Virginia Tech Information Systems for providing the motivation I needed to actually finish the degree.

Mike Emero worked in the area of automatically determining natural categories from computer science texts. His work was beneficial when I constructed the knowledge base and lexicon for Kalos. More importantly, Mike's friendship and support is appreciated.

Thanks to Phil Benchoff, KC4ZEN, for pointing out that a few C language *printf()* statements could be used to perform all the work that Kalos requires hours of CPU time to perform. I hope this dissertation will begin to convince him that natural language generation is a bit more complicated than he initially thought.

A special thanks goes to Greg Lavender. Like me, he worked professionally during part of his Ph.D studies. Knowing that he traveled that long, winding road to the end gave me hope and encouragement that, someday, I might also end my journey. He was supportive in many ways during the classes we took together, while studying for qualifiers, and during lunches at Arnold's. Our discussions of formal topics was always enlightening –to both of us, I hope– because our approaches were different. Of all the computer scientists I have known, I think Greg understands best my frustration at having written the same loop in countless languages over my years as a computer professional.

One cannot be engaged in this type of research without the proper tools. I thank Stuart C. Shapiro and company of the SUNY Computer Science Department for their excellent semantic

network package, SNePS-2.1. For exceptional support of this package, I thank Deepak Kumar and Hans Chalupsky. The other tool I used was Austin Kyoto Common Lisp (AKCL), a much enhanced version of Kyoto Common Lisp by Bill Schelter. Dr. Schelter's work on this extensive, portable version of Common Lisp is greatly appreciated.

My parents have always been supportive of my academic pursuits. Although not directly involved in this dissertation, it is only by their hard work, support, and encouragement that I came to have the choice to travel this road.

My wife, Sue Ellen, sacrificed more than anyone else during my pursuit of this degree. She gave up spending time with me and going to entertainment and social events so that I could have time to work on this dissertation. Sue Ellen was always positive and supportive, even as the years dragged on. The only sacrifice I regret is the time I missed being with her.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

> Q: Which is greater the spoken word or the written word?
> A: The written word, by far. It is of better quality, having benefited from planning, organization and revision. It has greater stability, making our memories look ephemeral by comparison. And it can reach more people over the course of time, including those not yet born.
>
> Marilyn Vos Savant, "Ask Marilyn." *Parade Magazine*

The use of natural languages by computers has been a long-term goal of computer science researchers. Computer systems with competent natural language interfaces would be applicable to a number of areas, including interfaces for computer novices and summarization by computer of written text. Furthermore, proficiency in language is one measure of human intelligence which, if a tantamount proficiency were implemented in a computer system, would be corroboration of true intelligence. Gabriel [1988] believes that natural language generation is the ultimate problem for artificial intelligence research.

However, proficiency in natural language by computers is currently not comparable to language usage by humans [Barr & Feigenbaum 1981]. People use language in a variety of situations, both spoken and written, and can adapt to novel linguistic situations. People even use language as a sort of game when they tell or listen to puns and jokes.

Computer systems, on the other hand, demonstrate only primitive language skills. Systems with natural language interfaces use language in a limited fashion, and adapt poorly if at all. For example, an expert system in the domain of cardiology might be able to explain the reasoning it used to come to some diagnosis, but would not even understand the question when asked where the pharmacy was located. Contrast this to a cardiologist, who can not only describe his reasons for coming to some medical conclusion, but who can also describe the way to the pharmacy and thousands of other facts that are not in the reach of the expert system [Lenat *et al.* 1990]. The human has command of a much larger set of knowledge than the expert system, but the expert system's deficiency is not only a matter of having a limited knowledge base. It lacks general competency in natural language.

Language proficiency can be divided into two skills: understanding and generation. A conversant must be able to derive meaning from the spoken or written text of other conversants and to convey ideas by generating speech or written text. In this dissertation, the focus is on the generation of natural language by computer. In particular, this research is in the area of connected text generation: the generation by computer of multiple sentence, formal text. Gabriel [1988] calls the

generation of formal, written text "deliberate writing." In deliberate writing, the author typically has no direct contact with his readers; they cannot ask him questions or provide other feedback. If the text is to succeed in communicating the intended ideas, the writing must be careful and considered. Not only must grammatically correct sentences be produced, but these sentences must be produced in the context of other sentences to form coherent paragraphs. These paragraphs must fit together in a coherent discourse structure. The result is a unified text.

Some research in natural language generation has focused on the task of generating short answers to questions. This type of system faces fewer difficulties than a connected text generation system because it does not have to deal with organizing concepts into a logically ordered discourse structure or the interrelationships between sentences and paragraphs. Short answer systems are required to operate in real-time if they are to effectively interact with a user. On the other hand, connected text generation systems typically do not need to operate in real-time.

State-of-the-art natural language generation systems are unable to generate polished connected text at anywhere near the proficiency of a typical human author (see section 1.1). Due to our lack of general generation techniques and limited knowledge bases, attempts to construct proficient natural language generation systems are presently doomed. A better approach for success in this area is to develop techniques that are useful in more limited generation systems such as the production of draft text that a human writer can then polish.

Consider the production of a users' guide for a microprocessor. Preparation of this type of documentation by design engineers may not be an appropriate use of their time, and design engineers may not have the necessary technical writing skills to produce a manual. On the other hand, a professional technical writer has the writing skills, but probably lacks the technical knowledge and knowledge about the specific microprocessor to produce a manual. A computer system that could automatically produce a draft of the users' guide from a knowledge base about the microprocessor would provide the technical writer with a foundation to produce a polished manual.

Although we are not ready to build a computer-based author, we can concentrate on developing computerized writing assistants that can automatically produce draft-level text. The human author would be relieved of the tedious task of gathering and organizing information. The author's skills would more appropriately be used to read and revise draft text.

To develop this type of automated writing tool, two items are needed:

- Knowledge bases that cover the subject areas for the discourse topics at hand must be available or must be able to be generated automatically.
- Generation techniques must be available to produce draft quality text.

For a natural language generation system that produces draft text to be useful, knowledge bases on

relevant subject topics must be readily available. If not, there is no economy of effort in pursuing the formidable task of creating a knowledge base to save effort solely in draft generation. Only where knowledge bases exist for other reasons or where they can be generated automatically from written text is automated draft generation appropriate. Currently, the automatic generation of knowledge bases is only attainable in limited domains where there are constraints on the text to be read.

Once a knowledge base is obtained, generation techniques are needed to produce draft text to satisfy some discourse goal, such as describing some device. These techniques must select and order facts from the knowledge base and convert them into natural language in a way that conveys to the intended reader messages that are appropriate to satisfy the discourse goal at hand. As described in the next section, this task is formidable. State-of-the-art natural language generation techniques are limited in the amount and robustness of the text they generate. Another problem is the exponential growth of computational power needed to handle generation of text from linearly larger knowledge bases [Reiter 1990].

The technology to construct a general natural language generation system, even one adequate only for producing drafts, is unlikely to appear in the near future. The difficulty of this task should not, however, cause us to avoid pursuit of better generation techniques. Workers in this area must continue to develop and test techniques in limited areas with the goal of making more practical, however limited, systems available and narrowing the distance between the current limited techniques and those able to produce comprehensive natural language generation.

The heart of this research is the development and testing of two interrelated natural language generation techniques that show particular promise as useful techniques for natural language generation systems currently under development and that open areas of further research that may prove useful in attaining comprehensive natural language generation systems. These techniques are revision and knowledge intensive generation. By revision, we refer to the iterative generation of text, where text from a previous generation pass is examined for defects in the current phase and regenerated in a way that improves the text. Revision allows the system to postpone some decisions in the early stages of text generation to a later stage, thus reducing the complexity of any given stage of generation. Furthermore, the use of revision by a natural language generation program provides a way to produce better text by allowing different stages of the generation process to interact through the revision processing. Finally, some text defects, such as unintentional rhyming, are best dealt with by revision processing. Human authors and editors use an analogous technique when they polish a draft text.

Knowledge intensive generation exploits explicit representations about all aspects of natural

language generation. Furthermore, a uniform representation scheme is used so that individual generation tasks can examine all the pertinent system knowledge including representations of intermediate results. I argue in Chapter 4 that a knowledge-intensive approach in which all types of knowledge are represented in a uniform knowledge base provides the only reasonable architecture for a revision system. The primary benefits of this scheme are that revision techniques can operate in a computationally economical fashion, not needing to infer results from a previous stage, and in an intelligent manner, by making decisions based on all pertinent knowledge. Part and parcel of using a monolithic knowledge base is the development of techniques to deal with the problem of computational explosion with large knowledge bases. Other benefits are derived from using a knowledge-intensive approach, such as easy extension of the system by the addition of knowledge. The sections on Revision and Knowledge Intensive Generation (sections 1.2 and 1.3, respectively) describe the usefulness of these techniques in greater detail.

1.1 Difficulties in Connected Text Generation

Language skills are difficult to program because the processes involved are not well understood. There are no general algorithms for dealing with natural language. Instead of algorithms, people are taught grammar and composition rules in English classes that are not easily adapted to computer systems. Rules for composition have subjective elements that are difficult for a computer system to deal with. For example, one possible rule of composition is "be concise," but it is difficult to quantify the conciseness of written text, let alone describe how to make a verbose text concise.

Traditionally, the problems associated with natural language generation have been divided into two broad categories: deciding what to say and deciding how to say it [McKeown 1985, McKeown & Swartout 1987]. The former refers to making choices about selecting and ordering facts to meet the discourse goal at hand, while the latter refers to making choices to render the selected facts into natural language. We now consider some of the key difficulties associated with these two problems.

- Text Organization: How can text be organized to satisfy some discourse goal?
- Surface Text and Cohesion: How can text be made cohesive?
- Knowledge Representation Issues: What are the knowledge representation issues for language?
- Reasoning About Conversants: How are the intentions and beliefs of the reader and writer handled?
- Task Decomposition: How can the text generation problem be decomposed to reduce complexity and enhance understanding of the problem?
- Text Goodness: What are the measures of good text?

1.1.1 Text Organization

Natural language generation systems must select and order facts from a potentially large knowledge base to satisfy some discourse goal. The result is a logically ordered set of facts analogous to a detailed outline that a human author might create when developing formal text. As Gabriel [1988] notes, the system must linearize facts and other statements that describe an object or action that is inherently "multi-dimensional." For example, missing facts that are yet to be presented must not cause the reader concern.

Selecting what should be said to satisfy some discourse goal is a formidable task. When a human is asked to produce text for a specific purpose, he will rarely state all he knows about the subject at hand. An engineer writing a users' guide for a microprocessor knows a great deal about the microprocessor and has a large amount of general computer knowledge. But a typical users' guide would contain only a portion of the computer-related knowledge the engineer knows. Some of the engineer's knowledge is excluded from the text because he assumes his intended audience already has this knowledge. For example, general purpose registers in a computer typically are both readable and writable under program control; the intended readers of a microprocessor users' guide know this fact, so it would not be stated. Other knowledge is excluded because the engineer considers it irrelevant to the discourse task at hand. For example, the purpose of most microprocessor users' guides is to impart the knowledge needed to design a circuit using the microprocessor and to allow a programmer to develop programs for it. Specifics on the CAD system used to lay out the gates on the microprocessor substrate may be of interest to the author, but not relevant to the tasks of using the microprocessor in a circuit or programming it.

Another problem related to text organization is how to determine the ordering of the facts to be presented. Readers and hearers of text assume that logical discourse organization will be used by authors and speakers to lead them to new information from the context of known information [Clark & Clark 1977]. But general techniques to order facts to meet some discourse goal do not exist.

Human authors tend to use standard patterns of discourse [McKeown 1985]. For example, when asked to describe an object, an author or speaker might begin relating the object to a taxonomy, followed by a list of attributes of the object, followed by a description of the parts of the object. The author or speaker could use this same discourse pattern in a variety of situations to describe a number of different objects. Standard discourse patterns can be combined in a variety of ways to meet a number of discourse tasks.

Similarly, a connected text generation system must have a repertoire of discourse structures, i.e., commonly used patterns of discourse. Finding an appropriate set of discourse structures for a

particular area of text generation is a difficult task. Although texts produced by human authors can be analyzed, this type of analysis is somewhat subjective. Given a particular sentence or phrase in a text, various readers might assign a different function to it [McKeown 1985]. In some instances, discourse structures need to be overlaid to fulfill multiple needs.

Another problem in selecting discourse structures relates to the number of structures selected. If too few discourse patterns are used in a generation program, the structure of the generated text may be repetitious. On the other hand, using more discourse structures complicates discourse structure selection.

Once a repertoire of discourse structures is selected, techniques are needed to allow the generation program to select an appropriate discourse structure for the task at hand. For example, if a program is to describe a computer, does it produce a description by describing each part in relation to some taxonomy, or does it describe the actions each part performs? Or, perhaps, it should use some combination of these two approaches. Several factors influence what types of discourse patterns are used. For example, in describing objects, Paris [1985] found that the expertise of the intended audience typically influences the level of description. Experts have knowledge about many objects and how they function, so they are given details about parts. They do not need process descriptions, i.e., descriptions of the actions an object performs. On the other hand, novices need process information because they do not have this type of information or the necessary knowledge to derive it in their background. Therefore descriptions aimed at novices contain both process descriptions that describe how objects work and details about the individual objects.

Discourse patterns may vary across different domains, and even across descriptions of different types of objects. For example, a person would typically use different discourse patterns for describing the directions for getting to some location than for describing a complex computer algorithm. A generation program that produces descriptions would have to be able to select both a general discourse pattern, e.g., description, and then select the type of this pattern suited to the discourse task at hand.

Selection of an appropriate discourse structure from a natural language generation program's repertoire is only part of the problem of selecting and ordering knowledge base concepts as the basis of coherent text. The program must be able to relate the discourse structure to information in the knowledge base. It has to be able to determine which facts are salient and which ones should not be stated. Stating the obvious is confusing to the reader, while leaving out useful information is frustrating for him [Cline & Nutter 1990]. Human authors writing about the same topic will disagree to some extent on what information is salient, which attests to the difficulty of producing good salience algorithms for the computer.

1.1.2 Surface Text and Cohesion

In addition to the problems of "what to say," there are problems that a natural language generation system must deal with in the area of "how to say it." After deciding what should be said, the generation system must convert the selected knowledge into natural language. Natural language is rich in its expressive power. There are a number of surface constructs to convey any given concept. But to make the implementation of natural language generation systems approachable, the vocabulary and types of sentences a natural language generation program can produce must be limited. A system must be designed to be expressive enough for the task at hand, while being limited to the point that it can be practically implemented. If expressiveness is too limited, the system will not be able to render necessary concepts or will be repetitious in its text. The selection of the proper vocabulary and sentence structure for a particular system is a difficult task.

Once a vocabulary and repertoire of sentence structures has been selected, there is a problem of selecting the most appropriate rendering for a concept. It is difficult in practice to determine which of the surface renderings is best suited in any situation. Further complicating this task is the that fact that lexical choice, i.e., selecting the word or words we use to express a concept, can impact the structure of the text [Danlos 1987].

In a connected text generation system, the sentences cannot be generated in isolation. They must be cohesive. Readers expect cohesive surface constructions to guide them through the text. To attain cohesion, a natural language generation system must use pronouns, compounding, and other surface-level techniques intelligently [Halliday & Hasan 1976]. But applying cohesive techniques can be difficult. For example, using pronouns correctly can require knowledge about conceptual ideas such as focus of attention [Grosz & Sidner 1986].

A connected text generator must avoid infelicitous, verbose, and ambiguous text. It must avoid awkwardness based on surface effects such as rhyming or repetition (e.g., "the D0 data register is a register"). Yet, techniques for avoiding poor text, or even detecting it, are not well-defined. To deal with these problems, a natural language generator must not only be able to deal with individual words and their meanings, but also the interaction between words.

1.1.3 Knowledge Representation Issues

The problem of natural language generation can be viewed as one of developing methods to make appropriate choices to select, order, and render in natural language facts that satisfy some discourse goal. These choices must be made in the context of potentially large knowledge bases describing domain, discourse structure, and linguistic knowledge, making the number of possible choices potentially computationally intractable.

A natural language generation system requires knowledge about some discourse topic and knowledge related to organizing and generating text. The knowledge base for a generation system must contain the facts needed to support the type of text desired. The facts have to be represented in such a way that they can be accessed and so that relevant relationships between facts can be found efficiently [McKeown & Swartout 1987, Gabriel 1988].

Domain knowledge bases for natural language generation systems are either crafted for the purpose of generation or for some other purpose. The latter type of knowledge base typically does not support robust natural language generation. For example, a knowledge base crafted for an expert system may be adequate for the expert system to demonstrate appropriate decision making; however, humans tend to ask questions that go beyond the simple decision explanation expert systems produce [McKeown & Swartout 1987]. This difficulty illustrates that the knowledge required for adequate natural language generation goes beyond the knowledge required for problem solving in some domain.

As natural language generation systems become more complex, larger knowledge bases will be required. The problems with large knowledge bases in natural language generation systems are well known [McKeown & Swartout 1987]. Traditional approaches have been to segregate large knowledge bases and use only the portion needed for the generation task at hand. But human authors appear to reason about many different types of knowledge as they write. Any knowledge base segregation scheme must not prevent necessary decisions from being made due to the segregation.

Another knowledge representation issue that impacts natural language generation relates to how humans use world knowledge in generating and understanding text. Human conversants share a large amount of knowledge that is fundamental to language understanding and generation. For example, if you are told that Joe went to a restaurant, ordered a pizza, and left, you would assume that after Joe ordered the pizza, one was cooked for him and either he ate it or took it away from the restaurant as carryout. You would also assume that Joe paid for the pizza. None of these facts are stated, but we know that people typically wait to be served after ordering and that people pay for items they order [Schank & Riesbeck 1981]. A hearer understands these things and makes the proper assumptions. A speaker also understands typical restaurant situations and avoids stating the obvious. Without access to world knowledge on the scale of that understood by humans, computer systems will not be generally proficient in using natural language. Effective use of large knowledge bases of this type is a difficult task [Lenat *et al.* 1990].

1.1.4 Reasoning About Conversants

The intentions, motivations, and beliefs of conversants are important to the study of text generation [Appelt 1985, Meehan 1981, Hovy 1988, Grosz & Sidner 1986]. As we converse with other people, we reason about what they know and how it changes as we converse with them. We also make assumptions about their intentions as an aid to understanding what they say. For example, if a teacher is instructing an advanced guitar student, she reasons about the student's knowledge. She would assume that the student already knows a number of fundamentals about his instrument such as how to hold it and the relationship of the notes to the strings and fret positions. She uses this assumption in selecting what to say to the pupil. As she begins teaching the student, she reasons about what he has learned. If she feels the student hasn't understood a particular lesson, she may choose to repeat it, or she might decide that the pupil understands the lesson and continue to the next one.

In connected text generation systems, reasoning about intentions and beliefs is typically less explicit then in conversational systems; however, the lack of reasoning about the reader on the part of the generation system seems to be a limiting factor. Human authors, on the other hand, do reason about the intentions and beliefs of their intended audience. They use this knowledge when deciding what facts to state and which ones should not be presented. They also reason about how the text at hand changes the knowledge and beliefs of the intended reader. For example, as facts are presented, the author may assume that the reader understands what has been presented. The author may also decide that a particular point is difficult for the reader to understand and present an example to make the point clearer. Such reasoning by natural language generation systems is difficult, in general, to implement and is computationally expensive.

One aspect of a natural language generation program, then, is to be able to model its intended audience in terms of what is known by typical audience members and what vocabulary they use [Cline & Nutter 1990]. Given the model, techniques are needed to reason about it and change it as facts are presented to the user.

1.1.5 Task Decomposition

Because natural language generation is a complex task, researchers in the field have typically decomposed the problem in a way to reduce overall system complexity. A good decomposition allows algorithms to be developed that address limited parts of the overall generation problem. Such algorithms are easier to develop and understand than those in a traditional natural language generation system.

Problems arise when the task at hand is not decomposed properly. Addressing natural language generation as a monolithic task results in complex algorithms and efficiency problems in dealing with large knowledge bases. On the other hand, if the problem is decomposed incorrectly, some decisions may be made based on incomplete knowledge or inter-task communication may be extremely complex. One problem area identified by several researchers is that traditional natural language generation systems do not allow lexical choice to affect conceptual decisions [Danlos 1987, McKeown & Swartout 1987]. This lack of interaction at the conceptual and stylistic levels is a result of the natural language generation architecture used by these systems.

Task decomposition of a knowledge intensive system involves the decomposition of both tasks and knowledge bases. The conflicting goals of robustness and efficiency make overall system architecture decisions difficult. I address these issues in section 3.2.

1.1.6 Text Goodness

Human authors reason about the goodness of their text. A writer of formal text typically reviews what he has written looking for defects, including text that does not meet his intended purpose. But computer-based measures of the goodness of text are relatively simple when compared to that of human editors. Computer writing aids use simple measures such as those based on vocabulary and sentence complexity [Kieras 1989]. Human authors actually read and understand the text that they are editing, looking for defects at both the surface and conceptual levels.

A natural language generation system could use text goodness measures in several ways. First, if poor text is detected, the computer system could indicate this to a human operator. If the measure of goodness was accurate, human intervention in reading and correcting the generated text could be limited to problem areas detected by the computer. Second, if text defects could be detected, natural language generation could attempt to effect repairs automatically.

1.1.7 Research Focus

Designing a computer system to generate formal, connected text is a difficult problem. Techniques are needed to ease the burden on the system designer who attempts to develop a natural language

generation system. This research focuses on two interrelated techniques: revision and knowledge-intensive generation. Revision allows the system to postpone some decisions in the early stages of text generation to a later stage, thus reducing the complexity of any given stage of generation. Furthermore, the use of revision by a natural language generation program provides a way to produce better text by allowing different stages of the generation process to interact through revision processing. Finally, some text defects, such as unintentional rhyming, are best dealt with by revision processing.

A knowledge-intensive approach in which all types of knowledge are represented in a uniform knowledge base seems to provide the only reasonable architecture for a revision system (see Chapter 4). Other benefits are derived from using a knowledge-intensive approach, such as easy extension of the system by the addition of knowledge. The following sections on Revision and Knowledge Intensive Generation (sections 1.2 and 1.3, respectively) describe the usefulness of these techniques in greater detail.

State-of-the-art natural language generation systems do not attempt to address all the problems with connected text generation due to the complexity of the task. To make progress in this field, researchers concentrate on limited parts of the problem. Although the primary focus of this dissertation is generation task decomposition and knowledge intensive generation, other generation problems are addressed in a limited fashion. Salience determination is addressed in section 6.1.1. Information on audience modeling is given in section 5.3.1. The knowledge requirements for a revision system are addressed in section 3.3.

1.2 Revision

Traditional natural language generation systems are based on a pipelined architecture consisting of two steps [Kantrowitz & Bates 1992]. First, a deep generation component determines what should be said. Second, a surface generation component constructs surface-level text from the concepts produced in the first step. (See Figure 1.1.) By contrast, human writers producing formal text typically create drafts that they revise iteratively to produce polished text. Several researchers in natural language generation [Vaughan & McDonald 1986, Yazdani 1987, Cline 1991, Cline & Nutter 1992, Cline & Nutter 1994a, Cline & Nutter 1994b] have proposed that a similar draft-and-revision paradigm could be used to improve the quality of computer generated multisentential text.

Figure 1.2 is a block diagram of a knowledge-based natural language generation system incorporating revision. Using a discourse goal, the deep generator selects which concepts in the system's knowledge base will be included in the generated text. For the text to be coherent, the concepts

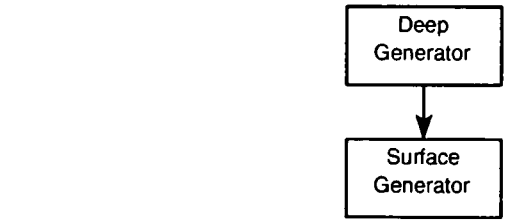


Figure 1.1 - Traditional Generation Architecture

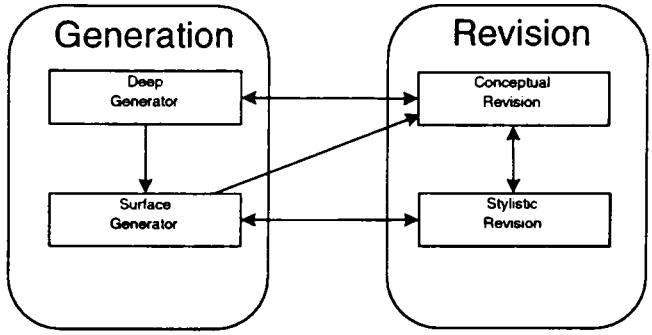


Figure 1.2 - Architecture for Revision

must be ordered and sentence and paragraph boundaries determined. The surface generator receives the concepts selected by the deep generator as input. It selects words and arranges them in conformity with natural language syntax to produce a text that conveys the ideas of the internal representation [McKeown & Swartout 1987]. The revision component, or revisor, analyzes the text produced by the generation components and produces suggestions to both generators for improving the text. The revisor is divided into two components: a conceptual revisor that makes suggestions to the deep generator and the stylistic revisor that makes suggestions to the surface generator. Revisions made by the deep generation component are called *conceptual revisions*, while revisions made by the surface generator are called *stylistic revisions*.

Revision is a useful generation technique for two broad reasons:

- Provides software engineering benefits
- Provides an accommodating architecture for several text generation techniques that are difficult to implement in a traditional system

From a software engineering standpoint, the revision module produces a redistribution and

reorganization of load that increases modularity, reduces overall system complexity, reduces the complexity of individual modules, and increases modifiability and maintainability. Increased modularity and reduced overall system complexity follow from the task of decomposition that separates the system into two major components, each with two separate modules, so that four conceptually coherent modules are doing what used to be done by two. Because the division results from a genuine task decomposition, the result is a more modular, less complex system. The decomposition also reduces the complexity of the components, and increases their cohesiveness. In the two-stage model without revision, the generation modules must make many decisions that involve reflection on their partially constructed outputs. The revision-based model defers these to the revision module. The generation modules can now produce text without reflecting back on what they have done so far, while the revision modules do not have to worry about how to construct text.

While this decomposition also affects modifiability and maintainability, there is a further gain from the nature of the decomposition. Domain specific linguistic knowledge, which may affect conceptual decisions, can be maintained in and applied to the draft generated text from the revision module. By isolating domain specific knowledge in a single module, a generation system is more easily adaptable to multiple domains.

The second broad advantage of including a revision module is that it provides a particularly welcoming platform for applying several specific techniques that are difficult to implement in a traditional natural language generation architecture. During revision, the text can be analyzed for ambiguity and awkward phrasings. Building this type of analysis into text generators would complicate them unnecessarily. Revision also lets conceptual and stylistic decisions interact. In a traditional system, all conceptual decisions are made prior to lexical choices [Danlos 1987, McKeown & Swartout 1987, Kantrowitz & Bates 1992]. Due to the iterative nature of revision, lexical choices can be made that result in changes at the conceptual level. More details are given in section 3.2.

The key requirements for implementing a revision process in a natural language text generation program are developing techniques to identify deficiencies in the text, developing strategies for improving the text once the deficiencies are detected, relating text representation to its corresponding domain knowledge, and relating both to the system's inference rules. The work reported here addresses these issues while demonstrating that revision is a useful generation technique.

1.3 Knowledge Intensive Generation

Knowledge intensive techniques have been applied to a wide range of problems in the field of artificial intelligence including natural language understanding [Neal & Shapiro 1987] and surface generation [Jacobs 1985]. The current work begins from the thesis that a uniform knowledge base

supporting an intensively knowledge-based generation process provides the only realistic approach for natural language generation systems that incorporate revision because the revision module needs access to knowledge used at each phase of text generation.

A human author, when revising a text, has access to at least the following: (1) the surface text, (2) his intent and goals for the text, (3) grammar rules, (4) discourse strategies, (5) text evaluation principles, and (6) knowledge of the topic domain. In revising, the author draws on all six sources in varying degrees, depending on the revision task at hand. Not all of these knowledge sources must be consulted for any particular revision task, but all the sources are available, and any two or more may be needed at some given time.

A natural language generation system that uses revision to improve text production also requires access to various knowledge sources if it is to perform its task in a realistic and efficient manner. For instance, without access to the underlying knowledge from which text was generated, a revision system would have to parse the surface text and try to determine the underlying goals of the text. A more realistic system would have all the system knowledge at hand.

The revision module contains a knowledge base of text evaluation principles that it uses to locate and correct weaknesses in the text. These principles include both stylistic ideas, e.g., how to use pronouns to make the text more cohesive, and conceptual ideas, e.g., quantifiers should be listed before qualifiers. The rules in the text evaluation knowledge base have antecedents in a variety of knowledge sources including domain knowledge, deep generation knowledge, the surface text, and surface generation knowledge.

A revision module must be able to examine draft surface text and produce suggestions on how to improve it. To perform this task, it must inspect the underlying linguistic and domain knowledge from which it was generated. The underlying structures allow the revisor to locate referents for nouns and pronouns and categorization information for objects. They indicate the intent of the text such as whether the text provides the attribute of an object or an example of how the object is used. The revision module can determine ways that the text can be regenerated by inspecting the choice points, both conceptual and stylistic, in the underlying structures. It is not enough to be able to reason separately about the separate kinds of information: some decision procedures have antecedents that refer to several kinds in one rule. The best environment for this analysis is a uniform knowledge base in which all the system knowledge (domain knowledge, knowledge about discourse strategies, linguistic knowledge, and knowledge about how revision is performed) is represented in a uniform manner and to which a uniform inferencing procedure can be applied. Traditional segregated knowledge base approaches are too inflexible to allow efficient revision techniques to be applied.

The primary problem with a uniform knowledge base is performance. As a knowledge base grows, the computational burden for matching and inferencing over the entire knowledge base increases exponentially. This drawback cannot be simply ignored. Indeed, some researchers have taken it as indicating that unified representation methods cannot succeed [McKeown & Swartout 1987]. But unified representation techniques need not imply blind search or unrestricted pattern matching. The author believes that the answer lies in intelligent access solutions. Three such techniques used in this study to reduce this computational burden are

- dynamic knowledge base partitioning,
- staging of the generation tasks, and
- path-based rather than match-based accessing.

No one phase of the generation process needs access to the complete knowledge base. For example, the surface generator does not need access to the deep generation schema templates, domain knowledge, or knowledge about revision. By partitioning the knowledge base and using only the required knowledge at any phase of generation, the system designer can construct a system with all the benefits of the complete knowledge base but with a reduced computational complexity. Furthermore, each phase of the generation process can be run as a separate stage, i.e., a separate program, with only the required sections of the knowledge base loaded, reducing the demand on system storage. Partial results can be passed between stages using temporary files or interprocess communications constructs.

Some generation tasks need access only to a limited amount of information in a knowledge base. The entire knowledge base can be used, but it is not needed. To reduce computational and memory demands on the system, a conceptually restricted version of the knowledge base, containing the limited knowledge needed by the task at hand, can be used with the same result as using the full knowledge base, but with improved performance. In some cases, the required knowledge can be trivially extracted from the knowledge base. In other cases, inferencing can be used to build the extract.

Path-based methods rely on association links: knowledge base connections between fragments of representation that can be constructed and later traversed with little computational burden. By associating the generated surface text with the underlying knowledge from which it was generated, computationally expensive matching and inferencing can be reduced or eliminated during revision. Underlying knowledge can be located by traversing the association links.

Further details on knowledge intensive techniques are given in Chapter 4.

1.4 Kalos

The Kalos* system explores the techniques of revision and knowledge intensive generation. It is built around a uniform knowledge base including discourse knowledge, linguistic knowledge, knowledge about revision, knowledge about computers, and knowledge about a specific microprocessor. It generates portions of a draft user's guide for the microprocessor. Revision is used to enhance the quality of the text produced.

A typical microprocessor user's guide includes an overview of the microprocessor, a description of the registers and buses, details about exception handling, an instruction set overview, details of addressing modes, and a detailed description of each individual machine instruction. Producing a user's guide is a time consuming task. Partial automation of this task is desirable to reduce labor costs. Since abstract machine descriptions are typically produced for other reasons (e.g., for retargetable compilers [Fraser 1977, Cattell 1980]), at least part of the processor-specific portion of the language generation knowledge base may already be available, thus reducing the cost of using a generation system for a new processor.

Kalos is implemented in SNePS-2.1 [Shapiro 1992], a semantic network package. The approximate sizes of the different knowledge bases in Kalos follow:

- 180 nodes in the domain knowledge base
- 120 nodes representing deep generation knowledge
- 1500 nodes representing surface generation and lexical knowledge
- 75 nodes representing revision knowledge

In addition to these knowledge bases, intermediate results are also represented as semantic networks. The size of these networks vary depending on the other knowledge bases.

1.5 The Thesis

The primary thesis of this paper has two major points:

- A revision architecture provides a beneficial decomposition of tasks for a natural language generation system.
- A knowledge intensive environment with a uniform knowledge base provides a welcoming architecture for a revision system.

There are three ways that the first point can be supported:

- The architecture allows some decision making that would typically be considered by traditional generators to be postponed for consideration by revisors

* Καλόσ is the ancient Greek word meaning *handsome* or *beautiful* with implications of *goodness*. Καλόσ was selected as the name of this system with the hope that this software would be both good and beautiful.

without the interaction between the generators and revisors greatly impacting generator performance.
- The architecture allows application of text generation and polishing techniques that are not available in a system with traditional decomposition.
- Empirical data from the demonstration system must support the claim.

For a revision architecture to be appealing, it must allow the text generators to be simpler than in a traditional system. This implies that some decisions can be postponed to the revisors that would normally be considered by the generators. Furthermore, the interaction between the generators and the revisors cannot be too complex or the benefits of the task decomposition are lost in the overhead of the architecture. To show that a revision architecture is beneficial for a natural language generation system, additional features not supported in a traditional system must be shown to be supported in a revision architecture, and the results of a significant demonstration system must be available.

The second point of the thesis is that a knowledge intensive environment with a uniform knowledge base is a supportive environment for a natural language generation system with revision. There are three ways to support this point:

- The environment allows revision to operate in an intelligent and efficient fashion.
- The computational burden of a uniform knowledge base does not overburden the natural language generation system.
- Empirical data from a significant demonstration system must support the use of this environment.

To show that a knowledge intensive environment with a uniform knowledge base is an accommodating architecture for a revision system, the types of knowledge needed for revision must be listed and it must be shown that the environment allows access to this required knowledge. Furthermore, it must be shown that a large, uniform knowledge base does not defeat its own use by putting an excessive computational burden on the system. Finally, empirical data from a significant demonstration system must support the use of this type of environment.

I will provide support along these lines for the thesis of this dissertation in section 6.5.

1.6 Outline of Following Chapters

Chapter 2 is a survey of natural language generation. It provides a discussion of deep and surface generation research, knowledge-based approaches to natural language generation, and current research into text revision. An introduction to discourse analysis is given as it relates to text revision.

Chapter 3 provides a detailed look at revision. A discussion of why revision is a useful technique is given. The chapter also describes conceptual and stylistic revision and discusses revision stopping

criteria. Finally, revision is cast in terms of rewrite systems, and complexity issues are discussed.

Chapter 4 discusses knowledge-intensive natural language generation. This chapter discusses the need for a uniform knowledge base in a natural language generation system with revision and efficiency issues associated with using a uniform knowledge base.

Chapters 5 and 6 describe the Kalos system in detail. Chapter 5 discusses knowledge representation issues and techniques for dealing with the computational burdens associated with a uniform knowledge base. Chapter 6 describes deep and surface generation and stylistic and conceptual revision processing. Sample output of the system is given. Support of the thesis of section 1.5 is discussed.

Finally, Chapter 7 provides summary and conclusions. Transportability of Kalos to other domains is examined, and future research directions are discussed.

Chapter 2
Natural Language Generation Survey

We first survey the plot, then draw the model.

William Shakespeare, *Henry IV*, Part 2

Natural language generation is the process of constructing prose by a computer program to fulfill a specified goal [McDonald 1988]. Existing natural language generation systems allow expert systems to describe their inferences, provide communications for tutorial and apprentice systems, and give alternate database front-ends for users unfamiliar with database system management languages. As knowledge bases and related systems become more sophisticated, there will be a need for even more sophisticated generation systems.

Some generation systems, such as tutoring systems, interact directly with users. These systems must function in real time to have reasonable interactions with humans. Typically, such systems produce short answers to user questions. Systems that produce more formal text typically have little or no interactive input from human operators. These systems operate with more relaxed time constraints and attempt to produce longer, more polished text. Kalos is a system of the latter type. This dissertation will focus on techniques for the production of polished text.

In Chapter 1 we looked at the organization of a typical, two-component natural language generation system with deep generation and surface generation modules. The deep generation component determines the content and organization of the text, while the surface generation component determines the words and syntactic structure of the text. In other words, the deep generation component determines what to say and the surface generation component determines how to say it. In the current literature on natural language generation, deep generation is also referred to as the conceptual or strategic component of generation, while surface generation is referred to as the stylistic or tactical component. Below, we look at each component in more detail.

2.1 Deep Generation

The deep generation component of a natural language generation system must select which concepts in the system's knowledge base will be included in the text to be generated. It must determine the content of the text being produced by selecting information from the knowledge base that will meet the discourse goal at hand. It must also organize the selected information into a coherent text. In organizing the text, the deep generation component must select a logical order for the text and determine sentence and paragraph boundaries [McKeown 1985, McKeown & Swartout 1987, Mann & Thompson 1987, Hovy 1988].

There are two major approaches to deep generation: the scriptal approach and planning [Hovy 1988]. In the scriptal approach, schemata are used to represent standard patterns of discourse. In planning, an AI planning paradigm is used to form utterances that satisfy some high-level goal. The scriptal approach is discussed in this section because it fits best in the traditional generation architecture. Planning is discussed in section 2.3.

A discourse structure is a pattern of text organization that can be used to accomplish some text production goal. For example, one discourse structure for describing an object is to first describe the object in terms of some taxonomy to which it belongs and then describe each of its components. Both McKeown [1985] in developing the TEXT system and Mann's group [Mann & Thompson 1987] in developing Rhetorical Structure Theory (RST) studied naturally occurring texts to classify text into a limited number of recurring patterns. Although RST is a more comprehensive theory than McKeown's, it is currently only a descriptive theory, while McKeown's is both descriptive and usable in text generation. Rösner and Stede [1992] point out the limitations of RST for use in automatic production of technical manuals in multiple languages and suggest improvements; however, these authors have not yet produced a generation system.

McKeown's TEXT system is a front end to a database containing information about weapons and military vehicles. TEXT accepts questions about items in the database. The user can ask what a database item is, what is known about a database item, and what are the differences between two items. TEXT produces brief answers to these questions.

In developing TEXT, McKeown studied fifty-six paragraphs related to the type of descriptions needed and found that four patterns could describe these paragraphs as long as pattern recursion and embedding were allowed. In TEXT, these patterns are represented as schemata: *attributive*, *identification*, *constituency*, and *compare and contrast*. The attributive schema is used to illustrate a particular point about an object or concept. The identification schema is used to define an object or concept. The constituency schema is used to describe an object or event in terms of its parts. The compare and contrast schema is used to compare an object to another object.

Paris [1985] added an additional schema, the *process* schema, to McKeown's list in developing the TAILOR system which produced descriptions of objects. Based on her study of encyclopedia entries, Paris discovered that descriptions for domain experts could be captured with the constituency schema, while novices required descriptions of how each part of an object functioned. The process schema traces causal relations in the underlying database in order to produce functional descriptions of objects.

A mechanism is required to fill schema slots. Schemata typically contain choice points and optional slots, so the slot filling mechanism must make intelligent decisions about what is relevant

to the topic at hand. McKeown's TEXT fills slots based on available knowledge and focus of attention [McKeown 1985]. TEXT's focus of attention constraints lead it to attempt to shift focus to a newly introduced item first, maintain current focus second, and shift to a previous focus third.

Another aspect of the deep generation process considered by some generation systems is a model of the intended reader. Such systems will attempt to produce different texts depending on the expertise, beliefs, and goals of the intended audience. Systems that use audience models can be divided into two types. In the first, the system obtains some feedback from the audience, e.g., the system user, that allows it to make some inferences about what the user knows or believes. For example, the ROMPER system by McCoy [1985] attempted to correct users' misconceptions about a taxonomy by interacting directly with the user. Systems that interact with the user are ideal for tutoring and computer-aided instruction (CAI) systems where the program, like a human teacher, can determine when the student misunderstands a concept. The system can then attempt to correct the misunderstanding. Before proceeding to a new lesson, the system can use student feedback to determine that learning has occurred.

In the second type of system where written text is being produced, the system has no direct feedback from the audience. One approach in this type of system is to generate text based on a single categorization of the audience given to the system at generation time. The system will then attempt to use vocabulary, sentence structure, and discourse structure appropriate to the intended audience. Cline and Nutter [1990] describe a shallow model of audience expertise in combination with a natural taxonomy that allows a generation system to select appropriate vocabulary. Paris [1985] implemented a system that produces descriptions of physical objects with part details for experts who already understood part functions while producing functional descriptions for novices.

2.2 Surface Generation

In a typical natural language generation system, the surface generation component receives an internal representation of what is to be said from the deep generation component. Based on this input, the surface generator must select and arrange words in conformity to natural language syntax in order to produce a text which conveys the ideas of the internal representation [McKeown & Swartout 1987]. The surface generator employs a lexicon and a grammar. The lexicon contains words which the system can use in expressing a text, and the grammar describes the syntax of the language.

There are a number of grammatical formalisms used in generation. Jacobs [1985] notes that it is difficult to identify strict distinctions between these formalisms due in part to their many similarities. Three important linguistic formalisms, augmented transition networks (ATN), systemic

grammars, and unification grammars, will be briefly discussed.

ATNs encode syntactic information in a directed graph [Woods 1970, Bates 1978]. The graph represents an extended finite-state automaton. The extensions allow actions on arcs, the use of registers for memory, the calling of subgraphs in subroutine call fashion, recursive invocation of subgraphs, and the use of backtracking to find a successful path through the network. ATNs were originally used for parsing natural language text, but several researchers (Simmons and Slocum [1972], Shapiro [1975]) demonstrated that ATNs could be used for text generation. When used for parsing, an ATN is written to examine text and build the corresponding conceptual representation. By examining the input text, the ATN decides which arcs to traverse. The actions on the arcs build a conceptual representation of the input text. When used for generation, an ATN is given a conceptual representation. The ATN tests the conceptual representation to determine which arcs to traverse. The actions on the arcs build the surface text that corresponds to the conceptual representation.

A systemic grammar consists of a hierarchy of "systems" which represent syntactic choices. Each system is a discrimination network. Associated with each system is a "chooser," a procedural function which may query conceptual, grammatical, or lexical information in order to compute one or more grammatical features such as mood or number [Hovy 1988]. Given a concept to express, the choosers query this concept and other features of the system in order to produce a surface text. One distinguishing feature of systemic grammars is that conceptual knowledge is part of the formalism instead of an awkward addition [Jacobs 1985].

Unification grammar (also called functional unification grammar) [Kay 1984], a recent addition to the systemic tradition, is a grammatical formalism using features to constrain linguistic choices. A feature is an attribute-value pair which may represent lexical, syntactic, or conceptual knowledge. A pattern is a special feature which specifies the surface location of each constituent of the text. Unification is used to match the functional description of the concept to be surfaced against the linguistic patterns of the grammar to form the surface text. One advantage of the unification grammar is the uniform representation scheme for various types of linguistic knowledge.

One disadvantage of all three of these syntactic formalisms is that they are typically represented in specialized knowledge structures that are not accessible to the primary inference mechanism of the generation system. Such "hidden" knowledge limits the ability of the deep and surface generation components to interact. The need for this type of interaction will be discussed in the chapter on revision.

A similar problem exists with lexicons used for generation [Nutter 1989]. Typically, lexicons are contained in specialized structures separate from the primary inference mechanism of the system.

Furthermore, lexicons contain essentially syntactic information. If semantic information is present at all it is contained in a different knowledge structure involving specialized access techniques. In the PENMAN system [Mathiessen 1981], semantic and syntactic structures of the lexicon are linked using concepts in the knowledge base. For example, this approach allows the system to determine that the semantic role of *Agent* can function as the syntactic role *Actor* if the semantic concept *SELL* were surfaced using the verb "to sell." Although this lexicon appears to overcome some of the limitations of purely syntactic lexicons, the approach is similar in function to using feature markers in a traditional lexicon.

The KING (Knowledge INtensive Generator) [Jacobs 1985] system is the one notable exception to the problem of implementing syntactic knowledge in specialized structures. KING uses a uniform representation for both conceptual and linguistic knowledge. This system is discussed in section 2.4.

2.3 The Use of Planning for Generation

Not all generation systems use schemata for deep generation, or grammars, such as ATNs or unification grammars, for surface generation. Indeed, not all generation systems are segregated into these two distinct components. Planning is a major approach to text generation.

The work of Appelt [1983, 1985] best illustrates the use of planning for generation. Appelt used a uniform planning mechanism at all levels of his system. Appelt's system also used procedures called "critics" to control interaction of plans at different levels of the generation process. The critics make decisions about content and wording. For example, a critic may combine plans from two levels that would normally be output as two sentences into a compound sentence.

Appelt's system contains a formal representation of the beliefs of the hearer and the speaker based on Moore's possible-worlds semantics. By reasoning about these beliefs, the system can avoid stating information that is already known by the hearer. Appelt's KAMP system, when planning to instruct a worker to remove a pump from a platform, can avoid referencing the platform if it believes that the worker already knows the pump is on the platform.

Hovy [1988] extended the planning formalism for text generation to include pragmatic concerns. Hovy's system PAULINE attempts to model human text generation in a social setting by acting on "interpersonal goals," characteristics of the hearer, and the setting of the conversation. For example, a human speaker will produce widely different utterances on a particular topic in the context of his own home than at a formal dinner party. Hovy's argument is that text generation software must take pragmatic concerns into consideration to produce appropriate text with varying goals and situations. He adds the question "why should I say it" to the traditional generation

problems of "what to say" and "how to say it."

McKeown and Swartout [1987] note that the planning formalism is unnecessarily complex for the domain Appelt addressed and that in more complex domains, a model of the speaker and hearer beliefs may be too complex for the planning system. Appelt recognized the complexity of combining deep and surface generation in a single planning model. In later versions of his KAMP system [Appelt 1985], he implemented a unification grammar using a planning mechanism that runs as a back-end to the deep generation planning mechanism.

2.4 Knowledge-Based Approaches

The use of knowledge-based techniques in text generation varies among systems. Two systems incorporating knowledge-based techniques are worth noting. The first is the ANA system [Kukich 1983], which reads input from a stock market database and produces a report summarizing the day's stock market activity. The system consists of four modules: the fact generator, the message generator, the discourse organizer, and the text generator. The first module is written in the C programming language, while the remaining modules are written in OPS-5. The system performs generation at the phrase level.

An important observation of Kukich is that it is useful to integrate a variety of knowledge types, including semantic and linguistic knowledge, using the same formalism. Kukich noted that although she attempted to separate semantic and discourse organizing knowledge, this separation was sometimes undesirable. Although ANA uses integrated knowledge in each of the OPS-5 modules, the knowledge for a particular module is isolated to that module. Kukick justifies this approach based on concerns for computational manageability, but she notes that a more valid psychological model of the report writing process would require a more closely coupled system, with feedback and interaction between modules.

The second noteworthy knowledge-based generation system is Jacobs's KING, a complete surface generation component [Jacobs 1985]. Jacobs uses a system for knowledge representation called Ace. Ace allows both conceptual and linguistic knowledge to be represented in the same framework. Both types of knowledge are represented hierarchically, and the two types of knowledge are linked together using arcs called structured associations.

KING uses a variation of a unification grammar represented using Ace primitives. However, in KING, the need for intensive unification is replaced by a simpler mechanism using structured associations. By following the structured association arcs, the system is able to link semantic roles with syntactic roles.

But KING only addresses the surface generation aspect of natural language generation. Al-

though KING and Kalos both share the use of knowledge intensive techniques, the uniform knowledge base used by Kalos must deal with all the knowledge used by the generation process, not just surface generation knowledge. Techniques are needed in Kalos to deal with the computational explosion caused by the uniform knowledge base covering all aspects of generation.

2.5 Revision for Natural Language Generation Systems

Chapter 3 discusses revision in detail. This section outlines other research in this area. Revision is an iterative process whereby text generated by deep and surface generators is analyzed for defects. The result of this analysis is passed back to the generators, which regenerate the text using the suggestions made by the revision module, or revisor. (See Figure 1.2 in Chapter 1.)

Several researchers in natural language generation have suggested a similar model for computer generated text. Vaughan and McDonald [1986] have suggested three phases of the revision process: revision, editing, and regeneration. In the revision process, the potential problems with the draft text are determined. In the editing process, a determination is made for which revision strategies are appropriate and which, if any, to employ. Finally, in the regeneration phase, the selected revision strategy is applied to the generation process at the appropriate point.

Vaughan and McDonald limit revision effects to the surface generation level only, on the grounds that only stylistic decisions are free of fixed constraints and that conceptual changes would change the meaning of the text. But other researchers have demonstrated that lexical and syntactic choices do affect conceptual choices [Appelt 1985, Danlos 1987, Hovy 1988]. For example, the amount of memory a microprocessor can address can be stated either in terms of the address bus size or the size of the processor's "address space." The former description expresses an attribute of the address bus, while the latter description expresses a direct attribute of the processor. The choice of the term "address space" affects a conceptual notion: whether the processor or one of its constituents is being described [Cline 1991]. Furthermore, Yazdani [1987] argues that revising should include both stylistic editing and high-level restructuring. For high-level restructuring, he sees revision as an independent process. He argues that a sophisticated system which cannot rebuild a mental model of its own creation is inadequate. A revisor should be able to compare the intentions inferred from the text with those used during generation. Discrepancies in these two sets of intentions indicates the need for revision. Yazdani argues that creative writers often start with a small piece of text representing a story idea and build a whole story around it. He sees this type of revision as the driving force of the creative process as opposed to a way of just improving text.

Meeter [1991] analyzed revisions made by professional editors with the goal of producing an

architecture for text revision. The revisions were all stylistic in nature, as Meeter wanted to avoid revision to content. By selecting professional editors and competent writers, she avoided dealing with conceptual revision. A broader model that considered conceptual revision would produce a more robust and psychologically satisfying architecture; however, Meeter's analysis of stylistic revision is still an important work.

Based on her analysis, Meeter proposes an efficiency model of generation. Consider the revision of the phrase "in a way that is independent" to "independently." It is assumed that a competent writer would know the word "independently" and would tend to use it over the longer phrase because of his desire to be concise. Meeter believes that in a familiar situation, such as answering the telephone, we have specific strategies we use to generate text. In an unfamiliar situation where we have no specific strategy, more general strategies must be used to generate text. These strategies combine other simpler strategies. In an unfamiliar situation, we typically do not backtrack, so we miss the opportunity to use optimal phrasing. Meeter's analysis leads her to conclude that a revision architecture that "favors efficiency and expressibility over initial optimality of expression" is a useful approach to natural language generation.

Although revision-based systems have been proposed for some time, only limited systems have been produced so far. The most advanced systems incorporating revision are the Yh system [Gabriel 1988], the weiveR system [Inui, Tokunaga, & Tanaka 1992] and the STREAK system [Robin 1993]. Yh uses planning to produce simple descriptions of computer algorithms. It produces draft text and then uses stylistic revision to improve the text. For example, one basis for revision in Yh is the rule "do not use too many adjectives." Yh demonstrates that stylistic revision is a useful technique; however, only a single, simple example is presented in Gabriel [1988].

The weiveR system is limited to stylistic revision of Japanese text. It focuses on repairing structural ambiguity and sentence complexity problems such as those associated with sentence length and depth of embedding. Although weiveR and Kalos were developed independently, both use a similar directed backtracking technique to modify surface text based on decisions made by the revisor. weiveR currently does not perform deep generation, but the authors of that system feel that revision should be more broadly applied, as in Kalos, because it has a psychological basis, allows interaction between deep and surface revision, and makes the implementation of a natural language generation system more tractable due to problem decomposition.

The STREAK system (Surface Text Reviser to Express Additional Knowledge) uses revision in a different manner than Kalos. STREAK generates draft text from a wire report, such as the report of a basketball game. The text is revised to add historical information. Historical information is required to allow the reader to understand the significance of the current report.

For example, saying that a particular basketball player scored only eight points in a game tells only part of the story. Adding the historical data that the player had been sidelined for seventeen games because of an injury casts his eight points in a different light. Robin argues that a traditional pipelined architecture is inappropriate for handling the addition of historical knowledge because of computational considerations and the need to add historical information in the context of both conceptual and stylistic knowledge. This latter point is based on a study of human-generated text where historical knowledge typically appeared at the phrasal or word level.

STREAK is interesting in that it considers both conceptual and stylistic concerns when making revisions. Also of interest is that the addition of information is performed at the micro (or word) level, allowing the addition of clauses and even single words.

STREAK is incomplete in several ways. First, the only type of revision it performs is the addition of information at the conceptual level. Second, the system reported by Robin [1993] is limited. It appears only to generate and revise a single sentence and portions of the system are incomplete. However, Robin's work illustrates the usefulness of a revision architecture.

2.6 Discourse Analysis

Work in the area of discourse analysis is related to the study of natural language generation with revision. Discourse analysis research can be divided into two areas:

- Text cohesion
- Deep structure of text

Theories of discourse analysis that deal with surface linguistic evidence for text cohesion are relevant to surface generation and stylistic revision. Discourse analysis theories that deal with the deep structure of text are relevant to natural language generation systems that produce multi-sentence text. However, much of discourse analysis deals with conversational situations that are not directly relevant to production of formal written texts.

A large body of research exists on the idea of coherence in text [Hobbs 1982]. One source of coherence is cohesion [Halliday & Hasan 1976]. Cohesion includes the use of anaphora and cataphora, use of words such as 'one', 'do', and 'so' as substitutions for nominal, verbal, and clausal phrases, use of conjunction, and use of ellipsis to make the text more readable. Kalos uses the ideas of cohesion in the production of anaphoric references and in the use of conjunctions to combine parallel constructs. It can identify appropriate cases to use cohesive constructs by reasoning about surface strings and the underlying concepts from which they were generated. The use of a full knowledge system gives Kalos access to both conceptual knowledge and surface strings and the relationship between the two.

Another source of coherence comes from the given-new contract, an implicit agreement between the writer and reader (or speaker and hearer) that requires the writer to "use given information to refer to information she thinks the listener can uniquely identify from what he already knows and to use the new information to refer to information she believes to be true but is not already known to the listener" [Clark & Clark 1977]. By writing text so that the given and new information is easily identified by the reader, the writer produces text that is more understandable. One technique for producing proper given-new information in sentences is thematic progression [Glatt 1982]. In thematic progression, given information should precede new information within sentences. Thematic progression also includes ideas about patterns of sentences that make them more coherent.

Four terms are used in describing thematic progression. *Theme* is the subject of the sentence (what is being talked about). *Rheme* is what is being said about the theme. *Given* information "refers to information that can be recovered from the previous linguistic or extralinguistic context" [Glatt 1982]. *New* information is information not recoverable from previous text. There are two primary thematic progression sentence patterns. In the first, the theme of both sentences is the same. In the second, the rheme of the first sentence is the theme of the second sentence. According to Glatt's study, it appears that sentences following the given-new contract in thematic progression order are easier to understand than other configurations.

Consider two sentences that are not in thematic progression order:

```
The SR contains condition codes and the interrupt mask.
The MOVESR instruction affects the SR register.
```

By placing the second sentence in passive voice, thematic progression can be obtained. After anaphora processing, the sentences would be

```
The SR contains condition codes and the interrupt mask.
It is affected by the MOVESR instruction.
```

A revision system can use these cohesion ideas as a basis for producing stylistic revisions. Further background on revision is discussed in Chapter 3.

Other theories in discourse analysis attempt to deal with the deep structure of discourse. Some of these divide discourse into a fixed set of rhetorical patterns (e.g., Mann and Thompson [1987], Grosz [1979]). Although the use of rhetorical patterns can explain surprising linguistic features [Reichman-Adar 1984], researchers working with different genres have developed different sets of patterns.

Other researchers account for mutual understanding of speaker and listener by appealing to "world knowledge." World knowledge is our general, nonlinguistic knowledge. As we participate in discourse, we use this general knowledge as we attempt to understand utterances. For example, if we see a sign that says "two hamburgers for the price of one," on the side of a building, we infer that there is an establishment in the building that sells hamburgers and other food. We use our general knowledge about fast food restaurants and about advertising to understand a simple phrase.

The world knowledge view of discourse understanding is based on the idea of speech acts. In this view, the ability of two conversants to understand a dialogue is based on their understanding of each others' intentions. The mutual understanding of intentions is the basis for understanding discourse meaning and flow [Reichman-Adar 1984].

One of the most satisfying theories of discourse analysis that is relevant to natural language generation was developed by Grosz and Sidner [1986]. This theory combines ideas related to linguistic structure, attention, and intentions. The linguistic structure is the actual sayings or writings of the text being considered. The structure is decomposed into segments that fulfill certain functions with respect to the entire discourse. Intentions refer to the purpose for which the utterance was produced. In other words, the goal of the speaker is useful in analyzing a discourse. Attention pertains to the objects, properties, and relations that are salient at any instance.

In this theory, for a discourse to be coherent, the hearer or reader must be able to recognize both the main discourse intention and intentions for each of the discourse segments. The hearer must also be able to recognize the relationships between the intentions, such as one intention supporting another. It is clear from this theory that for a text to be coherent, the purpose of the discourse must be shared between the speaker and hearer, and each utterance must contribute to this purpose.

Rhetorical relationships, such as McKeown's *amplification*, are less important to this theory than those based only on rhetorical patterns. In particular, the writer or speaker may have rhetorical relationships in mind when the text is generated; however, the hearer or reader may still understand the text without constructing or naming the rhetorical relationship.

The outcome of research in discourse analysis is applicable to the design of deep generators in natural language generation systems. The work in fixed rhetorical patterns (e.g., RST), text planning (e.g., Appelt's work), pragmatics [Hovy 1988], and focus [McKeown 1985] is based on research in discourse analysis. Discourse analysis research related to cohesion is useful in the design of stylistic revision techniques.

Chapter 3
Revision in Natural Language Generation Systems

> The Lord will be there and wait till I come on perfect terms . . .
>
> Walt Whitman, *Song of Myself*

The research reported here investigates two main techniques for enhancing connected text generation: revision and knowledge intensive generation. The former will be explored in this chapter, while the latter will be described in the next chapter.

3.1 Revision by Human Authors

Psychological researchers Hayes and Flower [1980] have developed a model of the human writing process. They suggest that writers cannot juggle the large number of constraints and demands they face while producing formal text [Flower & Hayes 1980]. To reduce cognitive strain, they suggest that writers produce a draft document and then polish it through revision. The Hayes and Flower model is divided structurally into three parts: The writer's long-term memory, i.e., his conceptual knowledge; the task environment, which contains the writing assignment and the text produced so far; and the writing process. See Figure 3.1.

Hayes and Flower divide the writing process into three major processes: Planning, Translating, and Reviewing. The planning process takes information about the goal of the desired text from the task environment and uses conceptual knowledge in the writer's long-term memory to develop a writing plan to meet the goal of the text. This major process contains three subprocesses: Generating, Organizing, and Goal-setting.

The translating process converts conceptual knowledge from the writer's long-term memory into surface language under the guidance of the plan produced by the planning process. The final process, reviewing, improves the quality of the text produced by the translating process. The reviewing process consists of subprocesses for reading and editing.

During the reading phase, the text produced by the translating process is examined, and a mental model of the text is recreated [Yazdani 1987]. Some text defects can be determined from the underlying concepts, but defects such as ambiguity and problems with the length of the text cannot be located without actually reading the text. The editing process detects problems with the text being read and makes corrections to improve its quality. Making corrections involves a variety of techniques, including deleting extraneous material, shortening lengthy paragraphs, using connectives, and changing sentence voice [Collins & Gentner 1980].

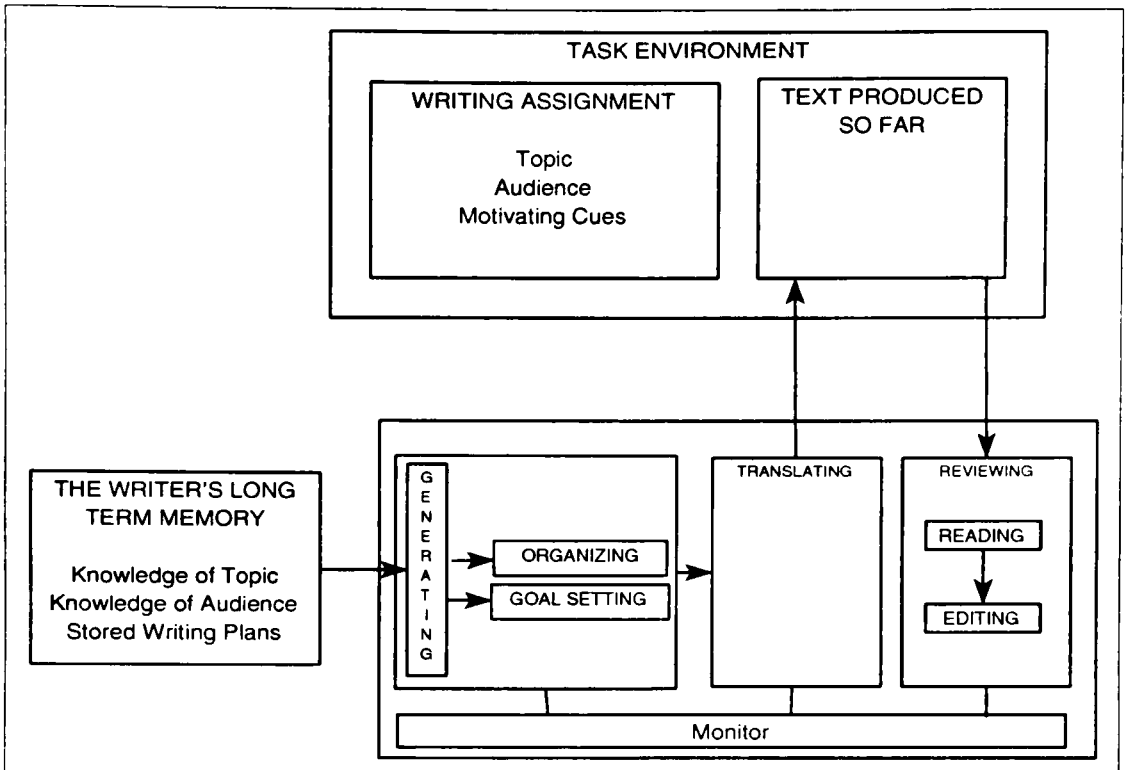


Figure 3.1 - Hayes and Flower Writing Model (Adapted from Hayes and Flower [1980])

In the Hayes and Flower model, invocations of planning, translating, and reviewing are not necessarily sequential. One process may interrupt another and may call the writing process recursively. For example, a writer may produce a sentence and immediately enter the reviewing process to polish it. The reviewing process may then invoke the translating process to rewrite the sentence. The reviewing process may even determine that additional explanation is required for a particular passage and so may invoke the planning process to include more conceptual knowledge into the writing plan.

3.2 Why Revision?

Revision is important for a natural language generation system for three reasons:

- It reduces natural language generation system complexity through task decomposition and modularity.
- It provides an architecture for dealing with infelicitous, verbose, and ambiguous text.
- It allows interaction between conceptual and stylistic decisions.

As noted in Chapter 1, natural language generation is a formidable task. Techniques that reduce the complexity of generation systems are beneficial. As noted above, revision reduces human cognitive strain by postponing some decisions while concentrating on others. The complexity of a generation system can be reduced similarly. Using a revision architecture makes generation modules simpler in design by postponing to the revision module decisions typically made in the generation modules. As Yazdani [1987] pointed out, this type of architecture is common in the construction of complex software systems such as compilers. To simplify their design, compilers are typically constructed as a pipeline of modules. This architecture lets the designer concentrate on some issues (e.g., lexical analysis) while postponing other processing (e.g., syntactic analysis) until later. As revision reduces cognitive strain on the human writer, a revision-based natural language generation architecture reduces cognitive strain on the system designer.

One way to view the increased efficiency of the generation modules in a revision system is from the point of view of decision making locality. In a traditional system, when the deep generator is considering how to fill a schema slot, it must consider previous decisions that affect the current slot, and it must look ahead to any interactions that filling the current slot will have on future decisions. For example, if the deep generator is describing a microprocessor and indicates that the microprocessor is a "16-bit microprocessor," it probably should not state in the same paragraph that the data bus size is sixteen bits, as this is redundant. But the term "16-bit microprocessor" is related to a schema slot that gives a direct attribute of the microprocessor, while the data bus size is related to a schema slot related to the data bus. A similar situation occurs in the surface generator. For example, to handle anaphora correctly, the surface generator may have to track focus of attention over a number of sentences.

In a traditional system, the locality of decision making is wide. But in a revision system, the locality for generation modules can be kept narrow. The deep generator need only be concerned with the particular slot being filled, and the surface generator can focus on single sentences. All decisions requiring wide decision making locality are postponed for consideration by the revision module. By postponing wide locality decision making to the revision module, the generation modules can be made simpler.

There is another software engineering benefit to incorporating a revision module into a natural language generation system. The revision module is a natural place to isolate domain specific linguistic knowledge and knowledge that relates to both surface and deep generation modules, thus producing a more maintainable generation system. For example, the application of domain-specific preferred terms can affect both conceptual and stylistic decisions. By isolating preferred term processing in the revision component, the generation component is more easily adapted to other

domains. Furthermore, by isolating the linguistic knowledge needed to make conceptual decisions in the revisors, the deep generator is less complex than if it considered linguistic knowledge directly.

In Chapter 1, it was stated that a generation system must avoid generating infelicitous, verbose, and ambiguous text. A revision component provides an appropriate architecture for dealing with these problems. For example, it is the ideal place to identify and eliminate ambiguities in the generated text. A text may be ambiguous either because of multiple meanings of the words used at the surface level, or more subtly, because of their interrelationships with other words in the text. Generation systems cannot simply avoid words with multiple meanings; too few words have only one. What must be avoided are texts which are construable –make sense– in more than one way. While it is conceivable that an initial generation module could contain measures to avoid ever generating ambiguous text, the complexities involved are overwhelming. By contrast, once the text has been generated, reading it to locate ambiguities is a far less demanding task. Hence it makes more sense to locate and eliminate ambiguities in a separate revision module, which has access to the surface text as well as to information about its origin.

Some awkwardness related to word sound, such as rhyming, is best dealt with by a revision module. For example, consider a knowledge base with two concepts *register* and *data register* that are related as superclass and subclass. A natural language generation system might produce, "The D0 data register is a register" (analogous in structure to "The F-150 pick-up is a light-duty truck"). The sentence is awkward because of the repeated use of the word "register." The compound term "data register" includes the fact that a *data register* is a subclass of *register*. Such surface-level problems could be dealt with during surface generation, but doing so would greatly complicate the generator. A revision module is an ideal place to deal with word sound problems.

A revision architecture addresses the problem of the lack of interaction between conceptual and surface decisions in a traditional generation system ([Rubinoff 1992] and [McKeown & Swartout 1987]). Traditional systems make conceptual decisions first and then generate surface text. This architecture does not allow lexical choices to influence conceptual decisions [Danlos 1987]. A revision architecture allows conceptual decisions to be changed in favor of the use of a particular word or phrase. The example given below shows how the use of the term *address space* affects the organization of the text and illustrates how a lexical choice can require a conceptual change. Kantrowitz and Bates [1992] describe this type of interaction between deep and surface generators as *interleaved*.

Consider as an example a description of the address bus of the Zilog Z-80 microprocessor:

- The address bus of the Zilog Z-80 microprocessor is sixteen bits wide.

Another way of stating this fact relates to the address space size of the microprocessor:

- The Zilog Z-80 has a sixty-four kilobyte address space.

The first sentence relates an attribute of the address bus, while the second sentence makes a statement directly about the processor. The second sentence both uses a preferred way of describing the processor's maximum memory size and gives an important feature of the microprocessor. It is thus desirable to include it in an overview paragraph of the microprocessor rather than in a following paragraph describing its buses.

3.3 Types of Revision

Revision, whether performed by a human author or a computer system, takes two forms. *Stylistic revision* occurs when the surface text is changed without altering the meaning of the text or the order of concepts. *Conceptual revision* occurs when the meaning of the text or the order of concepts changes. Replacing a noun phrase with a pronoun and compounding sentences are examples of stylistic revisions. Reordering, adding, or deleting text results in a conceptual revision. Examples of conceptual revisions are adding an example to existing text and reordering attributes of an object being described so that quantifiers are given first.

An example of a conceptual revision was given in the previous section. Consider the following example of stylistic revision. Sentences (1) and (2) are draft text:

- D0 is a register. (1)
- D0 is 32-bits wide. (2)

A simple stylistic revision is to render sentence (1) as the compound noun "the D0 register" and to use it as the subject of sentence (2), producing

- The D0 register is 32-bits wide.

The majority of research has concentrated on stylistic revisions. Vaughan & McDonald [1986] even suggested that only stylistic revisions be performed by a revision module. Yazdani [1987] suggested that conceptual revisions were worth studying. Cline [1991] suggested some pragmatic conceptual revisions. This chapter discusses both types of revision.

It is useful to look at the knowledge requirements for each type of revision. Consider the following sets:

- $Domain_C$: *Conceptual domain knowledge*
- $Domain_L$: *Linguistic domain knowledge*
- *Discourse* : *Discourse structure knowledge*
- *Linguistic* : *General linguistic knowledge*
- *Surface* : *Draft surface text*

$Domain_C$ is the set of conceptual knowledge that the system has about the domain at hand. For example, a natural language system that produces a description of a microprocessor would contains facts about the specific microprocessor and general facts about computers. $Domain_L$ is domain-specific linguistic knowledge, such as which words and phrases are preferred in the domain at hand. For example, in the area of computers, the term "address space" is typically used to express the amount of memory a processor can directly address. *Discourse* is the set of knowledge about discourse strategies, i.e., knowledge about how to select and order concepts to meet some discourse goal. In a traditional schema-based natural language generation system, this set includes facts about schemata and how to fill schema slots. The *Linguistic* set contains knowledge about words and about constructing surface text. This set includes knowledge about grammar rules. *Surface* is the set of surface text produced by the system. Revisors must take surface text into account to detect ambiguities and word awkwardness.

Conceptual revision examines

$$Discourse \cup Domain_C \cup Linguistic \cup Surface \cup Domain_L$$

while stylistic revision examines

$$Linguistic \cup Surface \cup Domain_L.$$

Notice that the conceptual revisor requires access to all the system knowledge. The primary knowledge sources for the conceptual revisor, i.e., the knowledge sources in which most revision rule antecedents are targeted, are the discourse knowledge base, the surface text, and the conceptual domain-specific knowledge base. However, revisor processing for preferred terms and lexical redundancy requires access to two additional knowledge bases: general linguistic knowledge and domain-specific preferred term knowledge.

The domain-specific preferred term knowledge base is small and contains knowledge frames to identify preferred words and phrases for the domain and to relate these to the domain knowledge base object types to which they should be applied. For example, the term "16-bit microprocessor" is a preferred term in the domain-specific linguistic knowledge base, and it should be used when describing a microprocessor with a 16-bit data bus.

The general linguistic knowledge base is used by the conceptual revisor to find ways that preferred terms can be applied. Given a preferred term, the conceptual revisor looks for grammar rules that can be used to produce the preferred term. If it finds a grammar rule that can result in the preferred word or phrase, it inspects the grammar rule to find the domain knowledge frame that will trigger the rule. See section 6.3.2 for details of this process in Kalos.

Only a small portion of the general linguistic knowledge base is needed by the conceptual revisor. First, the conceptual revisor needs a mapping between preferred terms that appear in the domain-specific linguistic knowledge base to the conceptual knowledge base frame types that will cause their generation. (See section 6.3.2.) Second, the conceptual revisor must also have knowledge of lexical terms that can cause redundancy, e.g., "The M68000 address register is an address register." (See section 6.3.1.) By extracting only the knowledge required to build the preferred term mapping and information on lexically-based redundancy, we can build a conceptually restricted version of the general linguistic knowledge base for use by the conceptual revisor that is much smaller than the full linguistic knowledge base. Although the conceptual revisor requires access to a large part of the system knowledge, the size used in practice can be limited. (See section 4.3 for more information on conceptually restricted knowledge bases.)

Conceptual revision suggestions must identify a choice point in the deep generation process and suggest a different choice. The choice point relates to a decision the deep generator makes when attempting to fulfill some discourse goal. For example, in a schema-slot filler, it is necessary to identify the slot being filled and the choice that resulted in the suboptimal text. Therefore, a conceptual revision suggestion is a 3-tuple:

$$\langle slot, choice\ point, new\ choice \rangle$$

Slot refers to an element of an instantiated schema, *choice point* indicates the deep generation state in which the defective text was produced, and *new choice* is the alternative selection that should be made to improve the text.

To apply a conceptual revision suggestion, the deep generator must determine that it is in the state where *slot* is being filled and *choice point* is under consideration. In this state, if *new choice* is a possible decision, it will be selected. A conceptual suggestion for a planning system would be similar.

Stylistic revision suggestions are also 3-tuples:

$$\langle slot, choice\ point, new\ choice \rangle$$

The surface generator determines when a given schema *slot* is being surfaced and *choice point* is reached in the surface grammar. At this point, it attempts to make the *new choice* decision.

Revision suggestions cannot always be applied. This is especially true in a system that produces conflicting revision suggestions. For example, consider sentences (1) and (2) above. It is possible to suggest that the subject of (2) be rendered as a pronoun and that (1) and (2) be combined into

- The D0 register is a 32 bits wide.

Both revision suggestions cannot be applied because if the sentences are combined, it may not be proper to render "D0" as "it." One way to deal with this problem is to have the revision modules remove conflicting revision suggestions before passing the suggestions to the generator modules. The scheme used by Kalos is to weight the structural impact each revision will produce and select the one that will have the greater effect. If two revision suggestions have the same weight, then one is selected nondeterministically.

The tuples given for revision suggestions contain the knowledge necessary to identify the suggestion and how it should be applied. In a practical system, suggestions may not take exactly this form. For example, a Kalos stylistic revision contains the *slot* being surfaced and a list of attribute-value pairs to be placed in the binding list of the surface generator when the *slot* is surfaced. The attribute-value pairs identify choice points where different decisions are to be made and identify the new choice to be made by causing backtracking in the surface grammar, i.e., they contain both the *choice point* and *new choice* information in one element of a suggestion.

3.4 Revision Stopping Conditions

Revision is an iterative process, so care must be taken to avoid an infinite loop in the generation process. Human authors typically stop when they believe that the text is good enough or when time requirements dictate that they stop polishing the text. In a natural language generation system, both of these measures can be applied.

How can a generation system determine that the text is good enough? The answer depends on the system design. In a simple system like Kalos, revisions are never retracted, and revision suggestions are applied in an order based on the amount of structural change that they make to the text. The application of some revision suggestions excludes the application of some others. Rules limiting sentence complexity further limit the number of applicable revision suggestions as the generation process progresses. In this type of system, the text is considered polished when no more revisions can be applied. In other words, the system designer decides *a priori* that the application of all possible revisions, using the conflict resolution rules of the system, results in polished text.

In a more robust system, revisions can be applied and later retracted. In this type of system, some stopping criterion is required to determine when the text is sufficiently polished. One possibility is to use the type of tests typically used in writing toolkits [Kieras 1989] to determine when a text is adequately polished. Since these measures typically have no direct knowledge of the meaning of the text but rely on secondary measures such as sentence length, they provide an inaccurate measure of text polish. As natural language understanding techniques improve, better

techniques for determining goodness of text will be available.

The system can also determine when to stop revising text by a direct or indirect measure of time. Using a direct measure, the natural language generation system continues to revise the text it is producing until no more revisions are possible or until a preset time expires. An indirect time limitation stops text production when either no more revisions are applicable or after a preset number of revision iterations. These techniques are best used during the experimentation phase when a natural language generation system is being developed and as an aid to prevent looping when an adequate degree of polish is never reached.

3.5 Revision and Rewrite Systems

Text revision can be viewed as a term rewrite system. The typical goal of rewrite systems is to reach a normal form [Dershowitz & Jouannaud 1990]. This is not the case in a revision system, as many "good" texts can be produced that convey the same information. The actual "goodness" of polished texts is in many ways subjective. However, the expressiveness of rewrite systems can be used to elucidate the process of revision.

Dershowitz and Jouannaud [1990] define rewrite systems informally as follows: "Rewrite systems are directed equations used to compute by repeatedly replacing subterms of a given formula with equal terms until the simplest form possible is obtained." Rewrite systems have the power of Turing machines, and can be thought of as nondeterministic Markov algorithms over terms instead of strings. They have uses in decision procedures (two terms are determined to be equal if both reduce to the same normal form), interpreters for equation systems, and in place of resolution for theorem provers.

Two types of revision systems will be considered. In the first type, revision rule application is nondeterministic. In the second type, some rule ordering principle is used. These two revision system types are analogous to term rewrite systems and priority rewrite systems [Baeten, Bergstra, & Klop 1987].

Since rewrite systems have the power of Turing machines, they have the power to capture the computational power of any computer-based revision system. We will base our discussion on revision systems as discussed in this dissertation. We first define set T that defines natural language sentences and the underlying knowledge from which they were generated:

$$T = \{[x, y, z] | x \epsilon C^+, y \epsilon G, z \epsilon S^+\}$$

where x is the conceptual knowledge from which the sentence was generated, y represents underlying knowledge about the sentence, and z is the surface text. C is the set of conceptual knowledge from

which sentences can be generated, G contains underlying information about sentences such as the mapping between sentence subjects and the objects they represent, and S is the set of words in the surface language. So the term $[x, y, z]$ represents a single natural language sentence. A sequence of n sentences,

$$t = \langle [x_1, y_1, z_1], [x_2, y_2, z_2], ... [x_n, y_n, z_n] \rangle$$

represents the output of a generation system.

We now define a term rewrite system which can be applied to t to revise the sentences. Define term rewrite system $(T, \Rightarrow)$ where $\Rightarrow$ is a binary relation on T [Jantzen, 1988]. The rewrite rules in $\Rightarrow$ are usually written in the form

$$l \rightarrow r,$$

indicating that the term l can be replaced with term r. The order of application of rules is nondeterministic.

By developing adequate definitions for sets C, G, and S and for $\Rightarrow$, the resulting term rewrite system, $(T, \Rightarrow)$, can be made to capture Kalos-style stylistic and conceptual revision. It is beyond the scope of this discussion to cast all of Kalos in terms of rewrite rules; however, to illustrate the correspondence between revision and term rewrite systems, we will consider a limited example.

Consider the following definitions for sets C, G, and S.

$$C = \{\langle mc, x, y\rangle, \langle size, x, y\rangle, \langle operand-size, x, y\rangle | x, y \epsilon O\}$$

$$O = \{D0, 32-bits, data-register, 1-8-16-32-bits\}$$

$$G = \{(m1, m2, m3, m4) | m1 \epsilon \{D0, +\}, m2 \epsilon \{data-register, 32-bits, 1-8-16-32-bits, +\},$$

$$m3 = \{decl, pred-nom, compound, +\}, m4 = \{na, qualifier, quantifier, +\}\}$$

$$S = \{D0, is, a, data-register, and, 32, bits, wide, can, handle, operands, 1, 8, 16\}$$

The elements of C capture the conceptual frames of class membership, size, and operand-size capabilities of microcomputer data registers. $\langle mc, x, y\rangle$ indicates that x is a member of class y. $\langle size, x, y\rangle$ indicates that register x is of size y. $\langle operand-size, x, y\rangle$ indicates that register x can handle operands of size y. x and y are members of set O, the objects we will consider. When two sentences are revised, their conceptual elements are concatenated.

Elements of set G capture both limited linguistic facts about sentences and functional relationships between objects in O and their usage in the sentences. For this example, elements of G record the subject of the sentence (*m1*), the object or predicate nominative of the sentence (*m2*), the type of sentence (*m3*), and if the sentence gives a qualifying attribute, quantifying attribute,

or no attribute about the subject ($m4$). When two sentences are revised, their linguistic elements can be combined with the + symbol.

The surface sentences are strings from S^+. These are represented as strings in the examples given below.

Variables x,y,z,f_i, and c_i are used.

We now give two rewrite rules:

$$r1: [c1, \langle f1, f2, f3, qualifier\rangle, x][c2, \langle f1, f4, f5, quantifier\rangle, y] \rightarrow$$
$$[c2, \langle f1, f4, f5, quantifier\rangle, y][c1, \langle f1, f2, f3, qualifier\rangle, x]$$

$$r2: [c1, \langle f1, f2, f3, f4\rangle, x\ y][c2, \langle f1, f5, f6, f7\rangle, x\ z] \rightarrow$$
$$[c1\ c2, \langle f1, f2 + f5, compound, na\rangle, x\ y\ and\ z]$$

Rule r1 is a conceptual revision rule that says that if a sentence gives a quality about some subject *f1* is followed by a sentence that gives a quantity related to the same subject, then the sentences should be reversed. Rule r2 is a stylistic revision rule that says that two sentences with the same subject can be rewritten into a compound sentence. Note that when sentences are combined, information about the underlying sentence types (*f3* and *f6* in rule r2) and attribute type (*f4* and *f7* in rule r2) is lost. In the system given here, this simplification does not cause a problem because once a sentence is compounded, no more revisions are applied to it. In a more complex example, information about the underlying sentences from which a compound sentence was produced would have to be saved.

Now consider a sequence of terms representing three sentences to be revised:

$$t = [\langle mc, D0, data-register\rangle, \langle D0, data-register, pred-nom, na\rangle,$$
$$\text{“}D0\ is\ a\ data\ register\text{”}]$$
$$[\langle operand-size, D0, 1-8-16-32-bits\rangle, \langle D0, 1-8-16-32-bits, decl, qualifier\rangle,$$
$$\text{“}D0\ can\ handle\ operands\ 1,\ 8,\ 16,\ and\ 32\ bits\ wide\text{”}]$$
$$[\langle size, D0, 32-bits\rangle, \langle D0, 32-bits, pred-nom, quantifier\rangle,$$
$$\text{“}D0\ is\ 32\ bits\ wide\text{”}]$$

t represents the three sentences

- D0 is a data-register.
- D0 can handle operands 1, 8, 16, and 32 bits wide.
- D0 is 32 bits wide.

If we apply rule r1 first, the order of sentences two and three are reversed:

$[\langle mc, D0, data-register\rangle, \langle D0, data-register, pred-nom, na\rangle,$
"*D0 is a data register* "]
$[\langle size, D0, 32-bits\rangle, \langle D0, 32-bits, pred-nom, quantifier\rangle,$
"*D0 is 32 bits wide* "]
$[\langle operand-size, D0, 1-8-16-32-bits\rangle, \langle D0, 1-8-16-32-bits, decl, qualifier\rangle,$
"*D0 can handle operands* 1, 8, 16, *and* 32 *bits wide* "]

If rule r2 is then applied, the first two sentences are compounded:

$[\langle mc, D0, data-register\rangle + \langle size, D0, 32-bits\rangle,$
$\langle D0, data-register + 32-bits, compound, na\rangle,$
"*D0 is a data register and is* 32 *bits wide* "]
$[\langle operand-size, D0, 1-8-16-32-bits\rangle, \langle D0, 1-8-16-32-bits, decl, qualifier\rangle,$
"*D0 can handle operands* 1, 8, 16, *and* 32 *bits wide* "]

The result represents the two sentences

- D0 is a data-register and is 32 bits wide.
- D0 can handle operands 1, 8, 16, and 32 bits wide.

If, however, we apply rule r2 to t first, the first two sentences are compounded:

$[\langle mc, D0, data-register\rangle + \langle operand-size, D0, 1-8-16-32-bits\rangle,$
$\langle D0, data-register + 1-8-16-32-bits, compound, na\rangle,$
"*D0 is a data register and can handle operands* 1, 8, 16, *and* 32 *bits wide* "]
$[\langle size, D0, 32-bits\rangle, \langle D0, 32-bits, pred-nom, quantifier\rangle,$
"*D0 is* 32 *bits wide* "]

Neither rule can be applied to this result and the resultant sequence represents the sentences

- D0 is a data-register and can handle operands 1, 8, 16, and 32 bits wide.
- D0 is 32 bits wide.

This example illustrates that given any particular output to be revised, a rewrite system need not produce a single output form. Since there is no normal form for natural language text, this result is not disturbing. Any number of good texts can meet the same discourse goal.

The Kalos system described in Chapters 5 and 6 differs from the example in important ways. Instead of applying rules in a nondeterministic fashion, Kalos uses a number of principles to order rule application. It applies conceptual revision rules first, followed by stylistic rules. Among these two sets of rules, conflicting rules are ordered based on a preference to rules that produce the most structural change.

A priority rewrite system [Baeten, Bergstra, & Klop 1987, Baeten & Weijland 1987] can be used to produce a rewrite system that behaves more like Kalos than the example above. A priority

rewrite system consists of a term rewrite system and a partial order $<$ on the rewrite rules that orders rule application. If $r_p < r_q$, then r_q cannot be applied if r_p is applicable. Care must be taken in the case where some term t is being considered and r_q is directly applicable, but r_p is not. If there is some sequence of rule applications such that t can be rewritten into t' where r_p is applicable, then r_q should still be blocked.

The term rewrite system can be modified into a priority rewrite system by defining $<$ as $r_i < r_j$ iff $i < j$. Applying this priority rewrite system to t above, rule r1 must be applied first. Rule r2 can then be applied, and the result is the same as in the term rewrite system when rule r1 is applied first. In the priority rewrite system, rule r2 cannot be applied first.

This example only gives the flavor of how to cast Kalos-like revisions in terms of rewrite rules. A rewrite rule system that performed at the level of Kalos would have to encode the knowledge that Kalos uses to make revision systems into the rewrite rules. Such a rewrite system would be extremely complex.

Research into term rewrite systems includes analysis of normal forms and termination [Dershowitz & Jouannaud 1990]. Since revision of natural language text does not produce normal forms, this area of rewrite systems is not applicable to revision. Theoretically, rewrite system termination analysis could be used to determine if a revision system is guaranteed to halt. In practice, this type of analysis may not be beneficial in a practical system. First, the analysis is tedious and costly. Second, it is not difficult to insure that revision halts in a system like Kalos where rule application is ordered. The application of high priority rules that determine the structure of revised sentences invalidates the application of many lower priority rules. This feature, along with rules that limit the amount of compounding and sentence complexity, reduces the chance for looping. Finally, in a system like Kalos that is used to produce draft text only, revision application can be time limited. It does not greatly matter whether revision terminates normally or is stopped by a time limit; either way, the output still needs human intervention for polishing.

3.6 Reflections on Revision Complexity

There are three questions relevant to the complexity of revision algorithms:

- What is the complexity of natural language generation in general?
- What is the complexity of revision algorithms in general?
- Can the complexity of revision algorithms be reduced by imposing some limitations?

Reiter [1990] gives a proof that generating descriptions that are accurate, valid, and free of false implicatures from a natural taxonomy knowledge base with defaults is NP-hard in the number of attributes in the description. Based on the maxims of conversation [Grice 1975], Reiter's algorithm

produces descriptions free of false implicatures. False implicatures occur when a speaker does not give the shortest utterance possible. For example, if a speaker says that there is a "dangerous fish" in the water, a knowledgable hearer will assume that either the speaker did not know the type of fish, e.g., shark, or that the fish was not a shark or other kind of commonly known dangerous fish. For natural language generation systems to communicate accurately with humans, they must produce descriptions that are not misleading. The potential for computational explosion occurs because, in general, an exhaustive search of the taxonomy is needed to find whether some combination of descriptive terms is valid in the sense of the maxims of conversation.

Reiter gives limitations on the structure of the knowledge base to insure that descriptions known to obey the Gricean maxims can be generated in polynomial time. However, Reiter's work only addresses formulating descriptions. Other generation tasks may also be NP-hard. For generation techniques to be scalable, the algorithms must operate in linear or polynomial time at least for the average case. (David McDonald has suggested that any cognitively realistic natural language generation must operate in linear time [Reiter 1990].)

General revision algorithms may be NP-hard in the worst case. Consider a general revision system where there are n revision rules, $r_1, r_2, ..., r_n$, and some test that operates in linear time that assigns a value based on some goodness measure of the text. Then, each revised text is characterized by an ordered sequence $\langle r_{i1}, r_{i2}, ...r_{ik} \rangle$, where $r_{i1}, r_{i2}, r_{i3}, ...r_{ik}$ refer to any of the n revision rules. For simplicity, we will assume that in any given revision sequence, a revision rule may appear only once. Also assume that the search for portions of the text where a rule can be applied operates in linear or polynomial time.

One way to determine the best revision sequence is to generate the text with all possible revision sequences and use the goodness measure to select the best revision. Note that the order of rule application may produce different text. For example, $\langle r_1, r_2 \rangle$ could produce better text than $\langle r_2, r_1 \rangle$. (See the previous section for an example of how revision rule ordering can affect the resultant text.) Also note that a longer sequence may not always produce a better text than a shorter one.

The sequences of revision rule applications to some unit of text can be thought of as a general tree (see Figure 3.2). Each arc is labeled with the name of a revision rule. The sequence of arcs from the root of the tree to a node gives one possible sequence of revision rules, and the node represents the revised text. If all revision sequences were possible and no search guiding technique were used, an exhaustive search procedure would be needed to find the best text. An exhaustive search would grow exponentially with the number of revision rules [Barr & Feigenbaum 1981].

But in practice, not all sequences of revision rules are possible for two reasons. First, all of the

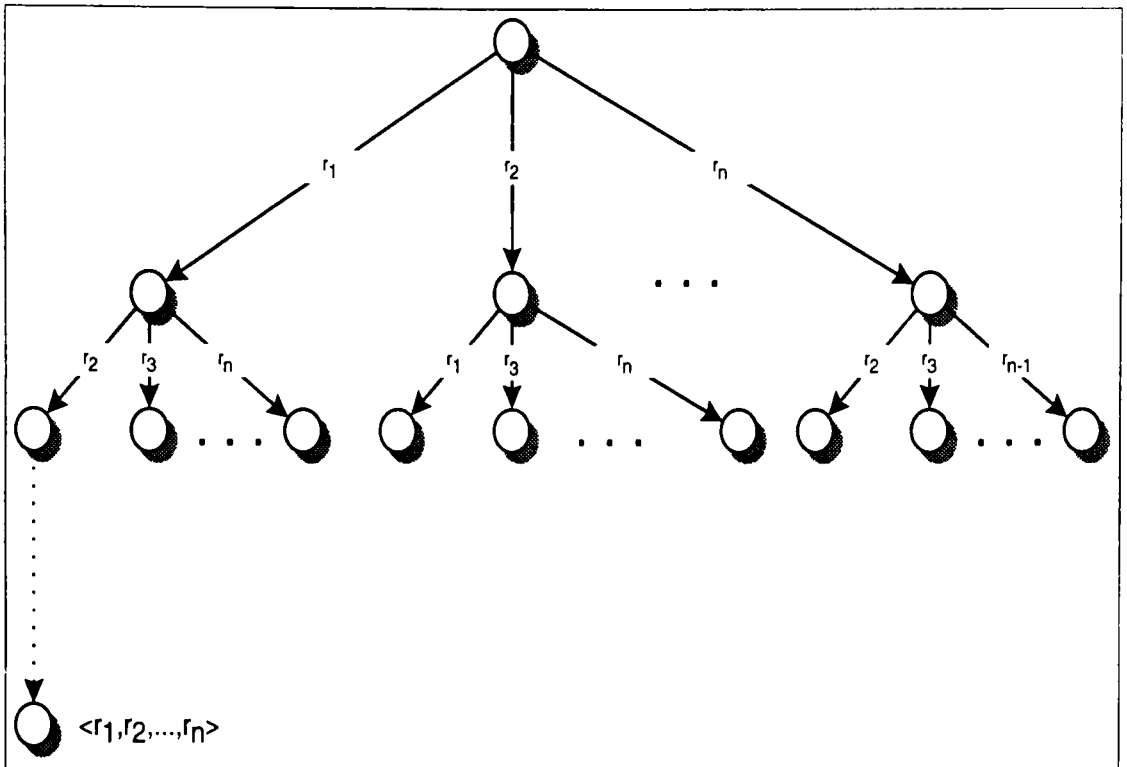


Figure 3.2 - Part of the Revision Rule Sequence Tree

rules may not be applicable to the text being revised. Second, some of the rules may be mutually exclusive. For these reasons, some leaves in the revision rule sequence tree will not be at level n. Because of this pruning of the tree, it is difficult to make a strong statement about the actual size of the tree. However, in the worst case with a text where many rules are applicable, it appears that computational explosion would occur.

Now consider a restricted revision system where rule application is ordered as in the priority rewrite system described in the last section. In such a system, no goodness measure of the text is used. The output of the system is the revised text created by applying the revision rules in the sequence governed by the rule ordering technique. As before, assume that a rule may only be applied once to a given unit of text and that the search for opportunities to apply revision rules to the text is not computationally expensive. Also assume that the rule ordering algorithm operates in linear time. Then for a system with n revision rules, the revision rule sequence can be constructed in n steps. This sequence corresponds to one path in the revision rule sequence tree of Figure 3.2.

This limited type of revision system operates in linear or polynomial time by pruning all the search tree paths except one. Since Kalos uses a revision rule ordering technique similar to this, this author believes that the algorithm of Kalos is scalable.

The problem with this type of limited revision rule ordering system is that it is not easily adapted to varying text situations. In such a system, it is assumed that applying rule r_1 before rule r_2 will always produce the best text. But it is unlikely that such a strict ordering will always produce good results. So, one of the most interesting questions in the area of natural language generation with revision is how to develop more robust revision techniques that are scalable. Such techniques must limit the area of the revision rule sequence tree that they search so that they can operate in linear or polynomial time. On the other hand, the pruning methods used must intelligently select the revision rule sequences to be explored.

3.7 Conclusion

In this chapter, I have argued that revision is a useful natural language generation technique because it provides software engineering benefits such as reduced complexity through decomposition and modularity; provides an architecture for dealing with infelicitous, verbose, and ambiguous text; and allows interaction between conceptual and stylistic decisions. A revisor should not be limited to surface-level changes but should perform both stylistic and conceptual revisions.

Knowledge bases required for revision and revision stopping conditions were also discussed in this chapter. Revision was cast in terms of term rewrite systems and computational concerns were discussed. In Chapter 6, the use of revision in the Kalos system will be described.

Chapter 4
Knowledge Intensive Natural Language Generation

I have taken all knowledge to be my province.

Francis Bacon

Natural language generation systems need knowledge about the discourse topic at hand, knowledge about discourse structure, linguistic knowledge, and lexical knowledge. McKeown and Swartout [1987] argue that different kinds of knowledge must be separated to let a generation system isolate the knowledge necessary to produce the required text without having to consider superfluous material. This chapter discusses knowledge base design, why a segregated knowledge base is inappropriate for a natural language generation system that uses text revision to produce improved text iteratively, and how the computational problems of using a unified knowledge base might be solved.

4.1 Knowledge Representation for Revision

Knowledge intensive techniques using a uniform knowledge base have been applied successfully to natural language understanding [Neal & Shapiro 1987] and to surface-level language generation [Jacobs 1985]. A basic thesis of the current work is that a uniform knowledge base supporting an intensively knowledge-based generation process provides the only realistic approach for natural language generation systems that incorporate revision.

The revision component of a generation system must detect instances of inferior text and develop strategies for improving it. A revision strategy involves identifying choice points in the generation procedure that are responsible for the shortcomings in the draft text, selecting different choices to improve the text, and producing a suggestion to the appropriate generation module to implement the revision. The development of a revision strategy places two burdens on the supporting knowledge base. First, to identify the choice points responsible for the draft text, the revision component must be able to identify the underlying knowledge from which the surface text was generated. Second, to select different choice points in the generation procedure intelligently to improve the text, it must reason about the surface text, surface generation knowledge (e.g., a generation grammar), deep generation knowledge (discourse schemata or planning rules), and domain specific knowledge.

Knowledge bases in traditional natural language generation systems usually do not meet these requirements for three reasons. First, surface generators are typically implemented using representation schemes different from the one used for the primary knowledge base. For example,

Augmented Transition Networks (ATNs) are generalized finite state automata that are used to generate surface texts. In a typical system, knowledge about the arcs of the ATN and the algorithms used to select and traverse the arcs is hidden from the inference system used by the primary knowledge base. The knowledge coded in unification grammars is usually similarly buried, accessible only to the chart parser.

Second, due to computational constraints, most systems use segregated knowledge bases. The deep generator typically has access to domain knowledge and knowledge about discourse structures. The surface generator has access to the message produced by the deep generator and the grammar it uses to convert the message to surface text. In some cases, the generators run as separate modules and do not have access to each other's knowledge. In other cases, knowledge used by each generator is encoded using different representation schemes that use different inferencing techniques. However, knowledge bases used in traditional natural language generation systems are too constraining to be used for the implementation of a revision system. In a system with highly segregated knowledge bases, revision software would only have access to the surface-level text and would have to perform its analysis of the text without benefit of the underlying structures. The lack of knowledge about the intent of the text and the possible alternatives during generation makes the revision task far harder than with a uniform knowledge base.

Third, a knowledge intensive system is more adaptive than one in which knowledge is encoded procedurely [Jacobs 1985]. Explicitly encoded knowledge has two advantages. First, explicitly encoded knowledge in a uniform representation may be extended more easily by adding more knowledge. In a system where knowledge is encoded procedurely, additional programs must be written to add knowledge. Second, explicitly encoded knowledge is more easily shared by multiple generation modules. Each module accesses the knowledge that it needs. In a procedural system, adapting routines for use by several different modules is more difficult.

To illustrate the knowledge requirements of a natural language generation revision component, consider an example of the stylistic revision of surface level text containing two parallel constructions:

- The SR register contains the condition codes.
- The SR register contains the interrupt mask.

To revise these sentences, the generation system must first infer that the two sentences have the same subject and verb and different objects. Next, it must pinpoint the choice point or points in the generation grammar that generated these two sentences as stand-alone sentences. Finally, it produces a revision suggestion to the surface generation component identifying the choices that produce a single sentence:

- The SR register contains the condition codes and the interrupt mask.

In a traditional system, the revision software would not have access to the underlying grammar rules used to generate the individual sentences. It would have to parse the individual sentences to determine that they have the same structure, subject, and verb. In a slightly more complicated example, this analysis would be more difficult. For example, if one of the sentences used a pronoun, the referent would have to be determined. However, in a system using a uniform knowledge base, the underlying grammar rules used to generate the sentences and information about referents are available to the revision software without the need to parse the surface text.

Now consider an example of conceptual revision in a system with a schema-based deep generation component. The revision component decides that the sentence

- The address bus of the M68000 is 24 bits wide.

should be rephrased to express a direct attribute of the M68000:

- The M68000 has a 16 megabyte address space.

To develop a strategy for this revision, the revision component must associate the first sentence with a schema slot in a schema describing the address register so that it can make a suggestion to the deep generation component to remove the contents of this slot. Next, it must infer which schema slot associated with the paragraph describing the microcprocessor must be filled so that the second sentence will be generated. This step requires the revision component to inspect the deep generation schemata and the system knowledge about how the slots are filled. Domain knowledge is required to infer address space size from address bus size. In a generation system with a traditional knowledge base, combining these different types of knowledge for the necessary inferences would not be feasible.

4.2 A Uniform Generation Knowledge Base

Figure 4.1 shows the relationship of knowledge types to surface text in a knowledge-intensive natural language generation system. The deep generation module applies discourse knowledge (schema filling knowledge or planning rules) to a domain knowledge base to select and order concepts. These concepts then go to the surface generator to be converted into surface text using its linguistic knowledge. For the revision module to produce revision suggestions efficiently, it must have access to the knowledge used to generate the surface text. Specifically, it must know the concepts that were surfaced and choice points used in generating the text, and be able to determine what other choices are available. The dotted lines in the figure indicate the association between the surface text and the underlying knowledge from which it was generated. Not only must a revisor have

access to all the system knowledge, it must be able to perform inferences over all this knowledge to produce intelligent revision suggestions.

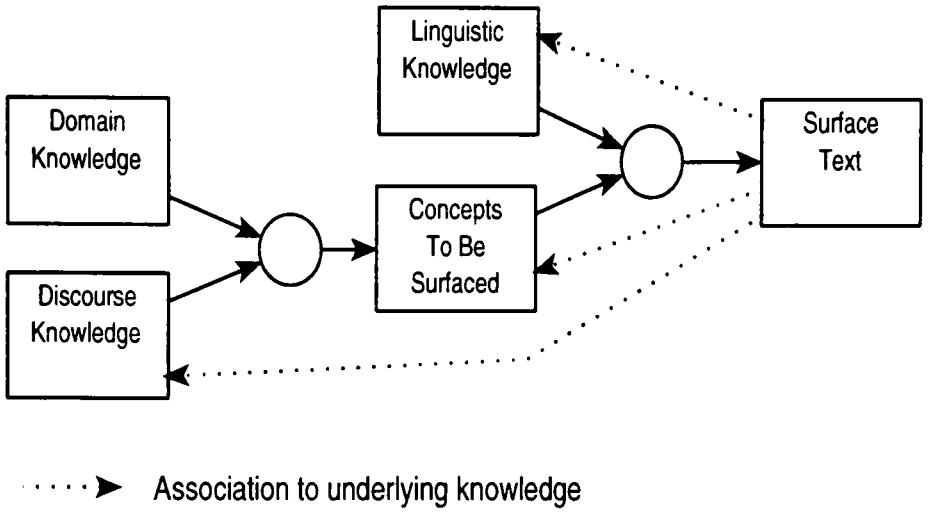


Figure 4.1 - Relationship of Knowledge to Surface Text

4.3 Efficiency Issues

The primary disadvantage of using a unified knowledge base is that the system must deal with superfluous information, requiring more computational power to perform inferences and complicating the process of selecting appropriate information. One solution to this problem is to partition the knowledge base dynamically. During deep generation, only domain knowledge and discourse structure knowledge is available. During surface generation, only the linguistic knowledge is available. For stylistic revision, only the surface text and related grammar rules are available. During conceptual revision, surface text, instantiated schema, associated schema templates, domain knowledge, and grammar rules are available. The partitioning of the knowledge base at each step reduces the size of the total knowledge necessary for inference during each step.

The idea of knowledge base partitioning can be taken one step further by running the generation process in stages. Each stage runs as an independent process loading the knowledge base partition it needs for the individual generation task at hand and pertinent knowledge produced by previous stages. Since each stage starts fresh, intermediate results produced by inferencing in

previous stages are not loaded, thus reducing the size of the knowledge base being considered. Furthermore, less demand is placed on the underlying storage management system (e.g., Lisp). Since each module is free to load any part of the knowledge base during any generation stage and since it typically requires only a portion of the system knowledge at any stage, the system has the advantage of using a uniform knowledge base but with increased efficiency.

The knowledge base is partitioned in two ways. Static partitioning is used at system design time to create the partitions for each generation task. For example, the deep generator uses a knowledge base partition containing only domain knowledge and knowledge about discourse structures. Dynamic partitioning is used to build partitions that contain a static partition plus a semantic network subnetwork determined from a message generated by a previous generation task. For example, the surface generation partition is constructed from a static partition containing linguistic knowledge plus the message to be surfaced that was created by the deep generator. Part of the dynamic partitioning process is to extract all knowledge base nodes referenced in the message given to the surface generator. In the case of Kalos, the message is an instantiated schema with links to the schema from which it was generated and the conceptual knowledge that was used to fill in the schema slots. All of this knowledge must be added to the knowledge base partition used by the surface generator. Due to the design of instantiated schemata and the domain knowledge base, only the subgraph dominated by the top node of the instantiated schema and the small set of subgraphs indicating numeric quantities need be copied. More details of how knowledge structures are passed between stages in Kalos is given in the next chapter.

In some cases, a processing stage may not need access to all the knowledge in a particular knowledge base partition. A conceptually restricted surrogate of the partition can be created that contains only the knowledge required for the particular stage. By using the restricted knowledge base partition instead of the entire knowledge base partition, the overall size of the knowledge base for the stage is reduced, improving inference performance. Overall performance is also enhanced since the inference steps needed to produce the surrogates is performed only once regardless of the number of revision iterations. If the entire knowledge base was used, these inference steps would have to be performed for each revision iteration over a larger knowledge base.

Some knowledge base restrictions can be constructed using circumscription [Genesereth & Nilsson 1987]. If a stage has antecedents in a knowledge base partition that can only be satisfied directly or indirectly by a small set of ground terms, then a conceptually restricted knowledge base partition can be generated that will allow inference to proceed with the same results as if the entire partition were used.

Conceptually restricted knowledge base surrogates can be generated from static knowledge

bases such as the knowledge base containing surface generation knowledge. For example, the conceptual revisor needs to map from a domain-specific preferred word or phrase to a schema filling choice that will cause the term to be generated. To do this, the revisor must inspect the surface grammar rules and lexicon to discover which knowledge frames or objects can trigger the generation of the word or phrase. This knowledge can be obtained by inference over the lexicon and grammar rules. But one of the antecedents to this inference is one of the preferred words or phrases. By building a small knowledge base containing only the mapping between preferred words and phrases and the corresponding knowledge frame or object that can trigger their generation, most of the knowledge contained in the surface generation knowledge partition can be discarded during the conceptual revision phase.

Surrogates for intermediate results can also be created to reduce the overall size of the knowledge for a given stage. Constructing this type of conceptually restricted knowledge base does not involve using circumscription. In this case, we remove knowledge that we know a particular stage will not need. For example, the structure that represents surface text includes surface text, the grammar rule from which it was generated, a binding list representing intermediate values produced while the surface text was produced, and the schema slot value from which the text was generated. The grammar rule and binding list knowledge are not needed during conceptual revision processing; however, they can be substantial during the generation of extensive text. To reduce the overall knowledge base size, these semantic network subtrees need not be included in the knowledge base used for conceptual revision.

Another aid in reducing the computational burden of a uniform knowledge base is to use path-based techniques that avoid matching and inferencing. Path-based techniques are used to associate the surface text and underlying knowledge from which it was generated, using an appropriate knowledge base construct (in the case of Kalos, an arc in a semantic network). Whatever technique is used to associate surface text and underlying knowledge, it should be computationally inexpensive to locate the underlying knowledge from the surface text. By following these associations, the revision module can locate the knowledge structures responsible for the generated text without performing computationally expensive matching and inference.

Other techniques to improve uniform knowledge base efficiency depend on the type of knowledge base used. The next chapter addresses specific solutions used in the Kalos demonstration system.

4.4 Conclusion

The need for a uniform knowledge base was discussed in this chapter. Traditional natural language generation systems use segregated knowledge bases and differing representation and inferencing techniques for different knowledge bases in the system. Since a revision system needs access to all system knowledge to make intelligent revision suggestions, a uniform knowledge base supporting a uniform inferencing technique is appropriate.

When using a uniform knowledge base, techniques are needed to reduce the computational burden. I have suggested three techniques for improving computational efficiency in a system with a uniform knowledge base:

- Static and dynamic knowledge base partitioning
- Staging and conceptually restricted knowledge base partitions
- Path-based techniques

Chapter 5
Kalos - Knowledge Representation

> Creative writers are always greater than the causes that they represent.
>
> E.M. Forster, *Gide and George*

5.1 Implementation Overview

Kalos is a testbed for developing revision techniques as described in Chapter 3 in a knowledge intensive natural language generation system as described in Chapter 4. The system is implemented in SNePS-2.1 [Shapiro 1992] and Common Lisp. This chapter gives an implementation overview and discusses knowledge representation and techniques for dealing with the computational burdens of a large, uniform knowledge base. Chapter 6 describes the techniques Kalos uses to generate and revise text, provides sample output from the system, and argues that revision in a knowledge intensive environment is a beneficial technique for a natural language generation system.*

At the center of Kalos lies a uniform knowledge base containing domain knowledge, linguistic knowledge, discourse strategy knowledge, and knowledge about revision techniques. All these forms of knowledge are represented together in a single SNePS-2.1 semantic network. Early versions of Kalos used the SNePS-2.1 belief revision system to improve efficiency. As Kalos grew and exceeded the capacity of the underlying computer system on which it ran, other techniques were developed to allow Kalos to deal with a large uniform knowledge base. The current version uses staging, partitions the knowledge base dynamically so that only relevant portions of the knowledge base are accessible for any given generation task, and applies path-based inference to improve inferential efficiency. (See Chapter 4.)

Kalos consists of a uniform knowledge base and two main modules: a generation module with deep and surface generation submodules, and a revision module consisting of conceptual and stylistic revision submodules (see Figure 5.1). The schema-based deep generator selects and orders concepts to meet some discourse goal. The surface generator converts the concepts selected by the deep generator into surface text. The conceptual revisor makes suggestions for improving conceptual defects in the text, while the stylistic revisor makes suggestions for improving stylistic defects in the text.

Unlike those of a traditional generation system, Kalos's deep and surface generators are relatively simple. Initially, they produce simple draft text. Many of the decisions made by traditional

* Chapters 5 and 6 assume that the reader is familiar with SNePS-2.1; Appendix A is a SNePS-2.1 tutorial for the reader unfamiliar with it.

generators are postponed for consideration by the revision modules. After generating the initial text, Kalos improves it iteratively in two cycles. The first cycle consists of the deep generator, surface generator, and conceptual revisor. In each iteration of the first cycle, the deep and surface generators produce text, and the conceptual revisor examines it for defects. If the revisor finds defects, it produces suggestions to improve the text. The deep generator uses the suggestions to regenerate the text. Revisions are cumulative, i.e., no revision suggestion is ever retracted in a later pass. The cycle ends when the conceptual revisor finds no further defects.

Neither the decision not to retract revisions nor the termination condition is fundamental to the model. Both were chosen for simplicity. Other systems, for example, could use a measure of text quality to determine when to stop the revision process.

The stylistic revisor cycle begins after all conceptual revisions are complete. This cycle begins with the stylistic revisor, which reviews the draft text and produces revision suggestions to improve it. These suggestions go to the surface generator, which uses them to regenerate the text. This cycle ends, and the final text is output, when no more revision suggestions can be applied.

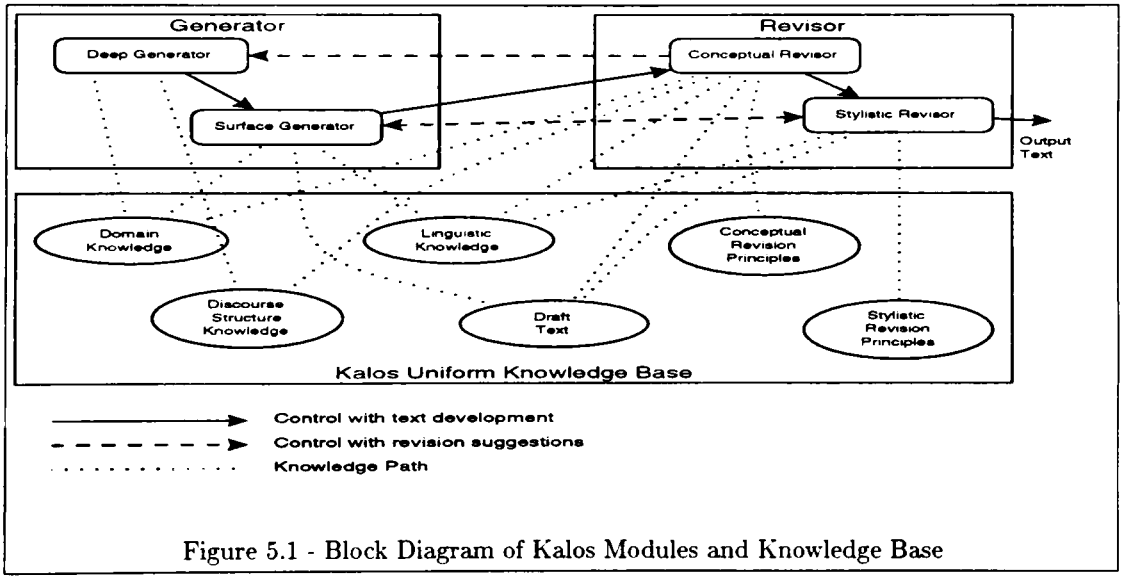


Figure 5.1 - Block Diagram of Kalos Modules and Knowledge Base

As Kalos is described in this and the next chapter, SNePS-2.1 semantic networks will be presented. Figures containing semantic networks have been simplified as much as possible without removing critical information. One complicating factor is the uniqueness principle used in SNePS-2.1: SNePS-2.1 does not duplicate a node that would look exactly like an existing node. For example, Figure 5.2a shows a semantic network representing two facts: a is of size 32 and b is of

size 32. The node for *32* is not duplicated. This representation complicates pattern matching in the network. The node for *32* has two *size-* arcs emanating from it, one pointing to node x and one pointing to node y. To avoid this problem, the representation shown in Figure 5.2b is used. The same concept is represented, but the tails of the *size-* arcs from nodes q and r do not originate from the same node, making pattern matching and arc traversal easier. The size of object a is represented by node q, and the size of object b is represented by node r. Given a size node, it is easy to locate the corresponding object node by traversing the *size-* arc followed by the *object* arc. But in Figure 5.2a, given the size represented by the node *32*, there are two *size-* arcs emanating from the node. The form in Figure 5.2b is used to avoid this ambiguity.

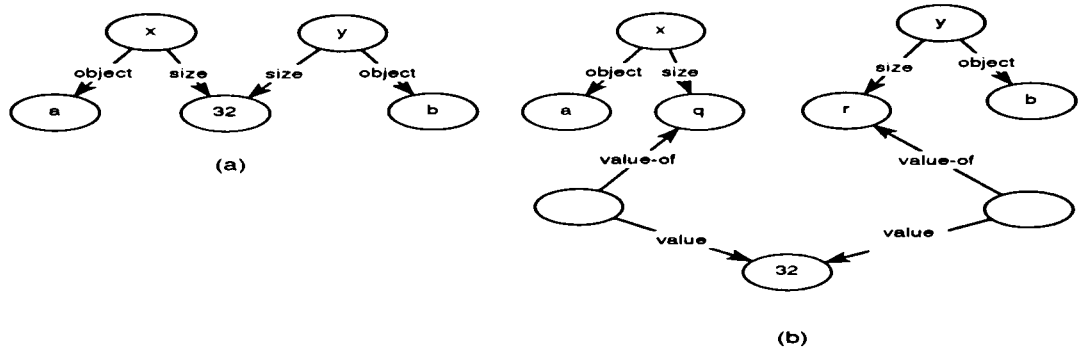


Figure 5.2 - (a) Uniqueness of "32" and (b) Kalos Representation.

The second representation is more visually complex than the first when displayed in a document. For that reason, the figures in this document will sometimes use the first representation, as well as other simplifications. For example, Figure 5.3a is a very small fragment of Kalos knowledge base information. The node b1 represents the program counter of some computer. The nodes labeled m1!, m2!, and m3! are assertion nodes representing that the object that b1 represents is called the "program counter register," the "PC," and the "program counter." The node m4! asserts that b1 is a register. To simplify the following diagrams, I factor out subdiagrams like m1! through m3!, eliminate system generated labels like m1!, etc., and replace the label "b1" by a convenient linguistic label (e.g., "PC"). Figure 5.3b shows the abbreviated rendering of Figure 5.3a.

One other liberty is taken with semantic network diagrams. A SNePS-2.1 base node with value x occurs only once in the network even if it is referenced several times due to the uniqueness principle. However, in many of the diagrams in this chapter, several different base nodes will be

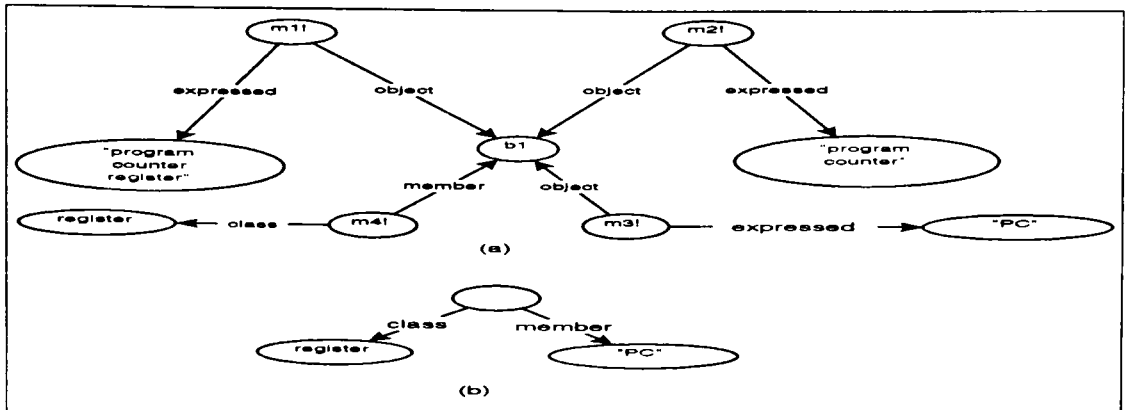


Figure 5.3 - Kalos Representation Simplification. Actual Network (a) and Simplified Version (b)

drawn that have the same value to avoid complicating the diagrams with crossing lines. The reader should be aware that these nodes actually collapse into a single node in the actual SNePS-2.1 semantic network.

5.2 Knowledge Types in Kalos

Kalos maintains domain knowledge, discourse strategy knowledge, linguistic knowledge, and knowledge about revision using a uniform representation and inferencing system, the semantic network system of SNePS-2.1. For Kalos to generate text about a microprocessor, it must have a domain knowledge base containing facts about a particular microprocessor and about general computer knowledge. It is from the domain knowledge base that the Kalos deep generator selects concepts to be used in describing a particular microprocessor. It performs this task using discourse strategy knowledge that contains text patterns that can be used to describe objects. The selected concepts are passed to the surface generator where linguistic knowledge is used to generate a surface language representation for them. Finally, the revisors use knowledge about text revision to look for defects in the text and to make suggestions on how to improve the text.

The domain knowledge includes general facts about computers and specific information about a specific microprocessor. A taxonomy is used to relate parts of the microprocessor, real and abstract, to categories of part types, and part/whole relations are used to relate parts to the objects of which they are a part [Emero 1992]. Attributes, e.g., bit sizes, are associated with objects. Process frames are used to describe the actions that result from individual instruction execution. Process frames are not included for other parts of the microprocessor, because the engineers for which a users' guide is written are assumed to know how general hardware elements such as registers work. If the

system were used to generate text for novices, process frames would be needed for all the elements of the microprocessor.

Some general computer knowledge is included in the taxonomy, such as the relationship between different types of processors. In addition to this taxonomic knowledge, there are SNePS-2.1 inference rules that encode general knowledge. For example, the fact that the address bus size of a processor is related to its address space is asserted as a rule.

Discourse structure knowledge is encoded as a set of schema templates and SNePS-2.1 rules that select nodes to fill schema slots. The schema slot-filling knowledge consists of some general rules about slot filling and specific knowledge about how each slot is filled. The slot filling knowledge consists of SNePS-2.1 constructs that trigger either pattern matching or inferencing to select concepts from the domain knowledge to fill the slot.

Linguistic knowledge consists of grammar rules encoded in SNePS-2.1 assertions. Assertions encode lexical data and grammar rules. The grammar rules form a functional unification grammar. Grammar features are encoded as attribute/value pairs that contain variables. The attribute/value pairs can reference all types of knowledge in Kalos, so both linguistic and conceptual knowledge can be tested in the grammar.

Conceptual revision knowledge is encoded as SNePS-2.1 inference rules. These rules are used to look for conceptual-level defects such as improper order of attributes and redundant information.

Stylistic revision knowledge is also encoded as SNePS-2.1 rules. These rules look for surface defects that can be repaired by the application of cohesive constructs such as sentence compounding and anaphora. Other stylistic revision rules are used to prioritize revision suggestions and select among conflicting suggestions.

The knowledge representation for encoding surface text is also important. Included in the surface text representation is the ordered surface text and association links to underlying knowledge structures. These association links are used two ways. They connect words with the nouns they represent, and they associate the surface text with the grammar rules and schema slots from which it was generated. The structures used are recursive so that each part of the surface text can be traced to the underlying knowledge from which it was generated. The revision modules need to access this underlying knowledge so that they can efficiently analyze the text for defects and look for alternate ways to generate the text.

5.3 Domain Knowledge Representation

Kalos selects the facts that will eventually be surfaced into natural language from the domain knowledge base, which includes general computer knowledge and facts about a particular micro-

processor. The microprocessor used in the examples in this paper is the Motorola M68000. The domain knowledge base also includes knowledge about typical features of a microprocessor, so that Kalos can avoid stating commonly known facts and state facts that are salient because they are atypical.

5.3.1 Natural Taxonomy

Domain knowledge is encoded as both SNePS-2.1 assertions (e.g., the M68000 is a microprocessor) and inference rules (e.g., the fact that the address space size of a microprocessor is related to its address bus size). Knowledge about the M68000 is encoded as a taxonomy of microprocessor parts. The taxonomy in question is a natural taxonomy in the sense of Rosch *et al.* [1976], so that Kalos can select appropriate class names when describing a part, determine salient facts by looking for atypical features of class members, and implement a shallow model of audience expertise [Cline and Nutter 1990]. Psychological research has shown that natural taxonomies contain a distinguished or basic level. Adult speakers use the names of these categories most frequently and can list a large number of attributes for them. They typically cannot list many attributes for superordinate categories and list few additional attributes for subordinate categories. Furthermore, the distinguished level for domain experts is at a lower level in the hierarchy than for non-experts and contains more attributes [Rosch *et al.* 1976]. Cline and Nutter [1990] and Reiter [1990] have used natural categories to select the appropriate level of description in natural language generation systems.

Emero [1992] developed a semiautomatic technique for constructing a taxonomy of terms related to computers and microprocessors. In this study, Emero analyzed several computer texts and microprocessor manuals. He then constructed a taxonomy and determined basic level categories statistically for two groups of users: novice and expert. He also identified a list of attributes and part/whole relationships used in computer texts. In Kalos, natural categories were selected based on the author's experience with microprocessor books and manuals and on Emero's research. The demonstration version of Kalos was designed to use terms in general use in microprocessor manuals whose target audience are engineers.

Natural categories allow Kalos to select the appropriate level of description for an item being described. For example, Kalos states that "D0 is a data register" because *data-register* is at the distinguished level in the taxonomy to which *D0* belongs. It would be inappropriate for Kalos to state that "D0 is a high-speed storage device" by selecting one of the superclasses to which *data-register* belongs. Stating that "D0 is a register," where *register* is one of the superclasses of *data-register*, would provide less information to the experienced engineer. Furthermore, natural

categories prevent Kalos from explaining objects that a typical reader already understands. A typical engineer already knows what a data register is. To explain it in a document intended to be read by engineers would be at best annoying and very likely misleading. Kalos avoids this situation by not describing objects at the distinguished category level.

Kalos can be reconfigured to describe objects for other audiences by changing the distinguished category levels. I do not describe this reconfiguration here because the current research focuses on revision and knowledge intensive generation. However, the design of Kalos makes such changes relatively simple.

The deep generator of Kalos must select salient facts when describing a microprocessor. Statement of the obvious is confusing to the reader, while the omission of salient facts causes the generated discourse to fail to meet the given discourse goal. Kalos uses natural categories as an aid to determining salient facts. Typical attributes for class members are encoded at the natural category. Attributes of class members that deviate from typical values at the distinguished level are considered salient. For example, registers are typically both readable and writable under program control. To state this fact when describing a typical register to an engineer would not be appropriate. But if a particular I/O register is only writable, this fact should be noted. Kalos uses the typical attributes encoded at the distinguished level to determine atypical attributes.

Kalos uses other salience rules that are described in section 6.1.1.

5.3.2 Representation

Figure 5.4 shows a simplified Kalos taxonomy for the *address-register* class, represented by the node "AREG" in the figure. AREG is a subclass of the *register* class and is a natural category that represents the class to which all address registers in all machines belong. Both *storage device* and *high-speed storage device* are superclasses of *register*. "M68AR" is a subclass of AREG and represents the address registers of the M68000. This subclass is divided into two subclasses for pure address registers and special address registers. The special address registers, A7 and A7', act as the user and supervisor stack pointers, respectively. All nine address registers belong to the class AREG and belong either to the pure address register subclass or to the special address register subclass. A7 and A7' also belong to the USP (user stack pointer) and SSP (supervisor stack pointer) class, respectively.

Figure 5.5 shows part of the SNePS-2.1 network used in Kalos to represent the address register taxonomy. *Superclass/subclass* frames are used to build the class hierarchy. Each class has a *category-type* frame that indicates if it is a superclass, a natural category, or a subordinate class. *Member/class* frames are used to associate an instance, such as *A0*, with its natural category, such

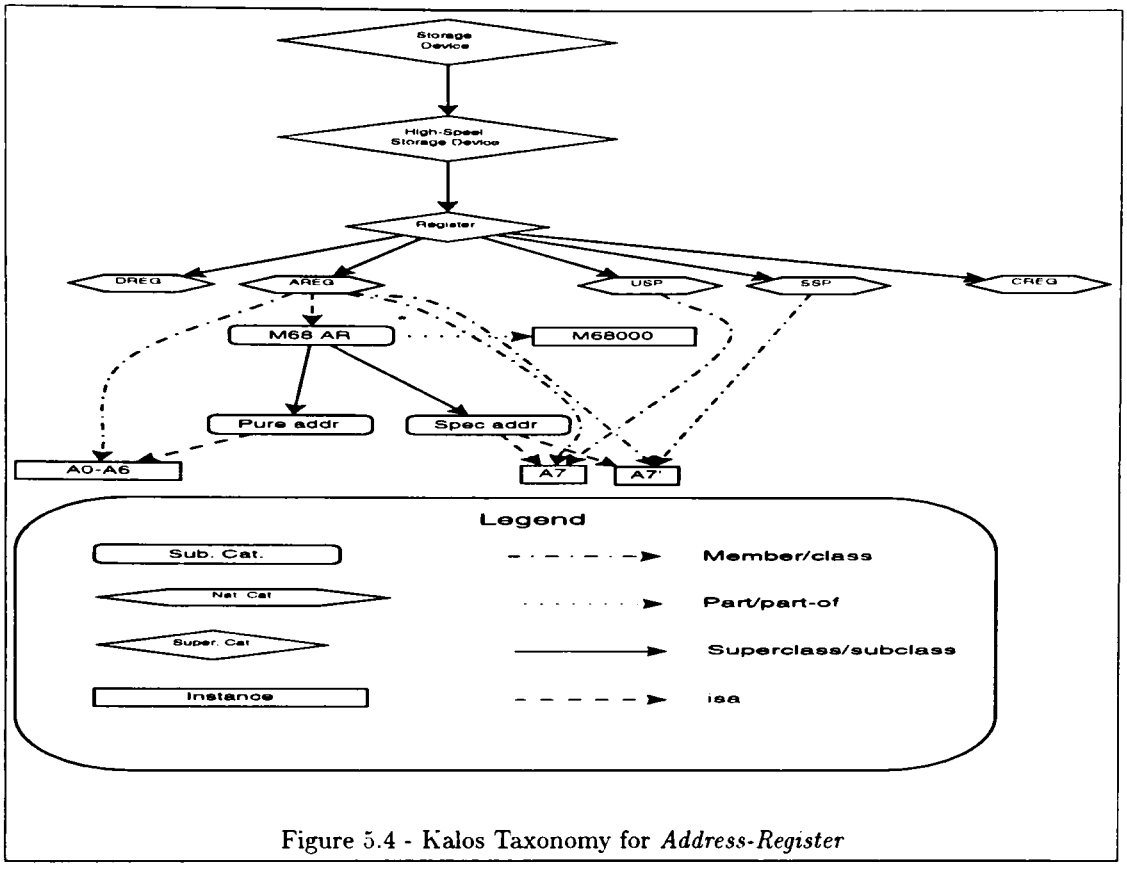


Figure 5.4 - Kalos Taxonomy for *Address-Register*

as *AREG*. An *isa* frame is used to associate an instance with its lowest level subordinate class. Instances inherit attributes via the *superclass/subclass* network. *Part/part-of* frames are used to indicate which components are constituents of other objects. In the diagram, each address register, such as *A0*, inherits that it is part of the M68000 from the M68AR class.

Most of the attributes for an object are attached to the node representing the natural category to which the object belongs. The remaining attributes are associated with subordinate classes to which the object belongs. To limit the amount of searching the system has to perform to locate attributes, *member/class* frames associate each object with its natural category and *isa* frames associate the object with its lowest level subordinate category. Computationally inexpensive arc traversal is used to locate the nodes at which the majority of the attributes for an object are located.

Kalos domain knowledge also includes attributes for categories and instances, represented using the notation of Figure 5.2b. Machine instructions are also represented in Kalos. Figure 5.6 shows the Kalos representation for the M68000 BCHG instruction. The BCHG instruction

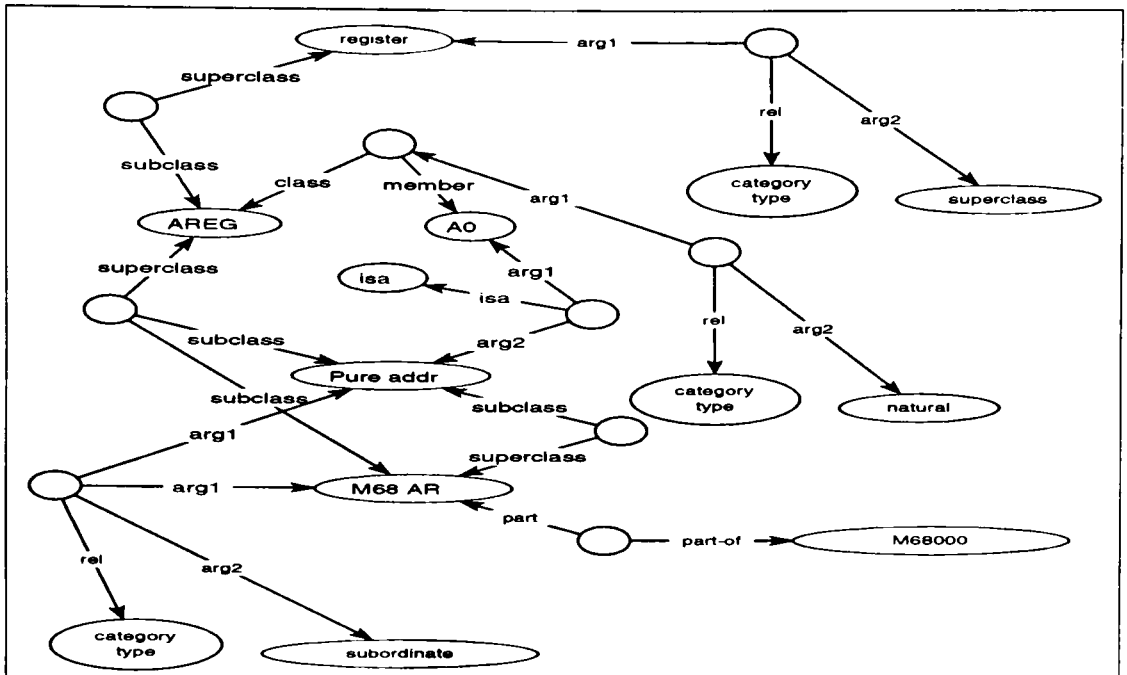


Figure 5.5 - Part of the SNePS-2.1 Network for Figure 5.4

complements the bit in the destination effective address specified by the source register and puts the complemented bit into the Z (Zero) status register. It then puts the complemented bit back into the destination operand, replacing the original bit value. The top level frame gives the instruction name and one or more actions performed by the instruction. The *action/next* form is used to create an ordered list in SNePS-2.1. First, the action indicated by node q is performed, followed by the action indicated by node r. The *ref/store* frame is used to specify a storage object pointed to by the *ref* arc, e.g., a register or instruction source or destination operand, and an access mode pointed to by the *store* arc. *SR* specifies the source register of the instruction, and *DEA* (Destination Effective Address) is the instruction destination operand. *Src* specifies that the specified memory object is to be read, while *Dest* indicates the register is to be written. The *bit/operand* frame indicates that the operand value pointed to by *bit* is the bit number to be accessed in the *operand* node. The *comp* arc indicates that the value it points to is to be complemented. The *to/from* frame indicates a register/memory transfer.

Instruction representations in Kalos capture the programmer model of what the instruction does. There is no attempt to capture the actual hardware operations or the actual order in which they occur. Typical vendor manuals for microprocessors take this approach. Kalos could, however,

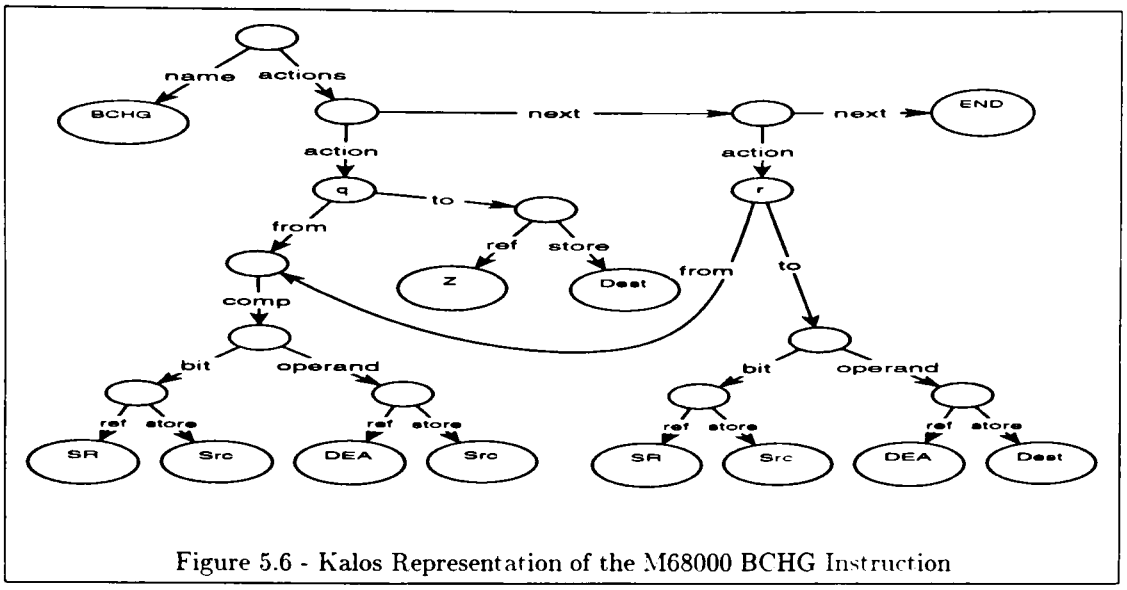


Figure 5.6 - Kalos Representation of the M68000 BCHG Instruction

be re-engineered to deal with a more detailed description level for instructions.

5.4 Deep Generation Knowledge

The deep generator uses discourse schemata to construct an ordered list of concepts that satisfy the goal of describing an object [McKeown 1985, Paris 1985]. A schema is a pattern of text that accomplishes a particular discourse goal. For example, one way to describe an object is first to relate it to a classification system, then tell what it is a part of, and finally describe its attributes (e.g., size and color). Kalos uses discourse patterns that first describe the microprocessor in terms of its relationship to a taxonomy of computer parts. Next, attributes of the microprocessor are listed. Then the parts of the microprocessor are listed, followed by a description of each of the parts. The schemata were created based on McKeown's [1985] analysis of human-produced text and a brief analysis of microprocessor users' guides.

Three types of knowledge are used by the deep generator: discourse patterns, salience rules, and knowledge about how to decompose object descriptions. The discourse patterns are stored declaratively as a set of schema templates. They contain attached knowledge that describes how to use inference and pattern matching to fill the schema slots. Salience knowledge is encoded as SNePS-2.1 rules. These rules are used to select salient concepts to fill schema slots. Other SNePS-2.1 rules are used to determine how to decompose an object into parts and part categories when producing a description of the constituents of an object.

In this section, I describe how discourse schemata are represented in Kalos. In section 6.1, a

complete description of Kalos schemata is given, salience rules are explained, and the schema filling mechanism is described.

5.4.1 Schema Template Representation

Figure 5.7 shows three simplified schema templates (*description*, *identification*, and *constituency*) in list format notation. (We describe a simplified schemata so that we can focus on the details of the representation without dealing with complicating issues related to deep generation.) The list form is not used directly by Kalos but provides a compact notation for describing schemata. These schema templates are used to illustrate how schemata are represented in Kalos.

Schema templates have two types of entries: slots and schema embeddings. Slot entries in templates describe how to extract knowledge base nodes to describe the object under consideration. Recursive schema embeddings indicate where another schema template should be used in describing the object. The slot filling mechanism uses the schema templates to build an instantiated schema that contains a structured list of concepts for describing the object at hand.

The simplified description schema contains no slots. It embeds an *identification* schema to describe an object in terms of its taxonomy and attributes. It then embeds a *constituency* schema to describe the individual parts of the object being described.

```
Schema Description X
  -> Identification X
  {-> Constituency X}

Schema Identification X
  (or (Taxonomy X = [member X/mainclass ?])
      (Taxonomy X = [subclass X/superclass ?]))
  {Attribute ?  = ?, :[object X/attribute ?]}*

Schema Constituency
  -> Description ?  = ?, :[part-of X/part-or-part-cat ?]
```

Figure 5.7 – Kalos Discourse Schemata

We now describe the simplified schema templates in terms of their representation. The list form is discussed first and then the actual semantic network representations are discussed. Schemata definitions take the following form:

```
Schema <name> X
 (<slot>|<embed>)+
```

where ⟨*name*⟩ is the name of the schema being described. "X" is a variable indicating the object being described. Following the first line are one or more lines (indicated by "+") that describe the schema slots. Each line describes a slot which is filled with one or more knowledge base nodes or the recursive embedding of another schema. ⟨*slot*⟩|⟨*embed*⟩ indicates this idea where "|" means *logical or*.

Slot descriptions contain attached knowledge that describe to the slot filler how to invoke SNePS-2.1 inference or pattern matching to select knowledge to fill the slot. Entries for embedding another schema may optionally have this same type of attached knowledge that describes to the slot filler how to locate the items for which schemata are to be embedded. We describe the representation for the attached knowledge here and give details on its use in section 6.1.1.

A line indicating a slot is of the form

```
<slotname> <var1> = <var2>, :[<pattern>|<inference>]
```

where ⟨*slotname*⟩ is the name of the slot, ⟨*var*1⟩ is a variable representing the object being referenced by the slot. ⟨*var*2⟩ is optional. If it is present, it is a variable representing the value used to fill the slot. ⟨*pattern*⟩|⟨*inference*⟩ indicates a knowledge construct that tells how to fill the slot. If the ":" is not present, the term is a pattern for matching; otherwise, the term is used to trigger inference. In either case, the term is shorthand for a Kalos knowledge base frame. In the case of pattern matching, the slot is filled based on results of pattern matching using the term. If inference is indicated, the system fills the schema slot based on the results of inferencing.

If ⟨*var*2⟩ is not present, the result of inferencing or pattern matching is used to fill the slot; otherwise, the values that were instantiated to the specified variable during inferencing or pattern matching are used to fill the slot or to select an object for which a schema will be embedded.

For example, consider the *taxonomy* slot in the *identification* schema:

```
Taxonomy X = [member X/class ?]
```

This line says that the slot name is "taxonomy," that it references the object indicated by variable "X" (the object being described by the *identification* schema), and that the slot is filled with all knowledge base *member/class* frames that have X as their *member* node. "?" matches any node.

Recursive embedding is indicated by the "→" symbol. Following this symbol is the name of the schema to be embedded, followed by a variable indicating the object being described. A schema can be embedded to continue the description of the current object or to describe the parts of the current object. If object parts are to be described, as in the *constituency* schema, an inferencing or pattern matching form, as in slot filling, can follow, indicating how to select the parts to be described by the embedded schema. For example, in the *constituency* schema, a *description* schema is embedded for all objects for which the system can infer a *part-of/part-or-part-cat* frame. The *part-or-part-cat* relation indicates the unique parts of an object and the categories of parts where there is more than one part in a group. For example, the M68000 has one data bus but eight homogeneous data registers, so the *part-or-part-cat* relation would refer to the data bus and the subordinate category of "M68000 data registers," not to the individual data registers. SNePS-2.1 inference rules indicate to the system how to construct the *part-of/part-or-part-cat* frames.

Braces indicate an optional slot or embedded schema. Braces followed by "+" indicate the slot or embedded schema is to be filled one or more times. Braces followed by "*" indicate a slot or schema can be filled zero or more times. Parentheses are used for grouping. "Or" indicates a choice point where one of two or more slots can be filled. Each choice in an *or* can be a single slot or a list of slots. Braces and parentheses are used to indicate each choice.

These discourse schemata are actually represented as semantic networks in the uniform knowledge base of Kalos. Figure 5.8 shows how these simplified schemata would be represented in Kalos. Each entry indicates either that a slot should be filled or that another schema may be recursively embedded. Slots and schema embeddings may have an optional modifier, associated with the *modifier/modified* frame, indicating that if the slot cannot be filled, generation failure is not necessary. For each slot represented in the template, there is attached knowledge that indicates how to fill the slot. The node pointed to by *arc* gives the slot name. A *pattern* arc in the template indicates that the required knowledge can be found by doing a SNePS-2.1 pattern match command, *findassert*, on the pattern represented as part of the *pattern* subgraph. An *infer* arc indicates that SNePS-2.1 inferencing should be initiated by the SNePS-2.1 *deduce* command. An *infer* subgraph can also have *filter* and *filter-by* arcs that tell the deep generator how to handle the result of inferencing. This feature allows pattern matching to isolate knowledge generated by the inferencing. The *filter* subgraph gives the arguments for pattern matching. The *filter-by* subgraph gives a SNePS variable whose value represents the node or nodes that should fill the slot. Kalos variables are strings enclosed in braces, e.g., ⟨*object*⟩, which is always instantiated to the object being described.

The *or* nodes represent ordered choices. The choices are ordered left to right in the figures, with the leftmost node being used first. The leftmost node that can be filled is placed in the

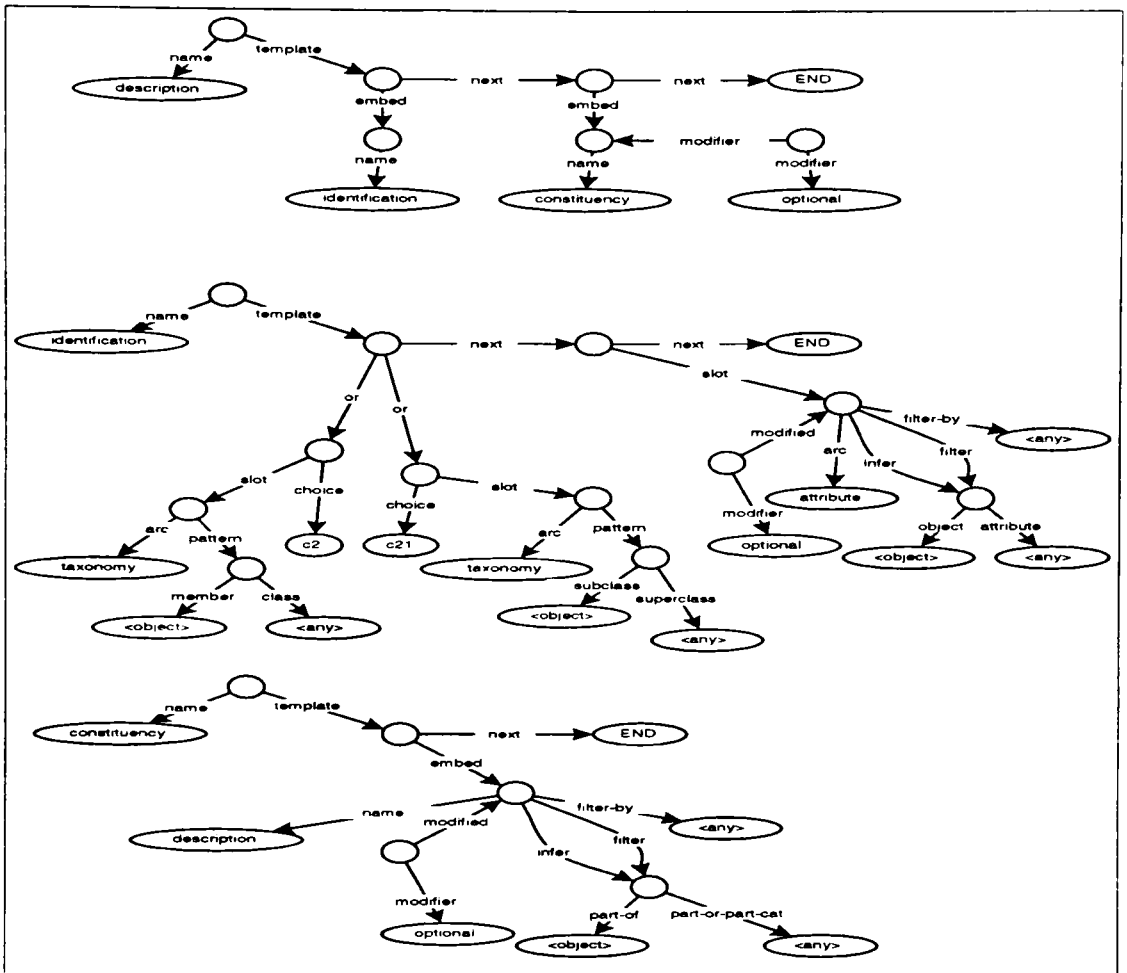


Figure 5.8 - Representation of Schemata in Kalos.

instantiated schema. The nodes at the end of the *choice* arcs are used to label each choice. The conceptual revisor uses these arbitrary node labels to suggest alternative choices.

The reader should compare this structure with the schema notation of Figure 5.7. The *pattern* arc in the SNePS-2.1 representation corresponds to the pattern matching construct, "[...]," in Figure 5.7. *Infer* corresponds to the inferencing construct, ":[...]." Since a SNePS-2.1 *deduce* command does not return the exact set of asserted nodes that match the inference pattern (sometimes SNePS-2.1 pattern nodes containing SNePS-2.1 variables are included), the *filter* arc allows pattern matching to follow inferencing. Typically, the *infer* and *filter* nodes will be the same. The *filter-by* provides the functionality of the $\langle var2 \rangle$ variable. The *modified/modifier* frame allows optional, zero or more, and one or more slots to be specified.

5.4.2 Instantiated Schema Representation

Figure 5.9a and 5.9b show the semantic network elements used to represent instantiated schemata that are built from schema templates. ⟨*Schema*⟩ represents a schema name, such as "description," and ⟨*slot*⟩ represents a slot name, such as "taxonomy." Figure 5.9a shows the representation for an embedded schema. Entries $n_1, n_2, ..., n_i$ represent the entries in the schema. Each entry can represent another schema or a slot. The *object* relation indicates the object associated with the filled schema. Figure 5.9b shows how slots are represented. Node *S* represents a knowledge base concept that fills the slot named ⟨*slot*⟩.

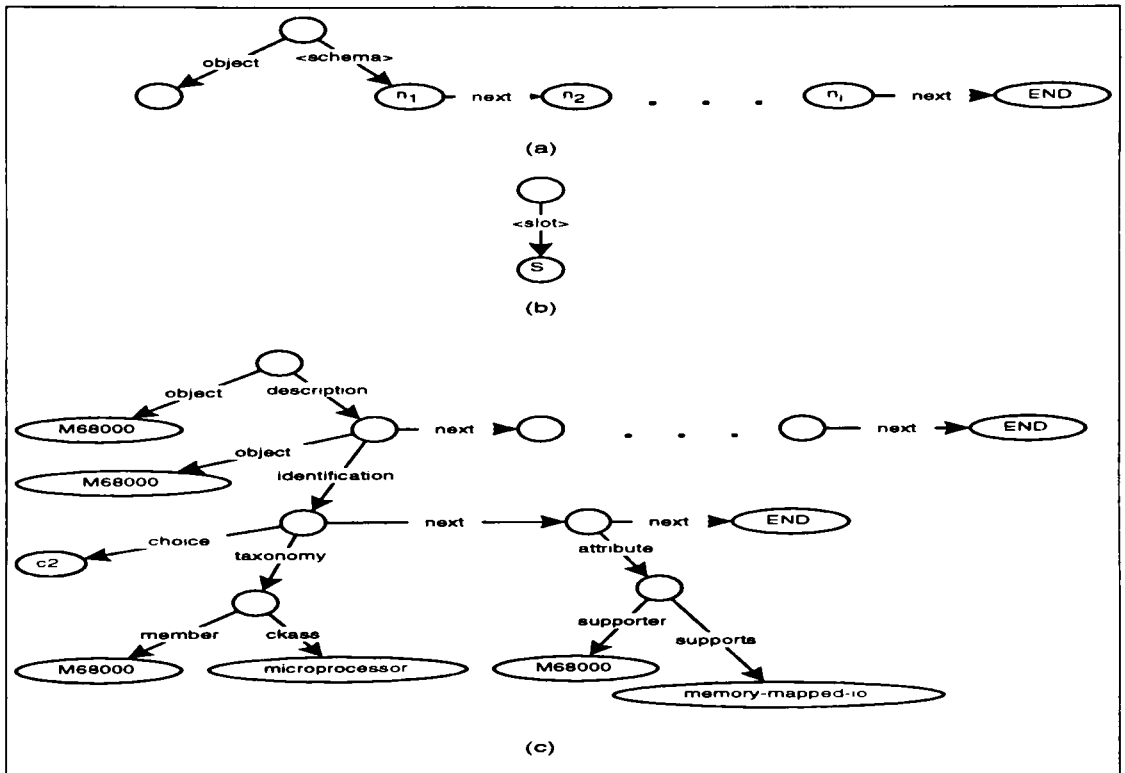


Figure 5.9 - Representation Elements for Instantiated Schemata (a and b) and an Example (c).

Figure 5.9c shows part of an instantiated schema for the M68000. The *description* schema embeds an *identification* schema that has two slots: taxonomy and attribute. A *member/class* frame fills the taxonomy slot and a *supporter/supports* slot fills the attribute slot.

An instantiated schema contains connections to the underlying knowledge from which it was generated so that the revisors can determine the intent of the conceptual text and quickly locate

alternatives for text regeneration. This information is included in two ways. First, arcs used in the instantiated schemata indicate the schemata and slots used. Second, the *choice* arc labels from the schema templates are encoded in the instantiated schemata to indicate which choices were used during its generation. Path-based techniques can be used to locate alternative choices in the schema templates given the choices used to generate the text.

As noted above, section 6.1 discusses how the Kalos deep generator uses these schema templates to construct an instantiated schema that contains an ordered list of concepts that, when surfaced, will satisfy the discourse goal of describing a microprocessor.

5.5 Surface Generation Knowledge

The surface generator is based on a unification grammar that uses lists of attribute/value pairs [Kay 1984, Jacobs 1985]. This section describes the knowledge representation used by the surface generator. Section 6.2 describes the algorithm the surface generator uses to produce natural language text.

5.5.1 Grammar Rule Representation

The surface generator knowledge is made up of grammar rules that represent how to construct sentences from nodes in an instantiated schema. There are three types of grammar rules:

- A top-level disjunction that contains alternatives for generating schema nodes as simple or compound sentences.
- Rules that represent how to surface particular schema nodes as sentences.
- Rules for sub-grammars and lexical entries for objects and concepts.

The building blocks used to construct these grammar rules are described in this section.

Kalos grammar rules are composed of two building blocks: *attribute-value pairs* and *grammar patterns* (which indicate the ordering of surface text). Attribute/value pairs take the form

(*var exp*)

where the lefthand side specifies a grammar variable consisting of a string enclosed in angle brackets, e.g., ⟨*object*⟩, and the righthand side is an expression consisting of a variable, string, or symbol. Surface generation software uses unification to assign values to variables. The variable bindings are kept on a binding list. Backtracking is used when unification fails.

In Kalos, lists of attribute-value pairs are constructed using the building blocks shown in Figure 5.10. The blocks are linked together using the *next* arc to create an ordered list. Figure 5.10.a is an *attribute/value* node. The *attribute* arc points to a node containing a grammar variable. The *value* arc points to a grammar expression which is either a variable, a string literal (e.g., "the"),

or a symbol (e.g., SINGULAR). When the node is evaluated, the surface generation software attempts to assign the value of the value expression to the attribute variable. If the variable is already bound to a different value at this level of the grammar, backtracking occurs; otherwise, the variable/expression pair is added to the grammar binding list.

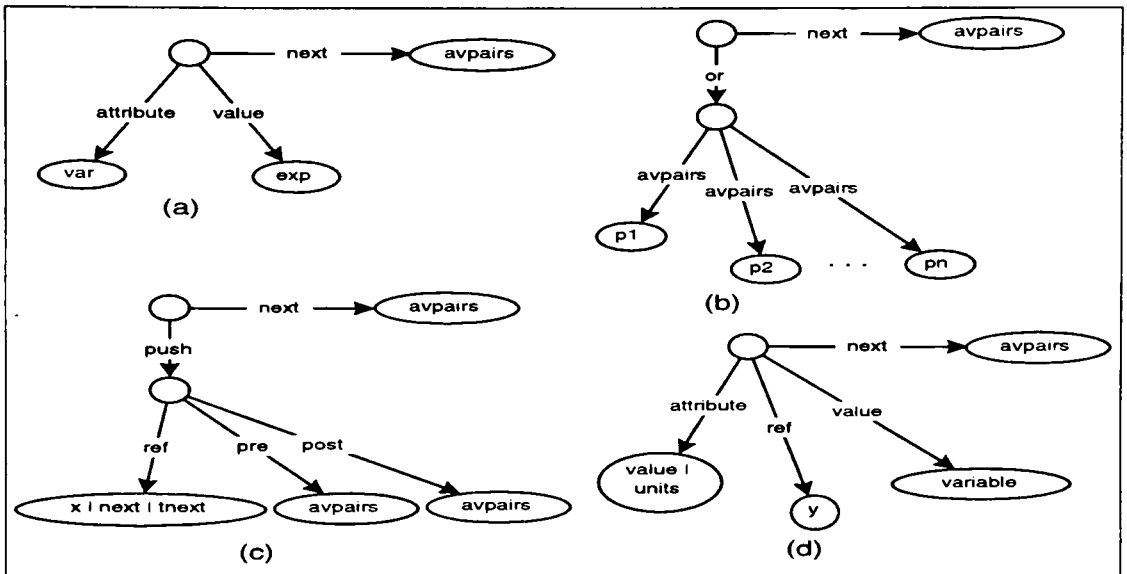


Figure 5.10 - Attribute-Value Pairs Building Blocks

Figure 5.10.b shows the second type of node that can occur in an attribute-value pairs list, an *or* node. An *or* node consists of two or more *avpairs* arcs that point to an ordered disjunction of attribute-value pairs lists, labeled p1, p2....,pn in the figure. When the surface generation software encounters an *or* node, it selects the lowest numbered node it has not yet tried and performs unification on the list. If unification fails, the software backtracks to the next lowest node. The actual ordering of the *or* nodes is based on the numeric part of the node name that SNePS-2.1 assigns. This number increases as nodes are created except if a node is identical to a previous node. In order to ensure that *or* nodes are unique and numbered in the order in which they are written, an extra *unique* arc is added from each *or* node to a uniquely named node. (The *unique* arcs are not shown in the figures.)

The third type of attribute-value pairs node, the *push* node, is shown in Figure 5.10.c. A *push* node is similar to a PUSH arc in an ATN. It allows the grammar to be invoked recursively. The *ref* arc points to a node that contains the name of a grammar rule to use, the symbol *next*, or the

symbol *tnext*. *Next* accesses the next schema slot node and executes the grammar rule associated with it. *Tnext*, i.e., "top-level next," is similar to *next* except that the next schema slot node is surfaced beginning at the top level grammar. (The differences between these two calls is subtle. *Tnext* is useful in compounding operations in that it prevents the binding list used in the caller of the *push tnext* from interacting directly with the binding list of the *tnext* sentence.)

Before push uses the grammar rule selected by its *ref* argument, the attribute-value pairs list specified by the *pre* arc is evaluated. The attribute/value pairs created by this evaluation, along with the current bindings, become the variable/value binding list that is in effect when the *ref* grammar rule is evaluated. After the grammar rule is evaluated, the *post* attribute-value pairs are evaluated. Only the variables bound during evaluation of the *post* list are added to the binding list at the level of the *push* node. The *post* attribute/value pair list has access to a special variable, ⟨*state*⟩, that contains the entire state (variable bindings and surface text) of the called grammar. This variable can be assigned to another variable in the *post* list so that it is added to the caller's binding list. The state information allows the calling grammar to extract the surface text generated by the called grammar. The *pre* bindings are not added at the calling leveling.

The last attribute/value pair building block is the *attribute/value/ref* node shown in Figure 5.10.d. It is used to retrieve the value from a *value/value-of* or *unit/unit-of* knowledge base frame (see Figure 5.2b). Given a knowledge base node *y* pointed to by the *ref* arc which is also pointed by *value-of* and *units-of* arcs, return the value of the node pointed to by either the corresponding *value* or *unit* arc depending on whether the *attribute* arc points to a *value* or *units* node. The value is placed in the variable specified by the node pointed to by the *value* arc.

Figure 5.11 shows the other grammar element, the *grammar pattern*, that is used in Kalos grammar rules. A grammar pattern is an ordered list that indicates the order of the surface text. Grammar patterns are optional in simple grammar rules. There are two types of grammar pattern nodes: *link* and *element*. A *link* node indicates that the surface text produced by a recursive invocation of the surface generator is to placed in the text at this point. The *link* node, indicated by node *p* in the diagram, is typically a variable that contains the state output of a recursive call to the grammar.

The *element* arc points to a node, *r* in the figure, that contains a string literal or a variable. The string or value of the variable is placed in the text at this point. Optionally, an *element* node can contain a *ref* pointer that indicates the object in the knowledge base represented by the surface expression given by the *element* arc. The *ref* information can be used during revision to quickly locate referents for surface expressions.

Figure 5.12 shows the two types of grammar rules used in Kalos. The primary difference

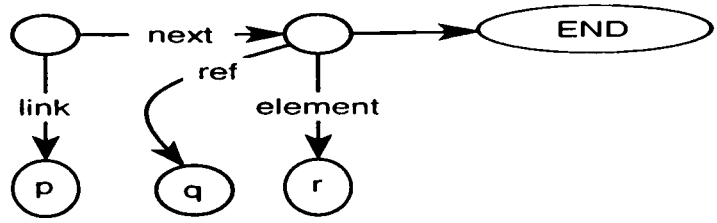


Figure 5.11 - Grammar Pattern Example

between the two is that the first is a disjunction of rules while the second is a single simple rule. Two rule types are used primarily for historical reasons.

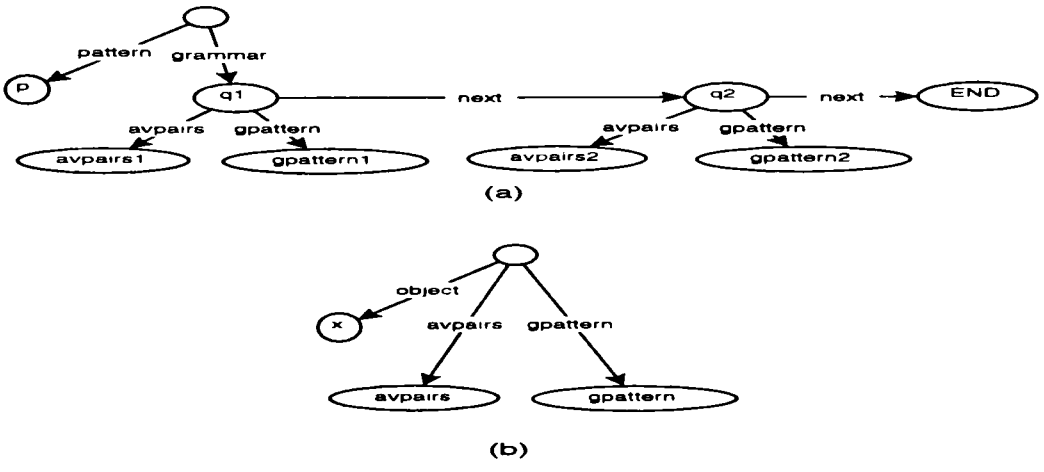
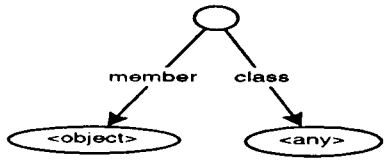


Figure 5.12 - Kalos Grammar Rules: (a) Pattern Rule and (b) Lexicon/Object Rule

Figure 5.12.a shows a *pattern/grammar* node that represents a Kalos grammar rule. Node *p* represents either a *name* node that specifies *top*, the top-level grammar, or a knowledge base pattern for which the rule can be used to produce surface text. Patterns typically use grammar variables. These variables are instantiated with the values taken from the knowledge base and added to the variable binding list before the grammar rule is processed. For example, consider the case where object *D0* is being described. The grammar variable ⟨*object*⟩ would be instantiated to *D0*. When a schema slot containing a *member/class* frame for *D0* is processed, it would match the pattern node



causing variable ⟨*any*⟩ to be instantiated to *data-register*.

Once the *pattern* arc has been processed, the surface generator begins by processing the *q1* node by applying the grammar software to the *avpairs1* attribute-value pairs list and the *gpattern1* grammar pattern list. If backtracking occurs, the next node in the list is processed.

Figure 5.12.b is the second type of grammar rule. This type of rule is used to provide a lexical entry for an object or concept and to encode sub-grammars. The *object* arc points to the node representing some noun or concept in the knowledge base or to the name of a grammar rule. The *avpairs* and *gpattern* arcs associate the object with the rule and surface pattern.

Figure 5.13 is an example of the grammar rule for the object *M68000*. The rule describes how to surface *M68000* depending on the value of the ⟨*nountype*⟩ and ⟨*number*⟩ variables where ⟨*nountype*⟩ is either *noun* or *pronoun* and ⟨*number*⟩ is either *singular* or *plural*. During evaluation of this grammar rule, the surface text for the object and its definite and indefinite articles are assigned to the variables ⟨*expressed*⟩, ⟨*def_art*⟩, and ⟨*indef_art*⟩, respectively.

We will return to this representation in section 6.2 where I show how the surface generator constructs natural language text from an instantiated schema using this representation.

5.5.2 Surface Text Representation

Besides representing grammar rules, the surface generator must represent the surface text that it builds in a way that will allow the underlying knowledge from which it was generated to be easily accessed. Figure 5.14 shows the representation for a surface structure. The *from-grammar* and *from-schema* arcs point into the grammar rule and instantiated schema, respectively, from which the surface text was generated. The *binding* arc points to an attribute-value pairs list that contains values for the variables as a result of the unification process. The *surface* arc points to a filled-in grammar pattern list that gives the surface text.

The surface structure is recursive in that *link* arcs in the surface representation can embed other surface structures which were produced when one grammar rule called another. This structure allows access to the knowledge structures at each level of the surface text. However, for efficiency reasons, only the surface generator-produced binding list at the top level is actually kept in the structure. *Push post* evaluations are used to bring relevant information to the top level.

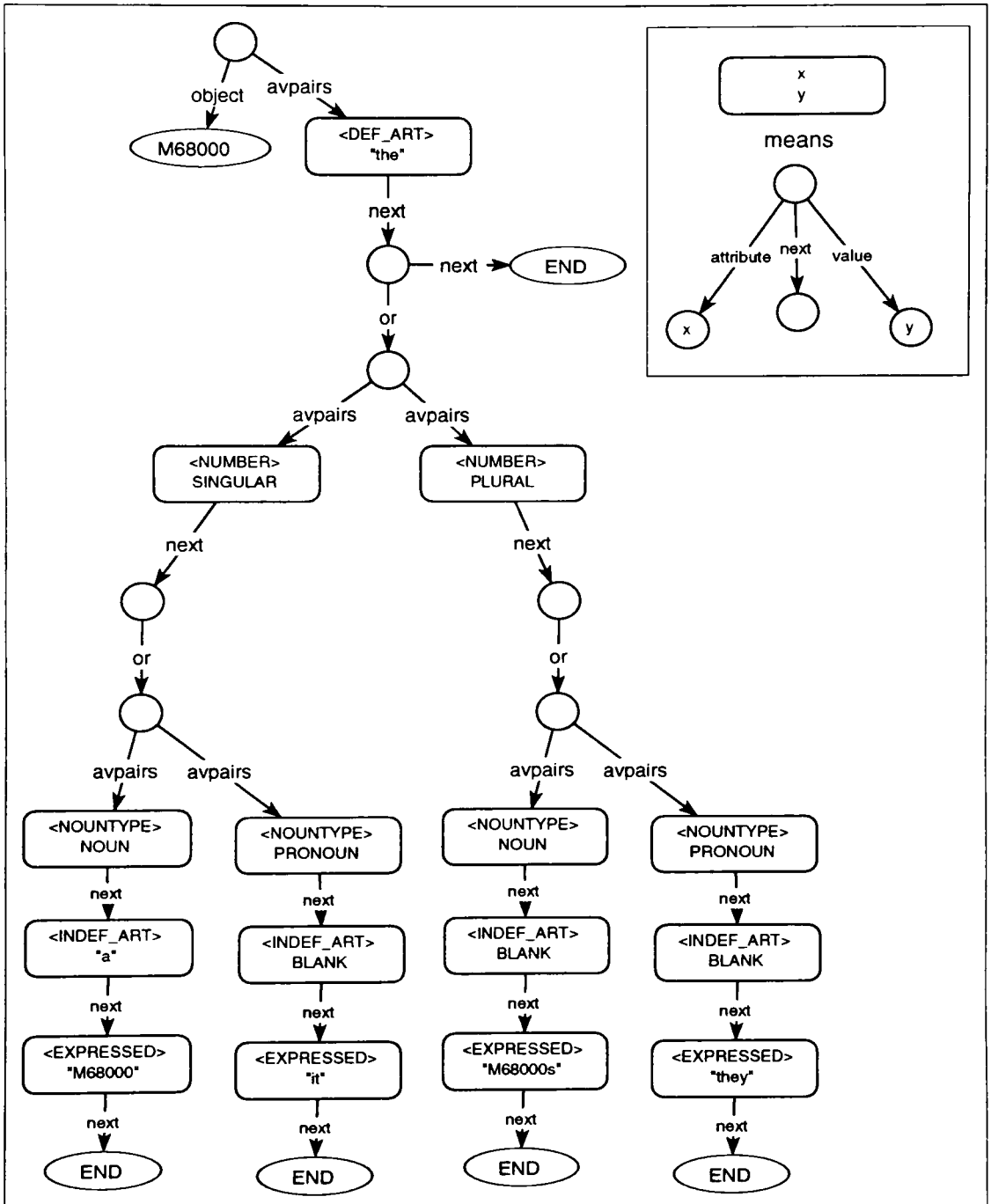


Figure 5.13 - Kalos Grammar Rule for Surfacing *M68000*

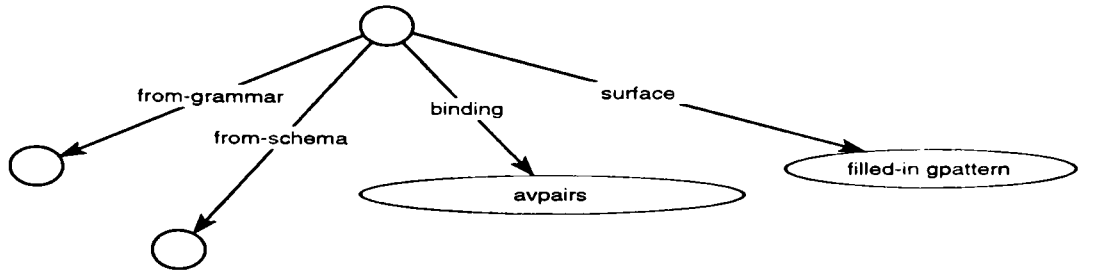


Figure 5.14 - Kalos Representation of Surface Text

5.6 Conceptual Revisor Knowledge

The conceptual revisor runs immediately after the surface generator. It looks for conceptual defects in the text and makes suggestions to the deep generator. The conceptual revision cycle, consisting of the conceptual revisor, the deep generator, and the surface generator, runs until no more conceptual defects are detected. Then the surface text is passed to the stylistic revisor and a stylistic revisor/surface generator cycle is entered. Section 6.3 describes the conceptual revisor in detail. In this section, the knowledge representations used by the conceptual revisor are discussed.

Conceptual revision knowledge is encoded as SNePS-2.1 inference rules. The rules inspect the surface structure of Figure 5.14, the domain knowledge base, and the preferred term knowledge base and makes decisions on removing, adding, and reordering concepts to better meet the discourse goal of describing a microprocessor.

Conceptual revision is performed using SNePS-2.1 pattern matching and inference over surface text representations that include associations to the underlying structures from which the text was generated (grammar rules and instantiated schema). Both the surface text structure (Figure 5.14) and the instantiated schema (Figure 5.9) are recursive to allow the formation of arbitrarily complex sentences and schemata. Kalos uses SNePS-2.1 path-based inferencing to examine these structures. (See the appendix for a description of SNePS-2.1 path-based inferencing.) The ability to use *or* and *kstar* (Kleene star) operators in a path specification allows SNePS-2.1 to locate embedded structures in a flexible manner. For example, to inspect the schema templates for the occurrence of a pattern that matches a *member/class* knowledge base frame, the *slotorschema* path can be used:

```
(define-path slotorschema (compose (or slot embed) (or pattern infer)))
```

By executing a SNePS-2.1

```
(find choice ?choice slotorschema (find member ?x class ?y))
```

the schema templates would be inspected for a *pattern* or *infer* pattern that matched a *member/class* frame. Subgraphs with either a *slot* or *embed* arc followed by either a *pattern* or *infer* arc would match the path. The SNePS-2.1 variable *choice* would be set to the choice label in an *or* subgraph that matched the pattern (see Figure 5.8). This *find* command would match the *taxonomy* slot in the *identification* schema in the figure and variable *choice* would be set to *c2*.

In addition to the knowledge about how to make conceptual revisions, the conceptual revisor adds four lists to the uniform knowledge base to indicate revision suggestions to the deep generator:

- List of slots/embedded schemata to be removed.
- List of new knowledge.
- List of choice point suggestions.
- List of attribute orderings.

The function of each of these lists is described in section 6.3 when individual techniques for making revision suggestions are described.

The list of slot/embedded schema removal suggestions contains slot/node pairs and schema/object pairs. During schema slot filling, the deep generator consults the removal list. Each time some knowledge base node is selected to fill some slot, the removal list is consulted to see if the slot/node pair is on the removal list. If it is, the slot is not filled and the deep generator behaves as if the slot were optional. When a schema is to be embedded for some object, the deep generator checks to see if the schema/object pair is on the removal list. If it is, the schema is not embedded for the object.

The list of new knowledge allows the conceptual revisor to add new knowledge to the domain knowledge base based on deductions it makes. The conceptual revisor adds the list of new knowledge to the uniform knowledge base so that it can be used by the deep generator without the deep generator having to infer it. The conceptual revisor adds new knowledge that will cause slots to be filled that were not filled during the initial deep generation phase.

The conceptual generator also passes a choice point suggestion list to the deep generator during text regeneration. The choice point list consists of object/choice pairs. Whenever the given object is being described, the deep generator tries to use the given choice point. Each choice refers to a schema choice point label that is part of the *or* construct used in schema templates. (See Figure 5.8.) When the deep generator encounters an *or* choice point, it checks the choice list to see if there is a recommended choice for the object being described. If there is, it checks the *or* node choices to see if the suggested choice is one of the available choices. If so, the deep generator uses the suggested choice.

The last list passed from the conceptual revisor to the deep generator is the attribute order list. Whenever the conceptual generator discovers that an attribute of quantity follows an attribute of quality for the same object, it suggests that the order be reversed. The order list consists of SNePS-2.1 nodes representing the proper order of the two attributes and the sentences associated with them. Whenever the deep generator fills an *attribute* slot, it orders the attributes based on this list. Figure 5.15 shows the format of a conceptual revisor suggestion to reorder attributes. The *alter-fill* relation indicates that the frame is an attribute ordering suggestion. This frame says that when attributes represented by knowledge base nodes *node1* and *node2* are used to fill an *attribute* slot of a schema, the deep generator should select *node1* first.

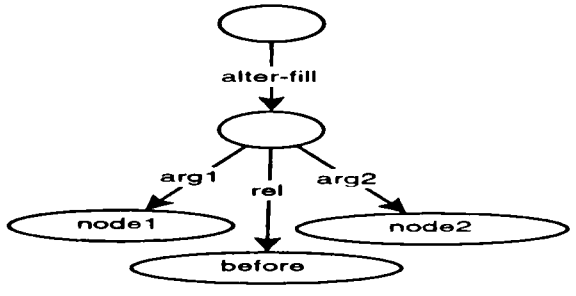


Figure 5.15 - Attribute Ordering Suggestion

5.7 Stylistic Revisor Knowledge

After the text has been revised by the conceptual revisor until no more changes can be made, Kalos enters a stylistic revision cycle. The stylistic revision cycle contains two processes: the stylistic revisor and the surface generator. A stylistic revision cycle starts with the stylistic revisor examining surface text and producing revision suggestions. These suggestions are used by the surface generator to regenerate the text. This cycle is repeated until no new revision suggestions are applicable and the final text is output. This section describes the knowledge representation used by the stylistic revisor. Section 6.4 describes how the revisor functions.

Stylistic revision knowledge is encoded in SNePS-2.1 inference rules. All the rules have the same subgraph structure as a consequence, that of a Kalos revision suggestion as shown in Figure 5.16. All possible revision suggestions are built and then conflicts between suggestions are resolved.

Figure 5.16 shows how stylistic revision suggestions are represented in SNePS-2.1. The *sentence1* and *sentence2* arcs represent the sentences that are affected by the revision. The *force* arc

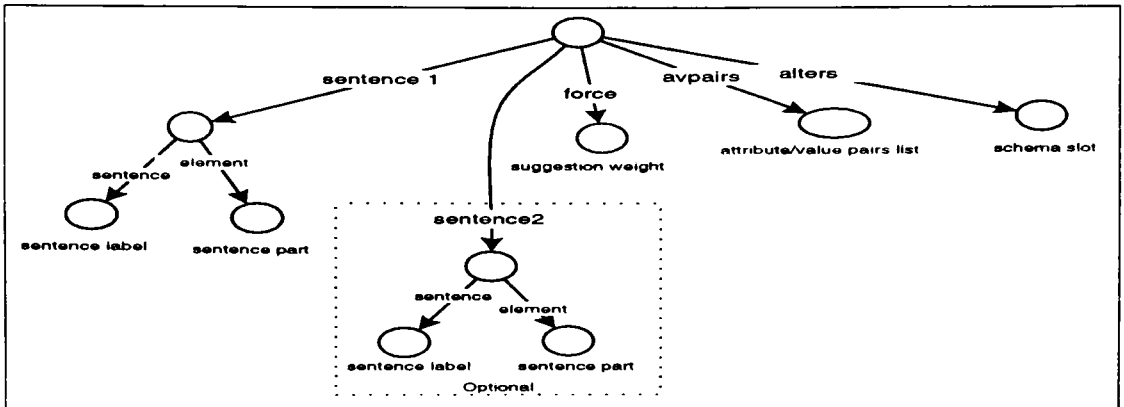


Figure 5.16 - Kalos Representation of Stylistic Revision Suggestions

points to a node holding an integer that represents the "goodness" of the revision suggestion. These three arcs are used to determine if two suggestions conflict, and if they do, which one to use. They are discussed in greater detail in section 6.4.

The *avpairs* arc points to a list of attribute/value pairs (like the *q1*, *q2* list of Figure 5.10a) and the *alters* arc points to an instantiated schema slot. A revision suggestion tells the surface generator that when it surfaces the slot pointed to by the *alters* arc, it should add the attribute/value pairs list pointed to by the *avpairs* arc to the current binding list. This allows the stylistic revisor to direct the surface generator by placing attributes on the binding list that cause backtracking with the effect of causing the surface generator to make a different choice that will improve the surface text.

For example, if the stylistic generator decides that, when a particular slot is surfaced, its subject should be a pronoun, it can build the attribute/value pair (⟨*subjtype*⟩ pronoun) into a revision suggestion for the slot. When the particular slot is surfaced, (⟨*subjtype*⟩ pronoun) will be added to the binding list for the surface generator (see Figure 5.13). When a noun is selected to represent the subject whose ⟨*nountype*⟩ is *noun*, an attempt will be made to set ⟨*subjtype*⟩ to the value of ⟨*nountype*⟩. Since ⟨*subjtype*⟩ has the value *pronoun* from the revision suggestion and ⟨*nountype*⟩ has the value *noun*, a conflict occurs. At that point, the surface generator will backtrack to the last choice point where the noun representation was selected, and the choice to surface the noun as a pronoun will be used.

5.8 Knowledge Base Partitioning

Kalos uses staging and conceptually restricted knowledge bases for efficiency reasons. (See section

4.3.) Early versions of Kalos used the SNePS-2.1 belief revision system to partition the knowledge base during each generation step to limit the number of SNePS-2.1 nodes over which inferencing was performed. As the system grew, memory and computational needs called for a better approach. The current version of the system runs as five separate programs. The first is a knowledge base preprocessing stage, while the remaining stages are the generation tasks of Figure 5.1. The preprocessing stage produces conceptually restricted versions of the knowledge bases and will be described later in this section. The remaining stages are executed in an iterative fashion as noted in the previous sections. The stylistic revisor is further staged. One stylistic revisor step is performed by running the stylistic revisor on pairs of consecutive sentences at a time and combining the results in a revision suggestion conflict resolution phase.

The pattern matching and inferencing a stage performs defines the knowledge frames it will use. Consider the surface generator that uses pattern matching and SNePS-2.1 arc traversal to read the set of grammar rules and an instantiated schema. It does not base any decisions on domain knowledge (other than the frames in the instantiated schema from which it is producing text), schema templates, deep generation rules, conceptual revision knowledge, or stylistic knowledge. By running the stage with only the knowledge that it needs, computational demands are greatly reduced with no reduction in results.

There are two sources of knowledge that a stage can use. The first is static knowledge that is available at the time Kalos starts the task of generating text. The second source of knowledge is intermediate results, such as an instantiated schema, that are created by one stage and used by another. The static sources of knowledge in Kalos are

- Domain knowledge
- Domain lexicon
- Domain-specific preferred terms
- Quantity frames
- Schema templates and deep generation rules
- Surface grammar
- Conceptual revision rules
- Stylistic revision rules

Domain knowledge consists of facts about the microcomputer to be described and general computer knowledge. The domain lexicon is in the same format as surface grammar rules and indicates one or more surface renderings for the objects in the domain knowledge base. Domain-specific preferred terms are kept in a separate knowledge base. Both revisors look for opportunities to apply preferred terms.

Quantities are built using the frames of Figure 5.2b. The SNePS-2.1 node names for nodes r and q in the figure must be the same in all stages and are generated by a SNePS-2.1 macro that

labels nodes based on their order in the knowledge base being loaded. So that these nodes have the same label in all stages, the quantity frames are kept in a separate knowledge base and loaded in each stage before any other references to quantities, insuring that the quantity nodes are the same in all stages.

The schema templates are SNePS-2.1 subgraphs of the form of Figure 5.8. They are contained in a knowledge base that includes other deep generation knowledge such as salience rules. The surface grammar, in the form shown in Figure 5.12, is also contained in a separate knowledge base. SNePS-2.1 rules for conceptual and stylistic revision are kept in separate knowledge bases.

The second type of knowledge in Kalos is produced by one stage and passed to another. The deep generator produces instantiated schemata like Figure 5.9c. These are inspected by both the surface generator and the conceptual revisor. The surface generator produces recursive surface structures like Figure 5.14 and passes them to the conceptual and stylistic revisors. The conceptual revisor produces a set of four lists, indicating deletions, choices, additional knowledge, and ordering information, which are used by the deep generator as revision suggestions during conceptual text regeneration. The stylistic revisor produces a list of revision suggestions. Each suggestion contains a list of attribute/value pairs and an instantiated schema slot. When the indicated slot is being surfaced, the attribute/value pair list is put in the binding list to direct surface generation.

To pass knowledge from one stage to another, the knowledge frames to be passed are first converted into SNePSUL, the SNePS-2.1 user input language, and put in a file that can be loaded by the next stage. Kalos uses a simple procedure to convert knowledge frames into SNePSUL. It handles singlely-rooted subgraphs in which the interior node labels are lost in the conversion process. All Kalos frames that are passed from stage to stage are singlely-rooted with the exception of quantity subgraphs (Figure 5.2b). Each stage loads the small quantity knowledge base separately, so the fact that the quantity subgraphs are not copied is not a problem. The SNePS-2.1-assigned interior node labels are not used by Kalos, so no useful information is lost when a frame is converted to SNePSUL. Anytime a label must be passed to another stage, a user-assigned label is used. For example, the *choice* arcs and labels of Figure 5.8 allow choices to be labeled uniquely without regard to interior node names.

Table 5.1 shows the sizes of the knowledge bases used by Kalos. The size listed for the discourse knowledge base contains schema template nodes and slot filling and salience rules. The linguistic knowledge base is split into the lexicon and grammar rules knowledge bases in the table. Conceptual and stylistic knowledge consists of SNePS-2.1 rules. The sizes for these knowledge bases in the table indicate the number of semantic network nodes used to represent the rules.

Staging greatly improves performance over a single program using a uniform knowledge base.

Knowledge Base	Number of Nodes
Domain	187
Discourse	119
Lexicon	891
Grammar	656
Conceptual Revisor	25
Stylistic Revisor	50

Table 5.1 - Kalos Knowledge Base Sizes

Even with staging there are limits to the knowledge base sizes that will support reasonable inference times for a given hardware/software configuration. In Kalos, additional measures were needed to support text generation for the M68000 knowledge base. The technique I used to further limit the size of knowledge bases for any given stage is the development of conceptually restricted knowledge bases. In Kalos, conceptually restricted knowledge bases are called surrogates.

Kalos uses five surrogates: *Lexicon Surrogate*, *Linguistic Surrogate*, *Grammar Rule Surrogate*, *Reduced Surface 1*, and *Reduced Surface 2*. Table 5.2 shows compares the sizes of the surrogates and the number of semantic network nodes that do not have to be loaded when the surrogate is used in the current version of Kalos. In the case of the surface text structure surrogates, the sizes are estimates for single sentences.

Surrogate	Derived From	Size	Savings
Lexicon	Lexicon & Domain	9	882
Linguistic	Lexicon & Grammar	7	1540
Grammar Rule	Grammar	49	607
Surface 1	Surface Structure	195	145
Surface 2	Surface Structure	12	28

Table 5.2 - Kalos Surrogates

Surrogates can be created from static knowledge bases and from partial results. Two surrogates are created from static knowledge bases for use by the conceptual revisor. The first, the lexicon surrogate, checks the domain knowledge base and the lexicon for occurrences of subordinate classes whose lexicon entry contains the lexicon entry for its superclass (e.g., "M68000 data register" and

"data register"). The linguistic surrogate, the second surrogate used by the conceptual revisor, is built from the preferred terms, the lexicon, and grammar rule knowledge bases to extract the information that the conceptual revisor needs to perform preferred word and phrase processing. The revisor needs to know which objects and knowledge frames cause preferred words and phrases to be produced. (See section 6.3.2.) Given this information, it can attempt to build an instantiated schema that contains those objects and frames.

These two surrogate knowledge bases are extracted using SNePS-2.1 inferencing from the grammar and lexicon knowledge bases before text generation begins. Since there are typically a small number of preferred words and phrases and redundant lexicon entries, the surrogate knowledge bases are much smaller than the grammar and lexicon knowledge bases from which they were built. Since the conceptual revisor must consult several knowledge bases, the savings in being able to load the surrogates instead of the knowledge bases from which they were generated is significant. A reduction in over a thousand semantic network nodes is seen in the current implementation of Kalos.

When the conceptual revisor inspects the surface text structure (Figure 5.14), it only needs access to the *surface* and *from-schema* parts of the structure. The *binding* and *from-grammar* parts of the structure can be very large, and they are not needed for conceptual revision. To reduce the knowledge base size for the conceptual revisor, a surrogate for the surface text structure, labeled *surface 1*, is loaded. This surrogate is produced when the SNePSUL version of the surface structure is produced by the surface generator. The surface generator produces two versions of the surface structure: one complete version for the stylistic revisor and an abbreviated surrogate for the conceptual revisor that does not contain the *binding* and *from-grammar* subtrees.

The stylistic revisor uses three surrogate knowledge bases. The first is the preferred term (linguistic) surrogate used by the conceptual revisor. The second, the grammar rule surrogate, is derived from the surface generator grammar rules and contains information about what structures can be produced by grammar rule alternatives. The $\langle major\_type \rangle$ variable in each rule indicates what structure, e.g., sentence or noun, can be produced by each grammar rule choice. The stylistic revisor uses this information to limit its suggestions to possible constructions. For example, before suggesting that a sentence be transformed into a noun phrase, the revisor checks the surrogate to see if the change is possible.

The stylistic revisor uses a third surrogate knowledge base, labeled *surface 2*, derived from the output of the surface generator (see Figure 5.14). The long binding lists are replaced with more efficient structures that contain only the binding list variables used by the stylistic revision rules.

SNePS-2.1 path-based inferencing is used in Kalos to improve efficiency. (Path-based infer-

encing is described in Appendix A.) A surface text contains a long attribute/value pairs list that contains the binding list that was created during surface generation. Revision rules need to look into the binding list to determine the value of variables which describe the surface text. For example, ⟨*subjref*⟩ has a value that indicates the object that the subject of the sentence refers to. The attribute/value pairs that refers to ⟨*subjref*⟩ can be anywhere along the attribute/value pairs chain. It is desirable to be able to locate the value of ⟨*subjref*⟩ without performing SNePS-2.1 inferencing.

A path-based rule of the form

```
(define-path pathav (compose binding (kstar next)))
```

is used to locate variable bindings. So a revision rule can test for

```
(build name *name pathav (build attribute <subjref> value *ref))
```

where *name is a SNePS-2.1 variable containing the name of the sentence being considered. The *pathav* relation is determined by pattern matching alone. The *ref variable is set to the object to which the subject of the sentence refers. The program that produces the surface text surrogate for the stylistic revisor uses this path-based inferencing to extract binding list variables.

5.9 Summary

This chapter has described the knowledge representation schemes used in Kalos. All the knowledge used by the system is encoded in a uniform knowledge base to which SNePS-2.1 inference and pattern matching can be applied. Schemes for dealing with the computational burdens of a uniform knowledge base were also presented.

The next chapter describes how Kalos uses the knowledge discussed in this chapter.

Chapter 6
Kalos - Knowledge Intensive Generation with Revision

I ride tandem with the random,
Things don't run the way I planned them.

Peter Gabriel

In this chapter, we describe how Kalos generates and revises text. To illustrate the techniques used in Kalos, we will present a single example that we will follow through all stages of generation. As each generation stage is described, we will use this example to illustrate generation and revision processing. The example is taken from the Kalos-generated opening paragraphs that describe the Motorola M68000. The deep and surface generators are described, followed by a description of the conceptual and stylistic revisors.

6.1 Deep Generator

The Kalos deep generator uses discourse schemata to construct an ordered list of concepts that satisfy the goal of describing an object [McKeown 1985, Paris 1985]. A schema is a pattern of text that accomplishes a particular discourse goal. For example, one way to describe an object is first to relate it to a classification system, then tell what it is a part of, and finally describe its attributes (e.g., size and color). Deep generation was discussed in section 2.1.

In Kalos, the deep generator is responsible for selecting and ordering concepts to meet the discourse goal of describing a microprocessor and its associated parts. It uses discourse patterns that first describe the microprocessor in terms of its relationship to a taxonomy of computer parts. Next, attributes of the microprocessor are listed. Then the parts of the microprocessor are listed, followed by a description of each of the parts.

Besides the discourse pattern, the deep generator needs additional knowledge to select concepts to be included in the description. For example, when describing a microprocessor, only salient attributes about it should be included. In Kalos, attributes of quantity (e.g., size and number) and atypical attributes are considered salient. Salience rules are described in section 6.1.1.

Deep generators are often the most complex part of connected text generation systems. By contrast, Kalos's deep generator is relatively simple. This simplicity reflects not restricted capacity but deferred work. Many decisions typically handled by the deep generator are postponed in Kalos for consideration by the revision module. Hence, the deep generator does not have to decide the order of attributes, detect redundancy, consider domain specific words and phrases that affect concept selection or order, or determine whether lists are too long to surface. The deep generator operates in the same mode during draft generation and during regeneration requested by the revisor.

During regeneration, it checks a list of revision suggestions that potentially modify its decisions; during initial generation, the suggestion list is empty.

6.1.1 Schema Filling Mechanism

The slot filling mechanism of Kalos could have been encoded directly in SNePS-2.1 inference rules; however, this approach would result in complex inference rules and the obscuration of the schema template knowledge. Instead, a slot filling mechanism was created as an extension to SNePS-2.1. Using this scheme, the schema templates are encoded in the format of Figure 5.8. This concise, declarative format is easier to maintain and modify than slot filling knowledge represented exclusively as SNePS-2.1 inference rules.

The Kalos slot filler is invoked with a node representing the object to be described. It uses the schema templates to build an instantiated schema that describes the object at the conceptual level. The schema templates indicate how slots or recursive schema fillings are to be handled. There are three ways that the schema filling software is guided by the templates:

- Absolute schema embedding.
- Filling based on pattern matching.
- Filling based on inference.

In the first case, an object is being described and a particular schema is embedded for the object. In the second case, information to fill a slot or the object for which a schema is to be embedded can be found in the knowledge base by pattern matching using the SNePS-2.1 *find* and *findassert* commands. In the third case, information on filling a slot or finding the objects for which a schema should be embedded is found by inference using the SNePS-2.1 *deduce* command. Inferencing works with a number of SNePS-2.1 rules that allows the deep generator to locate salient information and determine the level of description (i.e., part or category level). (Inferencing could be used exclusively; however, pattern matching is included for efficiency reasons. Since many types of knowledge frames about the microprocessor and its parts are explicitly encoded, these can be found quickly through pattern matching.)

Referring to Figure 5.8, entries in schema templates are indicated by *slot* or *embed* arcs. *Slot* arcs have associated *pattern* or *infer* subgraphs to tell the slot filler how to build a SNePS-2.1 *findassert* or *deduce* command, respectively, which, when executed, will produce a list of nodes to fill the slot. In the case of pattern matching, the node list returned is used directly; however, the node list returned by inference can be filtered by *filter* and *filter-by* constructions. *Filter* subgraphs tell the slot filler how to build a *findassert* command to locate nodes to fill the slot after inferencing has been performed. (This feature was needed with an older release of SNePS-2.1 that would return

variable nodes in the list returned by deduce.) The *filter-by* node indicates that a Kalos variable that was part of the inference/pattern match contains the nodes to fill the slot.

In the case of *embed* arcs, two forms are allowed. If the *embed* subgraph contains no *pattern* or *infer* subgraphs, then the schema is embedded for the current object. However, if a *pattern* or *infer* subgraph is given, then the operation is performed as in the case of *slot* arcs, and the list of nodes produced by the inference or pattern matching indicates the objects for which the indicated schema is to be embedded. This feature is used to embed *description* schemata for the part and part categories of an object.

If the schema filler performs the indicated pattern match or inference for a node and finds no nodes to fill a slot, it checks the status of a *modifier/modified* frame for the schema template entry. If the frame indicates that the slot is optional, the slot is not filled and processing continues to the next schema template entry. Otherwise, the schema filler aborts because a required slot cannot be filled. This approach is adequate for Kalos because the knowledge base contains facts for the required slots for all the objects that are considered. A backtracking mechanism could be used; however, the blind selection of alternatives would likely produce poor results. Another approach would be to defer handling of unfilled, required slots for consideration by the conceptual revisor, which could consider alternatives in the light of full system knowledge.

Determination of salience is an important task of the slot filling mechanism. The three premises of salience determination used by Kalos are typicality, the importance of quantities, and the importance of structure. Salience based on typicality is discussed in section 5.3.1. Kalos contains inference rules that search for attributes that are atypical when filling *attribute* slots.

Besides the salience rules based on typicality, Kalos also considers quantities, such as the number and sizes of registers, to be salient. This approach is based on the assumption that an engineer reading a typical microprocessor users' guide would be familiar with common microprocessor parts, such as registers and buses; however, an engineer would be interested in data path sizes, the number of registers, and other quantities, so these are considered salient. Inference rules are used to select quantities as salient attributes during schema slot filling.

The constituency of a microprocessor is also salient. An engineer who is familiar with typical microprocessors would be interested in the parts of a new microprocessor. For example, the types of registers and the instruction set are fundamental in understanding a microprocessor. This salience information is included in the structure of the schema templates and the schema rules that decide whether parts or a category to which they belong are to be described.

Although these rules are useful in selecting salient attributes of an object, additional research is needed to produce better salience rules. To make up for deficiencies of the current rules, a few

additional domain-specific salience rules were added to mark certain types of facts as salient to the description of a microprocessor. For example, the fact that the M68000 supports memory-mapped I/O is considered salient.

6.1.1.1 Kalos Schemata Overview

Kalos uses five schemata. They are tailored to the task of describing a microprocessor and other artifacts that can be decomposed into constituent parts.

- Description
- Identification
- Constituency
- Process
- Compare and Contrast

The *description* schema is filled to describe a microprocessor or one of its parts in detail. This schema contains no slots, but embeds other schemata to produce the description. It first embeds an *identification* schema for the object being described. The *identification* schema has slots for giving taxonomic information about the object, naming its members if it is a class, and listing salient attributes about it. The *constituency* schema is used to list and describe the parts of an object or category. When the *constituency* schema is filled for an object, the deep generator decides for each part of the object whether to describe the individual part or a subordinate category to which it belongs. Kalos uses the rule that subordinate categories are described except in the case of unique parts. Subordinate categories are used to handle groups of items, like data registers, that can be described as a group, instead of describing each data register individually. The knowledge base contains frames that indicate if the elements of a subordinate category are homogeneous, e.g., the category of M68000 data registers, or heterogeneous, e.g., the category of M68000 address registers. (Two of the address registers act as stack pointers.)

The *constituency* schema does not embed *description* schemata for the parts of homogeneous categories. After describing the category and listing its members, a description of the individual members of the category would offer no new information. This approach causes Kalos to describe the M68000 data registers as a group instead of individually. These registers are essentially the same except for their name and the bit combinations used in instructions to access them.

The deep generator handles part breakdown for heterogeneous categories differently. Since the elements of a heterogeneous category differ in some significant way, it is not sufficient just to describe the heterogeneous category. If the heterogeneous category is divided into a set of subordinate categories, the subordinate categories can be described; otherwise, the individual parts of the category must described. Consider the heterogeneous category for M68000 address registers.

It is divided into a homogeneous subordinate category and a heterogeneous subordinate category: one for A0-A6 which are pure address registers and one for A7 and A7' which also serve as stack pointers. Instead of describing each of the nine registers individually, Kalos describes A0-A6 as a group and A7 and A7' as a group. Since A7 and A7' differ, these are eventually described separately.

In Kalos, the *constituency* schema is used in two modes. In overview mode, it merely lists the parts or part categories of the object being described and gives a few important attributes about each part or category. Overview mode is used in the opening paragraph of a description of an object to list its parts. In detailed mode, the *constituency* schema embeds a *description* schema for each part or category it listed in overview mode. The embedded *description* schemata are filled to give a full description of all the parts and part categories of an object.

The *process* schema is used to give functional descriptions of objects. In Kalos, functional descriptions are only given for machine instructions since typically microprocessor users' guides follow this format. The typical reader of this type of manual is assumed to know how parts of computers, e.g., registers, work.

The *compare/contrast* schema captures a discourse pattern for comparing two different microprocessors. It is not used in the current version of Kalos, but is included for completeness.

During initial generation, Kalos fills optional slots if it can. When confronted with a choice point that is a disjunction of slots or recursive schema embedments, Kalos selects the first slot or embedment that it is able to fill. During text regeneration, the conceptual revisor can change the decisions made during initial deep generator by removing filled slots, selecting new choice points, reordering concepts, and indicating that some new facts are salient.

To limit the complexity of Kalos and to allow us to focus our energies on revision techniques, I have not explicitly encoded knowledge about the intentions, beliefs, and knowledge of the author and reader, and I did not design Kalos to reason about this type of knowledge. The intention to communicate, beliefs about the intention of the reader to learn new information, the effect of new knowledge on the model of the reader, and assumptions about the intended audience's knowledge are buried in the design of the system. However, as Gabriel [1988] points out, little reasoning about shared knowledge and the effect of new facts on the reader is required in generating simple descriptions.

6.1.1.2 Kalos Schemata Description

Section 5.4 described the representation of discourse schemata in Kalos. Figure 6.1 shows the five Kalos schema templates in detail: *description*, *identification*, *constituency*, *process*, and *compare/contrast*. *Constituency* and *identification* are further divided for handling specific situations. *Constituency* is divided into *constituency overview* for listing the parts of an object for an introductory paragraph and *constituency detailed* for giving detailed descriptions of the parts of an object. *Identification* is used to produce full descriptions, while *short identification* is used for producing descriptions for objects in lists. Schemata definitions take the form described in section 5.4.1.

The *description* schema is used to describe an object in terms of its place in the taxonomy to which it belongs and in terms of its parts. If the object performs some action, then a functional description of that action may be included. Optionally, an object may be compared to a similar object. The *description* schema embeds other schemata to select the knowledge required to describe a particular object. It is used at the top level to describe a microprocessor and is called at lower levels to describe the various parts of the microprocessor.

The *identification* schema selects knowledge to describe an object in relation to the taxonomy to which it belongs and to salient attributes. One of four different knowledge frames representing the object's relationship to the computer taxonomy is selected to fill a *taxonomy* slot. The four choices are

- member/mainclass
- member/class
- subclass/superclass
- arg1/isa/arg2

The first two choices and the last are used when an object is being described, while the third is used when describing a category. Some objects belong to more than one natural category, such as the M68000 A7 register which is an address register and a stack pointer. In Kalos, one of the natural categories is selected as the main class by inferencing based on its relationship to subordinate categories. In the case of A7, address register was selected as the main class so that it could be described in the context of the other address registers. (The fact that A7 belongs to two natural categories is salient, so the Kalos deep generator states that A7 is also a stack pointer.) If an object belongs to more than one category, the main category is listed in the *taxonomy* slot. If not, the taxonomy slot is filled with the object's only member/class frame.

If the object being described is a category, than its *taxonomy* slot is filled with a frame indicating its superclass. This slot allows Kalos to say "the status registers are control registers."

The final option allows the object to be related to a subordinate class to which it belongs.

```
Schema Description X
  -> Identification X
  {-> Constituency_Overview X}
  {-> Process X}
  {-> Constituency_Detailed X}
  {-> Compare/contrast X}

Schema Identification X
  (or (Taxonomy X = [member X/mainclass ?])
      (Taxonomy X = [member X/class ?])
      (Taxonomy X = [subclass X/superclass ?])
      (Taxonomy X = :[arg1 X/rel isa/arg2 ?]))
  {or (Instance_name X = :[object X/instance-name ?])
      (Homogeneous Instance X =  :[homogeneous-category X/homogeneous-part ?])
      (Instance X =  [arg1 ?/rel isa/arg2 X])
      (Subordinate Category X = :[sub_cat ?/category X/number ?N])}
  {Attribute ?  = ?, :[object X/attribute ?]}*

Schema Short_Identification X
  (or (Taxonomy X = [subclass X/superclass ?]
       Short_category_attribute ?  = ?, :[object X/short-cat-attr ?])
      (Taxonomy X = [member X/mainclass ?]
       Short_Attribute ?  = ?, :[object X/short-attr ?])
      (Taxonomy X = [member X/class ?]
       Short_Attribute ?  = ?, :[object X/short-attr ?]))

Schema Constituency_Overview X
  {or (-> Short_Identification ?  = ?, :[part-of X/part-or-part-cat ?])
      {Category_Division ?  = ?, :[part-of X/homogeneous-subcat ?]
       {or (Category_Division X, :[part-of X/heterogeneous-category ?])
           (-> Short_Identification ?  = ?, :[part-of X/heterogeneous-part ?])}}*}*

Schema Constituency_Detailed X
  {or (-> Description ?  = ?, :[part-of X/part-or-part-cat ?])
      (-> Description ?  = ?, :[part-of X/homogeneous-subcat ?]
          {or (-> Description ?  = ?, :[part-of X/heterogeneous-category ?])
              (-> Description ?  = ?, :[part-of X/heterogeneous-part ?])})}

Schema Process X
 {Action X = ?, [object X/actions ?]}+
 {Example X = [object X/example ?]}

Schema Compare/Contrast X
 OtherObject ?Y = ?Y, [object X/comparable-object ?Y]
 Similar X = [object X/otherobject ?Y/similar-attributes ?]
 Different X = [object X/otherobject ?Y/different-attributes ?]
```

Figure 6.1 – Kalos Discourse Schemata

Normally, objects should be related to the natural category to which they belong; however, the conceptual revisor can suggest this slot be filled for an object instead of the one using the member/class frame. For example, under initial generation, Kalos says that "the M68000 is a microprocessor," relating the M68000 to its natural category. While processing preferred words and phrases, the conceptual revisor attempts to use a phrase that categorizes the M68000 based on its data bus size, so it suggests that the deep generator relate the M68000 to the subordinate category "16-bit microprocessor." This suggestion causes "the M68000 is a 16-bit microprocessor" to be generated.

Next, an *instance name*, *instance*, *homogeneous instance*, or *subordinate category* slot can be filled if a category is being described. These slots list either the parts in a category or its subordinate categories. If the knowledge base contains a short name for listing the parts in a category (e.g., "D0-D7" for the M68000 data registers), then this name is used. If a short name if not available, the parts of the category are listed. The conceptual revisor can cause the *subordinate category* slot to be filled with a list of categories instead of individual part names if there are a large number of parts. In the case of M68000 instructions, it is not practical to list all eighty-two instructions in a single sentence or even in eight-two consecutive ones, so the *subordinate category* slot can be used to list categories of instructions ("arithmetic," "logical," "control," and "branch") instead of the individual instructions. (See section 6.3.4).

This schema also contains a slot for salient attributes of the object being described such as speed and size. These attributes are selected by inference rules. The *short identification* schema, a subtype of *identification*, selects taxonomy information and important attributes for lists of parts and part categories of an object used in overviews. It contains *short attribute* and *short category attribute* slots that are filled by inferencing and select only attributes related to size and the number of objects and category members. *Short attribute* is used when describing objects and *short category attribute* is used when describe a category.

The *constituency* schema is divided into two subtypes: *constituency overview* for building overview descriptions of parts of an object and *constituency detailed* for describing the parts of an object in detail. The former schema lists the parts, their sizes, and their numbers. The latter describes either the parts or their categories in detail. Both schemata must take into account whether a part or its category should be described. Every part of the object being described must be represented either directly or by having a category to which it belongs represented.

In the *constituency overview* schema, the first clause, the embedding of a *short identification slot*, is used when a concrete object is being described. This slot is filled with a list of parts and part categories that represents all the parts belonging to the object. (Each part is represented only once, either by being listed or by having a category to which it belongs listed. Categories are used to describe groups of items, like the eight data registers.) The remaining clauses are used when a subordinate category, such as M68AR (the group of M68000 address registers), is being described. Each category is broken down into homogeneous subordinate categories and either heterogeneous parts or categories. Each part of the category is represented only once as a part or as a member of a homogeneous category or a heterogeneous category. For example, M68AR can be decomposed into a homogeneous subordinate category *pure-addr-reg* (i.e., A0-A6, the pure address registers) and a heterogeneous subordinate category *spec-addr-reg* (i.e., A7 and A7', the special address registers)

or into *pure-addr-reg* and heterogeneous individuals A7 and A7'. The former form is used unless the conceptual revisor suggests otherwise.

The *constituency detailed* schema divides the parts of the object being described into the same set of parts and subordinate categories as the *constituency overview* schema. For each part or subordinate category, it embeds a *description* schema to produce a description for the item. When the deep generator attempts to fill a *constituency detailed* slot for a homogeneous category, no *description* schemata are embedded for the parts of the homogeneous category because descriptions of its individual parts would add nothing new to the description. For example, no *constituency detailed* schema is embedded for D0 when the homogeneous category "M68000 data-register" is described.

The *process* schema selects knowledge that describes a set of actions such as the actions that occur during the execution of a machine instruction or exception handling. This schema is filled with an ordered list of concepts describing the actions. An optional *example* slot can be used to give a specific instance of the actions being described. The *compare/contrast* schema extracts information about an attribute of the object being described that differs from a typical attribute value. This schema includes slots for the object being described, its attributes and values, the second object in the comparison, and its attributes and values.

Figure 6.2 shows the SNePS-2.1 semantic network for the *identification* schema. The representation used follows that described in section 5.4.1. (See Figure 5.8.)

6.1.2 Deep Generation

To illustrate deep generation in Kalos, consider generation of a description of the Motorola M68000 microprocessor. The deep generator begins by locating the schema template for the *description* schema. The first action is to embed an *identification* schema to select concepts from the domain knowledge to identify the M68000 in terms of its location in the Kalos taxonomy and to list its attributes. The template for the *identification* schema is located, and the deep generator builds an instantiated schema from this template.

The first step in filling the *identification* schema is to fill in the *taxonomy* slot for the M68000. A *taxonomy* slot is filled with a *member/class* frame in which the M68000 is pointed to by the *member* relation. There is exactly one frame in the Kalos taxonomy that meets this criterion. The node pointed to by the *class* relation represents the natural category to which the M68000 belongs. In the Kalos taxonomy, the natural category to which the M68000 belongs is *microprocessor*. To locate the *member/class* frame, Kalos uses SNePS-2.1 pattern matching invoked by the SNePS-2.1 *findassert* command.

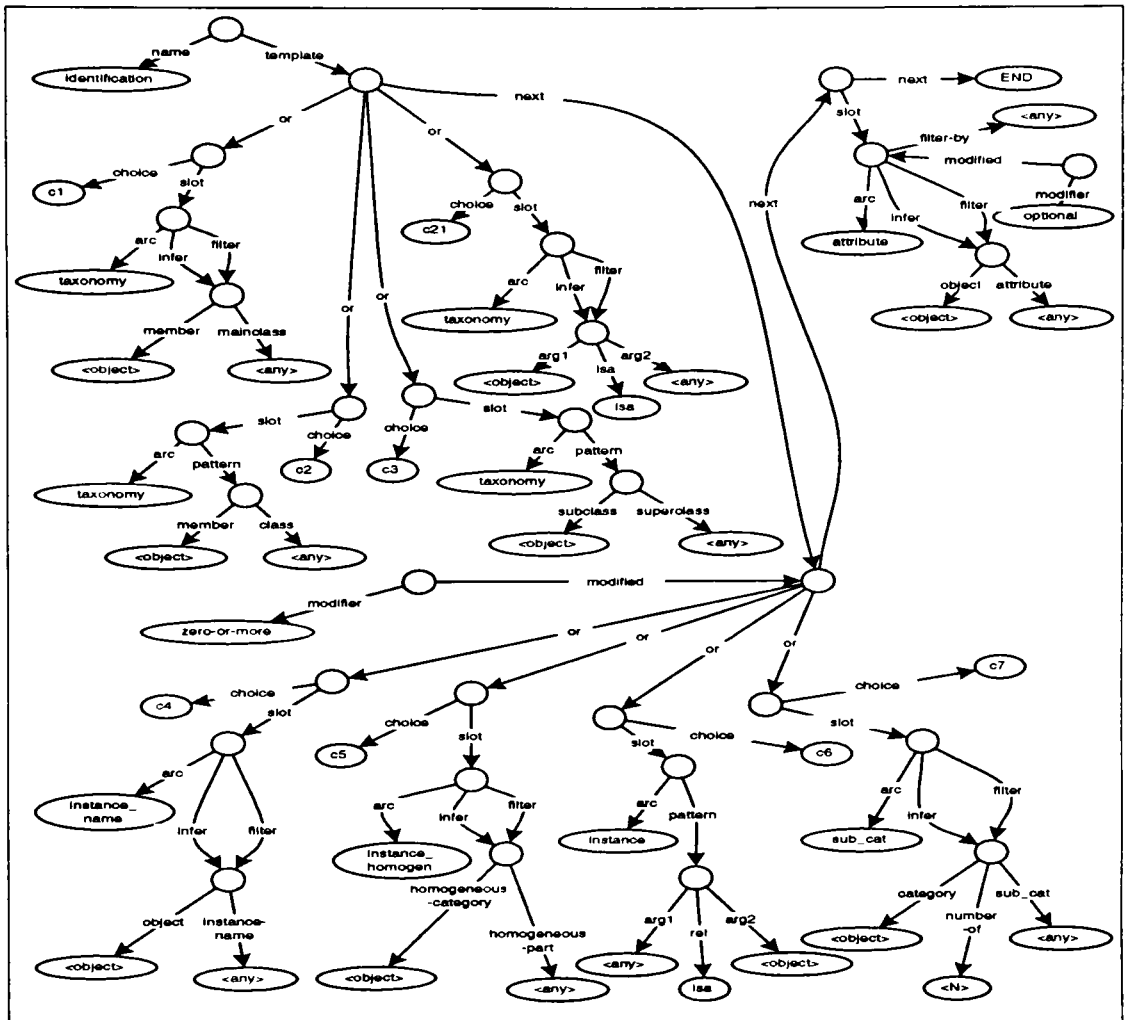


Figure 6.2 - Representation of *Identification* Schema in Kalos.

The next group of slots indicated by the "or" operator are not filled because they refer to subordinate categories, not objects such as the M68000. These slots are used to list the members of a category. Since this is an optional slot, the filled-in template will just not contain this slot; however, if the slot were not optional and could not be filled, slot filling would be aborted. This scheme is adequate for Kalos, but in a more complicated system, backtracking could be implemented.

Next, the *attribute* slot would be filled. In Kalos, SNePS-2.1 rules are used to infer which facts about the object being described are salient attributes. During initial generation, the only attribute of the M68000 considered by the deep generator to be salient is the fact that the M68000 supports memory-mapped I/O. Additional attributes may be suggested by the conceptual revisor.

After constructing a filled-in *identification* schema for the M68000, Kalos continues filling the *description* schema for it. The M68000 has now been described in terms of its relation to the computer taxonomy and its attributes. Next, a list of its parts are produced by embedding a *constituency overview* schema. The filled-in *constituency overview* schema embeds a *short description* schema for each part or part category of the M68000. The inference rule for the *part-of/part-or-part-cat* frame selects the individual object for unique objects and subordinate categories for parts that belong to a group.

The Kalos knowledge base contains no process information for the M68000, so the optional *process* schema embedded from the *description* schema is ignored. Next, the deep generator describes each part of the M68000 in detail using the *constituency overview* schema. *Description* schemata are embedded for parts and part categories such that each part is either described or is a member of a subordinate category that is described. Only the first slot is applicable to the M68000 since it is an object. The remaining slots are used when describing a subordinate category.

Finally, the deep generator attempts to embed a *compare/contrast* schema to compare the M68000 object to an object belonging to the same natural category. This schema is currently not used in Kalos but is included for completeness. It would be used to compare the microprocessor with other, competing microprocessors.

Continuing with the example of generating a description of the M68000 microprocessor, Figure 6.3 shows part of the instantiated schema for the M68000. The *description* schema embeds an *identification* schema which consists of *taxonomy* and *attribute* slots the for M68000. Next, a *constituency overview* schema is embedded to give an overview of the parts of the M68000. (Parts of the network are omitted and are indicated with dotted lines and nodes.) The figure shows the embedded *short description* schema for the subordinate category *M68AR*, the set of M68000 address registers. This schema has three slots. The first contains taxonomic information about the group of registers. Two *short category attribute* slots follow. The first contains the size of members in the *M68AR* subordinate category, and the second gives the number of members. (The taxonomy slot will be removed by the conceptual revisor because it produces surface-level redundancy. See section 6.3.1 for details.) The two *short category attribute* slots allow Kalos to say, "The M68000 has nine, 32-bit address registers." In the figure, the *constituency overview* schema also embeds *short description* schemata for the M68000 data bus and address bus.

Figure 6.4 is a continuation of Figure 6.3. It shows part of the *constituency detailed* schema embedded by the *description* schema. The figure shows that the *constituency detailed* schema includes an embedded *description* schema to describe *M68INST*, the set of M68000 instructions. Only the *identification* schema part of the *M68INST description* schema is shown. This schema

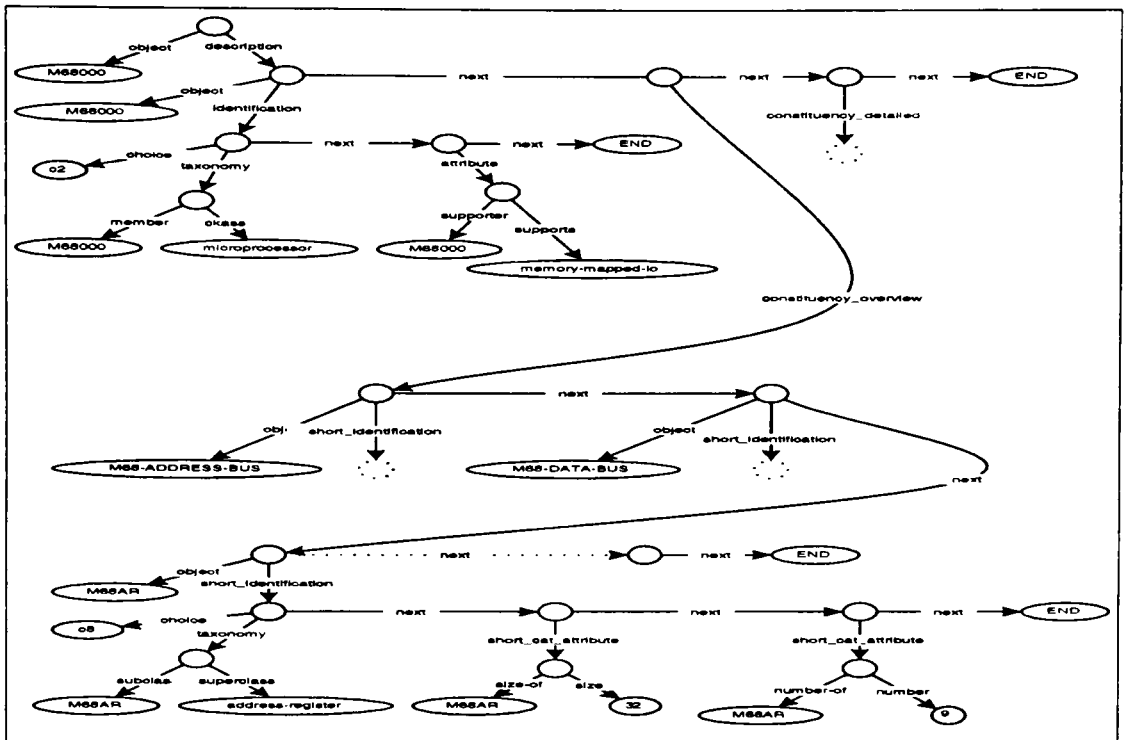


Figure 6.3 - Part of the Filled Description Schema for the M68000.

consists of a *taxonomy* slot, followed by an *instance* slot for each instruction, followed by *attribute* slots that indicate that the instruction set supports fourteen addressing modes. (Kalos does not embed *constituency overview* or *constituency detailed* schemata for instructions. Instead, a separate invocation of Kalos is used to produce instruction descriptions. This approach is used to keep individual Kalos runs relatively short.)

As the remaining parts of Kalos are described, we will return to this example to see how revision affects the filled-in schemata.

The surface level text produced from the instantiated schema produced by the deep generator is shown in Figure 6.5. Only six M68000 instructions are included to limit the size of the example. In this and following examples, Kalos behaves as if all eighty-two instructions were listed. Also note that no stylistic revisions have been applied, so there is a single, simple sentence for each schema slot.

6.2 Surface Generator

The Kalos surface generator is based on a unification grammar that uses lists of attribute/value

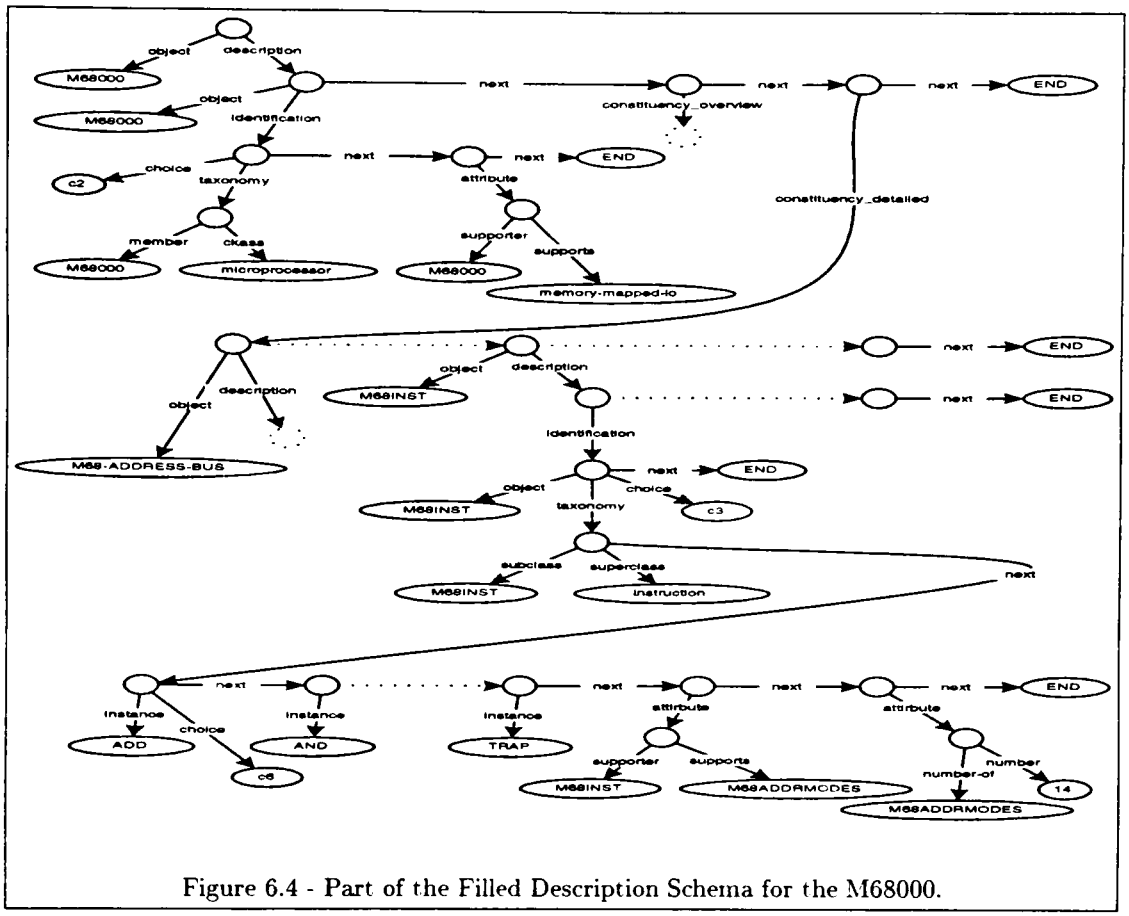


Figure 6.4 - Part of the Filled Description Schema for the M68000.

pairs [Kay 1984, Jacobs 1985]. This section describes the algorithms used by the surface generator. Grammar rule representation was discussed in section 5.5.

The surface grammar contains one or more rules for surfacing each knowledge frame that can occur in an instantiated schema. The first grammar rule associated with each knowledge frame describes how to surface the knowledge frame as a simple sentence. During initial generation, each slot in the schema is surfaced as a simple sentence. Alternative choices are only selected during regeneration based on revision suggestions.

Grammar rule application requires three operations: rule selection, rule evaluation, and surface text creation. Grammar rules are selected in three ways. First, the top-level grammar is always selected as the starting point for the grammar. This rule, as described in the next subsection, contains rules for compounding sentences and phrases. Second, rules to surface instantiated schema nodes are selected using pattern matching. The schema node is matched again the pattern nodes of the grammar rules for surfacing concepts (see Figure 5.12a). Third, grammar rules can be selected

```
((the M68000 is a microprocessor)
 (the M68000 supports memory-mapped I/O)
 (the M68000 address bus is an address bus)
 (the M68000 address bus is 24 bits wide)
 (the M68000 data bus is a data bus)
 (the M68000 data bus is 16 bits wide)
 (the M68000 address registers are address registers)
 (There are 9 M68000 address registers)
 (the M68000 address registers are 32 bits wide)
 (the M68000 data registers are data registers)
 (There are 8 M68000 data registers)
 (the M68000 data registers are 32 bits wide)
 (the M68000 instructions are instructions)
 (There are 82 M68000 instructions)
 (the M68000 address bus is an address bus)
 (the M68000 address bus is 24 bits wide)
 (the M68000 data bus is a data bus)
 (the M68000 data bus can handle operands 8 and 16 bits wide)
 (the M68000 data bus is 16 bits wide)
 (the M68000 address registers are address registers)
 (A0-A7 and A7' are the M68000 address registers)
 (the M68000 address registers can handle operands 32 bits wide)
 (the M68000 address registers are 32 bits wide)
 (the M68000 address registers can be divided into pure address registers)
 (the M68000 address registers can be divided into special address registers)
 (the pure address registers are M68000 address registers)
 (A0-A6 are the pure address registers)
 (the special address registers are M68000 address registers)
 (A7 is a special address register)
 (A7' is a special address register)
 (A7' is an address register)
 (A7 is an address register)
 (A7' is an address register)
 (A7' is also the supervisor stack pointer)
 (A7 is an address register)
 (A7 is also the user stack pointer)
 (the M68000 data registers are data registers)
 (D0-D7 are the M68000 data registers)
 (the M68000 data registers can handle operands 1 8 16 and 32 bits wide)
 (the M68000 data registers are 32 bits wide)
 (the M68000 instructions are instructions)
 (the ADD instruction is an M68000 instruction)
 (the AND instruction is an M68000 instruction)
 (the BCHG instruction is an M68000 instruction)
 (the BCLR instruction is an M68000 instruction)
 (the BRA instruction is an M68000 instruction)
 (the TRAP instruction is an M68000 instruction)
 (the M68000 instructions supports M68000 addressing modes)
 (There are 14 M68000 addressing modes))
```

Figure 6.5 - Sample Kalos Initial Generation

directly by name. Grammar rules of the form in Figure 5.12b have an *object* arc which associates the grammar rule with either a name or a concept. Named grammar rules allow one grammar rule to invoke another rule. Rules related to concepts are lexicon entries of the form shown in Figure 5.13.

Grammar rules are constructed from attribute/value pair building blocks consisting of attribute/value pairs, *or* nodes, *push* nodes, and quantity extraction nodes (see Figure 5.10). At-

tribute/value pairs are evaluated by evaluating the value part of the pair and assigning the result to the variable specified by the attribute part. Variable bindings are kept on a binding list. If a value is assigned to a variable that has no value, a binding pair for the variable and value is added to the binding list. If the variable already has a value and it is the same one as is specified by the value part of the attribute/value pair, then the binding list is unaltered and execution continues to the next attribute/value pair. If the variable has a value that differs from the value part of the attribute/value pair, backtracking occurs to the last untried alternative. If backtracking is invoked and there are no more alternatives to try, the surface generator aborts.

Attribute/value pair conflicts that cause backtracking have three sources, all of which are actions that can add variable bindings to the binding list. First, when a concept such as the *member/class* frame is selected for surfacing, pattern matching is used to find the appropriate grammar rule to surface the concept. When the rule is selected, the variables instantiated during the pattern matching, such as ⟨*object*⟩ and ⟨*any*⟩ above, are added to the binding list. Second, evaluation of grammar rules add variable values to the binding list. In Kalos, there are mechanisms for invoking one grammar rule from another, so considerable binding lists can be produced. When generating a sentence, for example, one of the first attributes set is for the variable ⟨*number*⟩ which is used to guide surface generation for "singular" and "plural" subjects. Third, the stylistic revisor can add bindings to the binding list when a particular concept is being surfaced to cause the selection of a grammar alternative. For example, the stylistic revisor can suggest the use of a pronoun by making a revision suggestion for a particular schema node with ⟨*subjtype*⟩ set to *pronoun*. This binding will force backtracking to select a pronoun as the subject of the sentence associated with the schema node.

Push nodes are evaluated by first creating a temporary binding list. The list consists of the current binding list and bindings created by evaluating the *pre* attribute/value pairs list. Next, the grammar specified by the *ref* node is evaluated. Finally, the *post* attribute/value pair list is evaluated. Variables that are assigned values in the *post* list are added to the binding list that was in effect when the *push* node was selected for evaluation. The temporary binding list created for the *ref* grammar is discarded.

When *ref* refers to *next*, the grammar rule to be evaluated is selected by pattern match, using the next instantiated schema node to select the grammar rule. To avoid binding list conflicts, values for variables instantiated during the pattern match, e.g., ⟨*object*⟩, temporarily override any values bound at the calling level. This feature allows compound sentences to be produced that require pattern matching for two schema nodes.

If *ref* is *tnext*, an empty temporary binding list is created and the top-level grammar is invoked

to surface the next schema node. The *post* node is used to extract useful variable binding from the execution of the top-level grammar. *Tnext* is useful for compounding operations to avoid binding conflicts between the grammars associated with two different concept nodes.

The value extraction nodes of Figure 5.10d are used to locate quantities of the form shown in Figure 5.2b and assign them to registers.

The last step in grammar rule evaluation is to use the surface order pattern of Figure 5.11 to construct the surface text. The surface order pattern is an ordered list of grammar variables and literals. By using this pattern and the current binding list, the ordered surface text is produced.

6.2.1 Top-Level Grammar

The surface generator begins execution by evaluating the top-level grammar rule. As this rule is evaluated, schema slot nodes are surfaced. Evaluation of the top-level rule may cause a single schema node to be surfaced as a single sentence or several schema nodes to be surfaced as a compound sentence. The top-level rule is evaluated repeatedly until all the slots in the instantiated schema have been surfaced.

The top-level grammar has seven choices:

- **Simple sentence:** Surface a simple sentence from a single schema node.
- **Noun phrase plus predicate:** Surface a sentence from two schema nodes that reference the same subject by surfacing the first schema node as a noun phrase and using it as the subject of the second sentence.
- **Compound predicate:** Surface a sentence from two schema nodes that reference the same subject by compounding the predicates.
- **Compound objects:** Surface a sentence from two schema nodes that reference the same subject, have the same verb, but have different objects by compounding the objects.
- **Compound sentences:** Surface a sentence from two schema nodes by combining a sentence for each schema node with "and."
- **Compound subject:** Surface a sentence from two schema nodes that are the same except for their subject referents by compounding the subjects of the two sentences.
- **Predicate:** Surface only the predicate of a sentence for compounding at a different level of the grammar.

Only the first rule is used during initial generation. Revision suggestions guide selection of the alternatives.

The top level grammar is an ordered disjunction of sentence types: simple, combined noun phrase and predicate, compound predicates, compound objects, compound sentences, and compound subjects. Each concept frame can be surfaced as a simple sentence. The remaining sentence types occur when one or more concepts are combined into a single sentence. The combined noun phrase and predicate is constructed by turning a predicate nominative sentence into a noun phrase and combining it with a second sentence that references the same object. For example

- D0 is a data register. (1)
- D0 is 32-bits wide. (2)

can be combined into

- The D0 data register is 32-bits wide. (3)

Sentences that have the same verb and object with different subjects are combined using the compound predicates rule. Sentences that have the same subject and verb but have different objects are combined using the compound objects rule. Two sentences can be combined using the compound sentences rule. Two sentences with the same predicate but different subjects are combined using the compound subject rule.

The stylistic revision rules that control compounding typically combine only simple sentences. This approach prevents inordinately long and complex sentences. There are cases, however, when a sentence that is constructed from two schema nodes is not too complex to be further compounded. For example, consider simple sentences (4) and (5) and their compounded version (6):

- The M68000 has 9 address registers. (4)
- The M68000 address registers are 32 bits wide. (5)
- The M68000 has 9 32 bit address registers. (6)

Although sentence (6) is constructed from two schema nodes, the two facts being expressed are closely coupled.

To allow sentences like (6) to be constructed and then compounded with another sentence, the ⟨*combine*⟩ variable is used in conjunction with ⟨*compound*⟩ to provide for multiple levels of compounding. The stylistic revisor affects compounding by assigning values to both these variables. See section 6.4.2 for details on how the stylistic revisor compounds sentences.

Figure 6.6 shows a simplified form of the seven choices in the top-level grammar. The grammar is shown in list form here, but it exists as attribute/value pair lists and grammar patterns in the uniform knowledge base of Kalos. Each choice has a label, a1-a7, a list beginning with *gpattern* which gives the order of strings and variables in the output, and a list of attribute/value pairs that are evaluated when the grammar rule is used.

Consider alternative a1 that is used to produce simple sentences. If ⟨*compound*⟩, ⟨*major_type*⟩, and ⟨*complexity*⟩ do not have values in the current binding list, "none," "sentence," and "simple" are assigned to them as the first attribute/value pairs are evaluated. Next, a push form is evaluated. "Next" indicates that the concept node from the next schema slot is selected. The empty list, the third form in the push list, means that no attribute/value pairs are evaluated prior to invoking the grammar to surface the schema slot. The grammar is invoked recursively to surface the schema node as a simple sentence. When the grammar is complete, control is returned to the post processing list

of the push list, and attribute/value pairs of this list are evaluated to selectively move attributes to the binding list at the top level. ⟨*state*⟩ contains the entire surface text and corresponding structures for the schema slot node. This value is placed in the ⟨*out*⟩ variable in the binding list at the current level. Similarly, the values for ⟨*verbroot*⟩, ⟨*minor_type*⟩, and ⟨*subjtype*⟩ are raised to the top-level binding list. The gpattern list is used to indicate the order of the output text. For a simple sentence, the order is trivial as the ⟨*out*⟩ variable containing the text is output.

When draft text is to be produced, a simple sentence is produced for each concept in the schema to be surfaced. During regeneration, the surface generator consults a suggestion list provided by the revisor. The suggestion list consists of a reference to a concept to be surfaced and an attribute-value pairs list containing values for some variables that will guide the regeneration. For example, suppose the revisor has analyzed sentences (1) and (2) above and suggests that the two sentences be combined using the combined noun phrase and predicate rule. It creates a suggestion with a reference to the slot filled with the concept that D0 is a member of the *data-register* class with an attribute-value pair that gives variable ⟨*compound*⟩ the value "nppred". When the top level rule is evaluated for the referenced node, the value of ⟨*compound*⟩ is placed in the binding list before the top-level grammar is evaluated.

When the top-level grammar is evaluated, the first rule, the one for generating simple sentences, is used. The rule fails because the the value for attribute ⟨*compound*⟩ is "none" in the rule and "nppred" in the binding list. The surface generator backtracks to the second rule which describes the desired compounding.

Other grammar rules, such as those specifying the surface text for an object, are evaluated as needed.

6.2.2 Surface Generation Example

To illustrate surface generation, we consider the instantiated schema of Figure 6.3. In this section, we will look at draft text production, i.e., no revision suggestions are considered. So the grammar is started with an empty binding list with the top-level grammar rules. Referring to Figure 6.6, alternative a1 is selected, and variables ⟨*compound*⟩, ⟨*major_type*⟩, and ⟨*complexity*⟩ are assigned values. Next the *taxonomy* slot node is selected from Figure 6.3 for surfacing by the push node in the grammar and the value of the *object* relation from the schema, M68000, is assigned to the ⟨*object*⟩ variable on the current binding list.

SNePS-2.1 pattern matching is used to select a *pattern/grammar* node (see Figure 5.12) for the *member/class* frame being surfaced. Figure 6.7 shows a simplified grammar rule for the *member/class* frame. The rule has two alternatives: b1 and b2. The first tells how to surface the

```
(a1 (gpattern <out>)
    (<compound> none)
    (<major_type> sentence)
    (<complexity> simple)
    (push next () ((<out> <state>)
                   (<verbroot> <verbroot>)
                   (<minor_type> <minor_type>)
                   (<subjtype> <subjtype>))))
(a2 (gpattern <np> <pred>)
    (<compound> nppred)
    (push next ((<major_type> np)) ((<np> <state>)
                                    (<subjref> <subjref>)
                                    (<subjtype> <subjtype>)))
    (push next ((<major_type> pred)) ((<pred> <state>)
                                      (<verbroot> <verbroot>)
                                      (<min> <minor_type>)))
    (<complexity> complex)
    (<major_type> sentence)
    (<minor_type> <min>))
(a3 (gpattern <s1> "and" <s2>)
    (<compound> pred)
    (push next ((<major_type> sentence)) ((<s1> <state>)
                                          (<subjref> <subjref>)
                                          (<subjtype> <subjtype>)))
    (push tnext ((<major_type> pred)) ((<s2> <state>)))
    (<complexity> complex)
    (<major_type> sentence)
    (<minor_type> compoundpred))
(a4 (gpattern <subj> <verb> <obj1> "and" <obj2>)
    (<compound> obj)
    (push next ((<major_type> sentence)) ((<subjref> <subjref>)
                                          (<verbroot> <verbroot>)
                                          (<subjtype> <subjtype>)
                                          (<subj> <subj>)
                                          (<verb> <verb>)
                                          (<obj1> <obj>)
                                          (<min> <minor_type>)))
    (push tnext ((<major_type> sentence)) ((<obj2> <obj>)))
    (<complexity> complex)
    (<major_type> sentence)
    (<minor_type> <min>))
(a5 (gpattern <s1> "and" <s2>)
    (<compound> sentence)
    (push next ((<major_type> sentence)) ((<s1> <state>)
                                          (<subjref> <subjref>)
                                          (<verbroot> <verbroot>)
                                          (<subjtype> <subjtype>)
                                          (<subj> <subj>)
                                          (<verb> <verb>)
                                          (<obj1> <obj>)))
    (push tnext ((<major_type> sentence)) ((<s2> <state>)))
    (<complexity> complex)
    (<major_type> sentence)
    (<minor_type> compoundsentence))
```

Figure 6.6a – Simplified Top-Level Grammar Alternatives (1 of 2)

knowledge frame as "the M68000 is a microprocessor," while the second indicates how to surface the frame as a noun phrase, "the M68000 microprocessor," for use as a subject in another sentence. As a result of the pattern matching, the variable ⟨*any*⟩ is bound to *microprocessor*. The b1 rule

first assigns values to three variables and then invokes the grammar software recursively to get the surface text for ⟨*object*⟩, ⟨*any*⟩, and for the verb "to be." The push rule for ⟨*object*⟩ would select and evaluate the rule shown in Figure 5.13. Based on the value of ⟨*number*⟩, choices are made to assign ⟨*expressed*⟩ the value "M68000" and ⟨*def_art*⟩ the value "the." Similar grammar calls are used to get surface text for "is" and "a microprocessor." Finally, the surface is assembled using the *gpattern* list, producing "the M68000 is a microprocessor."

```
(a6 (gpattern <out1> "and" <out2>)
    (<compound> subj)
    (push next ((major_type> subj)) ((<out1> <state>)))
    (push next ((major_type> plural_pred)) ((<out2> <state>)
                                            (<verbroot> <verbroot>)
                                            (<verb> <verb>)
                                            (<obj> <obj>)))
    (<subtype> compound)
    (<subjref> compound)
    (<complexity> complex)
    (<major_type> sentence))
(a7  (gpattern <out1>)
    (<compound> none)
    (<major_type> pred)
    (push next nil ((<out> <state>))))
```

Figure 6.6b – Simplified Top-Level Grammar Alternatives (2 of 2)

The surface generator would continue producing surface text until all schema slots are surfaced. (The complete text for initial generation was shown in Figure 6.5.) We will return to this example in later sections to see how revision can cause the surface generator to produce different text.

6.3 Conceptual Revisor

After the initial deep and surface text generation, a conceptual revision cycle is initiated. During each cycle, the conceptual revisor examines the current text for conceptual defects and makes suggestions to the deep generator. The text is regenerated by the deep and surface generator and a new conceptual revision cycle begins. When no more conceptual defects are detected, the surface text is passed on to the stylistic revisor and a stylistic revisor/surface generator cycle is entered. The stylistic revisor is described in the next section.

The Kalos conceptual revisor currently performs four types of conceptual revisions:

- Removal of redundant and superfluous information
- Application of domain preferred words and phrases
- Proper ordering of attributes
- Handling of inordinately long lists

```
(pattern (member <object> class <any>)
         (or (b1 b2))

(b1 (gpattern <det1> <subj> <xverb> <det2> <obj>)
    (<number> singular)
    (<subjref> <object>)
    (<objref> <any>)
    (push <object> nil ((<subj> <expressed>)
                        (<det1> <def_art>)
                        (<subjtype> <nountype>)))
    (push <any> nil ((<obj> <expressed>)
                     (<det2> <indef_art>)))
    (push be-verb nil ((<xverb> <state>)
                       (<verb> <expressed>)
                       (<verbroot> <root>)))
    (<major_type> sentence)
    (<minor_type> pred_nom)
    (<complexity> simple))

(b2 (gpattern <det> <n1> <n2>)
    (<number> singular)
    (<subjref> <object>)
    (<ref2> <any>)
    (push <object> nil ((<n1> <expressed>)
                        (<subjtype> <nountype>)))
    (push <any> nil ((<n2> <expressed>)
                     (<det> <def_art>)))
    (<major_type> np)
    (<minor_type> np)
    (<verbroot> none)
    (<complexity> simple))
```

Figure 6.7 – Simplified Grammar for Surfacing *Member/Class* Nodes

Each of these types of conceptual revision are described in the following sections.

In Kalos, both the conceptual revisor and the stylistic revisor are responsible for correct usage of domain specific preferred words and phrases. The conceptual revisor looks for situations where the use of preferred words or phrases will make conceptual changes, i.e., the addition, deletion, or reordering of text. A reason for performing this type of analysis in the revisors is to isolate domain specific linguistic knowledge in the revisors so that the surface generator is more easily adapted to other domains. Furthermore, it allows decisions based on linguistic information to be made without having to complicate the deep generator with linguistic information.

The conceptual revisor adds four lists to the uniform knowledge base to indicate revision suggestions to the deep generator:

- List of slots/embedded schemata to be removed.
- List of new knowledge.
- List of choice point suggestions.
- List of attribute orderings.

The function of each of these lists is described in section 5.6. The following subsections indicate how the individual types of conceptual revisions use these lists.

The revision suggestion lists are cumulative. Each conceptual revision step loads the lists from the previous passes. (The first conceptual revision pass starts with the suggestion lists being empty.) Revisions can be added to the list but not removed.

We now look at the types of conceptual revision performed by Kalos.

6.3.1 Revisions Causing Removal of Information

The Kalos conceptual revisor looks for opportunities to remove redundant and superfluous text. The revisions that are based on the removal of information fall into four categories:

- Redundant information due to surface effects.
- Redundant information due to restatement.
- Superfluous repetition.
- Domain specific revisions.

Surface effects can cause redundant statements to be generated. In Kalos, the knowledge base designer is free to construct subordinate categories and objects as needed and to assign one or more surface renderings in the lexicon for each subordinate category and object. In describing the M68000, we used subordinate categories to describe groups of registers. For example, the subordinate class *M68DR* refers to the eight M68000 data registers. It is a subclass of *data-register*, the class referring to all data registers in all machines. The lexicon entry for this class is "M68000 data registers." Kalos can then say things like "D0-D7 are M68000 data registers."

But a problem occurs when Kalos uses its schemata to describe this subordinate class because it attempts in both the *identification* and *short identification* schemata to state the relationship of the thing being described to the taxonomy to which it belongs. The surface form for the *subclass M68DR/superclass data-register* frame is "M68000 data registers are data registers." The problem is that the fact that the M68000 data registers are data registers is encoded in its lexicon entry for M68DR.

To avoid this problem, the conceptual revisor examines the instantiated schema for constructs of this form and targets them for removal by making removal suggestions to the deep generator.

The revisor compares lexicon entries for any objects that are members of *member/class, superclass/subclass*, or *arg1/rel isa/arg2* frames in the instantiated schema. If the lexicon entry for a subordinate object or subclass contains the lexicon entry for its class or superclass as a substring, then a suggestion is made to remove the slot containing the frame.

The conceptual revisor can also remove redundant information from the text if the redundancy occurs because of the restatement of some fact in a different way. In Kalos, the application of preferred word and phrase processing can cause information to be added to the text that restates information already present in the instantiated schema. Kalos gives priority to the statement that uses preferred words or phrases and deletes the restatement. For example, Kalos contains knowledge that, when describing a microprocessor, the term "address space size" should be used. It also contains the knowledge that the address space size is computed from the address bus size and how to perform the computation. A suggestion is made to the deep generator to add an attribute to the M68000 description giving the address space size. Since a statement of the address bus size in the same paragraph is redundant, the revisor suggests that the address bus be removed from the *constituency overview* schema for the M68000.

The conceptual revisor also removes text when it detects superfluous information. This happens when Kalos describes a subordinate category such as "spec-addr-reg," the subclass of M68000 address registers that also serve as stack pointers. The *instance* slot of the *identification* schema is filled with an *arg1/rel isa/arg2* frame that is surfaced "A7 is a special address register." Kalos then fills a *constituency overview* schema for A7 which contains a taxonomy slot. This causes the sentence, "A7 is an address register," to be surfaced. Finally, a *constituency detailed* schema is embedded for A7, causing another *taxonomy* slot to be filled. This slot results in "A7 is an address register" being surfaced.

There are two effects at the root of this problem. First, most of the salient attributes of the address registers are described as a group. A7 is only described because it is atypical in that it has a special function as the user stack pointer. This causes the taxonomy information to be restated in close proximity since there are few other salient attributes. The discourse structure based on *description*, *constituency overview*, and *constituency detailed* schemata fails at this level of the description. At a higher level in the description, the *description* and *constituency overview* schemata produce a single paragraph, followed by a paragraph or more for each part or part category produced by the *constituency detailed* schema. Because of the separation of the text produced by the *constituency overview* schema and the *constituency detailed* schema for an object, restatement of the taxonomy information is appropriate. But for an atypical object like A7 where only a single atypical attribute needs to be stated, the close proximity of the taxonomy information causes poor

text.

The second problem with this construct is that the *instance* slot for the description of the *spec-addr-reg* subordinate category casts A7 in terms of its membership in the subordinate category. Following this statement with the fact that "A7 is an address-register" is superfluous, particularly since this relationship has already be brought out in the description of M68000 address registers.

Kalos contains inference rules that looks for repeated taxonomy information following an *instance* slot that casts the object in a particular way. When this construction is found, a suggestion to delete the superfluous taxonomy frames is made to the deep generator.

The fourth type of deletion suggestion that Kalos can make is based on domain-specific information. These revisions are based on SNePS-2.1 inference rules that encode knowledge about the description of microprocessors and complement salience rules in the deep generator. For example, the deep generator includes attributes about an object that give a quantity about the object being described. So when a register or bus is described, its size and the size of the operands it can handle are listed. In some cases, the device can handle operands of various sizes up to the size of the device, and in others, only operands that are the size of the device are handled. For data registers, Kalos says

- The M68000 data registers are 32 bits wide.
- They can handle operands 1, 8, 16, and 32 bits wide.

Using the same rules when it describes the M68000 address registers, it says

- The M68000 address registers are 32 bits wide.
- They can handle operands 32 bits wide.

Since it is typical for a register to be able to handle operands that are its size, this text is defective. A domain specific rule causes the statement of a single operand size to be deleted when it is the same as the register size.

6.3.2 Application of Domain Preferred Words and Phrases

The second general category of revisions performed by the conceptual revisor is based on the application of preferred words and phrases. Preferred words and phrases are encoded in knowledge base frames that specify the preferred surface text and the object type to which it is applicable. For a particular preferred word or phrase, the conceptual revisor first determines if there is an instance of a description of some object in the instantiated schema that belongs to the class associated with the preferred word or phrase. If there is, the revisor inspects the surface generator grammar and the lexicon for ways to produce the preferred text.

If a particular grammar rule can be used to produce a preferred word or phrase for a particular

object type, the conceptual revisor extracts the knowledge base pattern that will trigger the rule, i.e., the pattern node *p* in Figure 5.12. It then inspects the deep generator schema templates for a choice point that will cause a slot to be filled for an object of the proper type that will trigger the pattern in the grammar rule. If it finds one, it suggests using the choice point by adding an entry to the choice list that is passed to the deep generator during text regeneration. If no choice point is applicable, it adds a frame for the object that matches the grammar pattern and flags it as an attribute of the object. This information is added to the list of knowledge to be passed to the deep generator during text regeneration. When the attribute slot is filled for the object, the new attribute will be included.

If the preferred text can be produced by a particular lexicon item, the revisor attempts to relate the object for which it is trying to use the preferred wording to the item whose lexicon entry will give the preferred wording. Inferencing is used to try to relate the objects using *member/class*, *superclass/subclass*, and *arg1/rel isa/arg2* frames. If such a relationship is found, the revisor looks for a choice point that will cause the frame to be included. If none are found, it adds the frame as an attribute.

There are two preferred phrases that are applicable to the *microprocessor* class in Kalos: "address space size" and "16-bit microprocessor." By inspecting the surface generator grammar rules, the conceptual revisor finds that "address space size" can be generated from an *object/addr_space_size* frame. It then inspects the deep generator schema templates for a schema slot that is filled with this frame. It finds none, so it deduces the address space size from the address bus size, creating an *object/addr_space_size* frame, and indicates that it is an attribute of the M68000 by building an *object/attribute* frame. This frame is added to the list of new knowledge that the deep generator will load during text regeneration. This new frame will be included in an attribute slot in the instantiated schema and cause the sentence "the M68000 has an address space size of 16 megabytes" to be generated.

The preferred phrase "16-bit microprocessor" also causes a conceptual revision to be made to the text. The list of preferred word knowledge base frames has an entry for a number of microprocessor sizes: "8-bit microprocessor," "16-bit microprocessor," "32-bit microprocessor," and "64-bit microprocessor." The conceptual revisor looks through the grammar rules for a rule that will cause one of these phrases to be generated. There are none, so it looks for an knowledge base object whose lexicon entry is one of the strings. It finds that the subordinate class *16-BIT-MICROP* can be surfaced as "16-bit microprocessor." The conceptual revisor then uses inferencing to try to relate the M68000, the only microprocessor being described, to the subordinate class *16-BIT-MICROP*. As part of SNePS-2.1 inferencing, it is deduced that, because the M68000 has

a 16-bit data bus, that it is a member of the subordinate class *16-BIT-MICROP*, i.e., an *arg1 M68000/rel isa/arg2 16-BIT-MICROP* frame is created. The revisor then inspects the schema templates for a slot this will cause this type of frame to be surfaced. It finds that the last choice in the first *or* of the *identification* schema (Figure 6.1) contains a *taxonomy* slot that can be filled with a *arg1/rel isa/arg2* frame. In the initial instantiated schema, the first choice was used to fill the *taxonomy* slot with a *member/class* frame. The conceptual revisor adds an object/choice pair to the choice point list that is passed to the deep generator to cause the fourth choice to be used instead of the first. When the text is regenerated, the surface rendering is "the M68000 is a 16-bit microprocessor" instead of "the M68000 is a microprocessor."

Figure 6.8 shows the knowledge links used to perform this revision. Note that knowledge from both the surface knowledge base and the deep generator knowledge base are needed. The arc labeled *1* shows the association between the preferred phrase "16-bit microprocessor" and the lexical entry for the subordinate class 16-BIT-MICROP. Kalos attempts to relate the M68000 to 16-BIT-MICROP causing the *arg1/rel isa/arg2* frame pointed to by arc *2* to be created by SNePS-2.1 inferencing. The schema templates in the deep generator knowledge base are examined for a slot that matches this frame. Arc *3* points to the part of the *identification* schema that matches the frame. A revision suggestion that will cause choice *c21* to be selected over choice *c1* when the M68000 is described is generated.

6.3.3 Proper Ordering of Attributes

The third type of conceptual revision relates to the ordering of attributes. The conceptual revisor tries to regenerate text so that attributes of quantity are generated before attributes of quality. The deep generator fills attribute slots using SNePS-2.1 inferencing. To reduce deep generator complexity, no attempt is made to order the attribute slots. As an example, consider two sentences generated by Kalos when describing the "D0" register:

- D0 can handle operands 1, 8, 16, and 32 bits wide.
- D0 is 32 bits wide.

Without conceptual revising, the two sentences would be combined by the stylistic revisor into the following compound sentence:

- D0 can handle operands 1, 8, 16, and 32 bits wide and is 32 bits wide.

The conceptual revisor uses SNePS-2.1 rules to locate attribute ordering problems. For example, consider the following rule:

```
(assert forall ($s1 $s2 $n1 $n2 $target)
  &ant ((build arg1 *s1 rel immedbefore arg2 *s2)
        (build name *s1 schema-node *n1)
        (build name *s2 schema-node *n2)
        (build name *s1 subjref *target)
        (build name *s2 subjref *target)
        (build quantifier *n2)
        (build min 0 max 0 arg (build quantifier *n1)))
  cq (build alter-fill (build arg1 *n2 rel before arg2 *n1)))
```

The rule says that for any two sentences, s1 and s2, if s1 immediately precedes sentence s2, s1 and s2 have the same subject, s2 surfaces an attribute of quantity, and s1 does not surface an attribute of quantity, then build a suggestion for the deep generator to reorder the attributes. When the deep generator is invoked to regenerate text, it looks for *alter-fill* relations and attempts to follow the suggested ordering. (See Figure 5.15.)

In the example, the revisor would suggest that the two sentences be reordered. After stylistic revision, a compound sentence would be produced:

- D0 is 32 bits wide and can handle operands 1, 8, 16, and 32 bits wide.

The rule generates an *alter-fill* frame that specifies the ordering of the attributes using an *arg1/rel before/arg2* frame. All of the *alter-fill* frames are built into the uniform knowledge base for use by the deep generator during text regeneration.

Note that the knowledge base contains rules about which attributes represent quantities and which do not.

6.3.4 Handling of Inordinately Long Lists

When Kalos describes a subordinate category such as "M68INST," the set of M68000 instructions, it fills an *instance* slot in the *identification* schema for each object that is part of an *arg1/rel isa/arg2* frame for the subordinate category. If the category of M68000 control registers were described, this structure would result in the text "SR and PC are M68000 control registers." But for "M68INST," a sentence would be generated listing all 82 M68000 instructions. Such a long sentence is inappropriate in an overview paragraph describing the instructions. Each instruction would eventually be described in detail by a separate Kalos run and would perhaps be listed in a table in a typical users' guide.

To avoid an overly long sentence, the conceptual revisor looks for another choice point that will generate a list of instruction subordinate categories, e.g., "arithmetic instructions," "logical

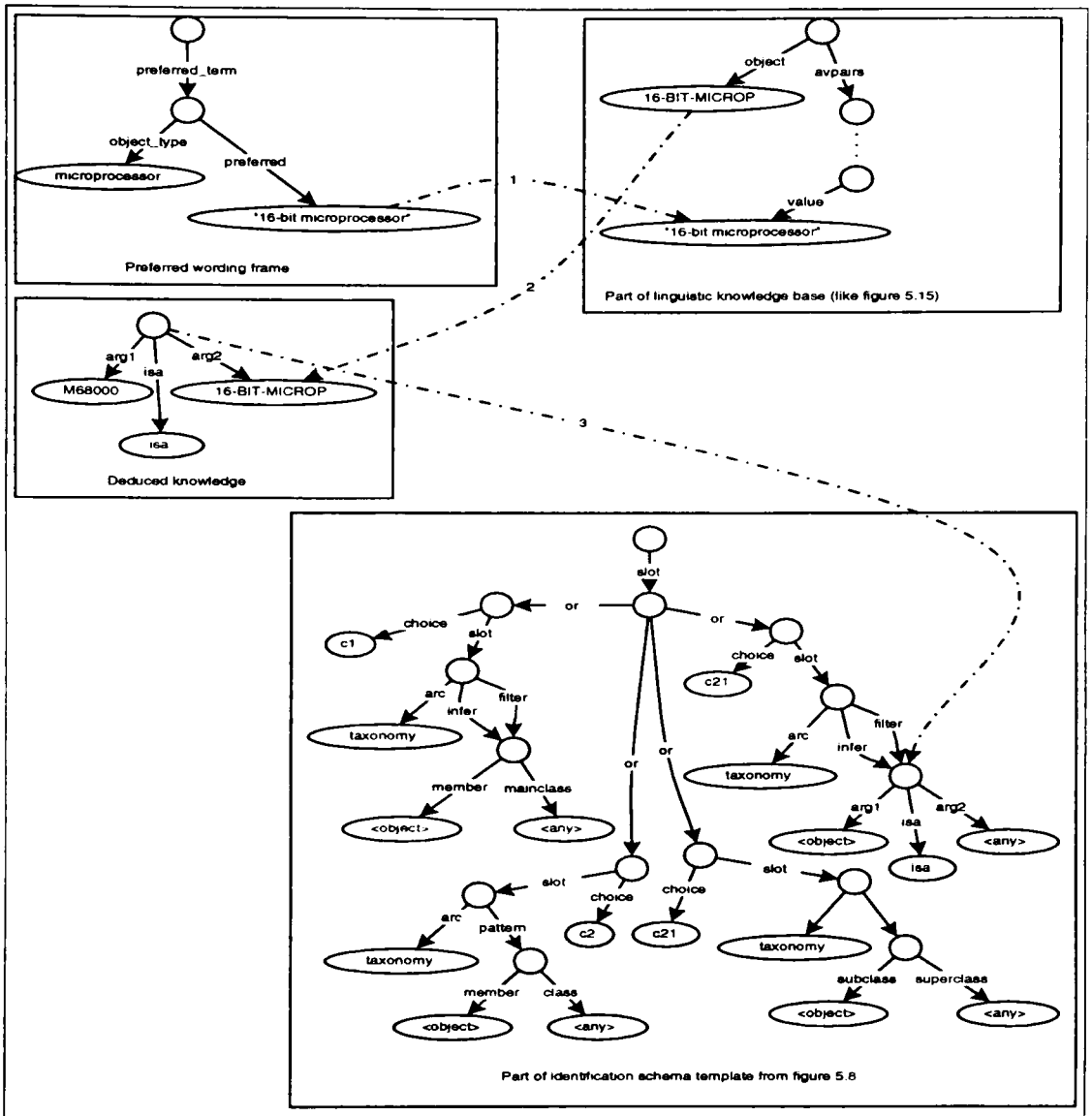


Figure 6.8 - Preferred Phrase Knowledge Base Usage Example

instructions," "branch instructions," and "control instructions," instead of listing all the instructions. By inspecting the *identification* schema, the conceptual revisor can select the final choice point in the *or* construct for instance names (choice *c7* in Figure 6.2). This choice would cause the subordinate category names and their sizes to be listed instead of the individual instructions.

SNePS-2.1 pattern matching is used to locate instance slots. If the category being described has an *arg1/rel catsize/arg2 bigcat* frame in the domain knowledge base, SNePS-2.1 pattern matching is

used to attempt to locate an alternate schema choice that will generate a list of categories. Pattern matching is also used to verify that the knowledge base contains subordinate categories that can be used to fill the instance slots.

Figure 6.9 illustrates how the schema of Figure 6.3 would be regenerated based on this type of revision.

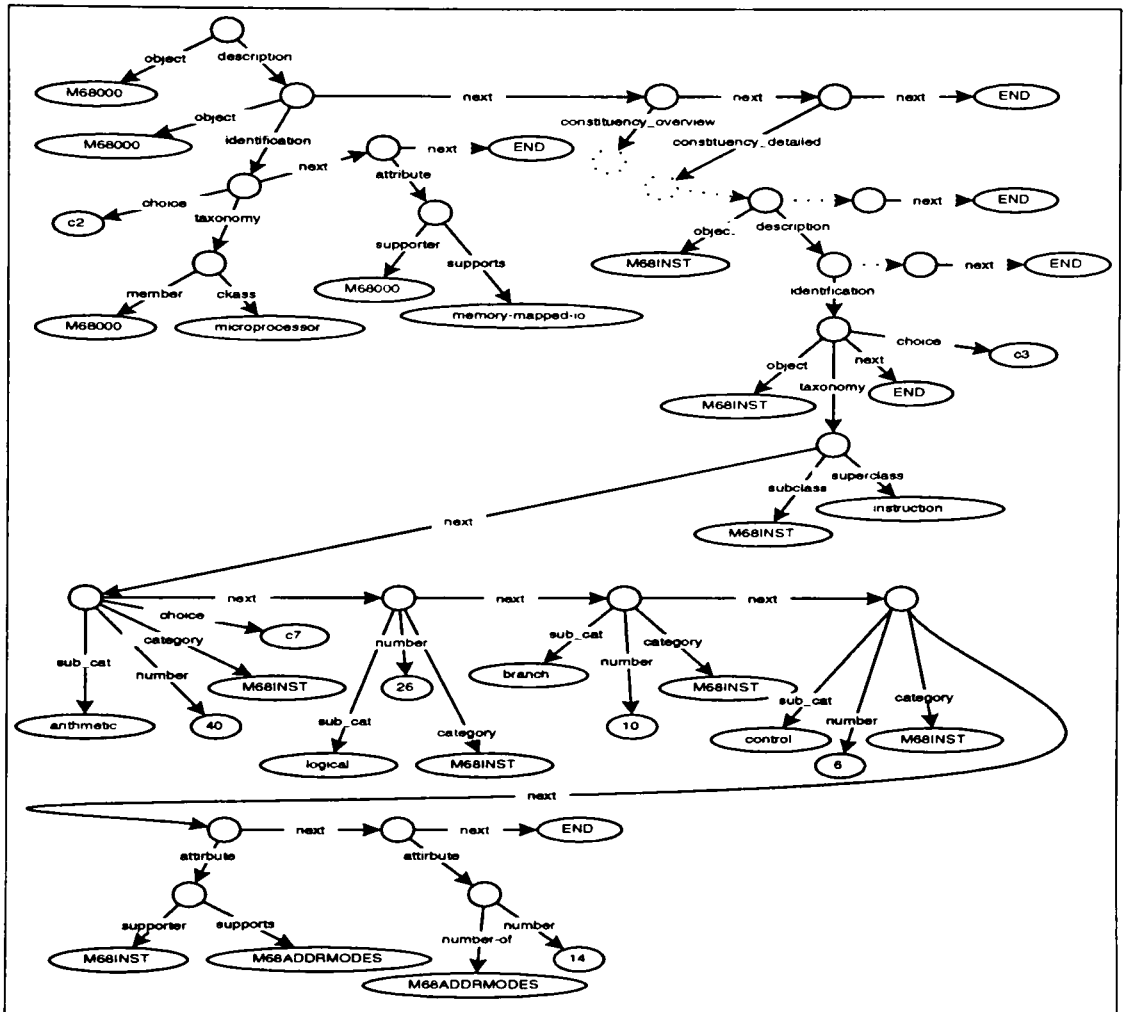


Figure 6.9 - Regenerated Schema after Sized-Based Revision.

6.3.5 Conceptual Revision Example

We now return to the example of text generation for the opening paragraphs of the M68000 description. Figure 6.3 shows the initial section of the *description schema* for describing the M68000 and

Figure 6.5 shows the initial draft text. Figure 6.10 shows the text generated after one conceptual revision step.

```
((the M68000 is a 16-bit microprocessor)
 (the M68000 has an address space size of 16 megabytes)
 (the M68000 supports memory-mapped I/O)
 (There are 9 M68000 address registers)
 (the M68000 address registers are 32 bits wide)
 (There are 8 M68000 data registers)
 (the M68000 data registers are 32 bits wide)
 (There are 82 M68000 instructions)
 (the M68000 address bus is 24 bits wide)
 (the M68000 data bus is 16 bits wide)
 (the M68000 data bus can handle operands 8 and 16 bits wide)
 (A0-A7 and A7' are the M68000 address registers)
 (the M68000 address registers are 32 bits wide)
 (the M68000 address registers can be divided into pure address registers)
 (the M68000 address registers can be divided into special address registers)
 (A0-A6 are the pure address registers)
 (A7 is a special address register)
 (A7' is a special address register)
 (A7' is also the supervisor stack pointer)
 (A7 is also the user stack pointer)
 (D0-D7 are the M68000 data registers)
 (the M68000 data registers are 32 bits wide)
 (the M68000 data registers can handle operands 1 8 16 and 32 bits wide)
 (the M68000 instructions can be divided into 40 arithmetic instructions)
 (the M68000 instructions can be divided into 26 logical instructions)
 (the M68000 instructions can be divided into 10 branch instructions)
 (the M68000 instructions can be divided into 6 control instructions)
 (the M68000 instructions supports M68000 addressing modes)
 (There are 14 M68000 addressing modes))
```

Figure 6.10 - Kalos Output After One Revision Iteration

The following revision suggestions were applied to the text in Figure 6.5 to get Figure 6.10.

- The removal of redundant taxonomy statements due to surface effects related to the subordinate categories "M68000 address bus," "M68000 data bus," "M68000 address registers," "M68000 data registers," "M68000 instructions," "pure address register," and "special address register."
- The removal of superfluous information due to restatement, affecting the sentences "A7 is an address register" and "A7' is an address register."
- The use of the preferred phrase "16-bit microprocessor" causes "the M68000 is a microprocessor" to be replaced with "the M68000 is a 16-bit microprocessor." The corresponding revision suggestion specifies that the last choice in the first *or* of the *identification* schema should be used instead of the first

choice when the M68000 is described. This revision also removes the sentence "the M68000 data bus is 16 bits wide" from the opening paragraph to avoid restatement of the data bus size caused by recasting the M68000 in terms of being a "16-bit microprocessor."

- The preferred phrase "address bus size" is used by adding an attribute about the M68000 that gives its address space size based on its address bus size. This statement is ordered before the other M68000 attribute because it specifies a quantity. The revision also causes the statement of the address bus size to be removed from the opening paragraph to avoid restatement of the address space size in terms of the address bus.
- Attributes of size for the data registers and data bus are ordered before attributes related to the size of operands for these devices.
- A domain specific revision caused deletion of the statement that the M68000 address registers can handle operands 32 bits wide since it is typical for a register of a certain size to handle only operands of that size.

Figure 6.11 shows part of the instantiated schema after one conceptual revision/deep generation pass. Note the changes that cast the M68000 in terms of being a "16-bit microprocessor" and the addition of an attribute giving the address space size for the microprocessor. The slots for the M68000 data bus and address bus have been removed to prevent restatement of the new facts.

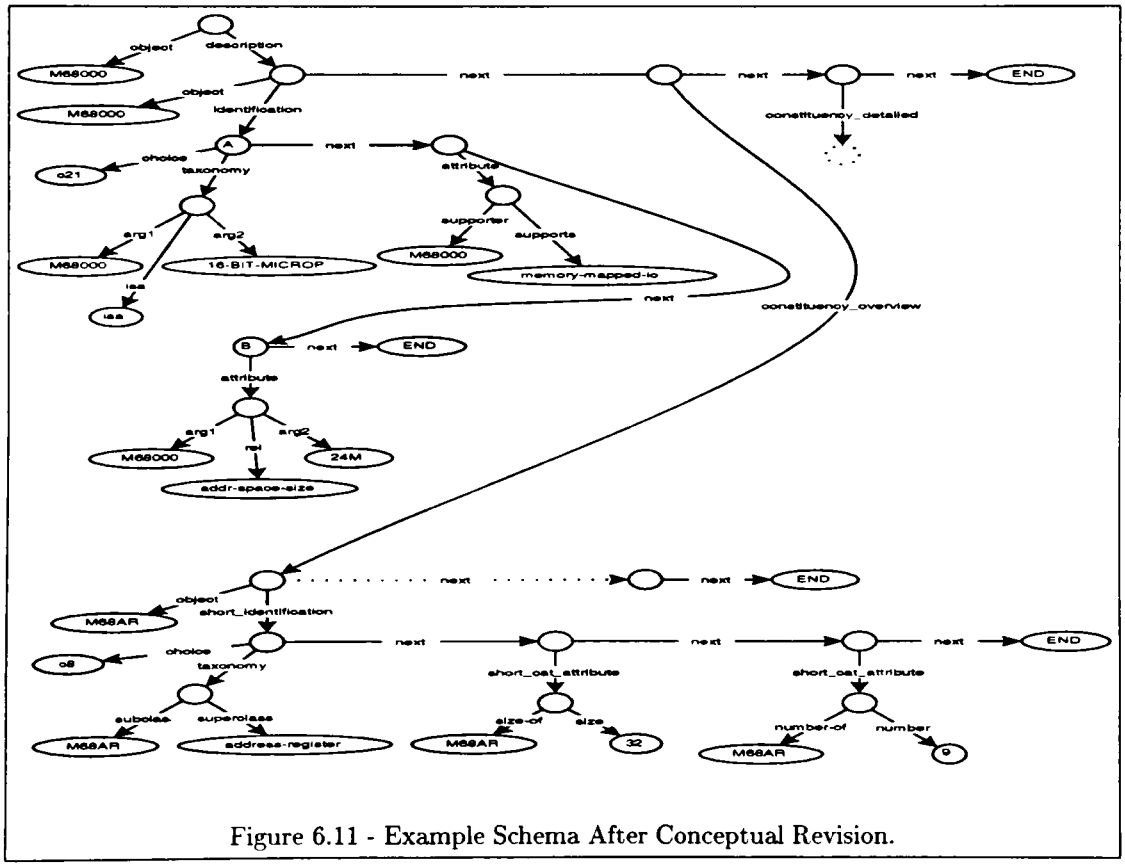


Figure 6.11 - Example Schema After Conceptual Revision.

The next pass through the conceptual revisor adds a single suggestion to the list produced by the first pass. The suggestion is to order the attribute related to address space size before the attribute about memory-mapped I/O. This revision results in the following opening text:

```
(the M68000 is a 16-bit microprocessor)
(the M68000 has an address space size of 16 megabytes)
(the M68000 supports memory-mapped I/O)
```

The remaining text is unchanged. On the third pass through the conceptual revisor, no new changes are produced, so Kalos enters a stylistic revision/surface generation cycle as discussed in the next section.

6.4 Stylistic Revisor

After the text has been revised by the conceptual revisor until no more changes can be made, Kalos enters a stylistic revision cycle. The stylistic revision cycle contains two processes: the stylistic revisor and the surface generator. A stylistic revision cycle starts with the stylistic revisor examining surface text and producing revision suggestions. These suggestions are used by the surface generator to regenerate the text. This cycle is repeated until no new revision suggestions are applicable and the final text is output.

The stylistic revisor examines the surface text and underlying structures looking for opportunities to apply cohesive techniques such as anaphora and compounding. The revisor examines sentences in groups of one and two looking for potential revisions. It builds a list of all potential revisions by inferencing and deals with conflicts between the suggestions before invoking the surface generator to regenerate the text using the suggestions.

A simple example illustrates how the stylistic revisor works. Consider the two sentences

- D0 is a data register. (1)
- D0 is 32 bits wide. (2)

The revisor analyzes these sentences and their associated underlying structures for opportunities to apply cohesive techniques to them. In this simple example, the revisor considers anaphora and compounding suggestions:

- Make the subject of (2) a pronoun.
- Compound (1) and (2) using a compound object form.
- Compound (1) and (2) by generating (1) as "the D0 data register" and using it as the subject of sentence (2).

The revisor then selects the suggestions with the most structural change and discards any other suggestions that conflict with them. In this simple example, the last suggestion would be

used, and the first two discarded so that the sentence

- The D0 data register is 32 bits wide. (3)

would be generated.

The stylistic revisor is implemented as a combination of Lisp and SNePS-2.1 inference rules. First, inference is invoked to apply a set of SNePS-2.1 rules that look for potential stylistic revisions that can be made to the text (see section 5.7). Next, the stylistic revisor uses SNePS-2.1 inference to look for conflicts among the revision suggestions. The filtering process analyses revision suggestions that target the same sentence. Each suggestion contains a weight of 1, 2, 3, 4, or 5 where a higher weight indicates a more desirable revision. Weights are assigned based on the amount of structure change the revision will make and on *ad hoc* considerations such as compounding two sentences giving attributes of the same object instead of compounding based on the order sentences are encountered. When two suggestions reference the same sentence, the one with the stronger weight is used, and the other suggestion is deleted. When two conflicting suggestions have the same weight, one is selected nondeterministically except in the case where the weights are both 1. Suggestions with weight 1 impact the sentence structure minimally, so all suggestions of weight 1 are applied.

By favoring suggestions with a higher degree of structural change, suggestions that combine sentences are preferred over changes with less impact such as the use of pronouns. Although some suggestions are deleted, they may be applicable during a later revision iteration.

The Kalos stylistic revisor performs the following stylistic revisions:

- Suggest use of anaphora.
- Suggest sentence and phrase compounding.
- Suggest the use of preferred words and phrases.
- Suggest thematic progression constructs.
- Suggest other cohesive constructions.

6.4.1 Application of Anaphora

The stylistic revisor looks for opportunities to use anaphora to make the text more cohesive. It has two rules associated with using pronouns. First, if two consecutive sentences have the same subject, a suggestion is made to make the subject of the second sentence a pronoun. Second, if the instances of a class are identified and the class is the subject of the second sentence, a suggestion is made to use a pronoun for the second sentence. As an illustration of the second type of pronoun usage, consider the following sentences:

- A0-A7 and A7' are the M68000 address registers.
- The M68000 address registers are 32 bits wide.

A pronoun can be used as the subject of the second sentence.

- A0-A7 and A7' are the M68000 address registers.
- They are 32 bits wide.

Because the stylistic revisor can inspect the underlying knowledge from which the sentences was generated, it has access to the referents of all the nouns in a sentence. With this knowledge, it can make intelligent decisions about using pronouns.

6.4.2 Application of Compounding Constructions

The stylistic revisor is responsible for taking the simple sentences output from the conceptual revision loop and applying compounding constructions so that the sentences are more cohesive. Overuse of compounding turns every paragraph into a single sentence with a number of conjunctions, while underuse of compounding produces choppy text. The stylistic revisor must choose the proper level of compounding and select between conflicting compounding constructions.

The Kalos stylistic revisor examines two sentences at a time to look for opportunities to apply compounding techniques. As noted above, all suggestions for a paragraph are made and then conflicting suggestions are removed based on a weight attached to each type of revision. The stylistic revisor applies compounding at three levels:

- Combine two simple sentences to form a simple sentence.
- Combine two simple sentences to form a complex sentence.
- Combine a complex and a simple sentence to from a complex sentence.

In the first case, two simple sentences are combined to form a simple sentence. This type of combining can occur in two ways. First, two sentences that are the same except for the objects of prepositional phrases are combined. Second, two sentences expressing attributes can be combined into a simple sentence.

As an example of the first type of combining of simple sentences that results in a simple sentence, consider the two sentences:

- The M68000 address registers can be divided into pure address registers.
- The M68000 address registers can be divided into special address registers.

These two sentences originate from two individual deep generator schema slots but are meant to be combined into a single simple sentence giving the two subsets of the M68000 address registers:

- The M68000 address registers can be divided into pure address registers and special address registers.

The resulting sentence expresses a single concept and can be further compounded by Kalos.

The second type of transformation that can combine two simple sentences into a third simple sentence deals with combining two sentences about some object where the second sentence expresses

a size or number associated with the object. For example, consider two sentences about the M68000 address registers:

- The M68000 has eight M68000 data registers.
- The M68000 data registers are 32 bits wide.

The stylistic revisor can combine these sentences into the simple sentence:

- The M68000 has eight 32 bit data registers.

This resulting sentence states the size and number of M68000 data registers. Since size and number are normally expressed together, this sentence is considered simple and can be compounded further.

The second type of compounding the stylistic revisor can perform combines two simple sentences into a complex sentence. There are five ways complex sentences can be produced from simple sentences:

- Two sentences have the same subject but the predicates differ. The predicates are compounded.
- Two sentences have the same predicates but the subjects differ. The subjects are compounded.
- Two sentences have the same subject and verb but the objects differ. The objects are compounded.
- The two sentences differ in subject and object but have the same form. The sentences are compounded.
- The two sentences have the same subject and the first sentence can be recast as a noun phrase. The first sentence is turned into a noun phrase and is used as the subject of the second sentence. This form is used where the first sentence is a predicate nominative.

The first three types should be obvious to the reader. The fourth type is illustrated by the following two sentences and their compounded result:

- A7' is also the supervisor stack pointer.
- A7 is also the user stack pointer.
- A7' is also the supervisor stack pointer and A7 is also the user stack pointer.

Normally, the stylistic revisor doesn't combine two unrelated sentences to avoid excessive compounding; however, these two sentences are parallel in their meaning and form and are compounded.

The fifth type of compounding of simple sentences that produces a complex sentence is illustrated by sentences (1), (2), and (3) shown previously in this section.

The third general type of compounding combines a complex sentence and a simple sentence. This type of compounding is rare in Kalos, but is allowed when combining a noun phrase/predicate construction with a second sentence when the predicate and the following sentence both express attributes of the subject. Since listing attributes is natural, combining these into a single sentence typically does not make it excessively complex. For example, consider the two opening sentences

describing the M68000:

- The 16-bit M68000 microprocessor has an address space size of 16 megabytes.
- The M68000 supports memory-mapped I/O.

These two sentences are compounded in Kalos to form

- The 16-bit M68000 microprocessor has an address space size of 16 megabytes and supports memory-mapped I/O.

In some cases, compounding rule application must be limited to prevent excessive sentence complexity. For example, a rule for compounding the objects of prepositional phrases was described above; however, if the rule is applicable to three or more consecutive sentences, the resulting sentence may be too complex to further compound. For example, consider the four sentences describing the divisions of the M68000 instruction set:

- The M68000 instruction set can be divided into 40 arithmetic instructions.
- The M68000 instruction set can be divided into 26 logical instructions.
- The M68000 instruction set can be divided into 10 branch instructions.
- The M68000 instruction set can be divided into 6 control instructions.

After combining, the result is

- The M68000 instruction set can be divided into 40 arithmetic instructions, 26 logical instructions, 10 branch instructions, and 6 control instructions.

Because this sentence has four parallel elements, the surface generator marks it as complex so that the stylistic revisor does not further compound it.

6.4.3 Application of Preferred Terms

The stylistic generator also looks for opportunities to use preferred words and phrases. Each preferred word and phrase is associated with a class of object. In addition to this information, there must be grammar rules that exploit the preferred words and phrases and support knowledge on how to build revision suggestions. All of this domain specific knowledge is isolated in the stylistic revisor.

Kalos knows that the word "execute" is a preferred word for the class instruction and that "relative branch" is a preferred phrase for members of the subordinate category branch-instructions. First, we will look at the handling of the term "execute." Kalos normally says that "the M68000 has 82 instructions." Kalos revises this sentences to use "executes" instead of "has." The stylistic revisor consults its domain knowledge rules to determine that "executes" can be used with instructions, members of instructions, and instruction subclasses, that it can be used with the suggested frames, and that the instructions or instruction classes are the objects of the sentences. It then builds revision suggestions to use "executes" for the appropriate sentences.

In the case of "relative branch," consider the surface text for the draft of the *process* schema for the BRA instruction (Figure 6.12). The draft text describes the semantics of the instruction; however, in a typical users' guide, the actual register transfers are listed in symbolic form, for example,

$$(PC) + disp \rightarrow PC$$

When this instruction is executed, the contents of the PC (Program Counter) register are added to the *disp* (displacement) field of the instruction and the result is placed in the PC. If the register transfer semantics are given, it would be beneficial to surface the text description using the term "relative branch."

Given the preferred phrase "relative branch" which can be applied to branch instructions that use the contents of the program counter register in the branch address calculation, the stylistic revisor examines the surface generator grammar rules for occurrences of rules that will produce the term "relative branch." It finds such a rule as an alternative to the draft surface text given for the BRA instruction. This rule assigns "PC-relative branch" to the $\langle obj \rangle$ register, so the revisor makes a suggestion to use this suggestion with the attribute/value pair ($\langle obj \rangle$ "PC-relative branch"). The resulting text is shown in Figure 6.12. Note that this type of revision seems simple, but it relieves the surface generation from trying to decide which surface rendering to use. The user of Kalos may decide not to include "relative branch" as a preferred word if he or she desires the explicit semantics of instructions to be listed.

```
DRAFT TEXT:

The BRA instruction causes the sum of the contents of the PC and the
disp field of the instruction to be transferred to the PC.

AFTER REVISION:

The BRA instruction performs a PC-relative branch using the disp field
of the instruction.
```

Figure 6.12 - Kalos Instruction Description Example

6.4.4 Application of Thematic Progression

The stylistic revisor also attempts to surface the text using thematic progression (see section 2.6). This revision technique is applied on a paragraph basis where the sentences from adjacent *identification* and *constituency overview* schemata are considered to be a single paragraph. To maintain thematic progression between two sentences, either the subjects must be the same or the object of the first sentence must be the subject of the second sentence. The revisor looks for adjacent sentences in the same paragraph that are not in thematic progression order. It can easily determine this by looking at the $\langle subjref \rangle$ and $\langle objref \rangle$ variables in the binding lists for the sentences. If two sentences are not in thematic progression order, it looks for other renderings of the second sentence in the linguistic knowledge base that will put the two sentences in thematic procession order. For example, consider the two sentences:

- The M68000 supports memory-mapped I/O.
- There are 9 M68000 address registers.

The second sentence can be restated using M68000 as the subject:

- The M68000 supports memory-mapped I/O.
- The M68000 has 9 address registers.

On the first revision pass, the second sentence would be generated as "the M68000 has 9 M68000 address registers." Another stylistic revision rule would remove the redundant reference to "M68000."

6.4.5 Other Stylistic Revisions

The stylistic revisor performs two other revisions. It removes surface-level redundancy and reorders the parts of compound nouns. Surface-level redundancy occurs in the surface text because lexical entries for objects are made independently of how the objects will be used in sentences. For example, in

- The M68000 has 9 M68000 address registers.

the second "M68000" is redundant. In general, however, the name for the set of M68000 address registers is not a flaw. Consider

- There are 9 M68000 address registers.

The stylistic revisor inspects sentences with surface-level redundancy and suggests that the less specific superclass name be used. In the above example, the superclass for the set of M68000 address registers, the class of address registers, is used so that the sentence becomes

- The M68000 has 9 address registers.

The other miscellaneous cohesive revision deals with ordering the parts of compound nouns. In particular, if a compound noun contains a quantity part, the revisor suggests that the quantity be listed first. For example, the compound noun "the M68000 16-bit microprocessor" is transformed to the more natural "the 16-bit M68000 microprocessor." For the stylistic revisor to make this revision, it must consult linguistic knowledge which says that the surface rendering for the class 16-BIT-MICROP contains a quantity. This knowledge is encoded in a *subclass/quantity_part/noun_part* frame. For the 16-BIT-MICROP subordinate class, this frame is *subclass 16-BIT-MICROP/quantity_part 16-bit/noun_part microprocessor.* It indicates that the lexicon contains entries for the quantity and noun parts of the surface rendering under the given knowledge base objects. The stylistic revisor can then use the split noun form of the surface rule for the *arg1/rel isa/arg2* frame to generate the smoother text.

6.4.6 Stylistic Revision Rules

The heart of the stylistic revisor is a set of SNePS-2.1 rules that inspect pairs of consecutive sentences and makes revision suggestions. Figure 5.14 shows how the surface text is stored. The stylistic revisor has access to the surface text, the grammar rules used to generate the text, the schema node that was surfaced, and the top-level attribute/value pair binding list created by the surface generator. The structure of Figure 5.14 is recursive in that the *surface* arc chain may contain *link* arcs to other structures of the form of Figure 5.14 that represent parts of the text.

The stylistic revisor considers pairs of consecutive sentences when looking for text defects. The current version of the stylistic revisor makes its decisions based on the top-level attribute/value pair lists, top-level grammar rule, and top-level schema nodes for each pair of sentences. All the grammar rules are designed to pass pertinent information up to the top-level binding list using the action of the *post* attribute/value pair list of the grammar *push* nodes. This approach allows the stylistic revision rules to operate efficiently by having all the useful information at the top level without sacrificing robustness.

The stylistic revision rules can inspect any of the variables on the top-level binding list; however, the current version makes use of the following variables:

- ⟨*subjref*⟩: The object which is referenced by the subject of the sentence.
- ⟨*subjtype*⟩: The type (noun, pronoun, or proper noun) that represents the subject of the sentence.
- ⟨*objref*⟩: The object which is referenced by the object of the sentence.
- ⟨*compound*⟩: The main compounding level of the sentence.
- ⟨*combine*⟩: The secondary compounding level of the sentence. Mostly used for compounding two simple sentences into a third, structurely simple sentence.

- ⟨*complexity*⟩: Records the complexity of the sentence. Used to mark a sentence as too complex to compound further.
- ⟨*verbroot*⟩: The root of the verb used in the sentence.
- ⟨*suggsubjref*⟩: The object suggested by the revisor during a previous text generation iteration to be used as the subject of the sentence. This variable allows the revisor to suggest a sentence transformation to create thematic progression ordering.
- ⟨*minor_type*⟩: This grammar variable records information about the structure of the sentence. For example, it indicates if a sentence is a predicate nominative.
- ⟨*slot*⟩: The slot label of the primary schema node for this sentence.

When sentences are compounded, they are generated from two or more schema nodes. The current version of the stylistic revisor only inspects the top-level schema node used in constructing compound sentences. This approach is adequate in the current implementation as conceptual information is only needed when combining two simple sentences. Any additional information needed for compounding at a more complex level can be found in the attribute/value pair binding list. However, since the surface structure of Figure 5.14 is recursive, all the schema nodes used to generate a sentence can be located and used in stylistic revision rules.

The stylistic revisor also inspects only the top-level grammar rule although all the grammar rules are available. Currently, the grammar rule is only inspected to determine if there is an optional grammar rule for a sentence that can be used to generate it as a noun phrase.

The revision rules used by the stylistic revisor have access to all the information about sentences as described above, knowledge about the ordering of sentences, and linguistic knowledge. From this knowledge, the stylistic revisor looks for surface text defects and generates stylistic revision suggestions. (Refer to Figure 5.16 for the format of stylistic revision suggestions.) The *sentence1* and *sentence2* arcs give information about which sentences are affected and which parts of the sentences are affected by the revision. (The *sentence2* arc is optional for revision suggestions that only affect a single sentence.) The *force* arc gives a weight of the goodness of the suggestion. These three arcs are used to remove conflicting revision suggestions when more than one suggestion affects a sentence.

The *alters* arc indicates the schema slot node that the suggestion affects. If more than one schema slot node is affected by a revision, only the first one is indicated. (Indicating the first schema node is adequate to indicate to the surface generator when to put the suggestion attribute/value pairs on the binding list.) The *avpairs* arc points to a list of attribute/value pairs that are to be put on the binding list when the schema slot is surfaced.

Consider, for example, the stylistic revision rule that suggests converting a predicate nominative sentence into a noun phrase that can be used as the subject of the following sentence. This

```
(assert forall ($subj $s1 $s2 $node $g)
&ant ((build arg1 *s1 rel immedbefore arg2 *s2)
      (build arg1 *s1 rel compound arg2 none)
      (build arg1 *s2 rel compound arg2 none)
      (build arg1 *s1 rel subjref arg2 *subj)
      (build arg1 *s2 rel subjref arg2 *subj)
      (build name *s1 grammar-node *g)
      (build gname *g  canproduce np)
      (build name *s1 schema-node *node))
cq (build alters2 *node
  sentence1 (build sentence *s2 element pred)
  sentence2 (build sentence *s1 element subj)
  avpairs (build attribute <compound> value nppred next end)
  force 4))
```

Figure 6.13 - Kalos Stylistic Revision Rule

rule is shown in Figure 6.13.

The first antecedent of the rule looks for an *arg1/rel immedbefore/arg2* frame that indicates that sentence s1 is immediately before sentence s2. The next two antecedents inspect the attribute/value pairs list for the two sentences to make sure that they have not already been compounded, i.e., their ⟨*compound*⟩ variables are set to *simple*. The next two antecedents inspect the attribute/value pair binding list to determine that the subjects of both sentences reference the same object. The next antecedent determines the grammar rule used to generate the simple sentence. The following antecedent inspects knowledge about the surface generator grammar rules to see if the first sentence can be cast as a noun phrase. Finally, the last antecedent inspects the surface text structure of the first sentence to extract the schema slot node associated with the sentence.

If all the antecedents are true, the consequence of the rule is built into the knowledge base. If the suggestion is not removed during suggestion conflict resolution, it will be used during the next surface generation iteration. When the schema node is being surfaced, the attribute/value pair (⟨*compound*⟩ NPPRED) will be added to the surface generator binding list. This value will guide the top-level grammar rule to select the rule to combine the next two sentences by turning the first sentence into a noun phrase and using it as a subject of the second sentence.

We now continue with the example of text generation in Kalos. Figure 6.14 shows the surface text after conceptual revision is complete. Note that the surface structure references the underlying schema and grammar nodes from which the text was generated. (Nodes A and B refer to the nodes in Figure 6.11 and a1 refers to Figure 6.6.) The revisor performs inferencing for *alters* nodes, resulting in these preliminary suggestions being included in its suggestions:

```
(assert sentence1 (build sentence m1 element subj)
        sentence2 (build sentence m2 element pred)
        force 4
        alters A
        avpairs (build attribute <compound> value nppred next end))

(assert sentence1 (build sentence m2 element subj)
        force 1
        alters B
        avpairs (build attribute <subjtype> value pronoun next end))
```

The first suggestion suggests combining sentences 1 and 2 by making the first sentence into a noun phrase and using it as the subject of the second sentence, resulting in "The M68000 16-bit microprocessor has a 16M address space." The second suggestion is for making the subject of sentence 2 a pronoun.

The next step in the revision process is suggestion conflict resolution. The revisor looks for pairs of *sentence1* or *sentence2* frames that reference the same sentences. When two are found, suggestions with *force* arc values of lessor degree are deleted. The second of the above two suggestions is removed, and the surface text is regenerated. When the first node is surfaced, ((*compound*) nppred) is placed on the surface generator's binding list, causing it to surface the node as "the M68000 16-bit microprocessor." This text is then used as the subject of the second sentence, resulting in the following text:

- The M68000 16-bit microprocessor has an address space of 16 megabytes.

The stylistic revisor is invoked again and analyzes the text. This time, only one suggestion is produced for the text:

```
(assert sentence1 (build sentence m1 element subj)
force 1
alters B
avpairs (build attribute <splitnoun> value true next end))
```

This revision suggestion causes the words in the noun phrase "the M68000 16-bit microprocessor" to be reordered as "the 16-bit M68000 microprocessor." This suggestion is based on the principle of listing quantifiers first.

Figure 6.15 shows the final output of Kalos after no more stylistic revisions are applicable. This text illustrates all the stylistic revisions discussed in this section.

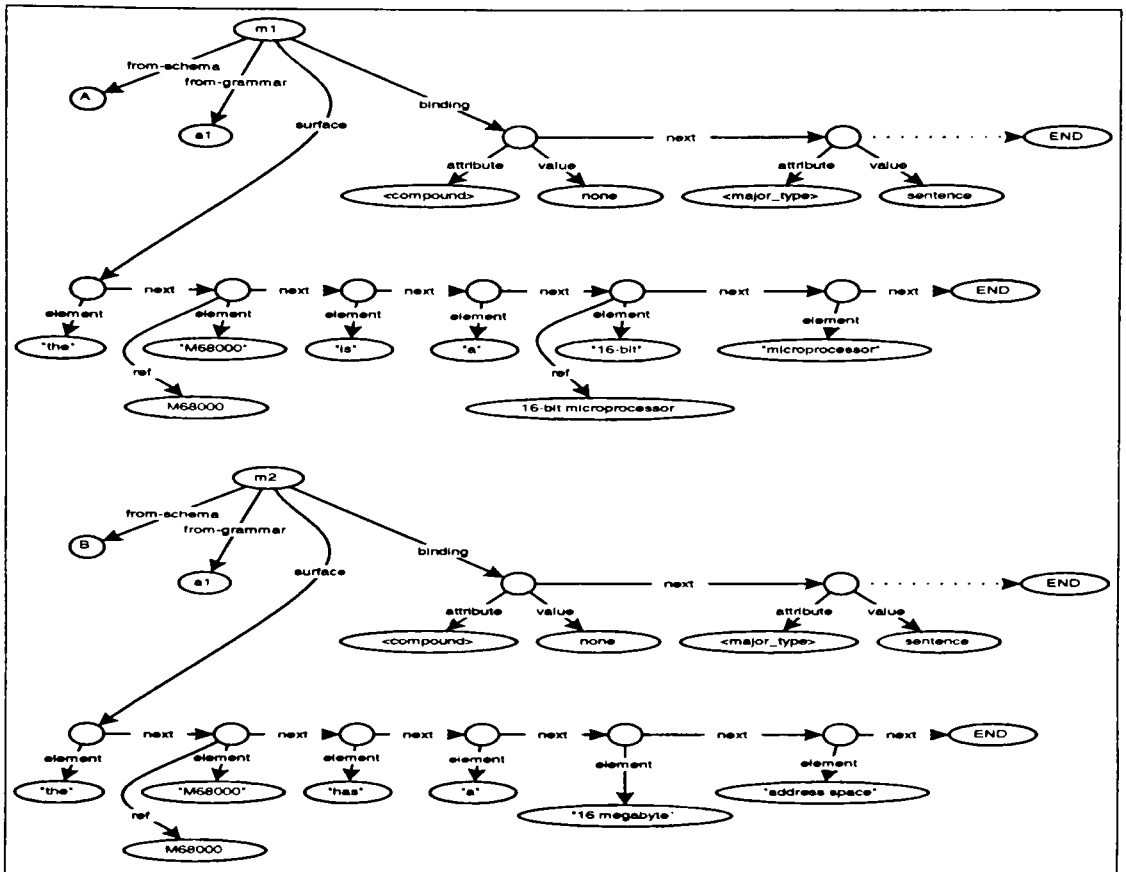


Figure 6.14 - Abbreviated Surface Representation for Text After Conceptual Revision.

```
((the 16-bit M68000 microprocessor has an address space size of 16
 megabytes and supports memory-mapped I/O)
 (it has 9 32 bit address registers and 8 32 bit data registers)
 (it executes 82 instructions)
 (the M68000 address bus is 24 bits wide)
 (the M68000 data bus is 16 bits wide and can handle operands 8 and
  16 bits wide)
 (A0-A7 and A7' are the M68000 address registers)
 (they are 32 bits wide and can be divided into pure address registers
  and special address registers)
 (A0-A6 are the pure address registers)
 (A7 and A7' are special address registers)
 (A7' is also the supervisor stack pointer and A7 is also the user
  stack pointer)
 (D0-D7 are the M68000 data registers)
 (they are 32 bits wide and can handle operands 1 8 16 and 32 bits wide)
 (the M68000 instructions can be divided into 40 arithmetic
  instructions 26 logical instruction 10 branch instruction and 6
  control instruction)
 (they support 14 M68000 addressing modes))
```

Figure 6.15 - Kalos Output After Stylistic Revision

6.5 Thesis Analysis

In section 1.4, I presented a two point thesis for the work described in this dissertation. I now argue that the thesis is supported.

The first point is that a revision architecture provides a beneficial decomposition of tasks for a natural language generation system. In section 1.4, I listed three criteria for evaluating the success of a revision system. First, significant decision making performed by a traditional generator had to be successfully postponed for the revisors. Furthermore, the interaction between the generators and the revisors could not significantly impact generator performance. In Kalos, I was able to postpone significant decision making, based on wide locality of decision making, to the revisors. The deep generator was relieved of dealing with redundancy, attribute ordering, domain-specific preferred terms, and inordinately long lists. The surface generator was relieved of dealing with cohesive constructions including anaphora, removal of surface-level redundancy, compounding, use of preferred terms, and thematic progression orderings.

Initial versions of both the deep and surface generators were constructed and tested before they were altered to look for revision suggestions in the uniform knowledge base. In both cases, the addition of revision suggestion processing had minimal computational effect on the generator and required little or no additional inference. Suggestions to the deep generator for removal of

redundant information, choice point suggestions, and attribute orderings are lists that the deep generator must search when making decisions. In practice, these lists are small and add little overhead. The fourth type of suggestion, added domain knowledge, also impacts deep generator decision making little. Added knowledge accounts for a small percentage of the overall knowledge base.

Interaction between the stylistic revisor and surface generator has little effect on the surface generator. Stylistic revision suggestions are schema slots and lists of attribute/value pairs associated with them. The only additional overhead incurred by the surface generator is to inspect the list of suggestions when a schema node is to be surfaced. If the node is on the list, its attribute/value pairs are added to the binding list.

Second, the usefulness of a revision architecture is supported by its ability to support techniques that are not possible in a traditional natural language generation task decomposition. In a traditional system, conceptual decisions derived from linguistic knowledge are not made. But in Kalos, some linguistic decisions that affect conceptual decisions are handled by the conceptual revisor. It removes concepts that are redundant because of the lexicon entries selected for the concepts. (For example, the frame *arg1 M68AR/rel isa/arg2 address-register* would be surfaced as "the M68000 address registers are address registers." But the relationship between *M68AR* and *address-register* is encoded in the surface strings associated with the concepts and therefore the sentence is superfluous.) A typical deep generator does not include linguistic information for efficiency reasons, but the conceptual revisor in a revision system is the ideal place to apply conceptual changes based on linguistic information. The conceptual revisor also makes conceptual decisions based on preferred phrases. This processing is another example of the application of linguistic information to make conceptual decisions.

Third, empirical data collected from Kalos supports the thesis that a revision architecture provides a beneficial decomposition for a text generation system. The current version of Kalos runs on an IBM RS-6000 model 560 under AKCL (Austin Kyoto Common Lisp) version 1.615 and SNePS-2.1. SNePS-2.1 is compiled, but the Lisp code specific to Kalos is currently interpreted. In this configuration, little function could be added to the deep generator without exceeding the memory management system of AKCL. The deep generator relies heavily on inference. The surface generator contains a large grammar (over 1000 nodes). Inference is not used in the surface generator as all functions requiring inference are postponed to the stylistic revisor. In the current configuration, the knowledge base of the surface generator is too large to support inference; however, inference would be required in a traditional system to make the decisions now made by the stylistic revisor. In other words, the traditional components of Kalos, the deep and surface

generators, produce poor text and cannot be extended on the current platform. However, by using a revision architecture, these same generators, in concert with the revisors, produce draft-quality text.

The second point of the thesis is that a knowledge intensive environment is a welcoming architecture for a revision system. In section 1.6, I listed three criteria for evaluating this point. First, the environment has to allow revision to operate intelligently and efficiently. In Chapter 3, I listed the types of knowledge required by the revisors. A uniform knowledge base provides access to all the system knowledge. Furthermore, the revisors can operate efficiently because they do not need to deduce knowledge inferred during another generator step. The use of association links helps the revisor locate the underlying knowledge associated with surface text easily.

Second, the computational burden of a uniform knowledge base must not burden the natural language generation system. In this respect, the uniform knowledge base is not entirely successful. As a natural language generation system grows, a uniform knowledge base places high demands on the underlying inferencing system. I dealt with these problems by developing knowledge base partitioning techniques and staging the individual generation tasks. These techniques allow a system to behave as if all the system knowledge were available, when in fact the knowledge that is not needed by a particular generation stage is not loaded.

Empirical data from Kalos supports the use of a knowledge intensive environment with a uniform knowledge base for a natural language generation system. Early versions of Kalos were able to use a uniform knowledge base. As the system grew, I dynamically partitioned the knowledge base using the SNePS-2.1 belief revision system so that only pertinent knowledge was available during each generation task. Finally, Kalos exceeded the capabilities of the underlying system on which it was implemented, so both staging and knowledge base partitioning techniques were used. Although a uniform knowledge base has negative features related to computational needs, Kalos shows that a uniform knowledge base can be used successfully when combined with techniques to manage the knowledge needed by individual generation tasks.

When analyzing the usefulness of algorithms, care must be taken to separate algorithm complexity concerns (section 3.6) from the performance issues associated with a particular implementation. Kalos was implemented as a research testbed and not as a production system. For this reason, little energy was spent on addressing performance problems.

Some initial performance problems were addressed by adjusting the initial memory allocations to the partitions to which AKCL applies garbage collection. Although this action reduced garbage collection and improved performance, another roadblock occurred when inference was applied to large semantic networks. I suspect two problems are at the heart of this issue. First, SNePS-

2.1 uses the uniqueness principle when building nodes into the network. If an identical node is already present, the new node is not constructed and the existing node is used. As the size of the network increases, the check to determine if the node already exists appears to consume significant resources. Second, inference in SNePS-2.1 is performed using a simulated parallel approach. As rules are activated, processing spreads throughout the network as combinations of variable bindings are sought that will satisfy the rule. In a large network with complex rules including a number of antecedents, the resources needed for inference are significant.

I have not explored these problems further. It is not clear if the performance problems I have experienced during the development of this project are the result of design decisions in AKCL or SNePS-2.1 or some interaction between the two systems. It is likely that tuning would produce improved results. (Neither package is a commercial product, but they both are useful and powerful tools.)

6.6 Conclusion

This chapter has illustrated the usefulness of revision and knowledge intensive generation in a natural language generation system that attempts to produce draft text. The relatively simple deep and surface generators of Kalos are able to produce reasonable draft-level text because of the power of the revision modules. More complex generators would have exceeded the resources of the platform on which Kalos was run; however, the staged generation system of Kalos was able to produce text within the memory and computational limitations of the given platform. Kalos also illustrates that although the use of a uniform knowledge base is constraining computationally, the techniques of staging, conceptually restricted knowledge bases, and path-based inferencing allow the benefits of a uniform knowledge base to be exploited.

The draft-quality text generated by Kalos could be extended by including more knowledge about the M68000 including semantics for all the machine instructions and information on the functions of the integrated circuit pins. With the extended text as a basis, a technical writer could begin to develop a polished users' guide. Some of the deficiencies that Kalos has that a human author would have to deal with are a limited number of ways Kalos can make statements, lack of knowledge about figures and tables, an inability to construct relatively complicated examples of hardware and software, and a lack of knowledge about the intended use of some special machine instructions. Kalos also lacks the ability to produce the marketing text that is typically found in the opening section of a microprocessor users' guide where the most salient features of the microprocessor, with respect to its predecessors and competition, are described and, sometimes, overstated.

The success of Kalos illustrates that systems that produce draft-quality text are feasible and that revision and knowledge intensive generation are useful techniques.

Chapter 7
Conclusions and Future Research

> I used to hate writing assignments, but now I enjoy them.
>
> I realized that the purpose of writing is to inflate weak ideas, obscure poor reasoning, and inhibit clarity.
>
> With a little practice, writing can be an intimidating and impenetrable fog!
>
> Academia, here I come!
>
> Calvin to Hobbes
> "Calvin and Hobbes" comic strip
> by Bill Watterson

7.1 Conclusions

Natural language generation is a difficult task. Work in this area has produced some useful techniques, but state-of-the-art is far below human proficiency in the area. Research must focus on improving machine handling of natural language to improve the quality of human-computer interfaces and to improve our knowledge of how language and intelligence works.

In this paper, I have demonstrated two interrelated techniques that can be used in a natural language generation system producing multiple sentence, draft-quality text. Revision and the use of a uniform knowledge base go hand-in-hand to form a powerful tool for natural language generation. The advantages of using these techniques are twofold:

- Provides software engineering benefits
- Provides an accommodating architecture for text generation and polishing techniques that are not accommodated by traditional architectures

Revision provides software engineering benefits to a natural language generation system. As compared to a traditional architecture, revision provides redistribution and reorganization of load that increases modularity, reduces overall system complexity, reduces the complexity of individual modules, and increases modifiability and maintainability. As noted in Chapter 1, connected text natural language generation is a complex task. By allowing some decisions typically made by the generation module to be postponed for consideration by the revision module, the complexity of the generators can be greatly reduced.

A revision module is also an ideal place to contain domain specific linguistic information such as preferred words and phrases related to the domain. By isolating domain-specific information in a revision module, the system is more maintainable and can be more easily adapted to other domains.

The second broad benefit provided by a revision system is that it provides an accommodating architecture for a number of techniques related to text generation and polishing. These techniques allow

- The interaction of conceptual and stylistic decisions
- The detection of infelicitous, verbose, and ambiguous text
- The application of text cohesiveness constructs

A revision architecture provides a means for the surface and deep generators to interact. In a traditional natural language generation architecture, all conceptual decisions are made before the surface generator is activated. So word choice decisions cannot influence conceptual decisions. But in a revision architecture, a number of iterations are made through the deep generation, surface generation, revision loop allowing greater interaction of the generators via decisions made in the revision module.

Some text defects, such as ambiguity and unintentional rhyming, are best detected after the surface text has been generated. Attempting to locate these defects during text generation would greatly increase the complexity of the generators. Some such defects, such as ambiguity, are effects of both surface text and word semantics, so they are best considered in terms of both the surface text and underlying conceptual structures. Revision provides the ideal architecture for this type of analysis.

Revision also provides an appropriate architecture for applying cohesive techniques such as anaphora and compounding. By postponing the application of cohesion until after text generation, the surface generator can be made less complex. Since some cohesive constructs, such as the use of anaphora, involve semantic notions, such as focus of attention [Grosz & Sidner 1986], a revisor with access to both stylistic and conceptual information can produce better cohesion.

Kalos demonstrates that revision is a useful technique for a natural language generation system and that both stylistic and conceptual revisions are beneficial. At the heart of Kalos is a uniform knowledge base that is shown to be an accommodating architecture for a revision system because it allows revisors access to knowledge at all levels of the generation process. Only by having access to all the system knowledge can a revision module perform its task intelligently and efficiently.

Techniques to deal with the computational burdens created by using a uniform knowledge also have been addressed in the Kalos system. Three techniques are used in Kalos to make the uniform knowledge base usable:

- Path-based techniques
- Static and dynamic knowledge base partitioning
- Staging and conceptually restricted knowledge base partitioning

Path-based techniques reduce the need for pattern matching and inferencing. Instead, associations between parts of the knowledge base are constructed during text generation so that the underlying structures from which the surface text was generated can be located efficiently. Traversing associations is computationally inexpensive.

The attractiveness of a uniform knowledge base is that the same inference technique can be applied to all the system knowledge, combining different knowledge sources in the same inference rule. But not all the knowledge is needed for any given task. For example, the deep generator never has need to access surface grammar knowledge. So to reduce computational burden created by pattern matching and inferencing in a large knowledge base, the system knowledge base can be partitioned so that only the knowledge necessary for a given task is available. In addition to this static partitioning, dynamic partitioning can be used to select intermediate results needed by a stage.

Static and dynamic partitioning helps to reduce the computational burden on the system but does not address the cases where the system needs only a small portion of one of the knowledge base partitions. Conceptually restricted knowledge bases can be used when only a portion of a knowledge base partition is needed. For example, when the conceptual revisor is analyzing grammar rules to determine which knowledge base frames will produce a preferred word or phrase, it does not need the entire linguistic knowledge base. It only needs access to knowledge that indicates which knowledge base frames produce the preferred words and phrases. By building a conceptually restricted version of the knowledge base that includes only the relevant knowledge from the grammar rule knowledge base, the size of the partition under consideration can be greatly reduced. This partitioning is accomplished in a semantic network system through inferencing to discover all the network nodes that can produce preferred text and building a surrogate knowledge base that includes this knowledge.

Conceptually restricted knowledge base techniques can be combined with dynamic partitioning to produce conceptually restricted versions of intermediate results. In the case of Kalos, both revisors need the surface structure produced by the surface generator, but the conceptual revisor does not need the surface generator binding list information or knowledge about the grammar rules from which the text was generated. To reduce the size of the large knowledge base that the conceptual revisor uses, the binding and grammar rule information of surface structures are not passed to it.

Staging takes the idea of knowledge base partitioning one step further by running different steps of the generation process as independent modules while keeping the power of a uniform knowledge base. Instead of switching between knowledge base partitions in a single program,

the stages load different knowledge base partitions. Dynamic knowledge base partitioning is used to copy relevant knowledge between the independent modules. The effect is the same as using knowledge base partitioning with the advantage of reducing demands on the memory system of the underlying programming system, e.g., Lisp, because intermediate results can be discarded between stages.

The use of a uniform knowledge base provides additional benefits in terms of adding knowledge to the system. All phases of the generation and revision process are expressed in the same uniform knowledge base giving the system the power of explicit knowledge and the ability to make decisions over all the system knowledge. This type of power is necessary for revision but also improves system maintainability because of the uniform architecture.

7.2 Transportability to Other Domains

Kalos is tailored for producing descriptions of artifacts. It currently produces a description of the Motorola M68000 microprocessor, but could be modified to describe other microprocessors or to generate text in other domains. To generate text for another microprocessor, the facts about the M68000 in the domain knowledge base would have to be replaced with facts about the new microprocessor. For typical microprocessors, few other modifications would have to be made. For microprocessors with atypical architectural features, some additional grammar rules would have to be added to the linguistic knowledge base so that Kalos could produce surface text for the atypical features.

Adaptation of Kalos for other domains is more complicated. Kalos was designed primarily as a testbed for revision techniques and not as a commercial natural language generator that could be easily adapted and used in new domains. It is, however, worthwhile to consider the changes necessary to support other domains.

Kalos describes objects (i.e., their attributes and part/whole relationships) and processes. We consider how these two types of descriptions can be transported to other domains. If the objects and the part/whole relationships of the new domain can be easily represented in the knowledge representation of section 5.3, then the Kalos deep generator can be applied to the new domain to produce object descriptions. Process descriptions in the current version of Kalos are tailored to microprocessor instruction descriptions and are transportable to few other domains. If process descriptions are required for the new domain, then new frames must be added to the domain knowledge base to describe the processes. Little change is required to the slot-filling mechanism to be able to select process frames for surfacing.

The first step in transporting Kalos to a new domain is to design and construct a new domain

knowledge base and domain-specific salience rules. (Salience rules for part/whole decomposition and those based on typicality from the current version of Kalos would still be applicable.) Next, a new lexicon would be needed to store surface representations for objects in the domain-specific knowledge base. Then the grammar rule knowledge base would have to be extended to be able to express any new frames added to the domain-specific knowledge base. The majority of the new rules would be included to surface new process frames.

The revisors would require minimal modification to function in other domains. The preferred term knowledge base would have to be replaced to reflect preferred terms in the new domain. Since the revisors use preferred term knowledge in an automatic fashion, no other changes would be needed to support this change. The fact that minimal change is needed in the revisors speaks favorably to the decomposition method applied to the generation task described in this dissertation.

(Section 6.3.1 describes one domain-specific conceptual revision rule that removes the statement of operand size for a register when the register handles only one operand size that is equal to the register's size. A more general redundancy rule that is not domain specific could be used, thus making the current Kalos revisors non specific with respect to the domain at hand.)

7.3 Reflections on Revision

Throughout this dissertation, revision has been presented as an engineering technique for the reduction of the overall complexity of a natural language generation system. But in this section, we reflect on these techniques with respect to human editors.

Meeter [1991] studied the types of revisions performed by human editors in reviewing text produced by competent authors. These editors mostly performed stylistic revision without changing the meaning of the text. But what of domain experts who edit text produced by competent writers who are domain novices? Let us speculate about the types of revisions such editors would perform. First, the domain expert might correct misconceptions of the domain novice author. Second, the domain expert may suggest salient facts that should be included in the text to support the given discourse goal. The domain novice may not be aware of the facts or may not understand the domain well enough to grasp that the facts are relevant.

The first type of revising, like that studied by Meteer, deals mostly with knowledge and techniques from the discipline of writing. If domain knowledge is used, it is used in a declarative fashion without the need for reasoning about the domain. For example, an editor may suggest the use of the term "user friendly" in favor of "easy to use" in an advertisement for software. The selection of the term is not based on some deep knowledge about human/computer interaction, but on knowledge about the current "buzz" words for software advertisements. These are the types of

revisions labeled *stylistic* in the previous chapters.

But consider conceptual revisions. Let us divide conceptual revisions into two types. First, there are conceptual revisions that have their roots in the discipline of writing and only use domain-specific knowledge in a declarative fashion. Reordering lists of attributes, shorting long lists by stating categories instead of category members, and removing redundant statements based on surface-level effects are examples of the this type of conceptual revision which I will call *discourse-based conceptual revision.* It appears that a professional editor with little or no domain knowledge could perform this type of revision.

The second type of conceptual revision is rooted squarely in reasoning about the domain at hand. Reasoning about what is salient in a particular domain with respect to a given discourse goal and deciding when an example is needed to help explain a difficult description are illustrations of the second type of conceptual revision which I will call *domain-based conceptual revision.*

Consider that Kalos performs stylistic and discourse-based conceptual revisions. A deep understanding of the domain at hand is not needed to apply these types of revision, i.e., domain knowledge is used in a declarative fashion. It therefore appears that the techniques that Kalos uses begin to capture the mechanism professional editors carry out when dealing with the text of domain knowledgeable, competent writers.

(At first glance, it appears that the application of preferred terms in Kalos are domain-based conceptual revisions since inferencing is used to relate the object being described to some frame that will result in the use of a preferred term. But the knowledge used is mostly about the taxonomy that describes the universe of objects. The system asks "how can I use this term to describe this object?" Deep knowledge about the domain is not needed, just how to locate facts that can be used to cast an object in terms of a preferred word or phrase.)

Consider domain-based conceptual revisions. Can this type of revision be made independent of the domain at hand? The same question can be asked about deep generators: Are the limitations placed on deep generators to make them implementable the cause of their domain-specific behavior or are there domain-specific elements of writing?

To begin to think about this question, consider writers who author popularized books about science. Some authors have covered a wide variety of fields from nuclear physics to biology. It appears such authors can apply writing skills, including reasoning about what is salient in a particular domain, to a number of fields, at least when writing at the tutorial level. If such authors indeed have domain-independent writing and analytical skills, then perhaps more advanced deep generators and conceptual revisions could be designed to use all domain knowledge in a declarative fashion, especially systems like Kalos that produce draft-quality text. In such systems, domain-

specific salient facts can be added by the human editing the draft text.

On the other hand, highly technical documents are written by domain experts. It may be that the acts of determining salience and organizing the text for this type of prose requires such deep reasoning about the domain at hand that the complete separation of domain and writing knowledge is not possible. Unlike tutorial prose where salient information may be located by reasoning about simple features like typicality, taxonomic relationships, and quantities, salience determination in highly technical prose is based on domain-specific facts and relationships.

It appears to the author that much of the domain-specific nature of deep generators and conceptual revisors is not an inherent limitation related to the task of natural language generation but a limitation of the techniques at hand. To make deep generation easier to deal with, system designers integrate domain-specific knowledge into them. For example, a few domain-specific salience rules can reduce the overall size and complexity of the system since the system does not have to perform deep reasoning about the domain, relationships between facts in the domain, and historical information to determine salience. But given an extensive knowledge base about the domain at hand, it is conceivable that a draft text deep generator could infer salience, much like a human author appears to do. Similarly, a domain independent conceptual revisor, working to produce draft-quality text, appears possible.

In summary, Kalos performs revisions like those made by a professional editor who has little or no domain knowledge to text written by domain-knowledgeable, competent writers. It appears that much of the domain-specific design of deep generators is a limitation with current generation techniques. A domain independent natural language generator that uses domain knowledge declaratively may be achievable, especially when the goal is to produce draft-quality or tutorial text, when more powerful reasoning techniques and large domain knowledge bases are available.

7.4 Future Work

The type of text that Kalos can generate is illustrated in Chapter 6. The system could be extended by adding more knowledge about the M68000 microprocessor, enhancing the schema templates and grammar rules, and adding more conceptual and stylistic revision rules. Currently, conceptual revision and stylistic revision are pipelined. The basic architecture of Kalos does not require this ordering which limits the interaction between conceptual decisions and lexical choice. I plan to relax the ordering of revision analysis so that greater interaction between the two types of revision can occur.

Another major area of consideration deals with the way Kalos performs revision. Once a revision is performed, it can not be removed by later revision iterations. Techniques to relax this

limitation are needed to improve the flexibility of the system (see section 3.6).

One other area of future research relates to the nature of SNePS-2.1 rules in Kalos. Although rules are represented in the uniform knowledge base of Kalos, it is not possible in SNePS-2.1 to make inferences about SNePS rules. If we need to have Kalos reason about SNePS rules, such as those used to infer the salience of knowledge base facts, we cannot do it directly. Instead, we have to include knowledge in the system that describes the rules themselves. An improvement on this system would be to develop Lisp code that would read SNePS-2.1 rules and build knowledge base assertions to describe them in a uniform way. With this capability, SNePS rules could be written to make inferences about SNePS rules in Kalos by inspecting uniformly built knowledge about Kalos rules. One use of being able to make inferences about rules would be to automatically partition the knowledge bases for each stage of generation. Surrogates could be built by tracing the antecedents of the SNePS-2.1 rules.

Kalos could also be extended to include more knowledge about paragraph boundaries. The conceptual revisor needs paragraph information when dealing with the removal of information that is restated in a paragraph. During stylistic revision, Kalos assumes that certain groups of schemata constitute a paragraph when applying revision rules related to thematic progression. One problem with the current method is that it performs well at the top level description of objects that are relatively complex in terms of their attributes and parts. For simpler objects, the current method could produce many, single sentence paragraphs. A better approach would be to include knowledge about what constitutes a paragraph and have Kalos infer where the boundaries occur.

A major enhancement to Kalos would be the addition of ambiguity checking techniques. The stylistic revisor is an ideal place to check for ambiguity because it has access to both the surface text and the underlying knowledge from which the sentence was generated. With an appropriate natural language understanding program and extensive lexicon, the surface text could be read for ambiguities. From the underlying structures of the surface text, Kalos could determine the intent of the text and look for other ways to generate the text. If no unambiguous text could be found, Kalos could indicate that fact so that a human author could polish the text.

The types of conceptual revisions suggested here are interesting, but somewhat limited in nature. Future research in the area of text generation with revision should concentrate on more complex conceptual revisions similar to what human authors do [Yazdani 1987]. Such revision analysis would examine draft text in light of broad discourse goals, potentially removing, adding, or extensively reordering the text.

Appendix A
SNePS-2.1 Tutorial

A little inaccuracy sometimes saves tons of explanation.

Saki

SNePS-2.1

Kalos was implemented as a testbed for developing revision techniques in a knowledge intensive natural language generation system. It was developed in the SNePS-2.1 semantic network processing package [Shapiro 1992]. "SNePS (the Semantic Network Processing System) is a system for building, using, and retrieving from propositional semantic networks"†. SNePS-2.1 is written in Common Lisp. This section briefly describes the features of SNePS-2.1 required to understand the discussion of Kalos that follows this section.

A semantic network is a directed graph in which nodes represent concepts and arcs represent binary relations. Figure A.1a shows a simple SNePS semantic network. Concepts A and B are related by R. In SNePS, propositions are represented by nodes, while relations are part of the syntactic structure of the nodes from which they emanate.

For each arc R, there is a hidden, inverse arc R-. If R emanates from node x and points to node y, then R- emanates from node y and points to node x. Inverse arcs are not shown explicitly, but they can be used for specifying paths in the network. Figure A.1b shows an inverse arc.

SNePS provides mechanisms to build semantic networks, retrieve information via pattern matching, and perform backward and forward inferencing. SNePS deduction rules are represented as semantic networks. SNePS includes a belief revision system which was used in an earlier version of Kalos to partition the semantic network knowledge base dynamically to improve inferencing performance.

SNePS nodes have names which may be user or system assigned. Once a node is created, it can be referenced by its name. A node may be asserted or unasserted. SNePS displays asserted nodes by appending a "!" to the name, e.g., node *M14* would be displayed by SNePS as *M14!* if it were asserted. The *assert* macro creates an asserted node, while *build* creates an unasserted node. Unasserted nodes nodes may be asserted using the *!* operator.

SNePS performs two types of pattern matching. In the simple type, a single relation is used to specify the pattern to be matched on. The second type of pattern matching is called path-based

† Shapiro 1992.

inferencing. A path is defined as an arc, an inverse arc, or a sequence of paths defined using operators such as composition, logical *and*, and logical *or*. Path-based inferencing is defined by

$$\forall(n1, n2) path(n1, n2) \Rightarrow relation(n1, n2)$$

If *path* is a SNePS path between node *n1* and node *n2*, then SNePS behaves as if there is a *relation* arc between the two nodes.

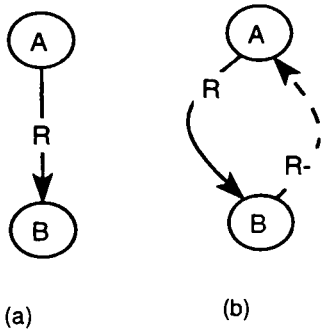


Figure A.1 - (a) SNePS Subgraph and (b) Inverse Relation.

Kalos uses three path operators: compose, kstar, and or. If for nodes $x_1, x_2, ... x_n$ and paths $p_1, p_2, ..., p_{n-1}$ where p_i is a path from x_i to x_{i+1}, then (*compose* $p_1 p_2 ... p_{n-1}$) is a path from x_1 to x_n. If path p composed with itself zero or more times is a path from node x to node y, then (kstar p) is a path from x to y. If any of the paths $p_1, p_2, ..., p_n$ is a path from node x to node y, then (*or* $p_1 p_2 ... p_n$) is a path from node x to node y.

The SNePS-2.1 *find* and *findassert* commands perform pattern matching. *Find* looks for the given pattern anywhere in the semantic network, while *findassert* looks only for asserted nodes. Variables can be used in SNePS-2.1 patterns using the "?" operator. For example, (findassert member m68000 class ?cl) will find and return a list of asserted nodes that have a *member* arc pointing to the M68000 node and a *class* arc pointing to another node. When the *findassert* command is executed, the nodes pointed to by the *class* arcs for all the subgraphs found by *findassert* are assigned to the SNePS-2.1 *cl* variable.

When pattern matching commands are embedded, such as

```
(find consider (find member ?x class ?y))
```

the embedded *find* locates all the nodes that are *member/class* frames[‡]. The outer *find* returns a list of nodes that have *consider* arcs pointing to any of the *member/class* frames. The nodes contained in the variables x and y contain a subset of the nodes that satisfy the *member/class* pattern. The variables are restricted to include only those nodes that are also pointed to by a *consider* arc.

SNePS-2.1 uses a number of non-standard logical connectives for its inferencing rules. Kalos uses three of these: and-entailment, or-entailment, and andor. And-entailment rules are of the form

$$(assert\ \&ant\ ((A_1, ..., A_n))\ cq\ ((C_1, ...C_m)))$$

where $A_1, ..., A_n$ indicate SNePS-2.1 nodes that are rule antecedents and $C_1, ..., C_m$ indicate SNePS-2.1 nodes that are rule consequents. The rules may contain variables. This type of rule means that the conjunction of the antecedents implies the conjunction of the consequents.

$$(assert\ ant\ ((A_1, ..., A_n))\ cq\ ((C_1, ...C_m)))$$

means that the disjunction of the antecedents implies the conjunction of the consequents.

$$(assert\ min\ 0\ max\ 0\ arg\ ((P_1, ..., P_n))$$

is the form of the *andor* rule used in Kalos. This rule means that nodes $P_1, ..., P_n$ are false.

SNePS-2.1 rules introduce variables using the *forall* arc. For example, the rule

```
(assert forall ($object $feature)
   ant (build object *object atypical *feature)
   cq (build object *object attribute *feature))
```

says that if some object has a feature labeled atypical, then the feature is a salient attribute of the object. The variables *object* and *feature* are introduced with the dollar sign operator and referenced using the * operator. SNePS-2.1 rules are consulted when inferencing is invoked using the *deduce* command.

If a SNePS-2.1 rule has two variables, x and y, consequents are not built in cases where $*x = *y$. This allows rules to be written where pairs of frames of the same type can be examined without comparing a frame to itself. For example,

[‡] The term *frame* is used to refer to the semantic networks used to represent domain knowledge in Kalos. Frames contain declarative knowledge in semantic network graphs with predefined internal relations [Barr & Feigenbaum 1981].

```
(assert forall ($x $y)
        &ant ((build member *x class microprocessor)
              (build member *y class microprocessor))
        cq (build pairs *x pairs *y))
```

This rule will build *pairs* o_1/*pairs* o_2 frames for every two sets of nodes o_1 and o_2 in the network that are members of the class microprocessor, but since *x cannot equal *y, *pairs/pairs* frames will not be built where both *pairs* arcs point to the same node.

The # macro in SNePS-2.1 creates a base (i.e., leaf) node and assigns a system generated name to it. In Kalos, this macro is used to create nodes like p and q in Figure 5.2b.

This appendix is a brief overview of SNePS-2.1. For more information, consult [Shapiro 1992].

References

Appelt, Douglas E. 1983. TELEGRAPH: A grammar formalism for language planning. *Proceedings of the 8th IJCAI,* Karlsruhe, August, pp 595-99.

Appelt, Douglas E. 1985. *Planning English Sentences.* Cambridge: Cambridge University Press.

Baeten, J.C.M., Bergstra, J. A., Klop, J.W. 1987. Term rewriting systems with priorities. In *Rewriting Techniques and Applications.* Lescanne, Pierre, ed., Berlin: Springer-Verlag, 83-94.

Baeten, J.C.M. & Weijland, W.P. 1987. Semantics for prolog via term rewrite systems. In *Conditional Term Rewriting Systems.* Kaplan, S. and Jouannaud, J.-P., eds., Berlin: Springer-Verlag, 3-14.

Barr, A. & Feigenbaum, E. A. 1981. *The Handbook of Artificial Intelligence,* Volume 1. Los Altos: William Kaufmann.

Bates, M. 1978. The theory and practice of augmented transition network grammars. In *Natural Language Communication with Computers.* Bolc, L., ed., Berlin: Springer-Verlag, 191-259.

Cattell, R. G. G. 1980. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems.* 2 (2), 173-190.

Clark, H. H. & Clark, E. V. 1977. *Psychology and Language.* New York: Harcourt Brace Jovanovich.

Cline, B. E. & Nutter, J. T. (1990) Implications of natural categories for natural language generation. In *Current Trends in SNePS - Semantic Network Processing System.* D. Kumar, ed., Springer-Verlag, Berlin, 153-162.

Cline, B. E. 1991. Conceptual revision for natural language generation. *Proceeding of the 29th Annual Meeting of the Association for Computational Linguistics.* (Student Session.) Berkeley, June, 347-348.

Cline, B. E. & Nutter, J. T. 1992. Knowledge-based natural language generation with revision. *Proceedings of the 5th Florida Artificial Intelligence Research Symposium* (Ft. Lauderdale, Florida, April 7-10). Florida Artificial Intelligence Research Symposium, 223-227.

Cline, B.E. & Nutter, J.T. 1994a. Generating and Revising Text: A Fully Knowledge-Based Approach. *International Journal of Expert Systems, Research and Applications,* 7(2).

Cline, B.E. & Nutter, J.T. 1994b. Kalos - A system for natural language generation with revision. *Proceedings of the American Association for Artificial Intelligence,* (forthcoming).

Collins, A. & Gentner, D. 1980. A framework for a cognitive theory of writing. In *Cognitive Processes in Writing.* Gregg, L. W. & Steinberg, E. R., eds., Hillsdale, New Jersey: Lawrence Erlbaum Associates, 51-71.

Danlos, L. 1987. *The Linguistic Basis of Text Generation.* Cambridge, Cambridge University Press.

Dershowitz, Nachum & Jouannaud, Jean-Pierre. 1990. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics.* van Leeuwen, Jay, ed., Cambridge: Elsevier and MIT Press, 243-320.

Emero, Michael. 1992 Using Naturally occurring texts as a knowledge acquisition resource for knowledge base design: developing a knowledge base taxonomy on microprocessors. Masters project report. August. Computer Science Department. Virginia Polytechnic Institute and State University.

Flower, L. S. & Hayes, J. R. 1980. The dynamics of composing: making and juggling constraints. In *Cognitive Processes in Writing.* Gregg, L. W. & Steinberg, E. R., eds., Hillsdale, New Jersey: Lawrence Erlbaum Associates, 31-50.

Fraser, C. W. 1977. *Automatic Generation of Code Generator.* Ph. D. Dissertation, Yale University.

Gabriel, R. P. 1988. Deliberate writing. In *Natural Language Generation Systems,* McDonald, D. D. & Bolc, L., eds., Springer-Verlag, Berlin, 1-46.

Genesereth, Michael R. & Nilsson, Nils J. 1987. *Logical Foundations of Artificial Intelligence.* Palo Alto: Morgan Kaufmann.

Glatt, B. S. 1982. Defining thematic progressions and their relationship to reader comprehension. In *What Writers Know: The Language, Process, and Structure of Written Discourse.* Nystrand, M, ed., New York: Academic Press, 87-103.

Grice, H. 1975. Logic and conversation. In *Syntax and Semantics: Volume 3, Speech Acts,* Cole, P. & Morgan, J., eds., New York: Academic Press, 43-58.

Grosz, B. J. 1979. Utterance and objective: Issues in natural language communication. *Proceedings, Sixth International Joint Conference on Artificial Intelligence.* Tokyo: August, 1067-1076.

Grosz, B. J. & Sidner, C. L. 1986. Attention, intentions, and the structure of discourse. *Computational Linguistics,* 12(3), 175-204.

Halliday, M. A. K. & Hasan, R. 1976. *Cohesion in English.* London: Longman.

Hayes, J. R. & Flower, L. S. 1980. Identifying the organization of writing processes. In *Cognitive Processes in Writing.* Gregg, L. W. & Steinberg, E. R., eds., Hillsdale, New Jersey: Lawrence Erlbaum Associates, 3-30.

Hobbs, J. R. 1982. Towards an understanding of coherence in discourse. In *Strategies for Natural Language Processing.* Lehnert, W. G. & Ringle, M. H., eds., Hillsdale, New Jersey: Lawrence Erlbaum Associates, 223-243.

Hovy, Eduard H. 1988. *Generating Natural Language Under Pragmatic Constraints.* Hillsdale, N. J.: Lawrence Erlbaum Associates.

Inui, K., Tokunaga, T., & Tanaka, H. 1992. Text revision: a model and its implementation. In *Aspects of Automated Natural Language Generation,* R. Dale, E. Hovy, D. Rosner, and O. Stock, eds., Springer-Verlag, Berlin, 45-56.

Jacobs, P. S. 1985. *A Knowledge-Based Approach To Language Production.* Ph. D. Dissertation. December. University of California, Berkeley.

Jantzen, Matthias. 1988. *Confluent String Rewriting.* Berlin: Springer-Verlag.

Kantrowitz, M. & Bates, J. 1992. Integrated natural language generation systems. In *Aspects of Automated Natural Language Generation,* R. Dale, E. Hovy, D. Rosner, and O. Stock, eds., Springer-Verlag, Berlin, 45-56.

Kay, M. 1984. Functional unification grammar: a formalism for machine translation. *Proceedings of the Tenth International Conference on Computational Linguistics.* Stanford, California, 75-78.

Kieras, D. E. 1989. An advanced computerized aid for the writing of comprehensible technical documents. In *Computer Writing Environments: Theory, Research, and Design.* Britton, B. K. and Glynn, S. M., eds., Hillsdale: Lawrence Erlbaum Associates, 143-168.

Kukich, K. 1983. *Knowledge-Based Report Generation: A Knowledge-Engineering Approach to Natural Language Report Generation.* Ph. D. Dissertation. August. University of Pittsburgh.

Lenat, D. B., Guha, R. V., Pittman, K., Pratt, D., & Shepherd, M. 1990. CYC: Toward programs with common sense. *Communications of the ACM.* 33(8), 30-49.

Mann, W. C. & Thompson, S. A. 1987. Rhetorical Structure Theory: description and construction of text structures. In *Natural Language Generation.* Kempen, G., ed., Dordrecht: Martinus Nijhoff, 85-96.

Matthiessen, C. M. I. M. 1981. A grammar and a lexicon for a text-production system. *Proceeding of the 19th Annual Meeting of the Association for Computational Linguistics.* Stanford, California, June, 49-55.

McCoy, K. F. 1985. The role of perspective in responding to property misconceptions. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence.* Los Angeles, August, pp. 791-93.

McDonald, David D. 1988. Natural language generation. Tutorial presented at 26th Annual Meeting of the Association for Computational Linguistics. State University of New York. Buffalo, New York.

McKeown, K. R. 1985. *Text Generation.* Cambridge: Cambridge University Press.

McKeown, K. R. & Swartout, W. R. 1987. Language generation and explanation. *Annual Review of Computer Science.* Volume 2, 401-449.

Meehan, J. 1981. Tale-spin. In Inside Computer Understanding: Five Programs Plus Five Miniatures, Schank, R. C. & Riesbeck, C. K., eds., Hillsdale: Lawrence Erlbaum Associates, 197-226.

Meteer, M. 1991. The implications of revision for natural language generation. In *Natural Language Generation in Artificial Intelligence and Computational Linguistics,* C. Paris, W. Swartout, and W. Mann, eds., Kluwer Academic Publishers, Boston, 155-177.

Neal, J. G. & Shapiro, S. C. 1987. Knowledge-based parsing. In *Natural Language Parsing Systems,* Bolc, L. (Ed.) 1987. Berlin: Springer-Verlag, 49-92.

Nutter, J. T. 1989. Knowledge based lexicons. In *Current Trends in SNePS - Semantic Network Processing System,* D. Kumar, ed., Springer-Verlag, Berlin, 97-106.

Paris, C. L. 1985. Descriptions strategies for naive and expert users. *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics.* Chicago, July, 238-245.

Reichman-Adar, R. 1984. Extended person-machine interface. *Artifical Intelligence.* **22,** 157-218.

Reiter, E. Generating descriptions that exploit a user's domain knowledge. In *Current Research in Natural Language Generation.* Dale, R., Mellish, C., and Zock, M., eds., London: Academic Press, 257-285.

Robin, J. 1993. A revision-based generation architecture for reporting facts in their historical context. In *New Concepts in Natural Language Generation: Planning, Realization and Systems,* H. Horacek and M. Zock, eds., Printer Publishers, London, 238-268.

Rosch, E., Mervis, C. B., Gray, W. D., Johnson, D. M. & Boyes-Braem, P., 1976. Basic objects in natural categories. *Cognitive Psychology,* Volume 8, 382-349.

Rösner, Dietmar & Stede, Manfred 1992. Customizing RST for the automatic production of technical manuals. In *Aspects of Automated Natural Language Generation,* R. Dale, E. Hovy, D. Rösner, and O. Stock, eds., Springer-Verlag, Berlin, 199-214.

Rubinoff, R. 1992. Integrating text planning and linguistic choice by annotating linguistic structures. In *Aspects of Automated Natural Language Generation,* R. Dale, E. Hovy, D. Rösner, and O. Stock,eds., Springer-Verlag, Berlin, 45-56.

Schank, R. C. & Riesbeck, C. K. 1981. *Inside Computer Understanding.* Hillsdale: Lawrence Erlbaum Associates.

Shapiro, S. C. 1975. Generation as parsing from a network into a linear string. *American Journal of Computational Linguistics*, Volume 1, 45-62.

Shapiro, S. C. 1992. *SNePS-2.1 User's Manual.* Department of Computer Science. State University of New York at Buffalo.

Simmons, R. & Slocum, J. 1972. Generating English discourse from semantic networks. *Communications of the ACM* 15(10), 891-905.

Vaughan, M. M. & McDonald, D. D. 1986. A model of revision in natural language generation. *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics.* New York, June, 90-96.

Woods, W. A. 1970. Transition network grammars for natural language analysis. *Communications of the ACM* 13(10), 591-606.

Yazdani, M. 1987. Reviewing as a component of the text generation process. In *Natural Language Generation.* Kempen, G., ed., Dordrecht: Martinus Nijhoff, 183-190.

Vita

Ben Eric Cline was born on October 12, 1951 in Wytheville, Virginia. After graduating from George Wythe High School, he obtained a Bachelor of Science in Computer Science from Virginia Tech in 1974 and a Master of Science in Computer Science and Applications from Virginia Tech in 1976. In 1985 he first seriously considered pursuing a Doctorate in Computer Science and formally entered the program at Virginia Tech in June of 1988.

He has worked in the Virginia Tech Computing Center, Systems Development, and Communications Network Services departments, beginning as a Computer Programmer in 1976 and progressing to a Communications Engineer Manager.

Dr. Cline has written two books: *Microprogramming Concepts and Techniques* and *An Introduction to Automated Data Acquisition*. He is a member of Upsilon Pi Epsilon, Phi Kappa Phi, the Association for Computing Machinery, and the American Association for Artificial Intelligence.

During 1991, he achieved a lifelong dream: becoming a radio amateur. He rapidly advanced to the Amateur Extra Class. He currently holds callsign AC4XO.

On August 12, 1978 he married Sue Ellen Jolly, and they lived happily ever after.