INTRODUCTION TO KNOWLEDGE SYSTEMS

MARK STEFIK

Introduction to
Knowledge Systems

Mark Stefik

Morgan Kaufmann Publishers, Inc.
San Francisco, California

Sponsoring Editor Michael B. Morgan
Production Manager Yonie Overton
Production Editor Elisabeth Beller
Editorial Coordinator Marilyn Uffner Alan
Text Design, Project Management,
Electronic Illustrations and Composition Professional Book Center
Cover Design Carron Design
Copyeditor Anna Huff
Printer Quebecor Fairfield

Morgan Kaufmann Publishers, Inc.
Editorial and Sales Office
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205 USA
Telephone 415/392-2665
Facsimile 415/982-2665
Internet mkp@mkp.com

Library of Congress Cataloging-in-Publication Data is available for this book.

ISBN 1-55860-166-X

Printed in the United States of America

99 98 97 96 95 5 4 3 2 1

Y\96\e

Contents

Introduction and Overview

The Building of a Knowledge System to Identify Wild Plants

There is always a tension between top-down and bottom-up presentations. A top-down presentation starts with goals and then establishes a framework for pursuing the parts in depth. Bottom-up presentations start with fundamental and primitive concepts and then build to higher-level ones. Top-down presentations can be motivating but they risk lack of rigor; bottom-up presentations can be principled but they risk losing sight of goals and direction.

Most of this book is organized bottom-up. This reflects my desire for clarity in a field that is entering its adolescence, metaphorically if not chronologically. The topics are arranged so a reader starting at the beginning is prepared for concepts along the way. Occasionally I break out of the bottom-up rhythm and step-at-a-time development to survey where we are, where we have been, and where we are going. This introduction serves that purpose.

The following overview traces the steps of building a hypothetical knowledge system. Woven into the story are some notes that connect it with sections in this book that develop the concepts further. Many of the questions and issues of knowledge engineering that are mysterious in a bottom-up presentation seem quite natural when they are encountered in the context of building a knowledge system. In particular, it becomes easy to see why they arise.

I made up the following story, so it does not require a disclaimer saying that the names have been changed to protect the innocent. Nonetheless, the phenomena in the story are familiar to anyone who has developed a knowledge system. Imagine that we work for a small software company that builds popular software packages including knowledge systems. This is a story of a knowledge system: how it was conceived, built, introduced, used, and later extended.

To Build a Knowledge System

It all began when we were approached by an entrepreneur who enjoys hiking and camping in the hills, mountains, and deserts of California. Always looking for a new market opportunity, he noticed that campers and hikers like to identify wild plants but that they are not very good at it. Identifying wild plants can be useful for survival in the woods ("What can I eat?") and it also has recreational value. He was convinced that conservationists, environmentalists, and well-heeled hikers have a common need.

The entrepreneur proposed that we build a portable knowledge system for identifying wildlife. He had consulted a California hiking club and a professional naturalist. He suggested that we begin by constructing a hypertext database about different kinds of plants, describing their appearance, habitats, relations to other wildlife, and human uses. Our initial project team is as follows:

- A hike representing the user community and customer.
- A naturalist, our domain expert on wildlife identification.
- A knowledge engineer, our expert in acquiring knowledge and knowledge representation.
- A software engineer, the team leader having overall responsibility for the development of software.

After some discussion within the company we agreed to develop a prototype version of the knowledge system using the latest palmsize or "backpack" computers. If the technical project seemed feasible, we would then consider the next steps of commercialization. We planned to use the process of building the prototype to help us determine the feasibility of a larger project. We recruited the naturalist and a prominent member of one of the hiking clubs to our project team. We called the group together and started to learn about each other's ideas and terminology.

Notes The participants are just getting started. They need to size up the task, develop their goals, and determine their respective roles on the project. They need to consider many questions about the nature of the knowledge system they would build. They ask "Who wants it?" because the situation and people matter for shaping the system. They also ask "What do these people do?" and "What role should the system play?" because these issues arise in all software engineering projects.

Connections See Chapter 3 for a discussion of the initial interview concepts and background on software engineering.

Our Initial Interviews

Our naturalist tells us he wants to focus on native trees of California. We begin with the famous California redwood trees, the *Sequoia sempervirens* or coastal redwood and the *Sequoiadendron giganteum* or giant redwood that thrives in the Sierras. Our naturalist is a stickler for completeness. He also adds the *Metasequoia glyptostroboides* or dawn redwood, which grew in most parts of North America. The dawn redwood was thought to have become extinct until a grove was discovered in China in 1944. At a blackboard he draws a chart of plant families as shown in Figure I.1. He tells us about the history of the plant kingdom:

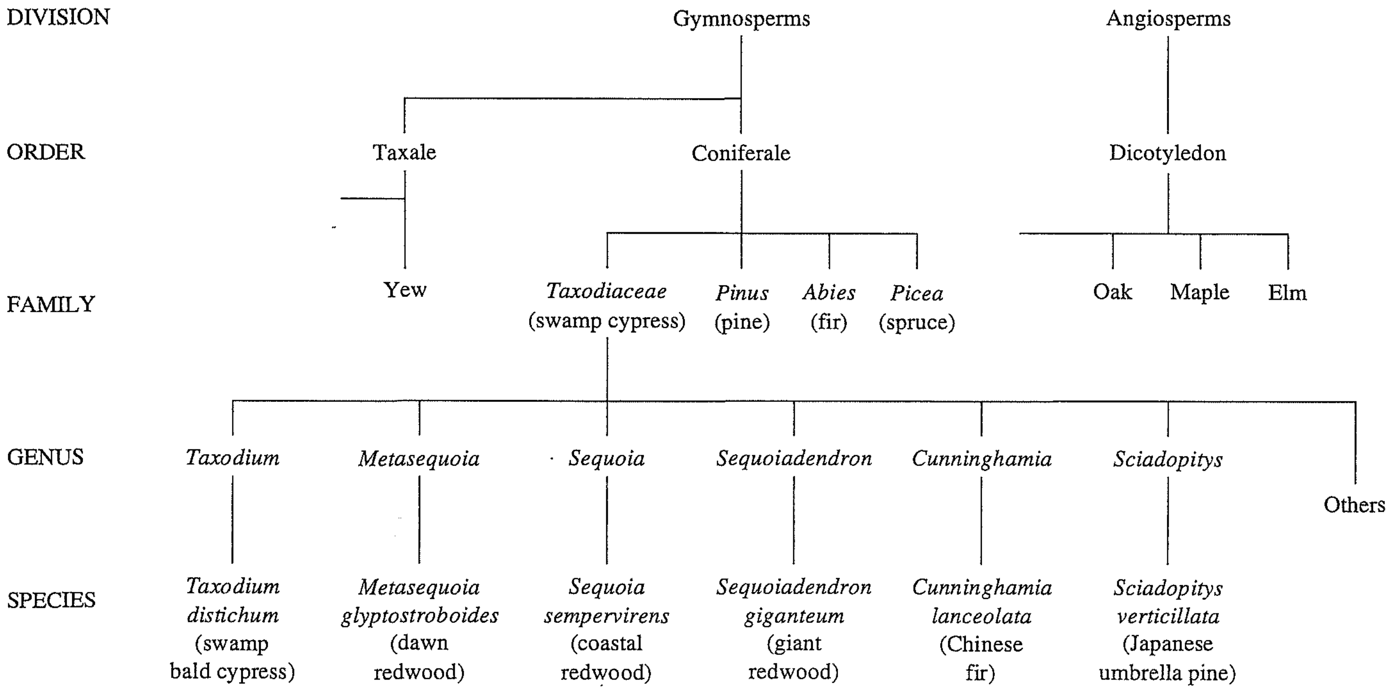

DIVISION
ORDER
FAMILY
GENUS
SPECIES
Gymnosperms
Angiosperms
Taxale
Coniferale
Dicotyledon
Yew
Taxodiaceae
(swamp cypress)
Pinus
(pine)
Abies
(fir)
Picea
(spruce)
Oak
Maple
Elm
Taxodium
Metasequoia
Sequoia
Sequoiadendron
Cunninghamia
Sciadopitys
Others
Taxodium distichum (swamp bald cypress)
Metasequoia glyptostroboides (dawn redwood)
Sequoia sempervirens (coastal redwood)
Sequoiadendron giganteum (giant redwood)
Cunninghamia lanceolata (Chinese fir)
Sciadopitys verticillata (Japanese umbrella pine)

FIGURE I.1. A partial taxonomy showing relations among plants closely related to California redwood trees. In our scenario and thought experiment, the naturalist was asked about redwoods and started lecturing about plant families.

> Plants evolved on Earth from earlier one-celled animals. About 200 million years ago was the age of conifers, the cone-bearing trees. Redwoods are members of the conifers, which were the dominant plant species at that time. They are among the gymnosperms, plants that release their seeds without a protective coating or shell.

Our naturalist is a gifted teacher but he tends to slip into what we have started to call his "lecture mode." After an hour of exploring taxonomies of the plant kingdom we begin to get restless. One of the team members interrupts him to ask the proper location in the taxonomy for the "albino redwoods," which are often visited in Muir Woods. This question jars the naturalist. Albinos do not fit into the plant taxonomy because they are not a true species, but rather a mutated parasite from otherwise normal coastal redwoods. Redwoods propagate by both seeds and roots. Sometimes something goes wrong in the root propagation, resulting in a tree that lacks the capability to make chlorophyll. Such rare plants would normally die, except a few that continue to live parasitically off the parent. Albino trees have extra pores on their leaves that make them efficient for moving quantities of water and nutrients from their host and parent trees.

At this point another member of the group, a home gardener, wants to know about trees she had purchased at a local nursery called *Sequoia sempervirens soquel* and *Sequoia sempervirens aptos blue*. Again, the naturalist explains that these trees are not really species either. Rather, they are clones of registered individual coastal redwoods, propagated by cuttings and popular with nurseries because they grow to be predictable "twins" to the parent tree, having the same shape and color. These registered clones are sometimes called cultivars. They would be impossible to identify reliably by visual examination alone and they are not found in the wild.

This leads us to a discussion of exactly what a taxonomic chart means, what a species is, what the chart is useful for, and whether it is really a good starting place for plant identification. Clearly the chart does not contain all the information we need about plants because some plants of apparent interest do not appear in it. We also have learned that there are some plants about which there is debate as to their lineage. After discussion we decide that the information is interesting and that it would be a good base for establishing names of plants, but that it would not be appropriate for us to proceed by just filling out more and more of the taxonomic chart. We decide to focus on actual cases of plant identification at our next session.

Notes In this part of the story the participants are beginning to build bridges into each other's areas to understand how they will work together. They bring to the discussion some preexisting symbol structures or representations, such as the plant taxonomic chart. Often there needs to be discussion about just what the symbols mean and whether those meanings are useful for the task at hand. As in this story, it sometimes turns out that these symbols and representations need to be modified. When the end product includes a knowledge system, then conventions about symbol structures must be made precise enough for clear communication and also expressive enough for the distinctions made in performing the task.

Connections See Chapter 1 for an introduction to symbols and symbol structures and the assignment of meaning to them. See Chapter 3 for a discussion of tools and methods for incremental formalization of knowledge.

1. The specimen is tall,
2. I'd guess about 30 or 35 feet tall.
3. So it's a tree . . .
4. symmetrical in shape.
5. From the needles in the foliage, it's obviously a pine,
6. but not one of the coastal pines since we're at too high an elevation in these mountains.
7. Could be either a *Pinus ponderosa*, a *jeffreyi*, or a *torreyana.*
8. Let's see (walking in closer) . . . dark green needles, not yellow-green,
9. about 7 inches long, and
10. in clusters of three.
11. Rather grayish bark, not cinnamon-brown.
12. Medium-sized cones.
13. Seems to be a young tree. Others like it are near, reaching heights of over a hundred feet tall.
14. It's probably a *Pinus jeffreyi*, that is, a Jeffrey pine.

FIGURE I.2. Transcription of our naturalist talking through the identification of one of the plants.

The Naturalist in the Woods

We prepare to study the naturalist's classification process on some sample cases. One member of the group sets up portable video and audio recorders at a local state park. Our hiking club member is our prototype user. We define his job as walking into the woods and selecting a plant to be identified. In this way we hope to gain insight into what plants he finds interesting and to test the relevance of the plant taxonomy. We ask the naturalist to "think out loud" as he identifies plants. Recording such a session is called taking a protocol. This results in verbal data where the naturalist talks about the bark coloration, surface roots, and leaf shapes. After these dialogs are recorded and pictures of the plants are taken, we transcribe all of the tapes. Figure I.2 shows a sample transcript.

After the session in the park, we go over the transcripts carefully with the naturalist, trying to reconstruct any intermediate aspects of his thought process that were not verbalized. We ask him a variety of questions. "What else did you consider here? How did you know that it was not a manzanita? Why couldn't it have been a fir tree or a digger pine? Why did you ask about the coloring of the needles?" Our goal is not so much to capture exactly what his reasoning was in every case, but rather to develop a set of case examples that we could use as benchmarks for testing our computer system. As it turns out, the naturalist does different things on different cases. He does not always start out with exactly the same set of questions, so his method is not one of just working through a fixed decision tree or discrimination network.

Notes At this point the group has begun a process of collecting knowledge about the task in terms of examples of problem-solving behavior. As we will see below, it is possible to make some false starts in this, and it is also possible to recover from such false starts.

Connections See Chapter 3 for discussion of the assumptions and methods of the "transfer of expertise approach.

Characterizing the Task

The knowledge engineer begins a tentative analysis of the protocols. He tells us he might need to analyze these sessions several different ways before we are done. He wants to characterize the actions of the naturalist in terms of problem-solving steps. His approach is to model the problem-solving task as a search problem, in which the naturalist's steps carry out different operations in the search. Figure I.3 shows his first tentative analysis of the session from Figure I.2. In this, he

Collect Initial Data
Determine height of plant: Plant is more than 30-feet tall. (1)
Shape is symmetrical. (4)
Foliage has needles. (5)

Determine General Classification
Infer: Plant is a tree. (3) Plant is a pine tree or a close relative.
Knowledge: Only pines and close relatives have needle-shaped leaves. (5)

Collect Data about Location
Mountain location.
Knowledge: Trees from the low areas and the coast do not grow in the mountains. (6)
Rule out candidates that do not grow in this region.

Form Specific Candidate Hypotheses
Mountain pine trees include the *Pinus ponderosa*, the *jeffreyi*, and the *torreyana*. (7)

Determine Data to Discriminate among Hypotheses
Knowledge: The hypotheses make different predictions about needle color and bark color.

Species:	*ponderosa*	*jeffreyi*	*torreyana*
Bark color:	cinnamon-brown	grey	brown
Needle color:	yellow-green	dark green	dull green
Needle clusters:	three	three	five
Needle length:	8"	7"	10"

Collect Discriminating Data
Needles are dark green. (8)
Needles are 7 inches long. (9)
Needles are clustered in threes. (10)
Bark is grayish. (11)
Cones are medium-size. (12)

Consider Reliability of Data
There are other trees in the area of the same character. (13)
Mature height of others is more than 100 feet. (13)
Infer that the specimen is representative but not yet full grown. (13)

Determine Whether Unique Solution Is Found
Only a *Pinus jeffreyi* fits the data. (14)

Retrieve Common Name
Knowledge: A *Pinus jeffreyi* is commonly called a Jeffrey pine.

FIGURE I.3. Preliminary analysis of the protocol from the transcript in Figure I.2. The numbers in parentheses refer to the corresponding steps in Figure I.2.

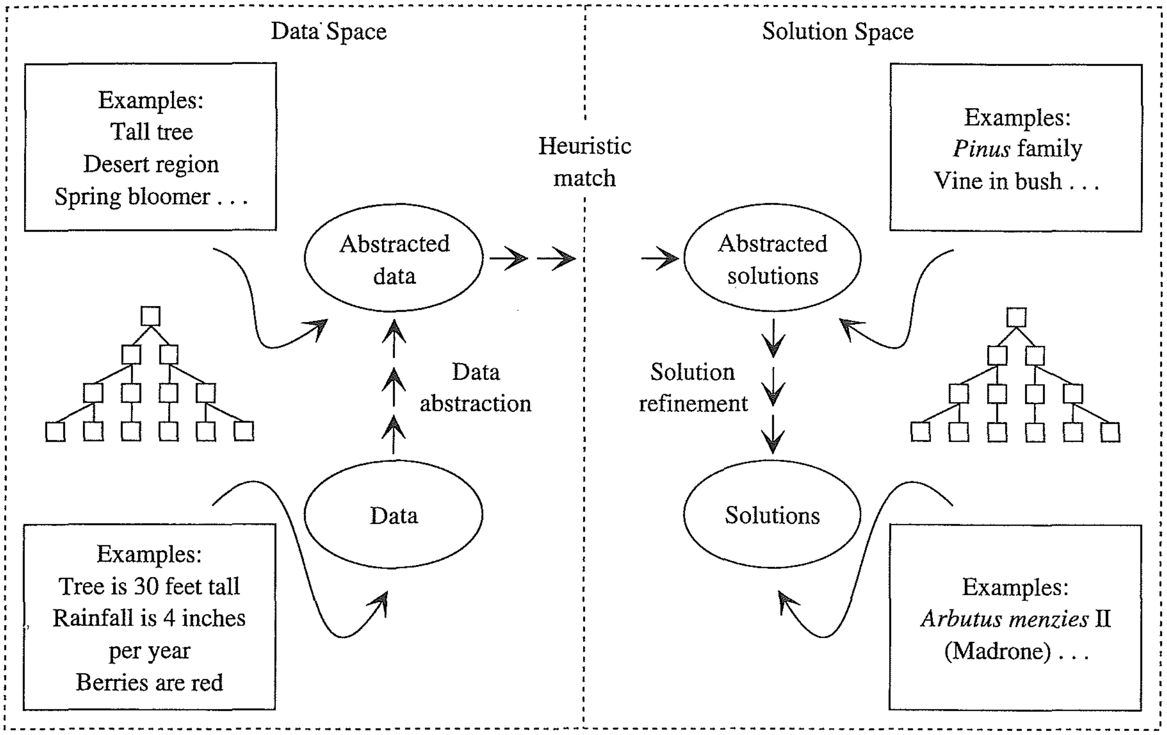


FIGURE I.4. The search spaces for classification. This method reasons about data, which may be abstracted into general features. The data are associated heuristically with abstracted solutions and ultimately specific solutions.

characterized operations such as "determining the general classification," "collecting data," "forming specific candidate hypotheses," and so on. These operations constitute a sketch of a computation model for the plant identification, which searches through a catalog of possible answers.

This tentative analysis of the protocol is consistent with a computational model that the knowledge engineer calls "classification." Someone in the group objects, arguing that the naturalist was not "classifying." Instead, he was merely "identifying" plants because the classes of possible plants were predetermined. The knowledge engineer agrees but explains that this is exactly what classification systems do. He draws Figure I.4 to illustrate the basic concepts used in this method.

To use this method, we needed to identify the kinds of data that could be collected in the field—the data space—as well as the kinds of solutions—the solution space. Data consist of such observations as the number of needles in a cluster. A final solution is a plant species. Classification uses abstractions of both data and solutions. A datum such as "3 inches of rain falls in the region annually" might be generalized to "this is a dry, inland region." A solution and species description such as *Pinus contorta murrayana* (lodgepole pine) might be generalized to pine tree. There are variations of classification, but they all proceed by ruling out candidate solutions that do not fit the data. Further analysis of protocols on multiple cases would be needed to determine what kinds of knowledge were being used and how they were used.

The knowledge engineer now has some questions for the naturalist. Suppose the solution space is given by a catalog of possibilities, such as the charts in the botany books we used on the project. The protocol analysis in Figure I.3 shows that the naturalist quickly ruled out the coastal varieties of the pine tree. But how about the many other species of pine that grow in the mountains? With book in hand, he asks why the naturalist had not considered a coulter pine (*Pinus coulteri*). The naturalist is taken aback. He answers that the coulter pine actually is a plausible candidate and asks to see the pictures of the specimen. After looking at it, he says the pine cones are too small and that the specimen does not have a characteristic open tree shape like an oak tree. Continuing, the knowledge engineer asks about the sugar pine. The naturalist answers that the cross-examination ferls like "lesson time," but that sugar pines are the tallest pine trees in the world, being more than 200 feet tall and that you would know immediately if you were in a sugar pine forest. However, the idea of systematically going through the catalog to analyze the protocols is appealing, so the two of them start working over them. The naturalist suggests that all of this post-protocol explanation and introspection might make him more systematic about his own methods.

As we continue to work on this, the significant size of the search space becomes clearer to everyone. One could be "systematic" by asking leading questions about each possible plant species. However, there are about 50 common species of just pine trees in California. Species of trees represent only a small fraction of the native plants. A quick check of some catalogs suggests that there are about 7,000 plant species of interest in California, not counting 300 or 400 species of wildflowers that are often discounted as weeds. It is clear that any identification process needs a means to focus its search, and that we need to be economical about asking questions. We begin to examine the protocols for clues about search strategy. We want to understand not only what he knows about particular plants, but also how he narrows the search, using knowledge about the families of plants and other things to quickly focus on a relatively small set of candidates.

Notes The group is developing a systematic approach for gathering and analyzing the domain knowledge. The protocol analysis has led to a framework based on heuristic classification. Usually protocol analysis and selection of a framework are done together. It is not unusual for the analysis to reveal aspects that were not articulated. Experts sometimes forget to say things out loud and sometimes make mistakes. For these reasons, it is good practice to compare many examples of protocols on related cases. Knowledge needed for a task is seldom revealed all at once.

Connections See Chapter 2 for characterizations of problem solving as search and for the terminology of data spaces, search spaces, and solution spaces. This chapter focuses on basic methods for search. To build a computational model of a task domain, we need to identify the search spaces and to determine what knowledge is needed and how it is used. See Chapter 3 for a discussion about approaches and psychological assumptions for the analysis of protocols. See Chapters 7, 8, and 9 for examples of the knowledge-level analysis and computational models for different tasks.

A "Naturalist in a Box"

As we build up a collection of cases and study the transcripts, we become aware of some difficulties with our approach. The first problem is that the naturalist is depending a great deal on properties of the plants that he can see and smell. Much of the knowledge he is using in doing this is not articulated in the transcripts.

Our hiking club representative kids the naturalist, saying he is "cheating" by just looking at the plants. We decide to take this objection seriously, and then notice three specific kinds of problems in the collected protocols. The first problem is that the naturalist makes his visual observations very quickly and often neglects to verbalize what he is doing at that point. Second, the naturalist does not articulate what guides his processes of perception. We need a systematic way of knowing where he is looking and then gathering the characterizations and inferences from what is seen. Third, we realize that the situation in which our system will be used introduces a new aspect of the task outside of the naturalist's field experience: communicating as though blind with an inexperienced observer. In short, the taking and analyzing of protocols seemed to be a good approach, but our approach was providing us with data for solving the wrong problem.

These problems mean that we have more work to do. For example, some additional "definitions" need to be captured, such as just what a "mottled pattern" is and what color is "cinnamon-brown." In using terms to refer to things in the world, we need to be sure that another observer can interpret the description and find the same thing. We call this the "reference" problem. We become nervous about the perceptual aspects of the naturalist's thinking because our "portable classifier" would not have capabilities for machine perception: In our projected system our users will need to observe the plants themselves. In addition, much technical vocabulary appeared in the protocols. We are becoming familiar with the naturalist's vocabulary as a result of working on the project. However, we recognize that our potential users and customers will not be comfortable with a question such as "Does it have radical leaves?" or "How many stamens and stigma are there?"

Again, we need a new approach to create a product that our potential customers will find usable. After a few hours of brainstorming, one of the group members proposes a knowledge acquisition set up that we later called "our naturalist in a box," as shown in Figure I.5.

In our setup the naturalist sits at a working table in a tent in the forest with whatever books and pictures he needs. The user walks off into the woods with a portable television. Each has a headset that allows them to speak by radio. In addition, the naturalist can show pictures from his books or drawings over the video link. This setup approximates the storage and display functions we would have with a portable hypermedia system, where the computer and stored images might perform the role of the "naturalist in the box." The voice communication would substitute for a pointing device and keyboard. All communications on the audio and video links are recorded for our later analysis. In addition, pictures of the plants and the user are taken, but not shown to the naturalist until later when we analyze the sessions.

At first the naturalist feels quite confined and hampered by the setup. A crucial question was whether the naturalist could properly identify the plants without seeing them. Soon, however, it becomes clear that the naturalist is able to function in this mode and that the setup is suitable for obtaining the information we need. We begin to discover specific requirements about the interactions with the user.

Along the way there are some interesting surprises. One fall day, a user wants to identify a brilliant red shrub at the side of the trail. As he describes the bush, the naturalist starts to ask whether it has shiny leaves organized in groups of three originating from the same point on the

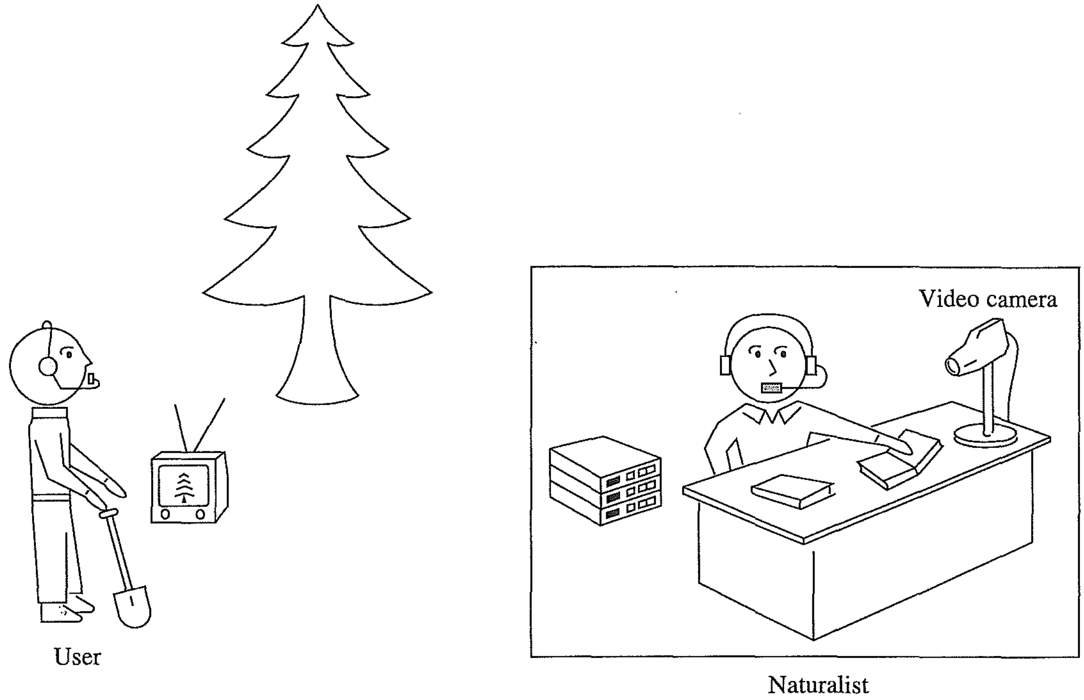


FIGURE I.5. Our "naturalist in a box" setup for gathering realistic problem-solving protocols.

stem. Suddenly the naturalist becomes alarmed and says, "No, don't touch that! It's probably *Rhus diversiloba*, I mean, poison oak!" This leads us to recognize the need for additional functionality in the performance program, beyond the simple drive to classify.

We begin to record various images of the plants from our cases and organize them for computer-controlled retrieval in a catalog of digitized images. The naturalist begins to use these images in his conversations with the "users." With a few keystrokes, he can retrieve an image and display it on the user's television. We make a simple interface to switch from general views of a plant to close-up views of its foliage, bark, or seeds. We also make it easy to arrange image arrays of similar plants next to each other for visual comparison. Together with the naturalist we experiment with different ways of retrieving and displaying images.

From the many video sessions it becomes apparent that our users are not willing to answer "20 questions" in order to identify plants. For one thing, they often do not understand words such as *pollen cones* or *deciduous*. Over a few weeks of experimenting with the setup, we begin to understand more about the flexibility inherent in the information exchanged in a session between the naturalist and a user. We notice that a user might volunteer some basic constraints at the beginning. For example, once it is determined that the region was "high desert" there is no point in considering plants that could grow only in less arid regions. Once it is determined that it is winter, there is no point asking questions about deciduous leaves that would no longer be on the plant. If the user is interested in shrubs, it is not necessary to enter that constraint for every specimen.

Our programmers create a template of constraints that can be carried over from problem to problem, without the need to reenter them or infer them from new data. To avoid tailoring our interactive approach to the requirements and idiosyncrasies of a single user, we rotate through a collection of users and test whether the vocabularies and kinds of interactions can be easily understood.

As we broaden our activities to different parts of the wilderness, our naturalist suggests that we enlist the help of other naturalists who specialized in those regions. The naturalists work alternatively as a team and as individuals checking each other's work. We find that they differ in their approaches. This raises general issues about combining expertise. We collect the interesting cases for staff discussion.

As we begin to use the system, we need a name for it. Someone proposes calling it SEQUOYAH, after the Cherokee for whom the redwoods are thought to be named. Born in 1760, Sequoyah was known for developing a written alphabet for Native American languages. His goal was to enable the tribes to communicate and to preserve their Indian dialects. We thought that the name was appropriate for honoring the Native Americans who lived in and knew so much about the wilderness. We also saw the knowledge system as a new kind of writing medium or active documentation, that would be an appropriate tool for a modern-day Sequoyah.

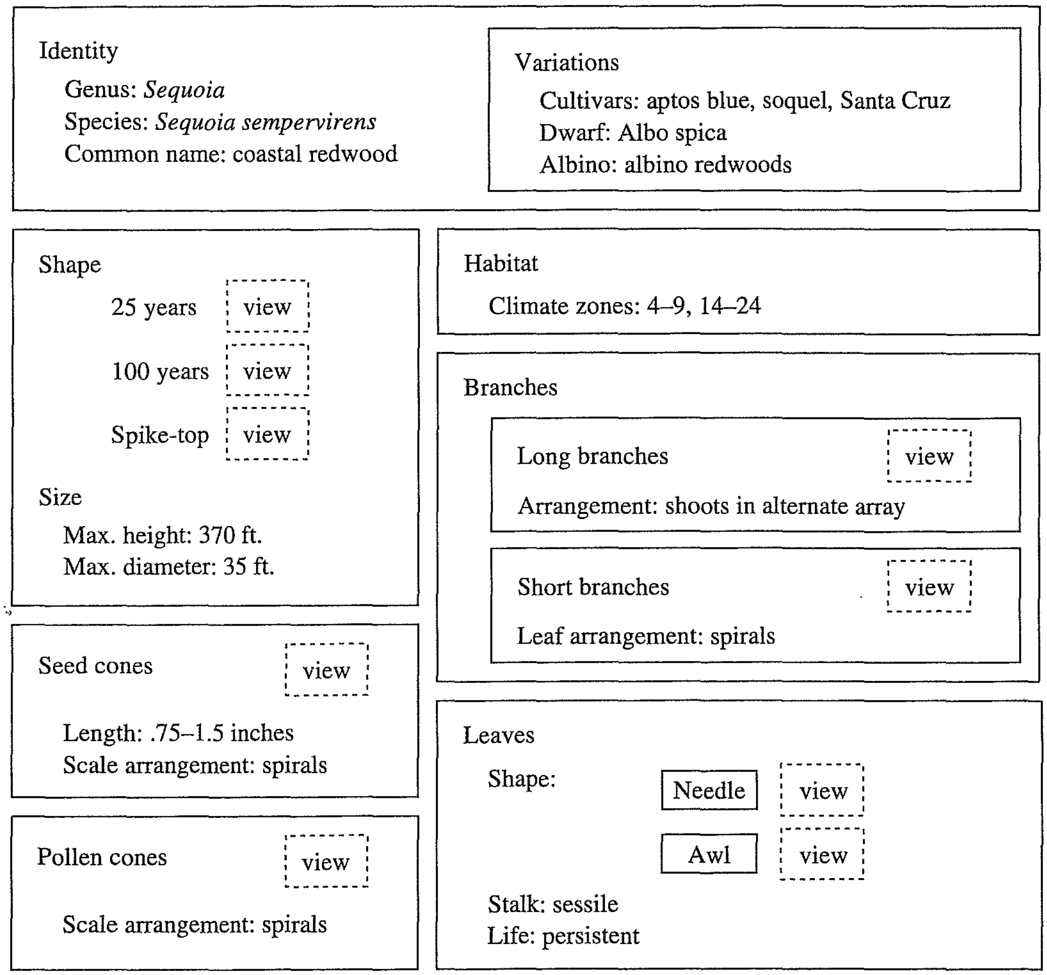
Notes The group has had some false starts. They needed to define the task better to focus their process of finding and formulating the relevant knowledge. The term *process* keeps coming up in the context of building and using knowledge systems. Knowledge systems for specific tasks must be designed to fit the processes in those tasks. In this example, the group discovered that it had made a crucial error in its assumptions about the way that the system would be used. In particular, it had implicitly assumed that the user would have the same observational skills as the naturalist. Addressing this exposed the need for different protocols.

Connections Chapter 3 considers general points from software engineering about defining tasks and about involving potential users and multiple experts in early stages of system design. When multiple experts are available, there is an opportunity for them to check each other's work, to find idiosyncrasies and gaps, and to develop alternative reasoning models. Chapter 1 discusses the use of multiple models in reasoning.

Developing Formal Representations

Working from the protocols and other background material, we begin to develop formal relations for modeling the identification of plants. These relations need to make distinctions adequate for representing the reasoning in our sample cases. For example, we need to define a "color" relation that would map consistently to a range of colors and color mixtures. Furthermore, we know that the bark, leaves, and cones could all be different colors. This leads us to formalize a vocabulary of standard "parts" for plants. Following the biologist's lead, we select standard terms for parts and use the same names for particular relations throughout our database.

The knowledge engineer develops an example of a frame for our consideration, as shown in Figure I.6. As he explains it, the frame is a representation that would be used for the reasoning done internally by SEQUOYAH and externally for our discussions of what SEQUOYAH could and should do. Internally, SEQUOYAH might employ multiple and different representations for some parts of its task, but we would define our understanding and view of them largely through the **vocabulary** made visible in frames. Externally, user interfaces would specify how we viewed



Identity
Genus: *Sequoia*
Species: *Sequoia sempervirens*
Common name: coastal redwood

Variations
Cultivars: aptos blue, soquel, Santa Cruz
Dwarf: Albo spica
Albino: albino redwoods

Shape
25 years view
100 years view
Spike-top view

Size
Max. height: 370 ft.
Max. diameter: 35 ft.

Habitat
Climate zones: 4–9, 14–24

Branches

Long branches view
Arrangement: shoots in alternate array

Short branches view
Leaf arrangement: spirals

Seed cones view
Length: .75–1.5 inches
Scale arrangement: spirals

Leaves
Shape: Needle view
Awl view
Stalk: sessile
Life: persistent

Pollen cones view
Scale arrangement: spirals

FIGURE I.6. Example of a composite frame tried for data entry. In designing these we seek ways that would ensure the use of a uniform vocabulary in the database.

and interacted with the frames. His purpose at this setting is to consider some of the meaning and vocabulary that we would use in creating SEQUOYAH. The general idea is that we will choose a uniform vocabulary to the extent that it is possible.

The first thing he explains about the frame in Figure I.6 is that it contains many different kinds of information. For example, the boxes marked "view" are intended to link to digitized photographs that can be shown to the user. They are not intended for interpretation by SEQUOYAH itself. The second point is that many of the entries have further elaborations elsewhere in the data base. For example, there is a frame for pollen cones, a frame for the *Sequoia* genus, a frame for climate zone 14, a general frame for leaves, and so on. Using the frames, we define vocabulary and relations for plants, their parts, their botanical classifications, their habitats, as well as observations and tests that could be carried out by our users. In addition we develop criteria for choosing what to observe, choosing what candidates to consider, combining constraints, and developing warnings about hazards.

We then discuss the different data fields, with an eye toward specifying more precisely what they should mean. The naturalist says that separating the common name from the botanical name is a good idea. In the case of the redwood, several other trees are commonly called redwoods that are not related. In Brazil, the Amazonia is often called a redwood. It has light red and orange wood. In Burma, there is a tree called the Andaman redwood. It has red or crimson wood streaked with red. In Europe, the wood from *Pinus sylvestris* or Scots pine is often called redwood. The naturalist suggests that we provide means for retrieving descriptions and comparing trees with similar names.

We then talk about descriptions of climate zones. We decide to use the zones established in the *Sunset New Western Garden Book*, which is widely used in the western states. However, there is an important issue about the meaning of *habitat.* Do we mean the places where the tree could be grown or the places where it grew naturally? Do we mean the places where it survives or the places where it thrives? Naturalists sometimes define habitat as the regions where the species will propagate naturally. It becomes clear that we needed to be able to distinguish several different meanings of habitat, both for consistent reasoning by the system and for consistent encoding of the database. The naturalist offers to provide a candidate set of standard terms and relations for the knowledge base.

Climate zones delineate places on the map. We notice in the protocols that the naturalist makes substantial use of information about location. If SEQUOYAH knew where the user was located, it could automatically take that information into account in its analysis, ruling out or deemphasizing plants that would not be expected to grow there naturally. This leads us to consider representations of map data, relating information about climate, latitude, longitude, and elevation. We also decide to include information about parks, towns, lakes, and so on. We recruit a specialist to the team who can investigate the availability and possible formats of map information.

A question comes up about the choice of units of measurement. In talking about length or width, should we use inches and miles or metric units? That issue turns out not to matter much because the computer can convert the information between different units of measurement.

One of the important uses of the frames is to represent the possible solutions to the classification task. Ideally, a solution corresponds to a single species. In going over the cases we had obtained from the "naturalist in a box" protocols, however, we find a few cases that challenge our representational capabilities. Some of the most interesting (and at first perplexing) cases are ones where several plants were growing so closely together that the observer did not realize that the "specimen" was actually two or more different plants. For example, vines can become so intertwined with their host plants that their leaves appear to belong to the host. To enable SEQUOYAH to reason about such cases, we decide to admit "composite solutions," generated by appropriately mixing together the attributes of plants that grow in this way. In cases such as this, decisions about what the system can represent determine the coverage of its problem-solving abilities.

Finally, we notice that we need to model some data about the seasons. Plants appear differently in different seasons. This is especially true for plants that show bursts of spring growth, plants that lose their leaves, and flowers with a limited blooming period.

Over the next few weeks, we build up a knowledge base about the domain. As the knowledge base becomes more complete, we compare its performance on test cases with that of our naturalist. Our goal is not so much to replicate the precise sequence of his steps as much as to approximate his skill and expertise over a range of cases.

Notes At this point the group is beginning to develop formal symbolic representations to model the domain, as in the example of the redwood tree frame. To judge adequacy of representations, the group needs to keep the task in mind. The world is quite open-ended, and every project needs to determine the bounds on the knowledge that it will formalize. Different designs for representation have different capabilities for making distinctions. Different choices for symbol structures can also have a large impact on the practicality of making different kinds of inferences. For example, before picking representations for space (map data) and time (seasonal data), we need to know what kinds of inferences we expect the system to make and which ones must be made quickly.

One powerful advantage of computer systems over books as knowledge media is that they can be executed. That is, the models can be run and debugged on cases. This encourages the systematic development and testing of models in an experimental mode.

Connections Chapter 1 introduces concepts about symbols and meaning with examples of objects and relations used in building models. Issues about representation and reasoning about space and time are considered in Chapters 4 and 5.

Testing the System with Users

Over the next few weeks we get ready to work with what we called our "beta-test" users. By "beta-test," we mean the first people outside of our working group to try out our system. Given our operational version of SEQUOYAH, we knew that people could give us quite specific feedback about what they thought about it. Our goal is to collect information that will enable us to improve our product, bringing it closer to our customers' needs.

We use a variety of approaches to gather information. One is to work with hiking clubs to observe their use of the system. We collect detailed records of their interactions with the system, including video records, trying to identify those places where SEQUOYAH confused them. We also interview them about the system and collect questionaires and suggestions. This process reveals bugs, proposed changes, and proposed new features, which we then consider in the meetings of our development staff.

One thing we discover is that our customers often have very good ideas. We decide that we need a way to encourage customers to send us new ideas and bug reports on a regular basis. This leads to a "message" feature that we add to our system. At any time during a session, a user can push a message key and type in a short message with a suggestion or bug report. This creates a file on the computer with the user's name, the version of the system, the context in the session, and so on. At some later time, the user can plug his or her computer into a modular telephone and it would dial an 800-number that passed the information to us. The information gathered automatically in this way enables us to reproduce the situation where the problem arose. To encourage contributions, we offer rewards for suggestions that we later incorporate into our products.

Notes Testing the early version of a system in realistic settings is a crucial part of every knowledge-system project. Custom systems that are tailored for a single group of users can be developed with those users from the beginning. Companies that try to market to a large set of users need to have effective ways to probe those markets and to assess the products. These studies can involve a joint effort between marketing and development staff.

Connections Marketing and testing are not major themes in this text. We mention them here to fill in some of the larger commercial context of knowledge systems. However, Chapter 3 contains some suggestions for approaches for developing systems jointly with user groups.

Add-on Systems

We now skip forward in the story. SEQUOYAH is in its second release in the market. Several organizations have approached us about "opening" our knowledge base so it can be extended.

One idea that provokes a lot of interest is extending SEQUOYAH to act as a "digital trail guide" to the hiking trails in the state and national parks. Several well-known hikers express an interest in collaborating on add-on systems, which would provide an interactive interface for hikers in choosing and selecting hiking trails. Different tour promoters specialize in different kinds of hikes. Some focus on challenges. Some focus on nature walks. Some would like to exchange "hiking instructions" that connect with each other and into the extensive map and plant database that we have already built. For example, software for guiding a nature walk could draw the hiker's attention to a particularly interesting redwood on a trail and could provide backup for questions drawing on our general database about wildlife. We see that by providing facilities for extending and exchanging databases, we can extend our market. One of the hiking magazines proposes to work with us to develop digital trail guides that they would distribute with their quarterly magazine. They would like to have a partnership arrangement with us for developing the software and also to give special agreements to their members for purchasing our systems. The simpler proposals for a digital trail guide would build on our map database by overlaying trail information, showing the distance between points, the altitudes, the slope of trails. Other more elaborate proposals include systems that would design day hikes for campers on demand, given constraints about distance, what they want to see, and so on.

In addition, a group of California nurseries approaches us about extending SEQUOYAH to advise on landscaping with native plants. With the difficulties created by drought in California, they have seen much greater interest in the horticultural use of water-efficient native plants. One nursery wants to extend our database of native plants and to develop a design assistant for landscapes. The database would be extended to include information about layout, planting, fertilizing, pruning, ecology, and forest management. They want to extend the plant selections to include plants overlooked by hikers, such as native grasses used for controlling erosion. They observe that many horticulturists are trying to catch up in their understanding about native plants. In an arrangement similar to one of the hiking clubs, one professional nursery proposes to create quarterly "digital feature reports" that recommend newly available plants and to extend the database and ideas into new areas.

We convene some company meetings to assess the commercial prospects and managment and technical issues for these proposals. One of the issues that we see is that there will suddenly be many different versions of our system. What will the maintenance issues be when someone is using last year's version of our plant database with a new digital trail guide from the hiker's club? We want the software to work together as well as possible, accessing the latest versions.

We also discover that the landscaping advisor system would need representation and reasoning capabilities beyond those that we had used in SEQUOYAH. One member of the development sketches out a mockup of the user interface in conjunction with one of the landscape specialists, a horticulturalist. The mockup, which is shown in Figure I.7, shows the system reasoning about plants and maps. The map representations are more detailed than the ones that we used for SEQUOYAH, but about the same as would be needed for the trail guide applications. They would, however, need to represent additional elements, such as water systems and lighting. Furthermore, the preliminary examination of the search processes for landscape design suggested that the methods used by SEQUOYAH, heuristic classification, would not be adequate. The landscaping system has a much more complex space of possible solutions. For example, a solution might include compatible plant selections; layout; land contouring; and the construction of sheds, patios, water sytems, and so on. There are quite a range of functions that the system could perform, including creating perspective drawings of the landscape, projecting plant growth over time using parameterized fractal drawings of plants, and comparsion views that show the effects of different choices. In general, it appears that the landscape advisor would be much more complex than SEQUOYAH to build. We need to identify software from other vendors that we could incorporate in the product. We recognize that a preliminary study is needed to determine what features would be desirable and what is affordable.

As we explore the technical and policy issues for our product, we recognize that there is not just the one business of selling "boxes" to hikers, but rather that there are many different businesses for providing data and services in our "knowledge medium." Some of the businesses could build on each other, and some of them require expertise beyond the means of our company. We need to understand which businesses make sense strategically and what the conditions are for making them commercially viable. If we turn our knowledge system into an "open system," we need to consider what protections would be required for our software.

As we consider the changes we are seeing in technologies and concepts for publication and communication, we realize that our products are examples of a new kind of writing for people who enjoy the wilderness. The products are enabling such people to make their ideas and expertise active in a way that is more adaptable than books. Reflecting back on the Cherokee Sequoyah's work to introduce an alphabet as a symbol system for American Indians, we feel that the selection of a name for our system is even more appropriate.

Notes As knowledge systems become larger, the issues for managing, updating, and sharing information become more crucial. In the current state of the art, people have not yet confronted the issues of making knowledge bases that can be used for multiple tasks. Although this point is not emphasized in the story, there are many open issues about copyright and distribution of electronic publications and electronic services. The first generation of electronic networks discouraged commercial participation and sidestepped the opportunity (and effort) to shape or inform enduring public policies. Many policymakers now believe that even the research networks seem likely to benefit from the use of market mechanisms to foster the creation and administration of new services.

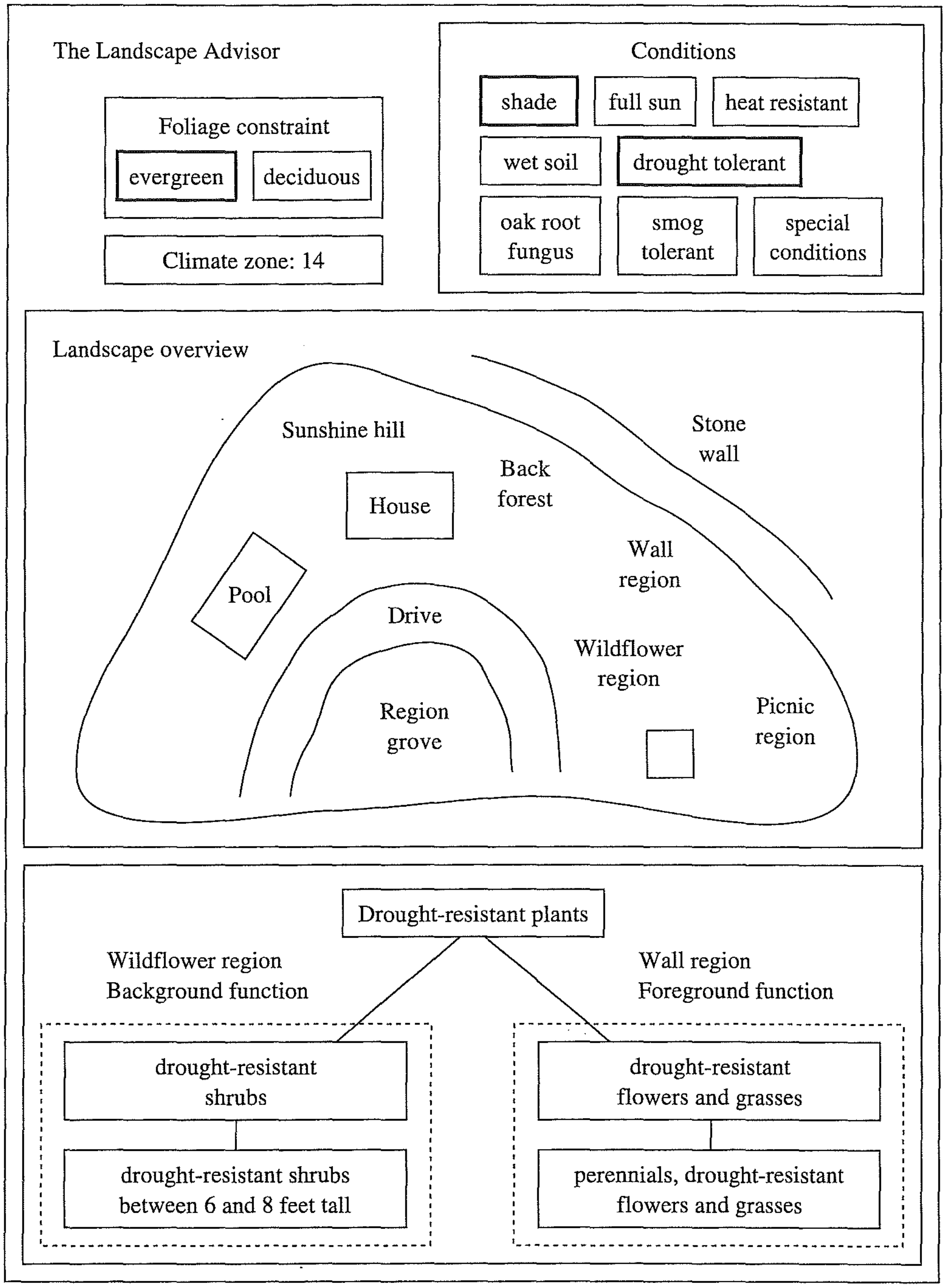


FIGURE I.7. Mockup of a user interface for a landscape advisor. This system needs to accept as input a description of a landscape, including details about land contours, existing vegetation and trees, and uses for different areas. It needs a vocabulary to identify different uses of plants in yards, such as for erosion control, screening, shade, and so on. It needs to represent a variety of constraints about kinds of foliage desired, soil condition, and user needs. A "solution" is an expression of any of the plantings, changes to land contours, or physical structures that make up the elements of a landscape.

The Story and This Book

The preceding story is a tour through topics relevant to building knowledge systems. It emphasizes the social and product issues over the technical issues of building them, which are the primary topics of this book.

The SEQUOYAH story is intended to "make the strange more familiar." We seek an integrated perspective on diverse topics: protocol analysis from information-processing psychology, project organization concepts from software engineering, search methods from artificial intelligence and operations research, and representation and reasoning concepts from computer science and graph theory. To understand about knowledge systems — from what knowledge is to how it is represented, or from how systems are built to how they are used — requires that we wear several different hats. For everyone, some of these hats fit better than others. This introductory story provides an orientation for why there are so many hats. We refer back to it occasionally in the text.

As you read this book, these increasingly familiar topics will reinforce each other. Protocol analysis may seem familiar as the art of conversation or the art of interview, but when applied in knowledge engineering it rests on models of problem solving as search and psychological assumptions. Choosing a data structure may seem like the art of programming, but in the context of knowledge engineering, the engineering choices depend on requirements for the reasoning and search. Building a model of a classification task or a configuration task may seem like formalizing common sense, but in the context of knowledge engineering it rests on foundations about search methods and solution spaces. In this book, I hope to sharpen our perceptions of the "unexamined familiar." People who participate in knowledge engineering in their own areas of expertise report a deeper sense about what they know and how they use it. They also gain new perspectives on the nature of knowledge, and the processes for creating it, using it, debugging it, and communicating it.

This chapter introduces symbol structures, representation, meaning, modeling, inference, and computation. It presumesonly a modest familiarity with programming and logic.

1

Symbol Systems

In their 1975 Turing Award paper, Allen Newell and Herbert Simon presented what they called the physical symbol system hypothesis:

> **The Physical Symbol System Hypothesis.** A physical symbol system has the necessary and sufficient means for general intelligent action.

By *necessary*, Newell and Simon meant that an analysis of any system exhibiting general intelligence would show that the system is a physical symbol system. By *sufficient*, they meant that any physical symbol system of sufficient complexity could be organized to exhibit general intelligence. By *general intelligent action*, they meant the same order of intelligent and purposeful activity that we see in people, including activities such as planning, speaking, reading books, or composing music. This hypothesis puts symbols at the core of intelligent action. Roughly, it says that intelligent systems are subject to natural laws and that the natural laws of intelligence are about symbol processing.

Symbols are central and familiar elements of natural language, mathematics, logical formalisms, and programming languages. As suggested by the physical symbol system hypothesis, they have also been traditionally associated with dominant theories of mind and intelligence. In recent years, symbolic and representational theories of mind have been challenged by other accounts based on nonsymbolic and subsymbolic structures. To pursue these topics here would defer us from our primary interest in knowledge systems and is deferred to the quandaries section of this chapter.

Whatever the case for natural systems and general intelligence, symbols are central to knowledge systems. We can build and study knowledge systems without resolving the issue of whether they are intelligent. However, we cannot build them or understand them without using symbols or without understanding the nature of symbols. This chapter is about symbols, what they are, how they acquire meaning, and how they are used in creating computational models.

1.1 *Symbols and Symbol Structures*

Symbols are the elements of our spoken and written languages. Here are some examples of symbols:

Paige
infers
3.141592654
computer
a-very-long-hyphenated-word

Symbols can be arranged into larger structures that we call **symbol structures** or simply **expressions**. Figure 1.1 shows several examples of symbol structures.

When we refer to symbols in the context of knowledge systems, we usually mean words, numbers, and graphics. These symbols appear on computer displays, are represented in computer memories, and are manipulated by our programs. Symbols and symbol structures are so familiar that we seldom pause to examine their nature. In the following, however, we define symbols and introduce terminology that will enable us to be precise about the properties that we attribute to them.

1.1.1 *What Is a Symbol?*

The dictionary defines a **symbol** as a written or printed mark that stands for or represents something. A symbol may represent an object, a quality, a process, or a quantity as in music, mathematics, or chemistry.

For us the dictionary definition of symbol is preliminary. It conveys basic intuitions about the term. Symbols are marks in a medium. They are used to represent things. Although this definition seems simple enough, it raises some questions: Is any marking a symbol? Does the notion of "write" include electronic or biological encodings? Can we determine from a symbol itself what it represents? Can two people disagree about what a symbol represents? Can a symbol represent itself?

Although these questions may seem obscure, simple confusions about symbols and representation repeatedly lead to difficulties in creating and using knowledge systems. Symbols are fundamental to creating and using computational models. Through a series of examples, this section develops terminology and a framework for understanding and answering these questions.

The dictionary definition suggests that two basic themes concern the nature of symbols: (1) What is a marking, and (2) What is representation or reference? Markings and reference are the topics of this section.

We begin with **registration**, which is about recognizing and identifying markings. Consider Figure 1.2. What symbols are in the figure? One person might say that there are two overlapping squares arranged so that part of one square is occluded by the other. Another person might say simply that there is a set of eight horizontal and vertical lines. There is nothing about

Paige

a+b

(SQRT (- (* b b) (* 4 a c)))

```
(forall (x) (if (and (instance-of x oak-leaf)
                     (equals season spring))
                (color x green)))
```

$(\forall x)\ f(x) \rightarrow g(x)$

Willy threw the ball to Morgan.

Subject:	Willy
Verb:	threw
Object:	ball
Receiver:	Morgan

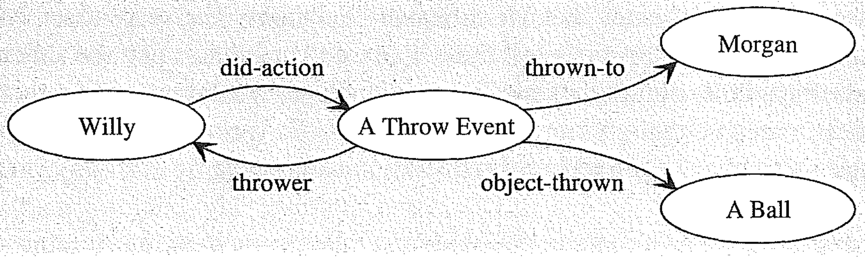


FIGURE 1.1. Examples of symbol structures. A symbol is a pattern in a physical medium recognizable by some interpreter. A symbol structure is a physical arrangement of symbols.

the figure itself that makes one person right and the other wrong. These different accounts of what symbols are in the figure are called different **registrations** of the figure. When different people look at markings they may disagree about what symbols are present. They may use different conventions about notation to identify symbols in particular kinds of documents, such as musical scores, books of poetry, or architectural drawings.

The issue of what markings constitute a symbol is intimately bound up in however a person or machine *recognizes* it. When we say that something is a symbol, we imply the choice of a recognizer. Typically, a recognizer has an associated alphabet or set of symbols that it can recog-

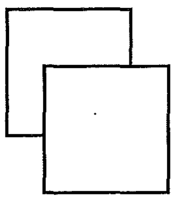


FIGURE 1.2. The registration of symbols: This example shows marks on a page. There they might be recognized as a single symbol, or as two overlapping squares, or as a set of horizontal and vertical lines. Each different registration corresponds to a recognition process for a different interpreter.

nize. It can tell whether some set of markings constitutes a symbol in the set, and also, identify the symbol. We use the terms **token** and **type** to distinguish individual physical symbols (tokens) from classes of symbols that are recognized by the recognizer as equivalent (types). For example, if a recognizer was designed to recognize the alphabet letters used in English as printed in a particular font, then not counting numerals and punctuation there would be 52 types — one for each upper and lowercase character. On a printed page in this font, each occurrence of the letter "a" corresponds to a token. Two tokens are equivalent for the purposes of recognition if they are of the same type.

We now give a definition of symbol that takes recognizers into account. A **symbol** is a physical marking or pattern, which can be read, recognized, and written by a recognizer. A **symbol structure** is an arrangement of one or more symbols in a physical medium. We also use the simpler term expression to refer to a symbol structure. So far, our definition of symbol mentions markings but not representation. We postpone discussing representation, that is, the assignment of meaning to symbols.

Not every piece of matter or energy is a symbol. The wind blows leaves across the ground and leaves markings, but these are not symbols. When we identify certain patterns as symbols, there must always be a recognizer that identifies the symbols. The recognizer can determine where each symbol starts and stops, can tell them apart, and can determine the salient features of the arrangement of symbols. Sometimes the recognizer is part of a larger system that can retrieve and sense symbols, compare them, write new symbols, and do various kinds of reasoning. Sometimes we use the somewhat vague term **interpreter** to refer to various kinds of larger systems that include a recognizer.

The registration issue even arises in mundane contexts such as interpreting text symbols written on a page. Consider the symbols in the following sentences on this page.

Willy lives across the street. He threw a ball to Morgan. (1)

In the typical account, the symbols in (1) are the images of the eleven printed words, and the symbol structures are the sentences, which are composed of adjacent words. Other registrations are possible. For a theory of English spelling and grammar, the symbols are the printed characters, that is, images of letters of the alphabet and the symbol structures are words composed of letters; the sentences in turn are composed of words. For a theory of typography and font design, the symbols could be printed strokes or dots and the symbol structures the letters that they form. Different registrations correspond to different recognizers.

The modifier *physical* is intended to emphasize the concrete physical realizability of symbols, and to preclude any confusion with "ideal" symbols that have no tangible existence, such as perfect letters of an alphabet. The patterns can be almost anything. They can be arrangements of electrical charge in a computer memory, arrangements of organic compounds in brain cells connected by nerves, patterns of electromagnetic waves with intensities arranged in time, patterns of brightness on a display screen, patterns of nucleotides in a gene, dark marks on white paper, or scratches and paintings on ancient Indian pottery. When we say that genes are symbols, we could have in mind recognizers that are either automatic machines that read genes from DNA, or naturally occurring biological mechanisms that read genes to build proteins. The symbols in a symbol structure can be adjacent in a medium or linked together in some other way.

A **physical symbol system** is a machine such as a computer that operates on symbol structures. It can read, recognize, and write symbols. A symbol system creates new expressions, modifies old expressions, makes new copies of expressions, and destroys expressions. Symbols may be communicated from one part of a symbol system to another in order to specify and control activity. Over time, a physical symbol system produces a changing collection of symbol structures.

Our definition of symbol admits a very wide variety of techniques for realizing symbols. They need to persist long enough to be useful for a symbol system.

1.1.2 *Designation*

In some branches of theoretical computer science and mathematics, symbols are taken as primitive and undefined terms. In these theoretical studies of computation, ideal machines with various kinds of finite and infinite memories read and write symbols. Turing machines with finite control stores and infinite tapes are examples of ideal machines of this sort. Like other such abstract machines, they are used for discussing formal theories about computability.

These theories are concerned with fundamentals of formal languages and mathematical objects. Not only are they removed from finite and physical implementations, but they are also removed from practical uses and familiar settings.For example, these theories are not concerned with issues involved in creating physical patterns and arranging for their flexible and reliable manipulation. They treat computers as isolated systems. Symbols go in. An idealized computer munches on them and then other symbols come out.

However, knowledge systems are built from real computers and compute things relevant to a real, physical world. They are part of larger systems and we need to be able to discuss how they are connected and embedded in a physical setting. To interact appropriately with an external world, computers must be able to reason about it and also be able to represent aspects of it with symbols. It is for worldly settings that Newell and Simon's physical symbol system hypothesis was developed and in such settings the concepts of knowledge, reference and representation are of interest.

An **embedded computer system** is one that is integrally part of a larger system. Examples of such larger systems include electronic ignition systems devices, electronic banking systems, inventory and manufacturing systems, aircraft carrier information management systems, electromechanical systems in a troubleshooting context, medical systems for patient monitoring, sales and inventory systems at point of sale terminals, and communications networks. The boundary of what constitutes the larger system depends on the purposes of our analysis. It could include machines or even people in a social organization. All computer systems can be seen as embedded systems.

Our dictionary definition of symbol says that symbols represent things. We now turn our attention to what it means to represent something.

When we talk about representation, we need to identify three things: the symbols, the situation, and an observer. As suggested in Figure 1.3, the observer is an agent, that is, someone or something that perceives both the symbols and the situation. The observer uses the symbols as a model of the situation. When the observer perceives the situation, he organizes his understanding of it in his mind in terms of different elements, often referred to as objects. These objects have

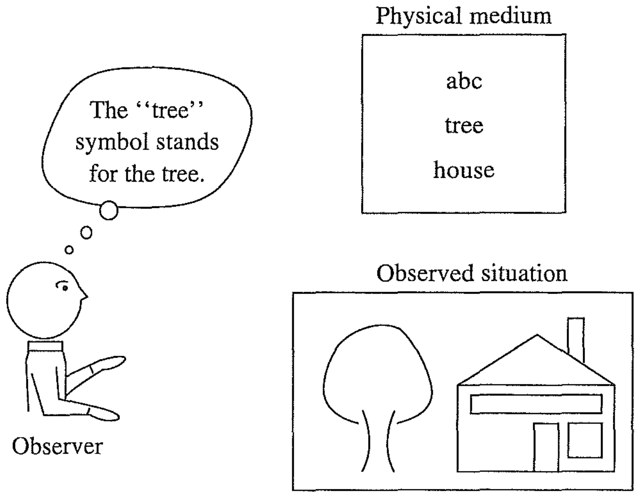


FIGURE 1.3. Designation is in the mind of an observer. It is a mapping between elements he perceives in the situation and symbols in the physical medium.

different relations to one another. For example, the observer might view a *car* in a situation as being *on the driveway*, or a *car door* as being *part of* a *car*. The observer uses symbols to stand for objects in the situation. When an observer picks some set of perceived objects in the situation and makes symbols for them, we say that he **reifies** them. When he associates symbols with objects in the situations, we say that the symbols **designate** the objects. The relation indicating what symbols stand for is called **designation**, or equivalently, **denotation**.

This brings us to an important point about symbolhood and defining designation: the need for an **observer**. The problem is that giving an account of what symbols represent requires a vantage point, usually outside of the symbol system. In most cases, the things that symbols stand for have been perceived by somebody. What a symbol stands for is not determined by a symbol's encoding or by its location. Restated, designation is not a physical property of the markings. Designation is assigned by and relative to an observer.

We introduce a distinction between a symbol system and its **environment** as shown in Figure 1.4. The boundaries that define the symbol system and environment depend on a point of view. The boundary around a symbol system delimits where the symbols are for particular purposes of analysis. A symbol system is contained in its environment and often its symbols refer to parts of that environment. We use the term *domain* or **symbol domain** to refer to a elements of interest in the environment.

In Figure 1.4, the observer of the computerized heating and cooling system says that "the 'reading' symbol stands for the room's temperature." From a cognitive science perspective, we may imagine that the observer's mind may itself have separate symbol structures: one standing for the reading symbol in the computer, one standing for the room's temperature, and another standing for the designation relation between the first two.

Exactly what do we mean when we say that the "temperature reading of 20" designates the room's temperature? There may be many symbols equal to 20 at various places in the computer's memory. For example, another 20 may be used by the computer to keep track of the number of heat vents in the building. We do not mean that every 20 pattern in the computer's

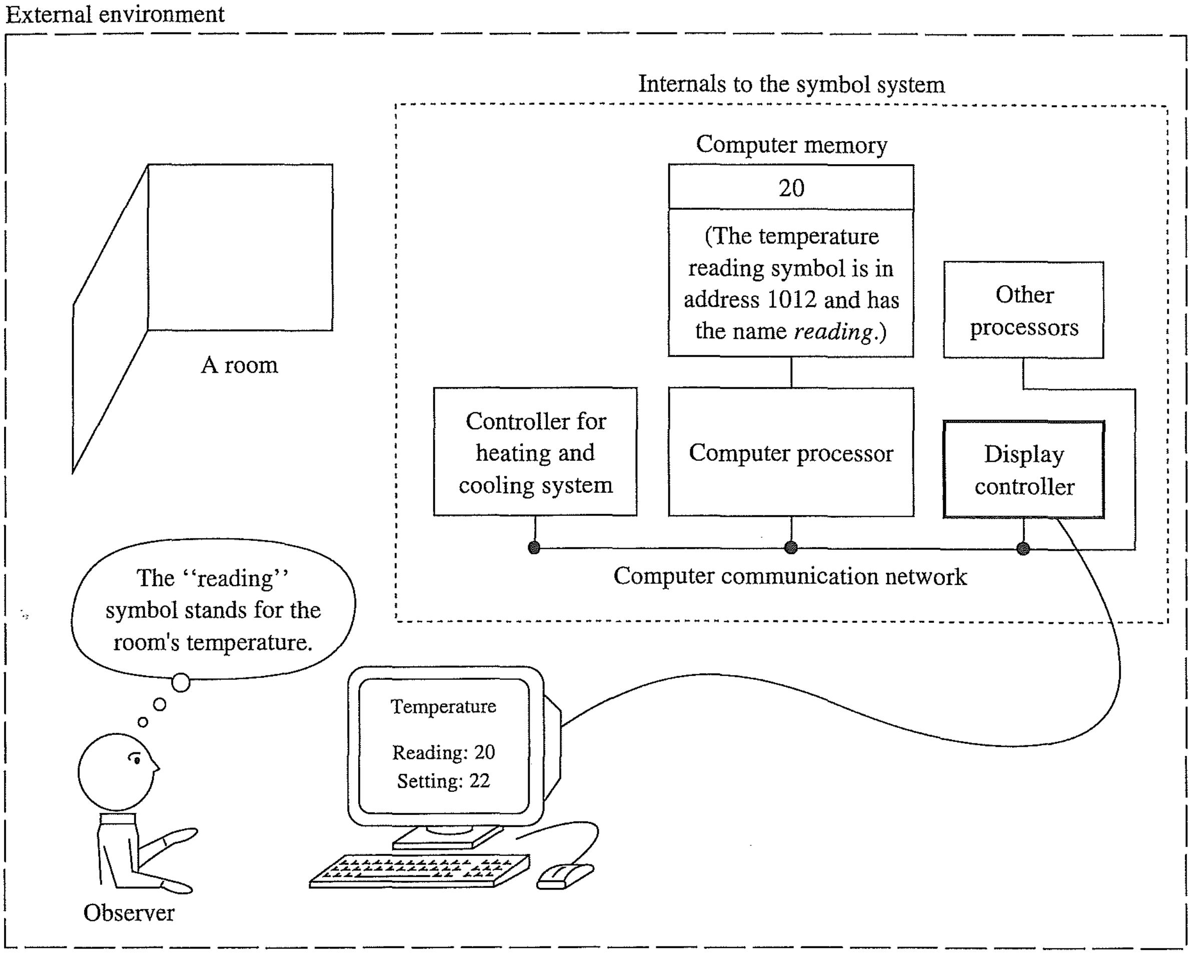

External environment
Internals to the symbol system
Computer memory
20
(The temperature reading symbol is in address 1012 and has the name *reading*.)
Other processors
A room
Controller for heating and cooling system
Computer processor
Display controller
Computer communication network
The "reading" symbol stands for the room's temperature.
Temperature
Reading: 20
Setting: 22
Observer

FIGURE 1.4. Designation is in the mind of an observer, who can give an account of both the symbol system and the manner in which its symbols designate aspects of an environment.

memory "stands for" the room's temperature. We probably associate a particular register in the machine as the "temperature variable." There must be some way for an observer to access a symbol. Two observers discussing the meanings of symbols must assure each other that they are referring to the same elements of the physical symbol system, such as by agreeing on the use of a particular addressing scheme. They might say that the symbol that is stored at memory location 1012 designates the temperature of the room.

Names are not the only means for identifying particular symbols. When a computer user looks at a display as shown in Figure 1.5, the display renders symbols and presents them for perception, identification, pointing and manipulation. A user interacts with a computer using the display and various keyboards or pointing devices. The picture elements or pixels in the display surface provide a view of information stored elsewhere in the computer, updated and maintained by a display controller. Thus, a user can identify a symbol by pointing at it, not just by naming it.

Figure 1.5 decomposes the process of designation for a variable into three steps. First an addressing scheme (memory address 1012 or "reading") is used to locate the variable or storage element. Then its value is determined (20). Then that value is assigned a referent in the environ-

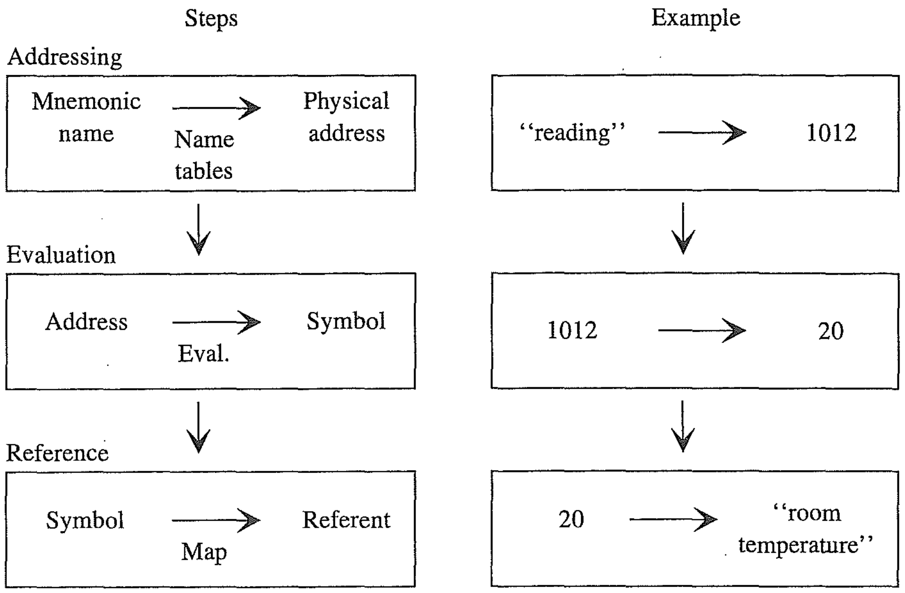


FIGURE 1.5. Designation for a variable involves addressing, evaluation, and mapping to some referent in the environment.

ment ("room temperature"). The addressing scheme provides each symbol with a unique address and supplies a reliable means for accessing the symbol through that address. In effect, the address is a name for the symbol. Computer languages often provide various external names that are mnemonics for referring to addresses.

1.1.3 *Causal Coupling*

All of the action of designation is in the mind of the observer. If an observer changes his mind about what a symbol refers to, that need not have any observable effect on the symbol or the environment. We now consider a more active relation where changes to the symbols or the environment can have effects.

Consider the heat-regulation system in Figure 1.6, which includes a thermocouple, a computer, and heating and cooling units controlled by the computer. In this system the electronics have been arranged such that the computer can access a symbol in its memory that it interprets as a reading of the temperature of a room on a numeric scale. The value of the temperature symbol is determined by a physical process starting with an interaction between the air in the room and a sensor. Electrical voltages corresponding to temperature are converted to digital signals by an analog-to-digital converter. The particular operating principles of the sensor and communication devices do not matter for the purposes of this example. The components are arranged so that when the temperature changes, a series of physical events follows, causing the temperature symbol — the contents of computer memory at address 1012 — to change. This kind of connection beween phenomena at one place and a distant symbol is called **causal coupling**. The term *causal coupling* can be expanded to include cases where all of the changes take place inside a single system, rather than just interactions between symbols in the system and entities in the environment.

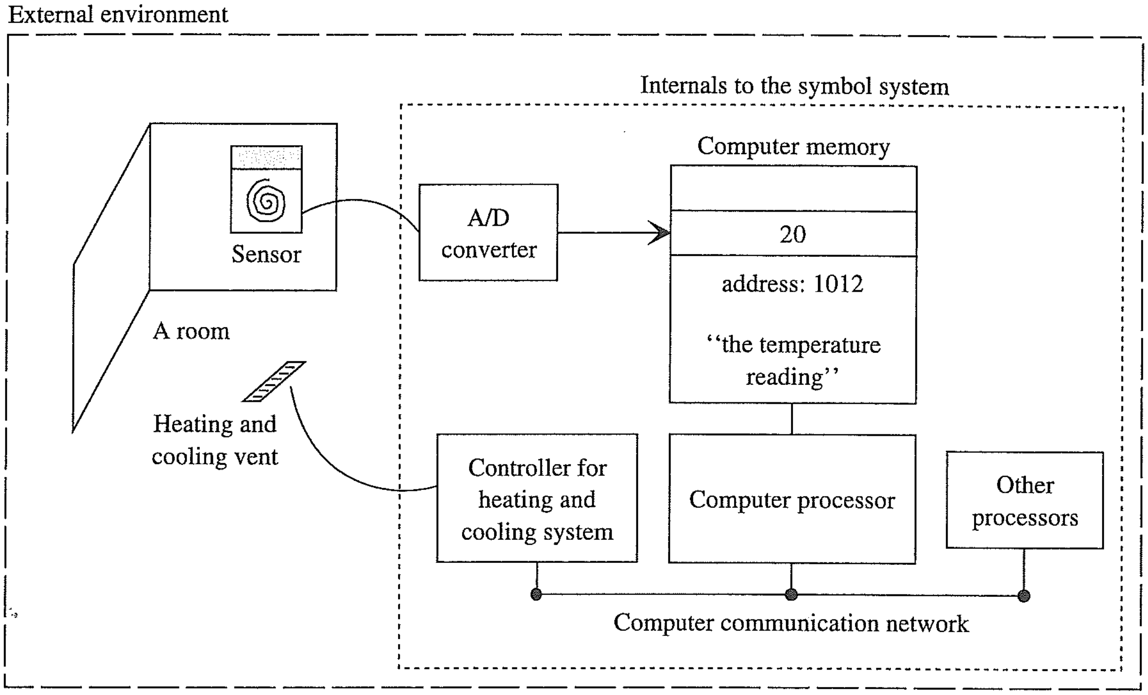


FIGURE 1.6. Causal coupling: In this example, a change in room temperature causes a change in the "temperature symbol" in a computer memory. Causal coupling can go from outside to inside as when a change in the sensor causes a change to a register in the computer's memory. It can also go from inside to outside, as when changing a register in the computer causes a setting to change in a heating vent. The key notion is that a phenomenon or a change in something causes a change in a symbol elsewhere.

There are always propagation delays between the time of the cause and the time of the corresponding change to a symbol. In large distributed systems, these propagation delays can be crucial in determining the dynamic behaviors of the systems. However, in the following examples, we will assume that the delays are negligible for the purposes of our analysis.

Causal coupling can go from outside a symbol system to inside such as from a sensor to a register in the example in Figure 1.6. It can also go from inside to outside, as when a change in a symbol in a controller in Figure 1.6 causes a change in the setting of a heating and cooling vent that controls the distribution of air. These two directions correspond respectively to sensing and motor control. Causal coupling can go between two symbols inside the computer such as when the system transmits a reading between different parts in order to decide whether to vent cool air into a room. Copies of the reading may be transmitted periodically along a computer communication network to various environmental control units in the building.

Causal coupling from an environment to symbols inside a computer is a very simple case of **machine perception**. In more sophisticated examples of machine perception, the effect on symbols is mediated by a potentially complex interpretation process. For example, consider a machine vision system that interprets digitized images from video cameras. The required computational process involves many levels of automatic analysis, and has many ambiguities and uncertainties associated with the sensor readings and the interpretation process. For example, the recognition of a person in a scene can involve processing of thousands of individual picture elements, recognition of edges and regions in the image, clustering of regions into larger images, comparing images from the sensor to stored images of objects expected in the scene, and so on until finally the image on the camera is associated with an identification symbol.

Causal coupling and designations are independent relations. A symbol may be causally connected to an object in an environment but not designate that object for some observer; similarly, a symbol may designate some object in the environment for an observer but not be causally connected with it. For example, suppose that the wire in Figure 1.6 becomes broken. In this event, the reading inside the computer probably will be incorrect and the tokens communicated to other systems will not be tied causally to the temperature of the room. The internal reading now contains misinformation. The neighboring systems may respond correctly, but based on misinformation. Indeed, reasoning about such disconnections illustrates an example where we need to reason separately about the two relations. Even though the wire is broken, we still want to say that the symbol "stands for" the temperature. Because the wire is broken, the mechanism is not operating correctly and the symbol may be wrong.

1.1.4 *Cognitive and Document Perspectives of Symbols*

Symbols in computational systems play two distinct and important roles: the document role and the cognitive role. When we are interested in the use of symbols to communicate with another person, we are taking a **document perspective**. In this role we are interested in symbols as **presentations**, that is, in the way that they can be used to explain things. Knowledge systems share properties with paper and electronic documents. They use symbols, they mediate communication, they are often constructed by groups of people, and people come to agreement about what the symbols mean.

When we are interested in the use of symbols for automated reasoning in the computer, we are taking a **cognitive** or **"mentalist" perspective**. In this role, we are interested in symbols as **representations**, that is, in the way that they model a situation, the properties of the symbols as the computer manipulates them in various ways to carry out reasoning, and in how we can use resulting symbols to infer things about the situation.

Figure 1.7 shows two users interacting with a graphical representation of scheduled temperature changes over time. In this case, the graph and various interactive menus for changing it constitute a user interface. They are external symbols that the users can see and talk about.

The situation in Figure 1.7 is representative of the context in which knowledge systems are created and used. Multiple people interact around a display. They create, discuss, share, and modify symbols that appear on the display. The symbols on the display are causally connected to symbols internal to the machine and perhaps to symbols on other machines. Part of the discussion is used to agree about meanings of the symbols. This is necessary in case multiple people are making changes to the system.

Sometimes we distinguish **model symbols** or domain symbols, such as the temperature reading symbol, from **view symbols** that represent and are causally coupled with presentations on the display. There are often many different kinds of view symbols, engaged in different parts of the process of rendering an appropriate image and supporting interactions with it.

Suppose two users are talking about the temperature control system, and one of them points to a display and says, "This is the room's temperature." He is interested mainly in the symbol structure on the display and probably means that it designates the room's temperature,

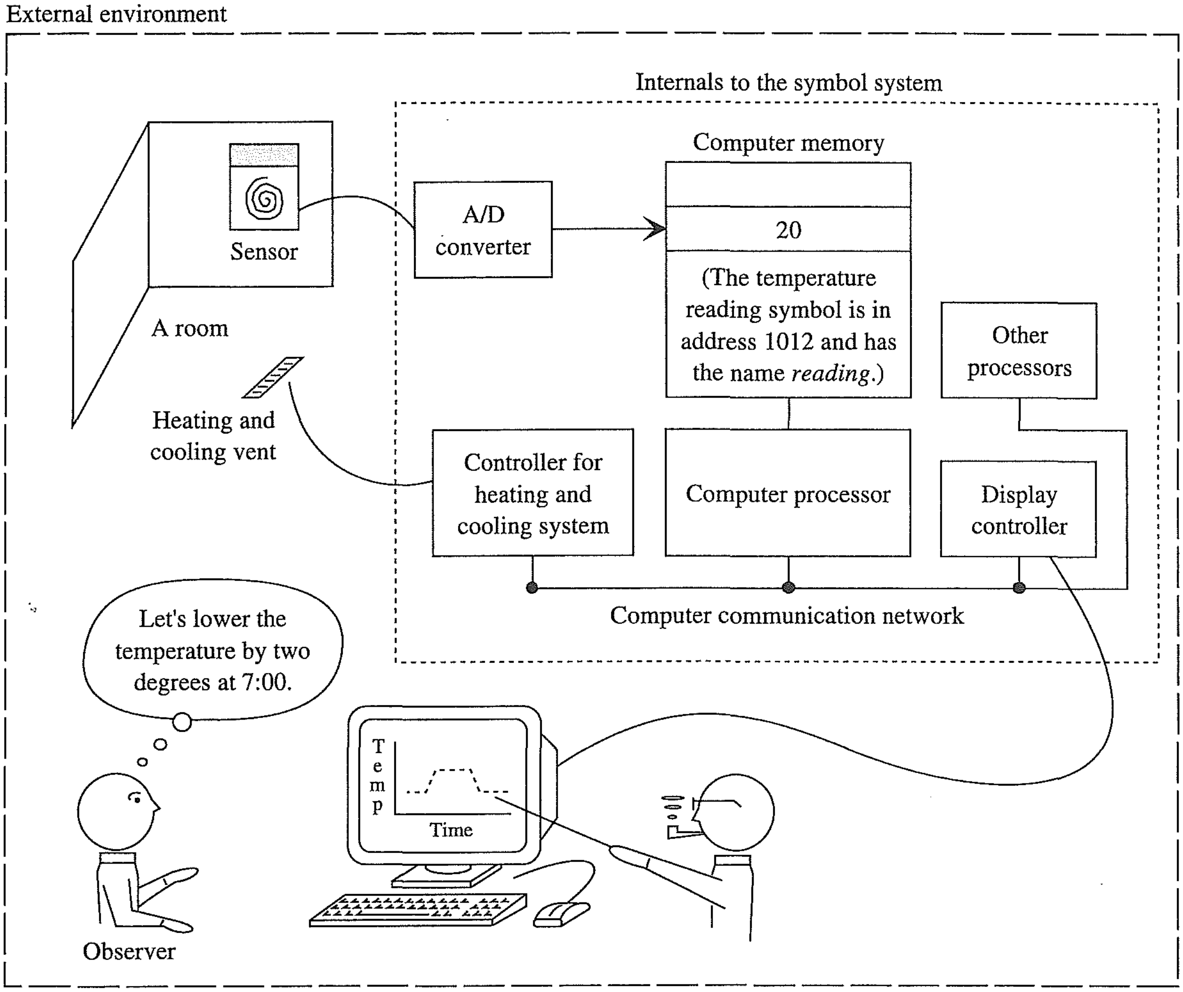

External environment
Internals to the symbol system
Computer memory
20
(The temperature reading symbol is in address 1012 and has the name *reading*.)
A/D converter
Sensor
A room
Heating and cooling vent
Other processors
Controller for heating and cooling system
Computer processor
Display controller
Computer communication network
Let's lower the temperature by two degrees at 7:00.
Temp
Time
Observer

FIGURE 1.7. Two people having a discussion about the planned control of room temperature over time. The mechanisms of the system including the sensing of room air temperature, the formatting of the display, the interactions with the user interface, and the control of the heating and air conditioning through the controller can be explained in terms of causal coupling. The assignment of meaning to symbols is described by the concept of designation.

that he expects the other party to be able to read and understand it, and that the indication it provides is up to date because of causal coupling. If the second person is a designer of the system, he might be concerned with many internal symbols. He or she would be familiar with all of its internal connections and mechanisms, could refer variously to a symbol on the display, a symbol used to construct the image rendering of the display, the reading symbol in computer memory, the "symbol" that is the digital electrical pattern at the output of the analog-to-digital converter, the symbol that is the analog voltage on the output line from the sensor, or even the symbol that is the configuration of bent metal inside the temperature sensor? To the designer all of these symbols play a role in the overall mechanism. When there is a chain of causal connections between symbols in a symbol system, it is sometimes convenient to ignore the differences between them in informal conversation.

1.1.5 *Summary and Review*

We began with a dictionary definition of symbols that says symbols are markings that represent things. This definition leaves open issues about the nature of markings and the nature of representation, leading us ultimately to introduce both a recognizer and an observer into our account of symbols.

We define markings relative to a recognition process. Symbols are physical patterns that can be read, recognized, and written by a recognizer. The identification of a set of markings as constituting a symbol is called registration. Symbol structures (or expressions) are arrangements of symbols in a physical medium. There can be multiple copies of a pattern or symbol in a symbol system. This gives rise to the terminological distinction between types and tokens. Types are classes of equivalent symbols in the alphabet of the recognizer; tokens are individual symbols in a medium.

We make few computational requirements on the physical patterns used for making symbols. The patterns must persist long enough for the workings of the symbol system and there must be a means for reading and writing the symbol structures.

We define representation relative to an observer. An observer is able to look both at the symbol system and a situation, and determines which elements of the situation are referred to by the symbols. This kind of meaning is a semantics of reference, and is called designation or denotation. The term "symbol" is sometimes used casually by computer professionals to mean roughly the same thing as "term" in propositional logic, or roughly, what we called a "marking." However, when more precision is warranted, symbolhood requires specifying both a recognizer to identify the markings and an observer to assign meanings to them.

An environment is a setting that contains a symbol system. Computer systems are embedded in larger systems. Causal connections link symbols to their environments in such a way that a change in the environment causes a change in the symbol or a change in a symbol causes a change in the environment. Our heat control system provides an example of this kind of connection, where changes in room temperature lead to changes in a reading.

We distinguish two perspectives on symbols in symbol systems. The document perspective emphasizes the observer's use of symbols for communication in presentations. The cognitive perspective emphasizes the computational use of symbols as representations.

Exercises for Section 1.1

■ **Ex. 1** [*CD-05*] *Warmups.* The following questions about symbols were raised early in this section. Indicate the answers with yes or no. If the question is ambiguous, explain briefly.

(a) *Yes or No.* Is any marking a symbol?

(b) *Yes or No.* Can the marking for a symbol be an electronic or biological encoding?

(c) *Yes or No.* Can we determine from a symbol itself what it represents?

(d) *Yes or No.* Can two people disagree about what a symbol represents?

(e) *Yes or No.* Can a symbol represent itself?

Ex. 2 [*CD-05*] *Recognizers and Observers.*

(a) Briefly, why are symbols (themselves) defined relative to a recognizer ?

(b) Briefly, why is designation defined relative to an observer?

(c) Briefly, what is the difference in concerns between part (*a*) and part (*b*) ?

Ex. 3 [*05*] *Designation for Symbols on a Computer Display.* Consider the following story. Two accountants are using a spreadsheet program to make financial projections about rental property. As is typical for spreadsheet models, they have defined variables that stand for things like rental income, utilities costs, mortgage payments, depreciation and so on. One of the two accountants points to a place on the screen and says, "The utility cost estimate for this month is too low because the tenants will probably run the air conditioner during the summer." Later they discuss whether monthly cashflow variables should include projected tax payments or depreciation.

In this chapter, we described a process of designation of program variables in terms of three steps: addressing, evaluation, and designation. Do these steps show up in the activities of the two accountants? Explain briefly.

Ex. 4 [*CD-05*] *Terminology.* For each of the following statements indicate whether it is true or false. If the statement is ambiguous, explain briefly.

(a) *True or False.* An embedded computer system is a computer that is hidden inside a larger system.

(b) *True or False.* In machine sensing, causal coupling means a change in the environment causes a change of a symbol or symbol structure.

(c) *True or False.* In machine control, causal coupling means a change of a symbol or symbol structure causes something to change in the environment.

(d) *True or False.* We define both types and tokens with respect to a recognition process.

(e) *True or False.* The document perspective of symbols in a knowledge system requires that symbols (or displays of them) need to be readable by observers who can discuss their meanings.

■ **Ex. 5** [*!-15*] *Storyboards and the Role of Assumptions in Understanding Narrative.* Several projects in AI have sought to create computer systems that understand stories, that is, that give summaries of stories and answer questions about them.

This open-ended exercise shows how the translation of natural-language sentences into formal representations requires knowledge about the situations that the sentences describe.

Consider the following two-sentence story about Paige and Morgan, a girl and her brother.

Paige rolled the ball to Morgan.
Morgan threw it back to Paige.

(a) A storyboard, such as is used in making movies and cartoons, is a sequence of scenes, each of which is a snapshot of the story world.

Imagine a sequence of scenes representing the sequence of events in the story. Fill in information for the intermediate scenes that would be represented in a storyboard for our sample story.

For all scenes.
Paige is a person.
Morgan is a person.
There is a ball.
Paige and Morgan are in a play area.
They are playing together.
(Various implicit facts about orientation, mass, roundness of balls, gravity, air, that are not really mentioned in the story.)

Scene 1
Paige has the ball.
She starts to roll it to Morgan.
(Various implicit facts about Paige's feet being on the ground, how she moves, etc.)

Scene 2
The ball is in motion, rolling from Paige to Morgan.
Paige no longer has the ball.
Morgan is paying attention to the arrival of the ball.

Scene 3
. . .

Scene 6
Paige catches the ball.
Paige has the ball.

(b) In the sample two-sentence story, does the story say explicitly who has the ball at the end? What assumptions might you use to answer the question? How does understanding the second sentence rely on understanding the first one?

Ex. 6 [*!-10*] *Assumptions and Nonsense*. The sentences we use for efficient human communication can be remarkably brief. This is possible because we use many clues from the context to understand what is being said. In doing that, we also make many assumptions. Consider the following answer that one person gave another when asked if he could have a ride to the airport.

1. I could give you a ride.
2. Except that my car is being fixed at the garage.
3. But it should be ready by now.
4. Except I don't have any money to pay for the repairs.
5. But I can borrow some from Dan.
6. Except Dan isn't around right now.
7. But he should be right back.

(a) Describe what a normal, competent and rational listener would believe about the availability of a timely ride to the airport after hearing each of these statements.
(b) What does this tell us about inferential processes for understanding stories.

Ex. 7 [*T*] *The Knowledge Representation Hypothesis.* Brian Smith (1982) proposed the knowledge representation hypothesis, which follows:

> Every intelligent physical symbol system includes symbol structures that we as external observers can take to be a propositional account of the system's knowledge. These symbol structures play an essential and causal role in determining the system's behavior. Furthermore, the system's behavior is independent of our account of its internal structures.

Smith does not require a trivial mapping of one symbol structure to one proposition, and the symbol structures can be graphs, arrays, or distributed symbols. The encoding process could be arbitrarily complex. For example, in a particularly perverse security robot the symbols could be recorded using an encryption algorithm, which presumably would make the "mentalese" elusive or computationally intractible to decipher. Thus, the knowledge representation hypothesis covers a wide range of representational approaches.

(a) Briefly, compare the knowledge representation hypothesis to the physical symbol system hypothesis. What does the knowledge representation add, if anything, to the physical symbol system hypothesis? How does Smith's notion of a propositional account differ from Newell and Simon's notion of designation?

(b) What does the knowledge representation hypothesis guarantee about an observer's ability to make sense of "mentalese"? What does it predict about the correspondence between symbols used by the observer and symbols in the observed system?

(c) Why is it useful to ascribe levels in representational theories of mind?

(d) Are these hypotheses the subject of extensive inquiry in artificial intelligence research? Of what use are they to the study of knowledge systems?

Ex. 8 [*CD-05*] *Defining Symbols*. The dictionary defines a symbol as a written or printed mark that stands for or represents something. In this section we have argued that there are several practical issues about this definition.

(a) Briefly, what issues does the registration problem raise about the dictionary definition of symbols?

(b) In defining designation, the text claimed that designation is not a property of the symbol itself. Why not? Briefly, what issue does this point about designation raise about the dictionary definition of symbols?

1.2 *Semantics: The Meanings of Symbols*

The term *semantics* is used to describe both natural language and computer languages. It is often contrasted with syntax and is popularly understood to refer to the meaning of symbols and expressions in languages. "Getting the semantics right" is a goal often lauded in computer science.

In this vein, Hayes (1974) and others have argued for principled and systematic design of semantics for representation languages used in computers. As he put it, representation languages ought to have a semantic theory. Later in this section we consider in more detail what Hayes meant by this. For now we note that proposals like this have taken on broader appeal as the knowledge bases being proposed have increased in size and as it has seemed worthwhile to be able to combine knowledge bases that were developed separately.

There are many pragmatic issues in developing semantic theories for knowledge systems. The problem is not just principled versus "sloppy" semantics. More fundamentally, there is confusion about what kinds of semantics are needed.

A **semantics** is an approach for assigning meanings to symbols and expressions. Different kinds of semantics differ in the kinds of symbols considered and the kinds of meanings they assign. In the following we will consider several kinds of semantics that are relevant for understanding knowledge systems.

We begin by reviewing the most studied approach to semantics, the declarative semantics used for the predicate calculus. We then broaden the discussion by considering some historical examples of how AI researchers assigned meaning to expressions in representation languages and graph structures. For this it is convenient to establish some terminology and notation for graphs and trees. The terminology introduced here will be sufficient to carry us through the first three chapters of this book. Finally, we consider how different kinds of semantics are needed for different purposes. We compare several kinds of semantics that are relevant to knowledge systems.

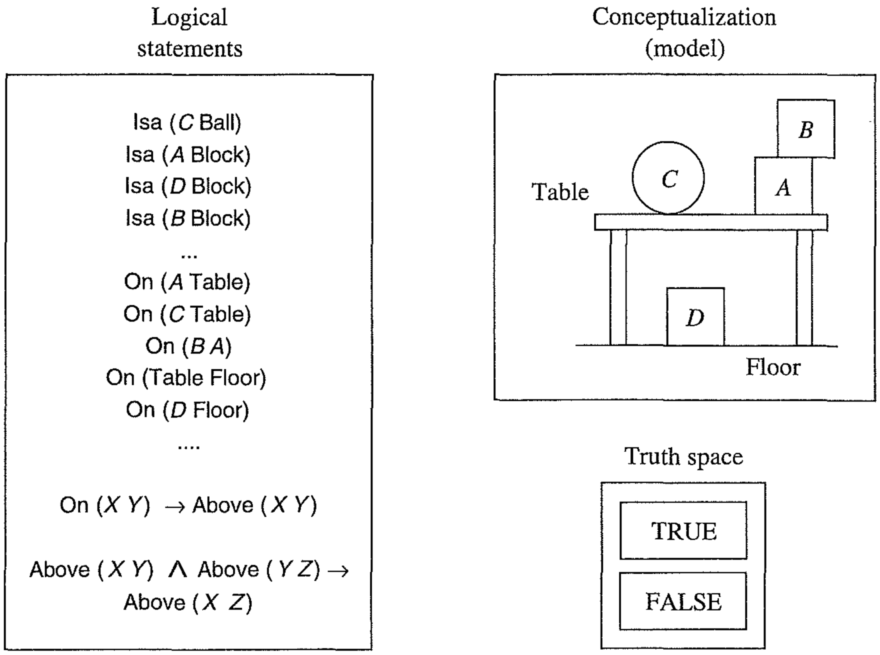


FIGURE 1.8. The elements of a declarative semantics for logic.

1.2.1 *Model Theory and Proof Theory*

We begin with the semantics developed for formal logic. We use this to establish a vocabulary and a basis for comparing other kinds of semantics. Our presentation is brief, since the goal is to summarize the concepts of semantics used with the predicate calculus.

Constants, Variables, Interpretations, and Models

Predicate calculus has two kinds of symbols: constants and variables. Constants are used to name elements in a special set known as the **universe of discourse**. In other words, this set includes terms for all of the things we are talking and reasoning about. Object constants name the objects in the set. Function constants are used to designate functions on members of the set. For example, arithmetic operations would be defined as functions. Relation constants name relations on the set.

Variables are used to describe properties of objects in the set without naming them. The predicate calculus includes relations for the logical operators ∧ (AND), ∨ (disjunctive OR), and the usual others as well. A first-order predicate calculus includes universal and existential quantifiers on objects, but not on functions or relations.

Here is an example of an expression in predicate calculus in the example in Figure 1.8:

On (Ball Table)

In this example, On is a relation constant. Ball and Table are object constants. Another is

Above (*X Y*) ∧ Above (*Y Z*) = >Above (*X Z*)

These statements are also called **axioms** in the model. Statements like these are what we write when we "axiomatise" a domain.

The first part of the semantics of predicate calculus is essentially the same as the semantics of reference from Section 1.1. We assume there is a memory or database of statements in the calculus. There is an observer and the observer has in mind a conceptualization, analogous to what we have called an environment. Designation in the mind of the observer associates the object, function, and relation constants with their counterparts in the conceptualization. In the terminology of logic, a mapping from the statements to the conceptualization is called an **interpretation**. Unfortunately, this is one of those words which has many different meanings in computer science.

For any term, we define its **extension** to be the elements in the conceptualization that it designates. Thus the extension of the term *C* would simply be the ball sitting on the table. A term like Block refers to a class of objects, whose members include the blocks *A*, *B*, and *D*. Similarly, an indefinite node might correspond to a variable whose referent has not yet been precisely determined, but that is perhaps limited to be some member of a class. So far we have not related the meaning of such terms to the inferences carried out on the model. To build a working system, claiming that terms refer to something does not signify much if the system does not reason with them. To formalize such reasoning, we need to add semantics of truth and semantics of proof.

The next part of the semantics involves the assignment of truth to logical statements. The basic idea is quite simple. First we consider ground statements or ground literals, which are ones that can be tested quite simply in an interpretation. For example, the statement that "*C* is a ball"—Isa(*C* Ball)—and the statement "*A* is on the table"—On(*A* Table)—can be readily checked. We say that they are satisfied in the obvious or intended interpretation of Figure 1.8. For a different interpretation they might not be satisfied. In Figure 1.8, the statement On(*B* Floor) is not satisfied. **Truth semantics** gives us a way of assigning a truth value to logical statements relative to an interpretation.

If an interpretation satisfies a sentence for all variable assignments, then it is said to be a **model** of the sentence. For example, the following sentence is satisfied by the intended interpretation of Figure 1.8 for all assignments to the variable *X*.

(∀*X*) Isa (*X* Ball) ∨ Isa (*X* Block) => Above (*X* Floor)

Variable assignment has no effect if there are no variables. Any interpretation that satisfies a ground sentence is a model of that sentence. A sentence is satisfiable if and only if there is some interpretation and variable assignment that satisfy it. An interpretation is a model for a set of sentences if it is a model for every sentence in the set.

The Meanings of Expressions

Much of the utility of predicate calculus arises from the systematic rules for assigning truth values to new expressions in the language. For example, if we are given that both of the following statements are true,

Isa(Block *A*)
Isa(Block *B*)

then we can conclude that the conjunctive statement is also true.

$\text{Isa(Block } A) \wedge \text{Isa(Block } B)$

The rules for this, which are learned by every beginning student of logic, are what people refer to as the well-founded semantics of logic. For another example, suppose we are asked whether the following formula is true.

$(\text{Isa (Block } A) \wedge \text{Isa (Block } B)) \vee \text{Isa (Block } C)$

We can assign

$X = \text{Isa (Block } A)$
$Y = \text{Isa (Block } B)$
$Z = \text{Isa (Block } C)$

In our standard interpretation, we have

$X = \text{True}$
$Y = \text{True}$
$Z = \text{False}$

The next step is to determine the truth value of the expression.

$(X \wedge Y) \vee Z$

One way to do this is with a truth table as in Table 1.1. In this table, we build up the values for the total expression from the truth values of the subexpressions. This is exactly the sort of property that Hayes requested of a semantic theory, that is, an account of how the meaning of a whole symbol structure is built up from the meanings of the parts. Of course, in complicated expressions there are often shortcuts and it is not usually necessary to write out a complete truth table. Nonetheless, at least in the propositional calculus, this is an approach than can be followed in case of doubt.

TABLE 1.1. A truth table.

X	Y	Z	$(X \wedge Y)$	$(X \wedge Y) \vee Z$
T	T	T	T	T
T	T	F	T	T
T	F	T	F	T
T	F	F	F	F
F	T	T	F	T
F	T	F	F	F
F	F	T	F	T
F	F	F	F	F

Formulas that are always true, regardless of the truth or falsity of their terms, are said to be **valid**. Such formulas are called **tautologies.** In propositional logic, truth semantics gives us a very simple and automatic way to determine whether a formula is valid. We simply compute its truth table and check whether every combination of truth assignments yields true for the formula.

Continuing with our example, if we have the statements

On (*A* Table)
On (*B A*)

then we can establish that the following statements are true without any further consultation with the model.

Above (*A* Table)
Above (*B* Table)

We can also ask questions about the truth of a statement without associating it with a particular interpretation. Some sentences are true for every possible interpretation. For example, the left statement below implies the right statement for all possible interpretations. In such cases, we say that the sentence on the left "logically implies" the sentence on the right.

$(\sim A \wedge B) \Rightarrow A \vee B$

The semantics of logic also deal with what is called provability. Predicate calculus provides prescriptions for establishing the truth or falsity of expressions, either from axioms or from expressions already established. Sanctioned inferences are described by **rules of inference,** which can be used to deduce new facts from old ones. The best known rule of inference is **modus ponens,** which says that if we have a fact *p*, and that *p* implies *q*, then we can deduce q. In the concise notation of logic, we would say

modus ponens: $A \wedge (A \Rightarrow B) \vdash B$

where the turnstyle figure means "proves." Figure 1.9 gives an example of the use of modus ponens to deduce that Felix is a member of the artificial intelligentsia. Roughly speaking, a **proof** is a sequence of statements where each successive statement follows from the preceding ones by some rule of inference, and the last statement is the "proved" conclusion.

A second important rule of inference is **universal instantiation.** It says that if something is true of everything, it is true of any particular thing. In logical notation, we would say

universal instantiation: $(\forall x)\ A(x) \Rightarrow B(x),\ A(a) \vdash B(a)$

Figure 1.10 shows how universal instantiation can be used to deduce that Ken is brilliant.

Truths, Proofs, and Decidability

The related ideas from truth theory and proof theory can be written using special symbols as follows. The following is a predicate calculus statement read as "*A* implies *B*."

$A \Rightarrow B$

```
Given the rule
    Can-pronounce (Student "heuristic")      ;IF a student can pronounce heuristic
    => Member (Student                       ;THEN he is a member of the
              Artificial-intelligentsia)     ;artificial intelligentsia

And the fact
    Can-pronounce (Felix "heuristic"         ;Felix can pronounce heuristic

Use modus ponens to deduce
    Member (Felix Artificial-intelligentsia) ;Thus, Felix is a member of the
                                             ;artificial intelligentsia
```

FIGURE 1.9. A logical deduction using modus ponens.

By itself this expression is neither true nor false. It is satisfied relative to an interpretation and variable assignment if and only if *A* is not satisfied (true) or *B* is satisfied in the interpretation. This rule is from truth theory. If such an implication is true for *every* interpretation and variable assignment, then we say "*A* logically implies *B*" and write the following.

$$A \models B$$

Note that this is not a statement *in* the predicate calculus, but rather, is a statement *about* predicate calculus statements. Similarly, we write the following if *B* is a tautology.

$$\models B$$

Finally, if there is a formal proof from *A* to *B* we write the following.

$$A \vdash B$$

```
Assuming the rule
    ∀(Student)                        ;For all students
    Undergraduate (Student)           ;IF the student is an undergraduate
    ∧ Institution (Student Cal-Tech)  ;at Cal Tech
    => Brilliant (Student)            ;THEN the undergraduate is brilliant

And the facts
    Undergraduate (Ken)               ;Ken is an undergraduate
    Institution (Ken Cal-Tech)        ;Ken goes to Cal Tech

Use universal instantiation to deduce
    Brilliant (Ken)                   ;Ken is brilliant
```

FIGURE 1.10. A logical deduction using universal instantiation.

A proof of a sentence is a finite sequence of sentences in which each element is a sentence chosen from a set, a logical axiom, or the result of applying a logical rule of inference. An important result of mathematical logic states that whenever a set of sentences implies another sentence, then there exists a finite proof of that sentence. For this reason the predicate calculus is said to be **decidable**.

Such proofs offer us yet a third possible semantics for logic, called a proof semantics. A **proof theory** determines whether a given expression is valid, that is, derivable, from a given database of facts. It is valid if there is a proof. For a given statement, there may be many possible proofs or no proofs at all. A **proof semantics** can be defined to associate a statement with a proof, or more simply, with "valid" or "not valid."

Proof theory establishes a standard of reasoning. It brings sense to many examples of confusing and illogical reasoning. For example, consider the following.

Given:	All fleas like some dog.	
	No fleas like any swimmer.	
Conclude:	No dogs are swimmers.	(1)

Even if we suppose that the first two sentences are true relative to some interpretation, the conclusion does not follow. For example, all fleas could like the same nonswimming dog. Proof semantics provides a careful and systematic account of when deductions are justified, that is, what follows rightfully from what. Reasoning that follows the appropriate logical principles is said to be sound. Sound is used here as a technical term to describe systems or methods that derive no more than can be supported according to explicit rules of logic. Thus, the conclusion in (1) is not sound. In contrast, the conclusion in (2) is sound but bogus. Bogus is not a technical term. The fault lies not in the inference but in the nonstandard interpretation.

Given:	Some fleas like all dogs.	
	No fleas like any swimmer.	
Conclude:	No dogs are swimmers.	(2)

But how can we tell which inferences are sound? The approach in logic is to characterize sound inferences systematically as those that follow from specified rules of inference.

In summary, there are three parts to the semantics of predicate calculus, the semantics of reference, the semantics of truth, and the semantics of proof. Each is a mapping from symbols and expressions to some kind of meaning, where the meaning may be elements in a model, the symbols *true* and *false*, or a proof. These are related respectively to model theory, truth theory, and proof theory. Sometimes this whole approach is called a **declarative semantics**.

1.2.2 *Reductionist Approaches for Composing Meanings*

A **language** is a set of expressions. In natural languages, a conventional unit of expression is the **sentence**. In English, sentences contain subjects and predicates. Crucial to the notion of a language is that there is a way of deciding whether any particular arrangement of symbols is a sentence in the language. In written and spoken natural languages, grammars are sets of rules that determine which arrangements of symbols are sentences. Thus, the expression "Shoe blue knob five running." is not a sentence and "He runs in blue shoes." is a sentence.

One remarkable fact about natural language is that indefinitely many linguistic expressions have meaning for people. Consider the following sentence:

The pink mouse flew her helicopter downtown to the opera. (3)

We have no trouble attributing a meaning to this silly sentence, even though it is unlikely that any reader of this book ever encountered it before reading it here. How can this be so? It suggests that the experience of understanding a sentence that we have never seen before is akin to the process of (say) adding two numbers we have never seen before or driving a car we have never seen before. We can do the addition because we view the numbers in terms of their smaller pieces, such as the digits in the ones column, the digits in the tens column, and so on. We have an algorithm that takes account the significance of the decimal representation and the rules for combining 1-digit numbers.

By this analogy, we expect it to be the case that there is a systematic way of interpreting expressions in natural language. This is a structural approach to composing meanings. Sometimes we call it a **reductionist** approach because the meaning of the whole sentence derives from the meanings of its parts.

The truth and proof semantics of predicate calculus both follow a reductionist approach in that the meaning of an expression is determined in a systematic way from the meanings of its parts. For example, the truth value of the expression $A \wedge B$ can be determined systematically from the truth value of A, the truth value of B, and the usual definition of $\wedge$.

It turns out that a reductionist view of the composition of meaning is not quite adequate for sentences in natural language. Consider the sentence "Is there any salt?" Asked of an environmental scientist measuring water quality, this sentence is probably a request for information about the results of his measurements. Asked of a waiter at a restaurant, the question would be interpreted as an indirect request to bring a salt shaker to the table. Part of the problem is that the sentence is not always the right unit of analysis. Context is provided by other sentences. Another problem is that this account of meaning fails to take into account the role of the speaker, the listener, and the situation.

For all of the shortcomings of a structural approach to semantics in natural language, this approach is important for systematic computer representations. We need to know how the meaning of an expression depends on the meaning of the terms in the expression. An account of the composition of meaning should draw on regularities in the arrangement of symbols. Restated, regularities in meaning should be reflected by regularities in symbol structures.

Computer systems use many different kinds of representations, not just sentences from a predicate calculus. Consider the following paragraph:

> When I write at home, I often look out the window at some of California's giant redwood trees. This morning on one such tree, I saw some branches high above a hammock outside where age and neighboring growth have combined to cause them to turn brown and die. Sometime in the months ahead these branches will come crashing down. I remembered that I should phone a tree surgeon to tend to them before they land in the hammock.

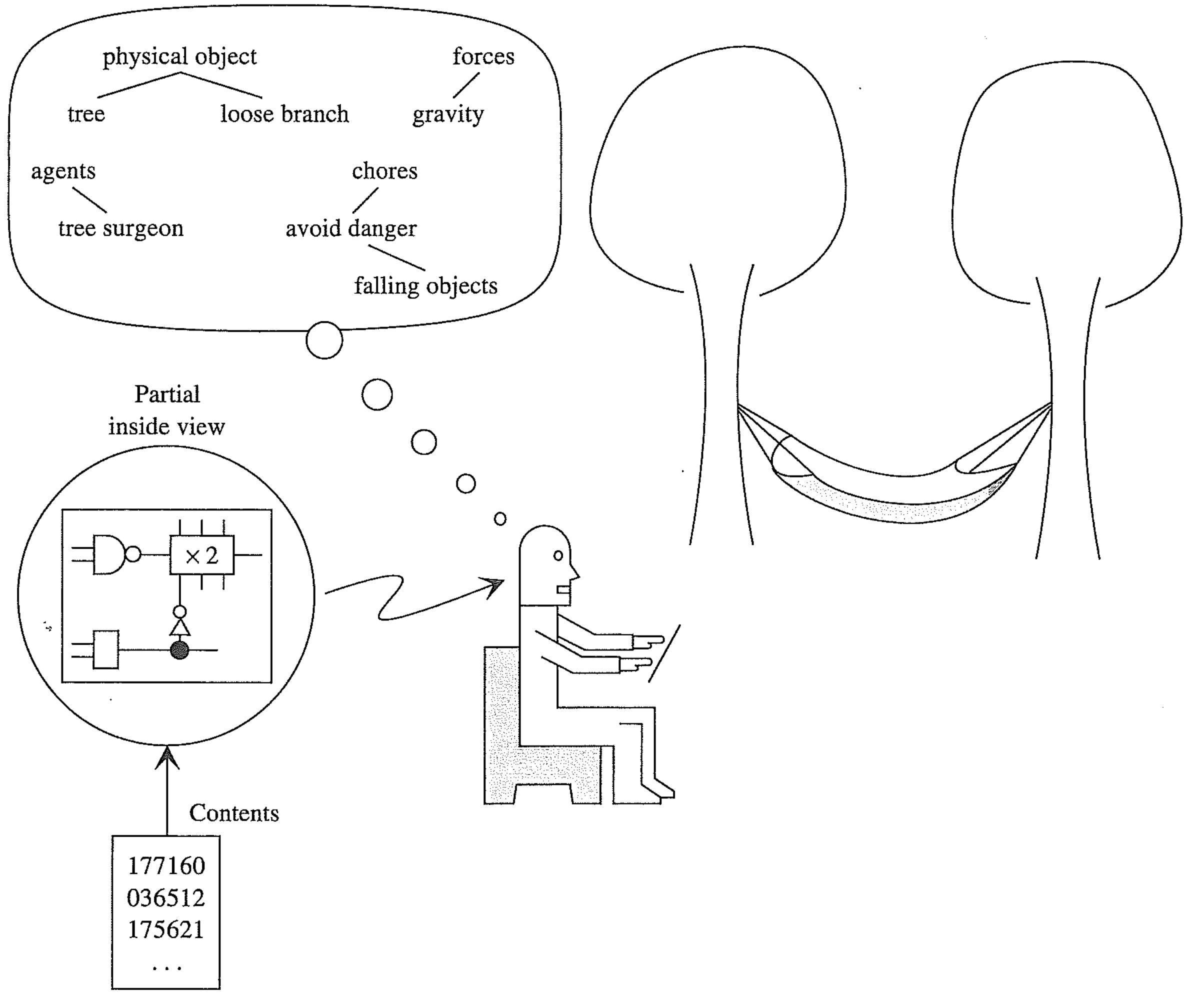


FIGURE 1.11. Deciphering the "mentalese" of the Mark-1 writing robot. This is analogous to trying to figure out how a computer works from the circuit diagram but without the operations manual.

According to Newell and Simon's physical symbol system hypothesis, the cognitive parts of this story—writing, observing trees, and planning activities—can be accounted for in terms of the processing of symbols, that is, marks in a memory and a processor that can read and manipulate those marks. Let us suppose for the purposes of this discussion that the agent in the previous paragraph is actually a manufactured physical symbol system, a "Mark-1" writing robot. As suggested in Figure 1.11, somewhere inside the Mark-1 there is a memory medium with marks that we expect correspond to "home," "giant California redwood trees," "hammock," "tree-surgeon," "phoning" and so on for whatever knowledge and beliefs the robot may be said to possess. In a well-ordered system we would also expect to find symbol-processing machinery that causes the anticipation of falling branches and the phoning of a tree surgeon. It is not necessary that the Mark-1 itself be able to tell us about the location and encoding principles for the underlying symbol system. However, if we as observers can discover the "mentalese" of the Mark-1, we expect to find symbols and processors that are causally connected with the behavior of the robot.

In artificial intelligence and cognitive science these expectations about symbols in memory and their connection with intelligent behavior are at the core of **representational theories of**

mind. In the context of knowledge systems, the emphasis shifts from deciphering mentalese to the systematic use of symbols in a computational knowledge medium.

By a **semantic theory** Hayes means an account of the way that particular configurations of symbols in a representation scheme correspond to particular configurations of the external world. By calling it a "theory," Hayes demands more than just allowing observers to assign arbitrary meanings to symbols. He wants a systematic approach for indicating how sentences in the language represent the subject matter. The grammar rules that determine whether a set of words is a sentence should also help us to determine what the sentence means. The regularities of this explain how we make any sense out of the sentence about the pink mouse in the helicopter.

Seeking a semantic theory shifts the focus from a concern with interpreting or decoding the symbol structures of a particular robot to the design of representation languages of adequate power for which we can give a principled account of what symbols written in them mean. The goal of this revised enterprise is technical: It is the development of engineering principles by which we can design symbol structures and knowledge systems whose properties are predictable and understood. In both cases—mentalese languages of the mind or representation languages—what we seek is a systematic way of assigning meanings to expressions.

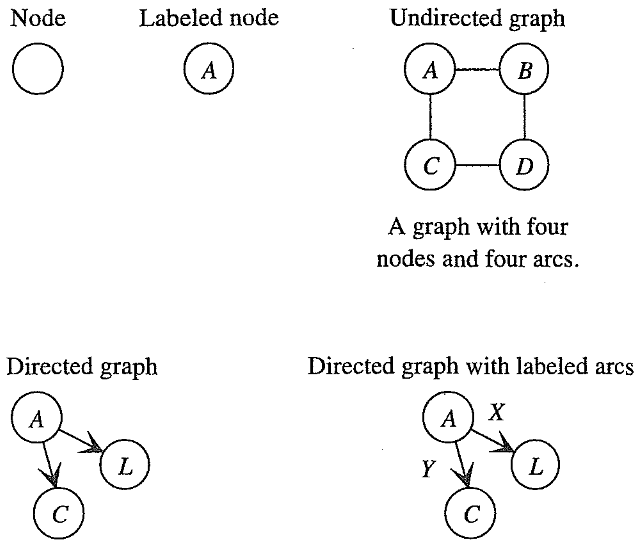
Returning to our analogy about the understanding of natural language, we would like to ask questions about the meaning of symbols and to derive answers using a computational process on the symbol structures in memory. So far, the declarative semantics of predicate calculus satisfies the requirements we have listed. Later in this section, we discuss why additional kinds of semantics have been developed to satisfy additional requirements. First, however, we will look at further examples of representational structures and at some of the approaches for assigning meanings to them that are in the same spirit as the declarative semantics although they are usually less-thoroughly developed.

1.2.3 *Terminology for Graphs and Trees*

Graphs are made up of two kinds of elements usually called **nodes** and **arcs**. Figure 1.12 gives several examples of graphs. The nodes are the circles and the arcs are the lines connecting them. When the nodes or arcs have distinguishing labels the graph is called a **labeled graph**. It is common in depictions of knowledge representations to use graphs with labels on the arcs and the nodes. Graphs are also distinguished as being either **directed** or **undirected**. Directed graphs have directed arcs, meaning the two ends are distinguishable. Directed arcs have an orientation, meaning that they start at one designated node and end at another. They are usually represented visually as arrows as shown in Figure 1.12.

A graph is **cyclic** if there is a path, starting from one of its nodes, that leads along the arcs from one node to another leading back eventually to the starting node. For directed graphs, the path must follow in the direction of the arcs. An acyclic graph is a graph with no cycles. Figure 1.13 gives examples of cyclic and acyclic graphs. Directed acyclic graphs are called **dags**.

We define **trees** as directed acyclic graphs in which every node (except the root node) has exactly one ancestor. The **root node** is the unique node in the tree having no ancestors. Sometimes the term **forest** is used to refer to a set of trees.



Node

Labeled node

Undirected graph

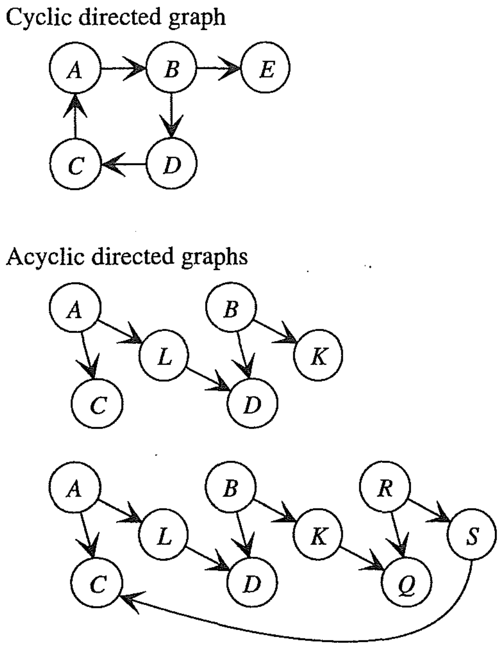
A graph with four nodes and four arcs.

Directed graph

Directed graph with labeled arcs

FIGURE 1.12. Some basic terminology about graphs.

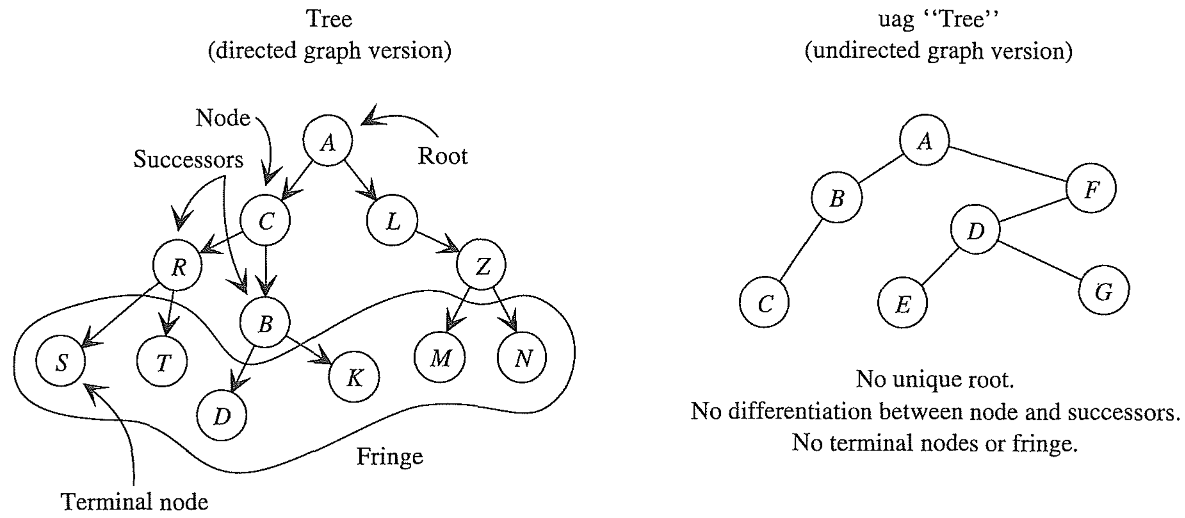
Trees are a very important special case in graph theory and computer science. As is apparent from the reference to "ancestors" and "root node" in the definition, trees have their own special terminology, which we will now make more precise. For trees, the directed arcs are also called **branches.** Directionality of the arcs point is important. Borrowing familiar language from family trees, we say that branches directly connect **parent nodes** with **children nodes.** Conven-



Cyclic directed graph

Acyclic directed graphs

FIGURE 1.13. Examples of cyclic and acyclic directed graphs.



Tree
(directed graph version)

uag "Tree"
(undirected graph version)

No unique root.
No differentiation between node and successors.
No terminal nodes or fringe.

FIGURE 1.14. Comparing two definitions of trees.

tionally, arcs point from parents to children. **Ancestor** and **descendant** nodes are defined in the obvious way. Mathematically inclined writers often prefer to use the term **successors** to refer to children nodes. Nodes with no successors are called **terminal nodes**.

Trees can be generated or drawn a little at a time. In this case, the term **leaf node** is used to refer to terminal nodes or to any other nodes showing no successors, even if further generation may potentially cause more successors to be presented. The set of nonterminal nodes is collectively called the **interior** and the set of leaf nodes is collectively called the **fringe**. In reasoning problems we often consider trees that are expanded incrementally. In these cases, the terms leaf node and terminal node are sometimes used to refer to nodes that have no successors yet, although they may later. We use the terms **dynamic fringe** or **frontier** in such cases to refer to the set of the deepest nodes in the tree that have been explored so far.

The term **branching factor** refers to the number of successors of a node. Often for the purposes of analysis, it is convenient to assume that all of the interior nodes in a tree have the same branching factor. When different nodes have different numbers of successors, we sometimes use an average branching factor for a set of nodes.

Before leaving this discussion of terminology, we note that although our definition of *tree* is the one most commonly used in computer science, it differs from the definition of *tree* most used in mathematics. In mathematics, trees are usually defined for *undirected* graphs rather than *directed* graphs. There are several equivalent definitions for trees as undirected graphs. (1) A tree is a graph that is connected and that has one more node than arc. (2) A tree is a connected, acyclic graph. Undirected trees are called "uags" or undirected, acyclic graphs.

Figure 1.14 compares trees based on directed and undirected graphs. In uag trees, there is no privileged node that stands for the root, no nodes are characterized or terminal or in the fringe, and there is no orientation to links differentiating nodes and their successors. It has been said that you can "pick up" a uag tree from any node so that the rest of it "hangs down." The directed version of trees is commonly associated with linked data structures in computer science and with search processes that have a starting place.

1.2.4 *Graphs as Symbol Structures*

In the following we will consider examples of a graphic descriptive language often called a **semantic network**. The term *semantic network* arises from Ross Quillian's Ph.D. thesis in which he used them as network models of information. Semantic networks are used for different purposes, are assigned meanings (semantics) in different ways, and are depicted by figures with nodes and directed arcs.

Since Quillian's thesis semantic networks have been used to model all sorts of nonsemantic things such as propositions in logic, the physical structure of objects, and the behavior of devices. At some time, virtually every one of these representations has been called "semantic" by someone. In AI and knowledge engineering, the term *semantic network* refers generally to a wide class of informal and formal symbolic representations. What these representations have in common is that they are all made of links and nodes. In hindsight, it is clear that this definition is indistinguishable from that of a **graph**. Graphs and graph structures have many useful properties as representations. However, *there is nothing fundamentally "semantic" about graphs.*

Graphs can mean things in just the same way that sentences in a language can mean things. We choose graphs as examples of representations simply because so many representations in knowledge systems are graphs. The issues are much the same whether the representations are graphs, grammatical sentences, or bitmaps.

When we use the term *semantic network*, we draw on vocabulary from the AI literature. We develop several informal variations of semantic networks in the following, using them to show why we need principles for systematic representation languages.

Consider the sentence:

Willy threw a ball to Morgan. (2)

Figure 1.15 shows one way to represent the information stated in this sentence using a graph. The syntax of our graph is simple. **Nodes**, depicted here as ovals, stand for various kinds of **objects** and **links**, depicted here as arcs with arrowheads and labels, stand for various kinds of **relations**. Arcs naturally have two ends, so they are most useful for representing two-part or binary relations. Relations involving more than two parts (*n*-ary relations) can be represented as nodes. Thus, in addition to the nodes that correspond to physical objects (Willy, Morgan, a ball) there are nodes that stand for relations. For example, the verb *throw* is represented as a particular 3-ary relation called a "throw event." The binary relations include did-action, thrower, thrown-to, and object-thrown.

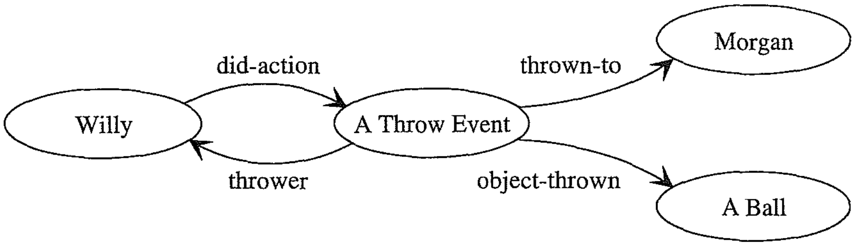


FIGURE 1.15. Example of a semantic network.

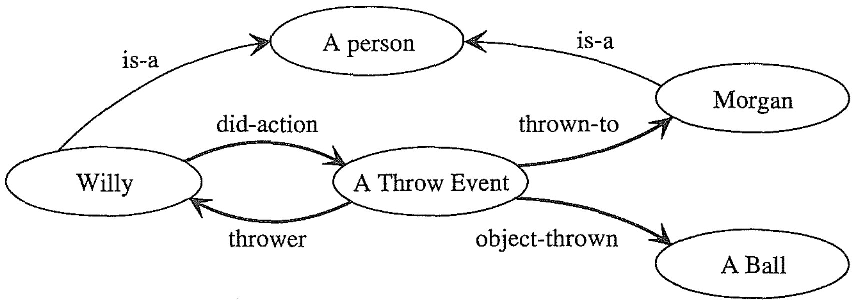


FIGURE 1.16. Extended example of the semantic network from Figure 1.15, showing some assumed relations (thin lines).

For such symbol structures to be useful in knowledge systems, there must be a way to structure knowledge in expressions and there must be a systematic way to determine the meanings of expressions from the individual terms and the structure of the network. Given the network in Figure 1.15, we could ask:

- Was something thrown?
- Is the ball a baseball?
- When did the event take place?
- Is Willy a person?

From Figure 1.15, we might guess that the presence of a "throw event" indicates that something was thrown. There is no indication at all of the kind of ball; nor is there any explicit indication that the kind of ball has not been determined. The form of the verb *threw* in (2) suggests that the event took place in the past. Unless the binary relation did-action indicates when the event took place, however, nothing is known about the time of the event. There is no explicit indication that Willy is a person, although this is a reasonable inference to make from the use of capitalization in the English sentence in (2). In a revised version of the semantic network in Figure 1.16, the graph is augmented with some thin-line arrows, intended to represent other relations, not stated in the sentence, but perhaps inferred from some context. The revised figure indicates (loosely speaking) that Willy and Morgan are people.

Unfortunately, this is much too glib. The history of representation languages in AI shows that it is easy and misleading to ascribe unwarranted knowledge and power to representations when we must use human intelligence to interpret them. A simple experiment demonstrates this. Consider how unintelligible the network becomes in Figure 1.17 when we substitute numbered symbols ("gensyms") for names: g0001 for is-a, g0002 for Willy, and so on. The loss of intelligibility reveals the amount of background that we unconsciously use to interpret drawn semantic networks. In effect we make guesses about what an interpreter would do. Changing the natural language symbols to gensyms removes the (possibly misleading) clues that guided our guesses.

Continuing this example, Figure 1.18 shows two sentences together with semantic networks that are intended to represent their contents. In this example, three elements take part in the across relation: Willy's *dwelling* is across the *street* from the speaker's *dwelling*. This three-part relation is expressed by the relation node with three binary relation arrows (across-object,

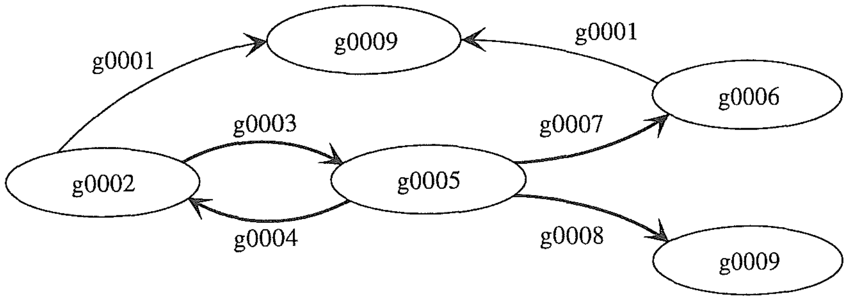


FIGURE 1.17. Semantic network of Figure 1.16 substituting numbered symbols ("gensyms") for names. The difficulty of making sense of this figure shows how easy it is to overlook the knowledge that people can bring to bear in interpreting semantic networks.

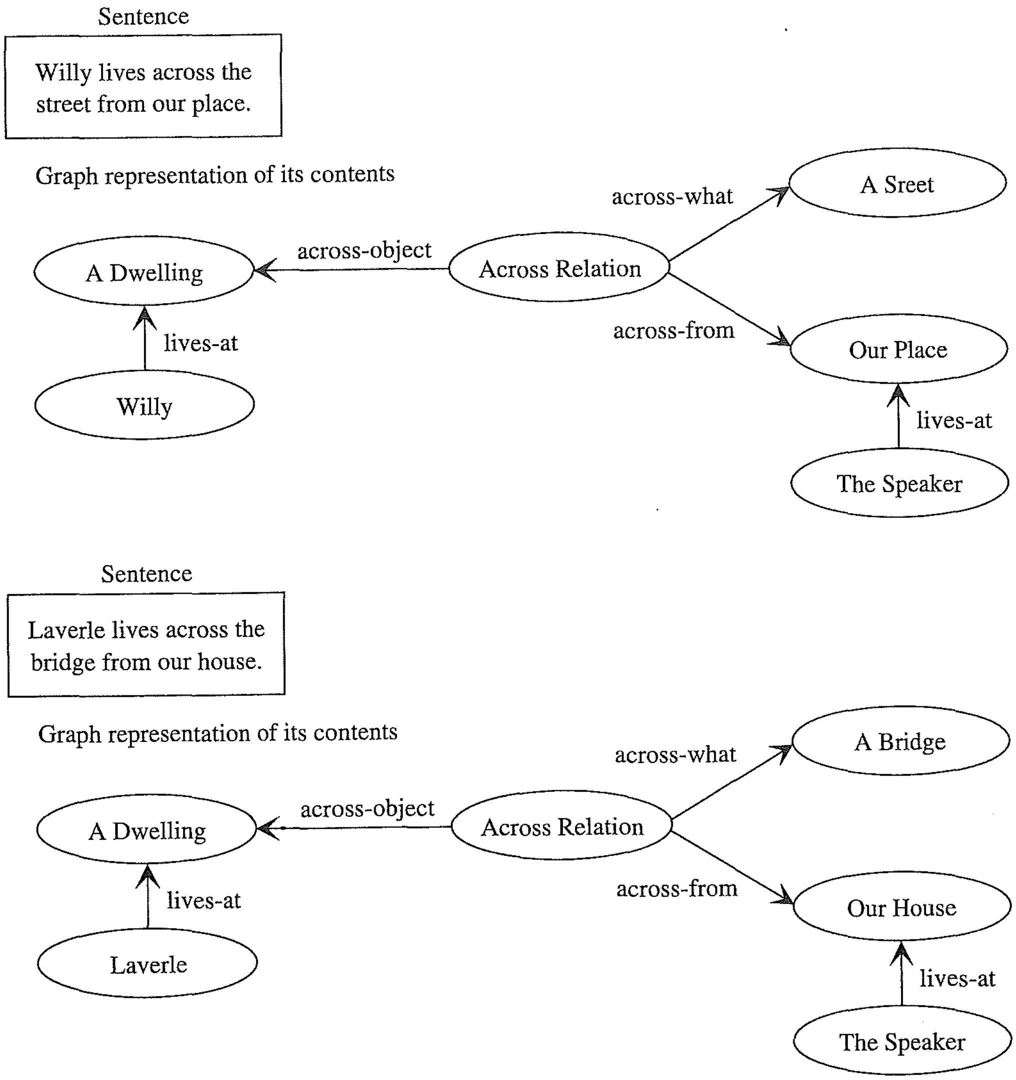


FIGURE 1.18. Sentences with similar meanings should be represented by similar symbol structures.

across-what, across-from) pointing to the other boxes. Such a representation provides specific predetermined places for putting expected kinds of information.

In summary, one of the requirements of having a semantic theory is that we give a detailed account of how meaning is ascribed to representations by their interpreter. For good engineering, the assignment of meaning should be systematic so that representations with similar meanings should have similar structures.

1.2.5 *The Annotation Principle and Metalevel Notations* ADVANCED

In the following, we illustrate a sequence of semantic issues using semantic networks. In this sequence, we retrace some of the history of ideas in the development of representation languages in AI.

We begin by considering a symbol system intended to reason about characters in old cartoons. Tweety the bird and Sylvester the cat are favorite cartoon examples traditionally used at least once in all AI texts. Just for fun we will treat objects from such cartoons as our domain or "world" in the following examples. Presumably the system would need to represent the fact that birds and, cats are animals, that birds have feathers and can fly, that canaries are birds, that Tweety is a neighbor of Sylvester and so on. One attempt to represent these facts in a simple semantic network is shown in Figure 1.19.

In describing such a representation, we must give an account of the processing that the system will carry out on the symbol structures in the course of its reasoning. For example, we expect the symbol system to infer that Tweety can fly. A common idea in such representations is that general information ought to be stored as high in a generalization hierarchy as applicable and inherited by nodes below it by means of a search process. This idea was motivated by psychological studies of human memory response. If more general properties are stored higher up in a generalization hierarchy, one would expect it to take more time for a subject to affirm a statement like "Tweety eats" than one like "Tweety is yellow."

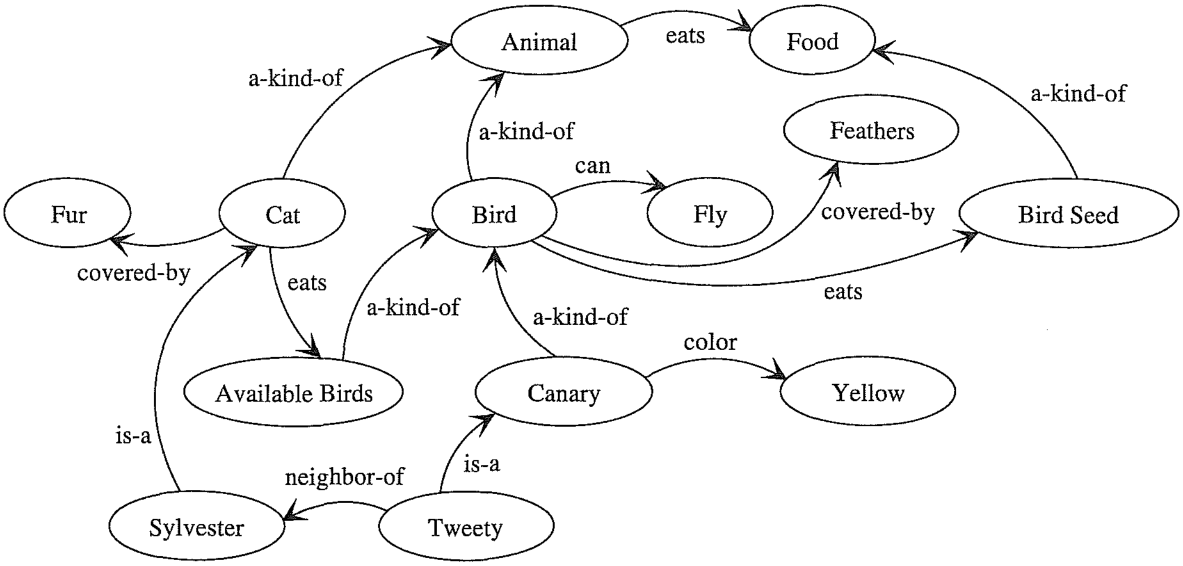


FIGURE 1.19. A simple semantic network to represent facts about Tweety and Sylvester.

One account of the inferential processing in an inheritance model follows:

> The link from the bird node represents that birds can fly. The canary node is linked to the bird node, so that the processor can add that canaries can fly. Continuing this process, the processor infers that Tweety can fly because the Tweety node is linked to the canary node.

This naive argument and organization captures some of the logical structure of the domain. We know that a bird is an animal and the answers to questions about birds will often be derived through general properties of animals. Semantic networks treat these deductions specially, leading to economies of computation for common cases.

However, from the same network we could give an analogous account showing how the processor would infer that Tweety has fur, as follows:

> The link at the cat node represents that cats have fur. The Sylvester node is linked to the cat node, so that the processor can add that Sylvester has fur. Continuing this process, the processor infers that Tweety has fur because the Tweety node is linked to the Sylvester node.

The silliness of the latter account derives from the last step, where the "fur" inference is implausible because Tweety is not a cat. More precisely, the implied processing acts as if we can infer that *y* has a property if *y* is linked to *x* by an arc and *x* has the property. This inference is too broad. The problem is that the network does not explicitly indicate which links convey inheritance of properties and it is clear that not all of them do. Being a neighbor of somebody does not usually imply that one has the properties of that person. Neighbor-of means something different from is-a and a-kind-of, so the processor needs to treat these different relations differently. But they are all represented in the same way in the graph, as directed arcs, albeit with different names. This brings us to the annotation principle.

> **The Annotation Principle.** Differences in intended processing should be reflected by differences in symbol structures. If two symbol structures are intended to be treated in different ways by a processor in a symbol system, the processor must be able to distinguish among some of the properties of the symbol structures. If two symbol structures are intended to be processed in the same way by a processor, then some of their relevant properties should appear the same to the processor.

The annotation principle makes explicit a structural approach to systematizing the composition of meaning. It is simple and perhaps obvious, but it comes up in many different guises in the design of symbol systems. The term *annotation* refers to the use of auxiliary symbols that are used to modify the interpretation of other symbols. These annotation symbols typically do not have the same kinds of meaning as the symbols that they annotate. For example, they usually do not designate objects in the environment.

Annotations are also called **metalevel notations** and **metadescriptions**. They are the basis for many of the representational frameworks and declarative notations used in knowledge representation languages. They are called *metalevel* because they do not refer to the same environ-

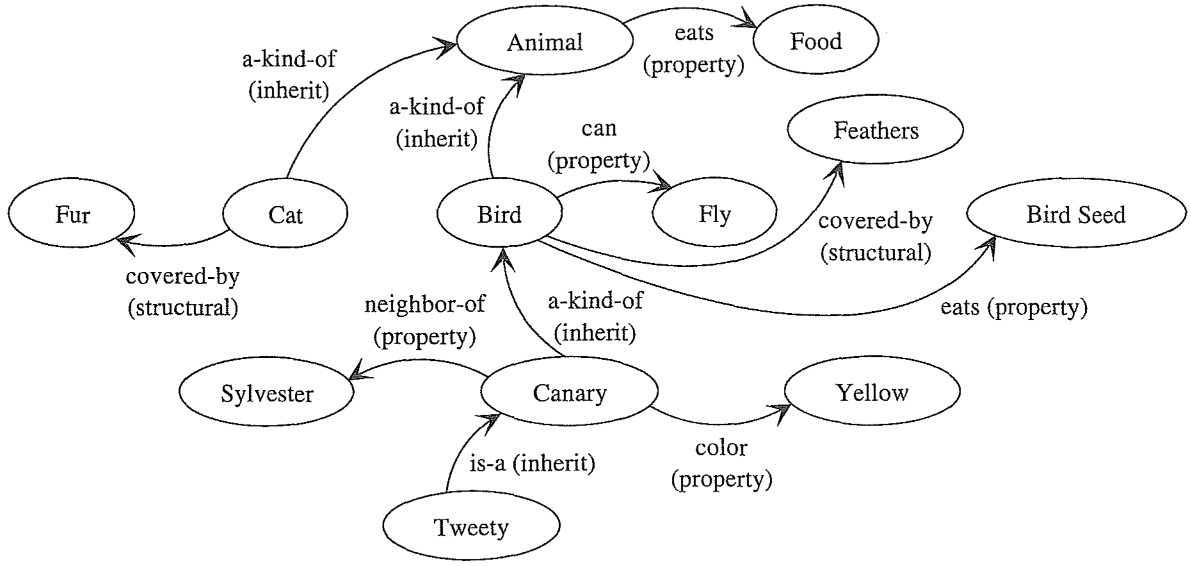


FIGURE 1.20. Augmenting the semantic network of Figure 1.19 with notations on arcs to guide processing.

ment that the cartoon characters in our example inhabit. Rather, they guide a processor in its processing of the symbol structures.

Programming and representation languages sometimes provide special mechanisms for adding annotations to symbol structures. For example, **metaclasses** in object-oriented languages are often used to define how the system instantiates classes or allocates storage. In this same vein, some languages provide ways of embedding structures within notations in a way that is invisible to programs using normal methods for accessing memory. Representation languages provide annotation mechanisms to simplify the design and development of interpreters or symbol structure processors. In this section, we will consider several examples of annotations. Many of these examples are drawn from problematic cases that were noticed by developers of early semantic networks.

In the example of the neighbor-of arc versus the is-a arc in Figure 1.19, we could modify our approach so that that the processor would differentiate among the arcs on the basis of their relation names. This would require that the processor have information for each different relation. An alternative is to augment the network with annotations.

In Figure 1.20, we extend the semantic network with additional annotations indicated by parenthetical labels associated with the arcs. This is a first step toward an engineering practice of developing taxonomic descriptions of relations. Such taxonomies characterize relations. For example, annotations can indicate which relations are used for inheritance, that is, for the propagation of properties from nodes that designate general classes to nodes that designate more specific ones. In Figure 1.20, is-a and a-kind-of are annotated as **inheritance** links. By such propagation the representation can serve to represent that canaries can fly and that Tweety is yellow. The amount of processing required to propagate properties is determined by the branching factor of the network and the depth to which the information must be propagated. To provent looping, inheritance networks are generally required to be acyclic.

Annotations can indicate which relations are **structural**, that is, which relationships describe information about physical structure in terms of subcomponents. In our cartoon example, cats have fur. If more detail were required we could show nested structural relations: Cats have paws, and paws have claws, and so on. As we make a representation more detail, we sometimes find it useful to increase the number of symbols used to represent the elements of a situation. Initially, it might be adequate to treat (say) properties of a cat's claws as properties of the cat. For example, we might just represent the cat's claws as being sharp. Later, however, we might need to represent claws on two different feet. Some claws are dulled and some are sharp. Furthermore, we may find that we want to reason in a similar way about bird claws and cat claws. At some point, however, the overall complexity of the representation may be reduced if we separate the representation of claw properties from cat properties, that is, if we reify the claws as separate objects with their own properties. Then the overall represent becomes a composite consisting of a cats, legs, paws, claws, and any other reified parts that are convenient. This makes it possible to simplify the overall hierarchy of classes into reusable representational elements.

Finally, some links could indicate information about nodes used for **documentation** by a database maintenance routine. Examples of this are the relations creator and date-created, which would connect nodes to representations of the knowledge engineer who created them and the dates that they were created. Such relations are used for purposes of bookkeeping and updating and can be ignored by a processor concerned strictly with the subject matter of cartoons.

These examples show how annotations enable an interpreter to distinguish cases for a small number of different annotations corresponding to **relation types** rather than a large number of different relation names. In particular, it opens up the possibility of defining classes of relations with common behavior. In our first account of processing in Figure 1.19, we assumed that the processor knew the names of the different relations in order to decide how to process them. With annotations, a symbol processor can treat is-a and a-kind-of as identical for the purposes of inheriting properties. Exercise 7 considers an alternative processing architecture for realizing this same effect. Regularities in the ways that symbols are to be interpreted can be exploited in terms of regularities in annotations and in the architecture of their processors.

Annotation in a semantic network can guide not only the processing of arcs but also the processing of nodes. How can the processor determines which objects designate things that can appear in a cartoon? Clearly, Sylvester and Tweety can appear and both are shown in the network as nodes. But canary is also a node. Does it make sense to say that "canary can appear in a cartoon" or "canary can appear in the node animal"? When we say that "Canaries are yellow," we do not refer to any particular canary. We referred to a **class** of small yellow birds. A class does not correspond to bounded physical entity in the world. You cannot "see" a class of birds. A class is a cognitive artifact that we use in organizing our thinking.

Making the meaning of classes such as "canary" philosophically and genetically precise requires more work. Does the class include ancestral birds from the age of the dinosaurs that differ increasingly from today's canaries? The value of precision in sorting out and representing such matters depends on the expected uses of the symbol system.

Figure 1.21 adds node annotations shown as parenthetical labels to indicate whether nodes designate classes or individuals. Even so, this is not yet enough to answer our question about what nodes designate things that can appear in the cartoons. For example, the network shows that cats eat available birds. What is this strange node available-birds? Again, it seems to describe a

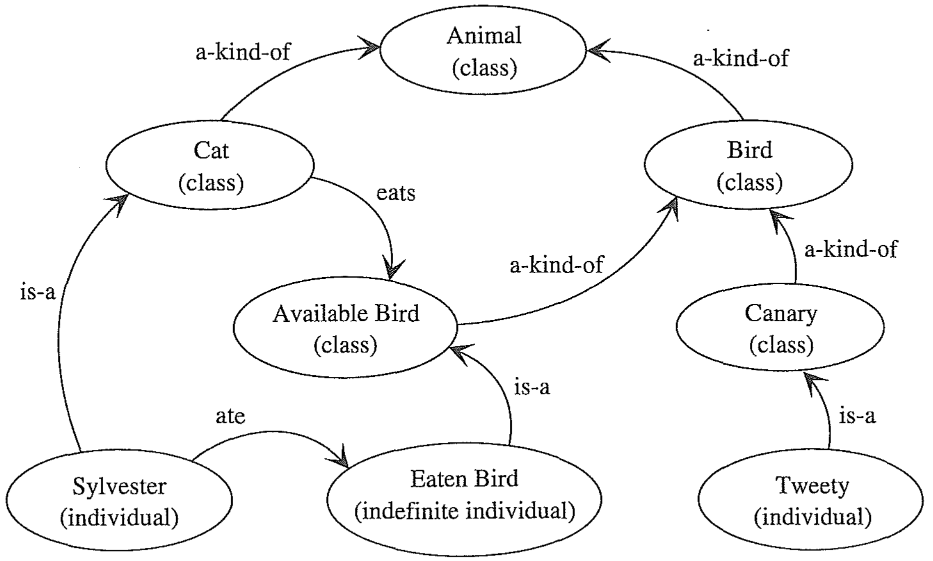


FIGURE 1.21. Augmenting the semantic network of Figure 1.19 with notations on nodes to guide processing.

class. Early developers of semantic networks were quite free in defining such nodes, but lax in characterizing precisely what they meant. The issue here is not that such nodes have no place in a semantic theory. Rather, the issue is that there is a fair amount of work required to be precise about what such nodes are intended to mean, and also work in arranging that the operations of a system are correct relative to the intended meaning of such representations.

We consider one more example. It would be convenient to be able to refer to particular birds that have not yet been identified with known individuals. For example, we might see a grinning Sylvester in the cartoon with yellow feathers in his mouth. We may want to reason about the fate and status of the unfortunate consumed bird, without knowing whether it was Tweety. Mystery stories make much use of this kind of reference. They tell us about a murderer, before telling us whether the foul deed was committed by the pretty maid, the spoiled son, the spinster aunt, the wily lawyer or the sinister butler. Figure 1.19 introduces *indefinite individual* nodes to refer to concepts such as the unknown eaten canary or the unidentified murderer.

What these notations about nodes have in common is that they are all about reasoning about the identity of individuals. We assumed without saying so that distinct nodes refer to distinct things. Class nodes refer to classes of objects. In our simplest examples, we have one node for Sylvester and another node for Tweety. The indefinite nodes change the way an interpreter reasons about identity. If we have a node for "the murderer" and other nodes for the maid, the spoiled son, and so, then later inferences about the possible identity of the murderer may leave us with more than one node referring to the same thing. If the system infers that the butler did it, then we would say that the butler node and the murderer node are **co-referential**.

More generally, two representations are said to be co-referential if they refer to or designate the same thing. For example, the symbols "the first U.S. president" and "George Washington" would usually be co-referential. Technically, the question of whether two representations are co-referential must be determined relative to an observer. Annotations can be used to support reasoning about identity and co-referentiality. Thus, our example of indefinite individual annotations for nodes support such reasoning. In addition, some representation languages include specific co-reference relations intended to indicate where different symbols are known to be co-referential. Examples of such annotations and reasoning with them are given in the exercises.

1.2.6 *Different Kinds of Semantics*

We have now established the background to consider different approaches to semantics. A semantics assigns meanings to symbols. Different kinds of semantics differ in the kinds of symbols considered and the kinds of meanings they assign.

To understand the different approaches it is useful to understand the goals of people who use them. Following Woods' discussion of semantics in Woods, 1975, we compare these approaches using caricatures of different points of view. The first set of caricatures is concerned with assigning meanings to symbols in programs and representation languages. These include the Logician, the Programming Language Designer, the Systems Engineer, and the Representation Language Designer. The second set of caricatures is concerned with assigning meanings to symbols in natural languages. These include the Computational Linguist, the Social Linguist, and the Social Scientist. These caricature names do not fully characterize the different fields. They are intended to exemplify some of the different kinds of issues that are relevant when people create semantics.

Semantics for Programs and Representation Languages

The Logician is concerned with specifying the meaning of a formal notation. To this end she is usually concerned with a formal definition of truth in a set theoretic model, sometimes called a Tarskian semantics. She wants a systematic way to determine when expressions in the notation are "true" propositions, when they are false, and what follows from what. For example, she wants to know how the truth of an expression like "(Socrates is a man) and (Socrates is short)" depends on the truth assignments of the two parenthesized expressions and the conjunction *and*. In the previous section described the semantics of the predicate calculus in terms of a **reference semantics**, a **truth semantics**, and a **proof semantics**.

The predicate calculus by itself does not embody any computational process. Programming languages and practical knowledge representation languages, however, need to specify processes for computing and reasoning. To accommodate this, additional kinds of semantics have been proposed.

The Programming Language Designer is interested in providing a formal specification of computation in programming languages. He wants to be clear about how syntax indicates what computation is to be performed, so that he can build reliable and portable compilers and compiler-compilers. He prefers formal specifications so that the omissions, contradictions, and ambiguities typical of informal language specifications may be avoided. His semantics describe how the output and final states of a program depend on its inputs and input states. There are different approaches to specifying this.

The idea that a proof system can give meaning to a programming language, and hence to programs, is due to Hoare. This enterprise is sometimes called the **axiomatic semantics**, although it is more commonly called the **denotational semantics**. This differs from the usual sense of semantics in logic. Ordinarily, the formal semantics of a proof system is given by relating it to a model theory defined in set-theoretic terms. Here, a proof system is used to specify the semantics of a programming language or a class of its implementations.

The **denotational semantics** involves three interpretation functions. One interpretation function maps programs onto mathematical functions that relate inputs to outputs. A second maps language expressions and program states onto values. A third maps commands in the language onto state transition functions. The details and variations of this approach are intricate and beyond the scope of this section. The main point is that the semantics map syntax onto descriptions of a computation.

The Systems Engineer is also a computer scientist, but she is chiefly concerned with aids to building and maintaining large computer programs. She is concerned with how a large program is built up from many subprograms and hardware subsystems. Big programs are written by groups of people, used by different people in constructing their own subprograms, and modified as needs change. She wants to be able to modify her subprograms without changing the programs that use them. She also wants to be able to use subprograms without knowing the details of how they work. The semantics useful to a Systems Engineer are about the requirements of subprograms. She is interested in external data representations, subprogram parameters, control regimes, and computational resources required by subprograms. Generically, we say that such approaches are concerned with **interface semantics**. In practical examples this involves a mixture of informal and formal descriptions.

The Representation Language Designer is interested in the reasoning phenomena that arise when people gain new information while working on a problem. People make assumptions about defaults and about what events and values are possible and likely. They change their minds in the light of new information, sometimes retracting things they believed earlier. The Representation Language Designer would like to have ways of describing computational methods and goals that could guide such nonmonotonic reasoning in computers. To the extent that they effect the operation of an interpreter, he is also interested in distinctions such as those discussed earlier in this section between class nodes, individual nodes, and so on. He may characterize this as knowledge for control or metalevel reasoning. He does not want to hide this knowledge inside a "black box" interpreter or to intermingle it with the domain knowledge. He wants to enter the statements declaratively and have the interpreter find them and use them when it decides what to do next in the reasoning process. Generically, we call the association of such knowledge with domain symbols a **reasoning control semantics**. This semantics maps statements in the representational language to computational processes for reasoning. The semantics for programming and representing languages are summarized in Table 1.2.

Semantics for Natural Languages

We now turn to semantics for natural languages. The Computational Linguist is concerned with the translation of sentences in natural languages into formal representations of their meanings. She is interested in characterizing how the same sentence can sometimes mean different things and that some sentences mean nothing at all. She would like to find an unambiguous notation in which to express the different things that a sentence can mean. Thus, the Computational Linguist is concerned with the translation of sentences and expressions from natural language into formal notations such as predicate calculus or a well-defined semantic network. We call this approach a **logical language semantics**. Note that this approach maps sentences in one language to sentences in another.

TABLE 1.2. Different semantics for programming and representation languages.

Kind of Semantics	*Used for*	*Symbols*	*Meanings*
Reference semantics	Identifying how symbols in a computer refer to things in an observer's environment.	Symbols and expressions in a physical symbol system or in a representation language.	Designation. A description of things known to an observer.
Truth semantics	Identifying what terms and expressions are true.	Symbols and expressions in logical formulae with a given conceptualization.	True or false.
Proof semantics	Identifying what terms and expressions are valid.	Symbols and expressions in logical formulae with a given database of formula.	Valid or not, as supported by a proof.
Denotational semantics	Characterizing how the syntax in a programming language specifies a computation.	Symbols and expressions in the syntax of a programming language.	A characterization in terms of mathematical functions indicating how final states and output are determined by initial states and input.
Interface semantics	Characterizing the operations and requirements of modules in a computer program.	Program modules.	Abstractions described as protocols, arguments, types, and operations.
Reasoning control semantics	Characterizing how symbols should be treated in a nonmonotonic reasoning process.	Symbols and expressions in a representation language.	Reasoning processes for default reasoning priorities, and so on.

Before passing on, we note in passing that is not generally adequate to consider sentences one at a time when assigning meanings. Consider the following two pairs of sentences.

(1) It is 12:30. Morgan is out to lunch.

(2) His memos never make sense. Morgan is out to lunch.

In this example, the first sentence gives us an important clue about an idiomatic interpretation of the second sentence. Thus sentences do not necessarily provide independent chunks that can be analysed or processed independently to determine their meaning. Natural language requires a context to determine meaning. This is in striking contrast to the truth and proof semantics of logic, and contrary to the suggestion that the meaning of an expression in a representation language be determined by the meaning of its parts.

Returning to our tour of kinds of semantics, the Social Linguist recognizes that many sentences uttered in conversation are chosen for their effect on the listener rather than to communicate statements about an external situation. For example, the English statement, "The car is almost out of gas" may be uttered as an indirect request to a driver to stop at the next gas station. Even statements that seem to contain logical operators often have unusual meanings. Only a perverse sense of humor would allow one to answer "yes" to English question "Are you left-handed or right-handed?" This is a request for information. One of the expected responses is "right-handed." The Social Linguist refers to the semantics studied by the Computational Linguist as merely the "literal meaning." He classifies statements as kinds of "speech acts," whose purpose in conversation is to indicate agreements, disagreements, commitments, priorities, goals, understandings, and other aspects of the communication and negotation process involving the agents. We call this approach **action semantics** because the meanings of the sentences are actions intended to have certain effects.

The Social Scientist is also concerned with the use of symbols in human interactions. She is interested in how it is that people come to agree about the meaning of terms that they use together in speaking and writing. She recognizes that people do not immediately understand each other's terms and that they develop models of each other and their use of language. She focuses on phenomena related to the change and elaboration of meaning. She characterizes the meaning of terms as being socially constructed and negotiated, as people sharpen or broaden what they mean by words. The point here is not the need for another kind of semantics, but rather, a need to focus on different properties of meaning. In our previous discussions of meanings, we acted as though meanings were fixed and unchanging. In contrast, the Social Scientist studies the evolution and convergence of meaning.

Before leaving these examples of kinds of semantics, we note that the list is not exhaustive and that there are many further variations. In the 1890s, the philosopher Frege, who invented the declarative semantics of logic, was also concerned with the relation between equality and the designations of terms in natural language. He proposed two natural language sentences, which have often been cited as perverse examples.

(1) Necessarily, the Morning Star is the Morning Star.
(2) Necessarily, the Morning Star is the Evening Star.

In these sentences, the term *Morning Star* refers to a bright star observed near the sun at sunrise. The term *Evening Star* refers to a bright star observed near the sun at sunset. From an astronomical point of view, the two terms refer to the same physical object, usually the planet Venus. From Frege's point of view, however, the first sentence is true and the second false. The two terms do not have the same "meaning."

People still argue about exactly what Frege meant with this example. Some people relate Frege's idea to the idea of transformational grammars. These grammars transform sentences to either a standard form or a logical statement. This approach is similar to the logical sematics described earlier. In this account, the terms *evening star* and *morning star* have different meanings because there are no rules for transforming them to the same form. The reason that the two terms are not made equivalent is that the transformational rules involve knowledge about variations in syntax but presumably not knowledge of astronomy.

A different explanation of Frege's point is based on consideration of the term *intensional* which means "of the senses." In this view, the semantics of *evening star* and *morning star* refer to the process by which they are perceived. They are perceived differently because one is seen in the morning and the other in the evening. This approach can be seen as a more sophisticated view of a reference semantics. It focuses on the operations, processing, and interpretation of the senses. We cannot simply refer to "the world" as though it were something that we necessarily all

TABLE 1.3. Kinds of semantics for natural languages.

Kind of Semantics	*Used for*	*Symbols*	*Meanings*
Logical language semantics	Characterizing what sentences in natural language mean.	Example sentences and expressions from natural language.	Sentences in a formal notation, such as predicate calculus.
Action semantics	Characterizing how sentences in natural language are used to cause action.	Example sentences and expressions from natural language.	The goals, agreements, and commitments of agents in a conversation.
Intensional semantics	Characterizing how terms in natural language refer to things that are perceived.	Terms in natural language used in writing and speech.	The operation, processes, and interpretations of the senses.

see the same way. We must say more about how we go about perceiving and understanding it. We call this approach **intensional semantics**. Table 1.3 summarizes approaches for assigning meaning to sentences in natural language.

Other philosophers take intensions to correspond to concepts, ideas, or things that can be imagined. Further discussion of related topics such as the semantics of necessity and intensional logic would take us too far afield.

1.2.7 *Summary and Review*

A semantics is a way of assigning meanings to symbols. There is no single "true" meaning or true way of assigning meanings, at least in the academic fields. Different fields have different traditions for assigning meanings. We considered examples of semantics for programming and representation languages and also for natural languages.

Looking back over the different examples of semantic theories, we can group them roughly into three families. First is the **referential family** of semantic theories. In this family, meaningfulness comes in the relations of symbols to objects of different kinds. Members of this family include the reference semantics, the intensional semantics, and perhaps the denotational semantics of programming languages where the "objects" are mathematical.

Next is the **cognitive family** of semantic theories. Meaningfulness arises from the systematic ways that subject matter is mentally and computationally represented and how reasoning processes are sanctioned over those representations. This family includes the truth semantics, proof semantics, operational semantics, reasoning control semantics, and interface semantics.

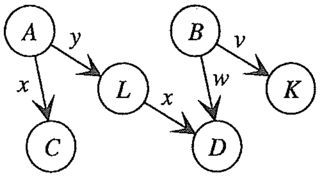
Third is the **social family** of semantic theories. This family emphasizes communication. In this approach, meaningfulness derives from the ways that agents use symbols in their interactions with each other. This family includes the action semantics.

These approaches to semantics are complementary. In discussing knowledge systems we draw on different approaches for different purposes. When we think of knowledge systems as embedded systems whose symbols refer to the world we draw on the referential family. When we argue about the conclusions that knowledge systems should reach and their computational characteristics we draw on the cognitive family of semantic theories. Finally, when we interact with people to incrementally define symbols to be used in a knowledge system that they will use and when we consider the interactions of knowledge systems in a human organization, we draw on the social family of semantic theories.

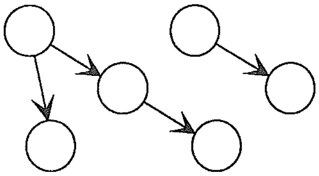
Exercises for Section 1.2

Ex. 1 [05] *Identifying Graphs and Trees.* Classify each of the following graphs as (1) directed or undirected, (2) labeled or unlabeled, (3) cyclic or acyclic, and (4) graphs, trees, or forests.

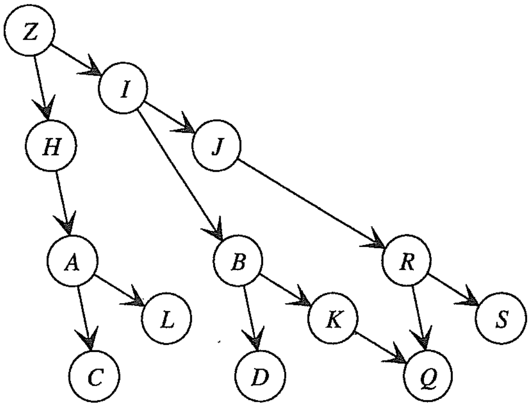
(a)



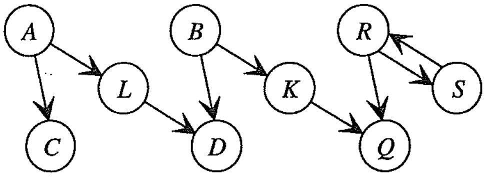
(b)



(c)



(d)



Ex. 2 [*05*] *Terminology*. For each of the following statements indicate whether it is true or false. If the statement is ambiguous, explain briefly.

(a) *True or False*. The main thing that different so-called semantic networks have in common is that they are composed of nodes and links (possibly labeled). This is the same as a definition of a kind of graph and there is nothing inherently semantic about graphs.

(b) *True or False*. A reductionist and structural approach to the meaning of a representation says that the meaning of an expression is determined by the meaning of its terms and by the pattern of their arrangement. Restated, the meaning of an expression composed of terms is a composition of their meanings.

(c) *True or False*. The annotation principle says that formal representations should be decorated with informal annotations that help people to understand them.

(d) *True or False*. Two nodes in a graph representation are defined to be coreferential if they both have arcs that point directly to a common node.

(e) *True or False*. The declarative semantics of predicate calculus consists of a reference semantics, a truth semantics, and a proof semantics.

Ex. 3 [*05*] *Terminology*. For each of the following statements indicate whether it is true or false. If the statement is ambiguous, explain briefly.

(a) *True or False*. A cyclic graph is one in which there are two directed arcs leading to the same node.

(b) *True or False*. In reasoning by inheritance, properties are propagated from general nodes to more specific ones.

(c) *True or False*. In drawings of is-a hierarchies, the arcs point in opposite the conventional direction for trees. That is, they point from specialized nodes to generalized nodes.

(d) *True or False*. The annotation principle is intended to make the design of interpreters simpler for systematic representation languages.

(e) *True or False*. The usual definition of a tree in computer science is equivalent to a dag.

■ **Ex. 4** [CD-*!-10*] *Reasoning about Identity.* Indefinite-individual nodes are used to refer to individuals whose identity has not yet been established. They provide a notation to express incremental reasoning about identity. One proposed framework defines two kinds of relations for reasoning about such nodes: **anchor** relations, which link an indefinite-individual node to an individual node, and **co-reference** relations, which link two indefinite nodes to each other. In the usual interpretation, different individual nodes designate different objects in the environment, anchor relations mean that an indefinite designates the same object in the environment as the individual node it is linked to, and co-reference relations mean that two indefinite nodes designate the same object in the environment even if it has not been identified yet.

(a) Consider a semantic network with individual nodes Huey, Duey, and Louie representing duck characters in a cartoon story and indefinite nodes for dessert-eater and brown-coat-wearer. Draw a semantic network to indicate that the dessert-eater and the brown-coat-wearer are known to be the same individual.

(b) Describe appropriate processing on this representation that would infer who ate the dessert, given that the brown coat was worn by Huey.

(c) Suppose that in a different situation, the processor had anchored dessert-eater to Huey and brown-coat-wearer to Louie. In what sense would it then become semantically inappropriate for the processor to put a co-reference link between the two indefinite-individual nodes?

■ **Ex. 5** [CD-*!-15*] *Closed World and Other Assumptions.* In murder mystery stories, the closed (or locked) room scenario is one in which all of the possible suspects are locked together in the same room (or train), so that no one could enter or leave during the crucial period when the foul deed was done. Furthermore, in detective stories, there is a kind of "fair play" assumption amounting to a contract between the author and a reader, which says that it must be possible for the reader to solve the mystery from the evidence given. Introducing a revenge-seeking cousin in the last scene or aliens with special powers from outer space is not considered fair play in the genre.

Similarly, in AI systems the **closed-world assumption** means that all of the objects of interest are described in the database. Usually many assumptions about interpretations influence modeling and representation.

In this exercise we consider an adventure of the three cartoon character ducks Huey, Duey, and Louie, nephews of Donald Duck. The three ducks were locked alone in a room with the dessert, so that none of them could escape and no one else could enter. Suppose also that the dessert could only be eaten by a duck, and that the dessert disappeared while they were in the room.

(a) How could a knowledge system infer who ate the dessert, given that neither Duey nor Louie wore the brown coat and that the brown-coat-wearer ate the dessert? Show how this requires the use of a closed-world assumption?

(b) Professor Digit says, "Although we can use quite simple reasoning models to infer the answer to problems like this, real world (and cartoon world) possibilities are quite endless. How do we know that the dessert did not simply evaporate? There are many different assumptions about the world that a system could make in solving problems like this." Do you agree with Digit? If yes, give some examples.

(c) In everyday reasoning, we are able to imagine a wide range of possibilities, and yet we are not overwhelmed by them on simple problems. Briefly, describe an approach for knowledge systems that provides this capability.

■ **Ex. 6** [*15*] *Representing Physical Parts.* Representations are designed for a purpose and the adequacy of a representation is judged relative to that purpose. In this exercise, we consider representations of physical parts. We are given the following statements:

A Buick is a kind of automobile.
All automobiles can be driven.
A Skybird is a kind of Buick.
The body of a Skybird is a sport body with an asymmetric shape.
The particular Skybird with serial number 100 has a red body.
A Skybird body has a left door and a right door.
The right door of a Skybird body is a passenger door.
The left door of a Skybird body is a driver door.
A driver door is shaped as trapezoid pattern 1.
A passenger door is shaped as trapezoid pattern 2.
A car door has a handle.
The engine of a Skybird is a model 600.
A model 600 engine has four cylinders.

(a) Assuming that you will use a graph representation, discuss how you distinguish between the following:

Representations of part relations versus representations of other relations
Representations of classes versus representations of particular objects

(b) Suppose that our task is to compute the list of parts of an object, from the largest part down to the smallest one.

In a rigorous discussion of what it means to be a part, we would need to define more what it means to be a part. For example, coatings such as paint are not considered to be parts. Substances used to make up materials such as metal alloys are also not considered to be parts. There is also a practical issue of granularity. Are we really interested in listing the lock washer on the bolt that holds the hand to the door shaft? Assume for this problem that the granularity of interest corresponds to the granularity in the network.

Show a systematic graph representation for the statements above.

(c) Describe your method for computing the list of the parts of a Skybird, using your graph representation as the database.

(d) Describe your approach for inheriting part descriptions. Specifically, explain how your approach includes door handles for both the passenger and driver door in the inventory.

Ex. 7 [*!-40*] *Partitioned Semantic Networks.* In 1975, Gary Hendrix observed that semantic networks were clumsy when compared with predicate calculus for representing quantified statements. He proposed a mechanism (Hendrix, 1975) for partitioning semantic networks into "spaces" that contained nodes and links and that were convenient for indicating the scope of quantified relations. Spaces are well suited for this purpose in that the node and arc variables encoding information within a space look exactly like the representations of specific facts. Furthermore, such variables are effectively isolated from constant information by partition boundaries.

Although interest in using semantic networks for representing statements in predicate calculus has been rather limited, notations for partitioning networks have provoked more interest as general representational mechanisms. Partitions are used for delineating clusters and defining boundaries for information hiding. For example they have been used to represent contexts, alternative worlds, and plots.

(*a*) Figure 1.22 represents the statement "Every dog has scratched a flea." The partitions in this figure are SA, the outermost space, and S1, the space that indicates the form over which the variable *d* is scoped. Arcs labeled with an *e* indicate element-of relations for the sets to which they point. Similarly, arcs labeled with an *s* indicate subset relations. The presence of each node and arc within a space are interpreted as implicit statements of exis-

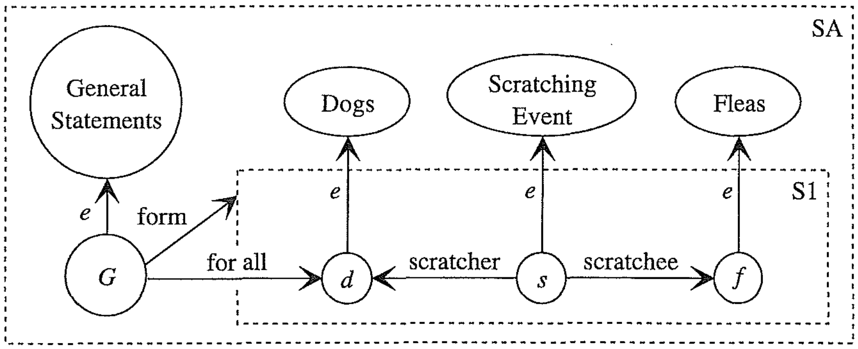


FIGURE 1.22. Roughly, "Every dog has scratched a flea."

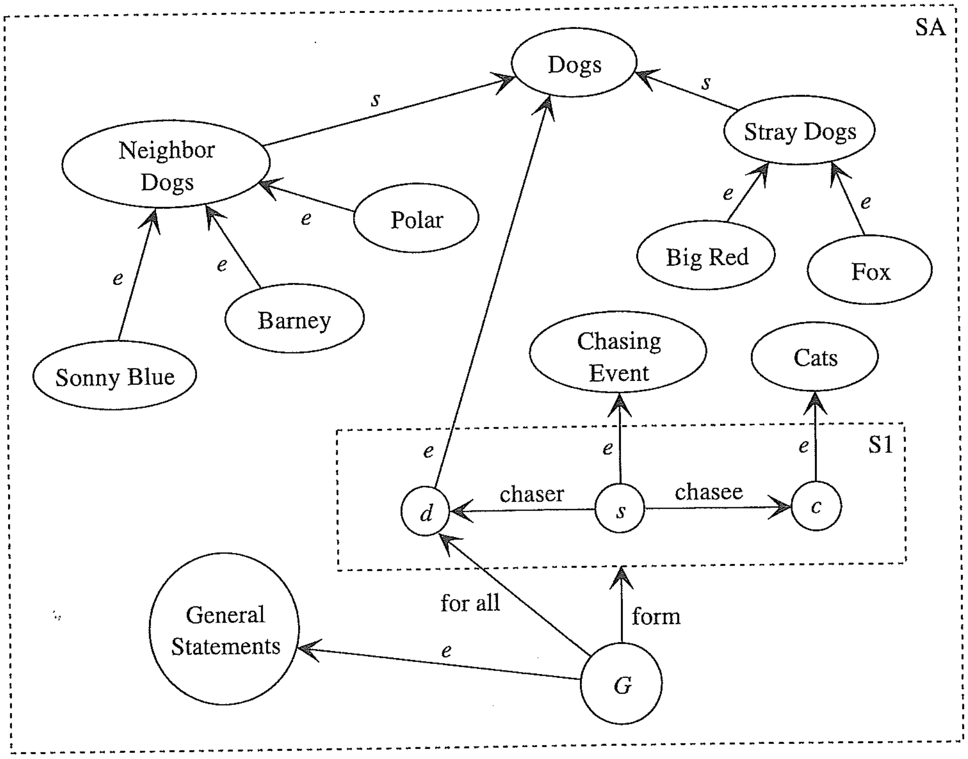


FIGURE 1.23. The "invisibility" of the contents of spaces.

tence about objects and their relations. Thus, the nodes in the figure assert that the following:

There are dogs.
There are fleas.
There are scratching events.
Every dog, *d,* has participated in a scratching event, *s,* in which it was the scratcher and some flea, *f,* was the scratchee.

In the same network, there are some metalevel statements, the details of which we will be a bit vague about.

G is a general statement.
It has a form (the space S1) and a universally quantified variable within that form (*d*).

Use this same graph and boundary notation to express the statement: "Every dog has visited every fire hydrant." Assume that both quantifications are part of the same general statement.

(b) In Hendrix's scheme, partitions determine what nodes are *visible* to a routine search. We do not detail Hendrix's particular approach here. In this exercise, we assume that if a space is contained in another space, the contained space can "see" everything in the containing space, but the contents of the contained space are invisible to the containing space. We assume further that the spaces are organized in a strict containment or subset hierarchy.

For example, in Figure 1.23, a search for instances of dog would find Polar and Barney and Sonny Blue, but not *d.* What other mechanisms discussed in this section could serve to distinguish variables from constants?

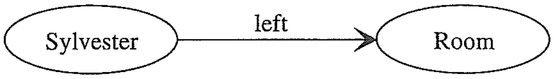


FIGURE 1.24. Semantic network representing "Sylvester left the room."

(c) Two philosophers, Dr. A and Dr. B, have gotten together to discuss partitioned networks as presented in this exercise. After several minutes of deep thought, Dr. A exclaimed, "Eureka! Partitioned nets are not so strange. The partitions function just like parentheses in predicate calculus notation." Dr. B thought about this, and countered, "No. They are just notations for grouping network elements into sets. They could be used for other purposes, too."

What feature of partitions enables them them to do more than delimit sets?

(d) Explain how partitions are more flexible for representing sets than parentheses in a linear notation, assuming the usual rules for well-formed expressions?

Ex. 8 [*!-20*] *Modeling Beliefs of Multiple Agents.* Sometimes it is useful for representation languages to indicate when to "escape" to alternative interpretations. We consider this starting with an example, drawn from the cartoons. Tweety is a bird, Sylvester is a cat, and they play tricks on each other.

(a) Consider the sentence:

"Sylvester left the room."

Using the notation from this section, we could represent this in a semantic network as in Figure 1.24.

Consider now the revised sentences:

"Sylvester was hiding in the room."
"He knew that Tweety believed he had left the room."

What is at issue in semantics if we use the same representation as before to indicate what Tweety believed?

(b) Professor Digit says he can always distinguish beliefs by inventing new relations. For example, to represent the example from part (a) he proposes relations like knows, believes, and believes-that-another-agent-believes. Why is this approach problematic?

(c) Propose an approach that would distinguish what Tweety believes from what Sylvester believes. Describe some of the requirements for interpreting symbols in your approach. Will your approach extend to cover the situation where there is deliberate deception because Tweety really saw Sylvester hiding and acted as if he did not see him in order to fool him. (Hint: See preceding exercise.)

Ex. 9 [*10*] *Kinds of Semantics.* The following cases refer to the kinds of semantics described in this section.

(a) What is the difference in concerns between a naive reference semantics and an intensional semantics? Why might a social scientist find a reference semantics naive?

(b) Programming formalists who want guarantees of the correctness of systems advocate the decoration of programs with annotations describing expectations and invariants. What approach to semantics would they use?

(c) What kind of semantics is called the "literal meaning" of natural-language sentences?

Alternative #1—Chinese Restaurant

Pro Arguments
Chinese food is healthy.
A variety of food is available.

Con Arguments
Chinese lunch is expensive.

Alternative #2—Hotdog Stand

Pro Arguments
Hotdogs are prepared quickly.
Hotdogs are inexpensive.

Con Arguments
Hotdogs contain a lot of fat.
No place to sit down.

Alternative #3—Ice Cream Parlor

Pro Arguments
Ice cream is available quickly.

Con Arguments
Ice cream, by itself, is not satisfying.
Ice cream, by itself, is not healthy.

FIGURE 1.25. Some arguments for lunch alternatives.

(d) What kind of semantics is most relevant to computer-aided software engineering (CASE) tools, that is, tools for coordinating the programming activities of a programming team?

(e) What kind of semantics is relevant for describing a reasoning process based on assumptions and defaults with different probabilities ?

Ex. 10 [*CD-!-30*] *Argumentation versus Proof.* It is often observed that arguments are not always won by "logic." This exercise considers ways that argumentation may be characterized in a way that is meaningful and computational, but which requires elements beyond those of the usual semantics of logic.

In this exercise, we consider three people who are discussing where to have lunch. We will call the lunchers A, B, and C. They have identified three possible restaurants: a Chinese restaurant, a hot dog stand, and an ice cream parlor.

(a) The three lunchers agree to write out arguments for and against the different alternatives. They come up with a chart like that in Figure 1.25.

One of the three lunchers, who is studying logic, observed that the process they were going through was quite different from the process of proving a theorem. It was not much like writing down a theorem "Chinese food is the best lunch" and then trying to prove it. Do you agree? If yes, explain the essential ways in which the process differs from proof.

(b) Once the arguments were written down, the lunchers then decided that some of the arguments depended on various assumptions. They wrote down a list of dependencies including those shown in parentheses in Figure 1.26.

Alternative #1—Chinese Restaurant	*Assumptions*
Pro Arguments	
Chinese food is healthy.	(Depends on assumption that the vegetables are fresh and not overcooked.)
A variety of food is available.	
Con Arguments	
Chinese lunch is expensive.	(Depends on assumption that they order separately. It is cheaper if they share a couple of main dishes.)
Alternative #2—Hotdog Stand	
Pro Arguments	
Hotdogs are prepared quickly.	(Depends on the assumption that they get there before the usual large lunch crowd.)
Hotdogs are inexpensive.	(Depends on whether the low-cost vendor is there today.)
Con Arguments	
Hotdogs contain a lot of fat.	
No place to sit down.	(Depends on the assumption that they are unwilling to sit at the fountain on the patio.)
Alternative #3—Ice Cream Parlor	
Pro Arguments	
Ice cream is available quickly.	(Depends on the assumption that they get there before the large lunch crowd.)
Con Arguments	
Ice cream, by itself, is not healthy.	

FIGURE 1.26. Some arguments for lunch alternatives, showing supporting assumptions.

Reflecting on this process, one of the lunchers noted that the process was taking on some of the elements of deduction. Do you agree? Explain briefly.

(c) As the lunchers pondered the assumptions, they noticed that they didn't believe all of them.

Luncher A believes all of the assumptions except three. He does not believe that the low cost hot dog vendor will be there today. He does not believe that sharing main dishes in the Chinese restaurant will reduce costs. He also does not believe that they can get to the ice cream parlor before the lunch crowd does.

Lunchers B and C also have different beliefs in the truth of the assumptions.

At this point, one of the lunchers notices that their process has a kind of "truth theory," but that it is different from that of logic because it admits assumptions. Briefly, explain the significance of this difference.

(d) After the different beliefs in the assumptions were tallied, the lunchers noticed that there was still more they wanted to discuss before drawing a conclusion. For example, luncher A noticed that he believed the pro argument "Chinese food is healthy" and also the con argument "Chinese lunch is expensive." However, he cared about the former argument more than the latter because the amount of money he spends on lunch has never been significant to him. Similarly, the other lunchers evaluate the arguments in their own ways.

How does this part of their decision process require concepts beyond the declarative semantics of logic? Explain briefly.

Note: This exercise is based on the idea of an **argumentation spreadsheet**, as described in Stefik et al, 1987, which was part of the Colab project at Xerox Palo Alto Research Center (PARC).

Ex. 11 [!-L-*15*] *Graphs and Logic as Representations.* In many graph representations, an account of the meaning includes an account of processing by the physical symbol system. For example, a process might specify how information associated with one node can be propagated to other nodes to which it is linked in particular ways.

(a) Create a graph representation (semantic network) for the following three sentences. Referring to the elements of your graph representation, describe how you would infer that "Bill can fly":

All birds can fly.
A pelican is a kind of bird.
Bill is a pelican.

(b) Instead of using the semantic network representation, translate these statements into formulas of the predicate calculus. Give a short proof that "Bill can fly." Indicate the rules of inference that (such as modus ponens and universal instantiation) you use in the proof steps.

■ **Ex. 12** [*CD-05*] *The Changeable Mind of the Observer.* Professor Digit is disturbed by the idea that the meaning of symbols is in the mind of an observer.

For example, suppose in a medical domain about infectious diseases that some of the microorganisms responsible for the disease "sniffleitus" develop a resistance to a particular antiobiotic treatment. In that case, doctors and other medical practitioners would extend the symbol sniffleitus to include the new dominant variant of the disease and would also prescribe a different treatment for it.

Professor Digit is concerned about the philosophical consequences of this, especially in the case that the knowledge system itself is not modified. He asks: "When people change their use of some symbols used in a knowledge system, does that change what the program means?"

(a) Briefly, show how the answer to the professor's question depends on our choice of semantics. Compare the answers for reference and denotational semantics.

(b) Suppose that the knowledge engineer now updates the program so that it prescribes a different treatment for sniffleitus, respecting its acquired resistance. Again, does this change the meaning of the program for denotational and reference semantics?

(This exercise was inspired by an example from William Clancey.)

1.3 *Modeling: Dimensions of Representation*

Like many computer programs, a knowledge system is a computational model of something, where the "something" is the domain or the situation. For example, a knowledge system for diagnosis and repair of radios would typically include a model of the physical parts of a radio, a model of the functions of the parts and of their electrical operation, a model of how the components can fail, a model of the diagnosis task, and a model for the repair task. For each model, representations are required and the representations must satisfy particular properties in order to be adequate.

The design of representations is a central concern in building computational models, involving many considerations and sometimes several changes until appropriate designs are found. In talking about the suitability of representations, we are concerned not only with the properties of the symbols themselves, but rather with their properties as *representations*. In the previous section, we discussed representational properties involving reference semantics, truth semantics, and proof semantics. These properties are concerned mainly with identity and inference.

In this section we broaden our discussion of representational properties, including many that bear on the computational use of representations. We discuss fidelity and precision, abstractions and implementations, primitive and derived propositions, explicit and implicit representations, canonical forms, use of multiple representations, space, time and structural complexity, and the broad implications of parallel processing for efficient manipulation of symbol structures. These dimensions arise in the design of all kinds of computational models.

The concepts introduced in this section are typical of an engineering approach to system design. For example, we ask about the adequacy of representations for making particular distinctions. We ask about the efficiency of particular operations on symbol structures. We ask how that efficiency differs according to different assumptions about the nature of the computational processing elements.

If you have had programming experience, many of the representational properties discussed in this section will be familiar. A programmer must design representations, making sure sure that they cover the cases of interest, that all of the important distinctions can be represented, and that computations run in reasonable time and space. These practical concerns are inherent in building knowledge systems and are encountered by everyone who builds computational models.

1.3.1 *Fidelity and Precision*

Fidelity refers to the correctness of statements or predictions about the world. When applied to representations, it refers to the correctness of the meaning that we ascribe to them. Another term for fidelity is accuracy. **Precision** refers to the degree of detail in predictions about the world. Similarly, the precision of representations refers to the degree of detail in the meanings that we ascribe to them. For example, a computation that is given to ten decimal digits is more precise than one that is given only to six digits. In a circuit diagnosis system, a representation that predicts that voltage will rise to "eight volts plus or minus one" is more precise than one that simply predicts that voltage will rise.

Representations can have fidelity with little precision or have great precision without fidelity. For example, the statement "The sun will rise tomorrow morning and set in the evening" has fidelity, but little precision. It does not give the precise time of the sunrise or say anything about its path across the sky. In contrast, the statement that there are 1,456,853,123 molecules of air in a tiny bottle in my study has great precision but is not accurate.

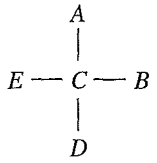
Different tasks have different requirements for accuracy and precision. Consider in Figure 1.27 two representations of a chemical structure for different tasks. The first representation presents molecules as topological structures, characterized by nodes (atoms) and arcs (bonds) be-

Topological representation:

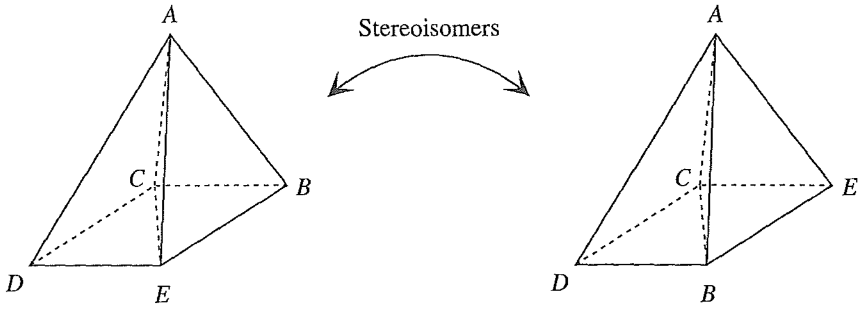
Logical clauses

(Connected *C A*)
(Connected *C B*)
(Connected *C D*)
(Connected *C E*)

Illustration



Three-dimensional representation:



Clause for left molecule

Clauses for both
(Center *C*)
(Top *A*)

Clause for right molecule

. . . Topological clauses from above

(Clockwise-Order *B E D*)

(Clockwise-Order *E B D*)

FIGURE 1.27. Stereoisomers are mirror-image molecules. They have the same atoms and the same corresponding chemical bonds, but they are distinguished by their shape.

tween them. A topological representation describes which atoms are connected. It is adequate for determining the chemical composition of a molecule in terms of its individual atoms.

However, a representation of a molecule limited to topology is inadequate for reasoning about molecular interactions. Molecules can have the same topological properties and yet differ in their three-dimensional structures. For example, organic enzymes must fit against other molecules and their interactions depend on the particulars of their three-dimensional structures. When two molecules are mirror images of each other, they are called **stereoisomers**. Stereoisomers have the same topology but act differently in some chemical reactions.

Precision and fidelity are not usually independent considerations. Representations that are correct but imprecise often lead ultimately to incorrect predictions. Continuous systems are ones in which a small change in the initial conditions leads to a small change in the final conditions. In continuous systems, small compromises in precision typically cause few problems in fidelity, if the results are not extrapolated too far. Chaotic systems, however, are ones in which arbitrarily small changes in the initial conditions can lead to large changes in the results. Even with great computational precision, there is little confidence in the fidelity of predictions for such systems.

Fidelity and precision reflect a tension between our representational goals and limitations when we use computational models. There are usually trade-offs in representations involving efficiency, fidelity and precision. Typically, an increase in precision requires a decrease in efficiency.

Practical knowledge systems employ multiple representations with different trade-offs in precision. In a diagnostic reasoning system, representations with low precision may be adequate for making coarse predictions about circuit behavior. For the purpose of isolating and identifying a faulty component, this may be enough to rule out major blocks of the circuit. More precise representations may then be called on to make a more detailed analysis on selected parts of the circuit. For this reason, the design of a computational model sometimes involves a suite of complementary representations with degrees of precision suitable for different purposes.

1.3.2 *Abstractions and Implementations*

Abstractions are high-level descriptions that say what a representation must do but not precisely how it must do it. Abstractions are essential in large programming projects. Abstractions are sometimes explicit as the "exported interface" of a program. When one subprogram refers to another and depends only on its exported interface, then implementations of the called program can be changed without changing the calling program. This facilitates the use of alternative implementations. **Implementations** are specific data structures and associated methods for storing information and carrying out operations relative to a given computational interpreter.

Unfortunately, the word *abstract* is misleading. All representations are abstract in that they leave out some features. In the terms of the previous section, all practical representations lack something in their precision, so they are necessarily "abstract."

The terms *abstraction* and *implementation* are better understood as describing relations between representations. When the relation between two representations is akin to the relation between high-level specifications and low-level and detailed descriptions, then we call the former abstractions and the latter implementations. This distinction is common in programming methodology and software engineering. Figure 1.28 suggests how an object-oriented language could specify abstractions in terms of "message protocols" that an implementation must include. In this context, an implementation is a description of how it carries out the required protocol. In idealized programming practice, we first build specify abstract models and then develop implementations. Typically, however, these processes are intertwined.

We define abstractions in terms of a set of features with meaning, a set of operations that change the features or return information about them, and a set of invariants on the features that must be preserved by the operations. This style follows object-oriented approaches to programming. For example, we could define an abstract representation of a figure as an entity with a shape, an origin and dimensions, certain computable properties such as area that are related to its dimensions, and operations for changing its dimensions, moving it about, and determining whether it covers arbitrary points in the plane. In this example, there are several invariants. The dimensions and shape of a figure do not change when it is moved. However the origin of a figure is not an invariant; it changes when the figure is moved. A complete characterization of the abstractions should characterize the required precision of the representation as well.

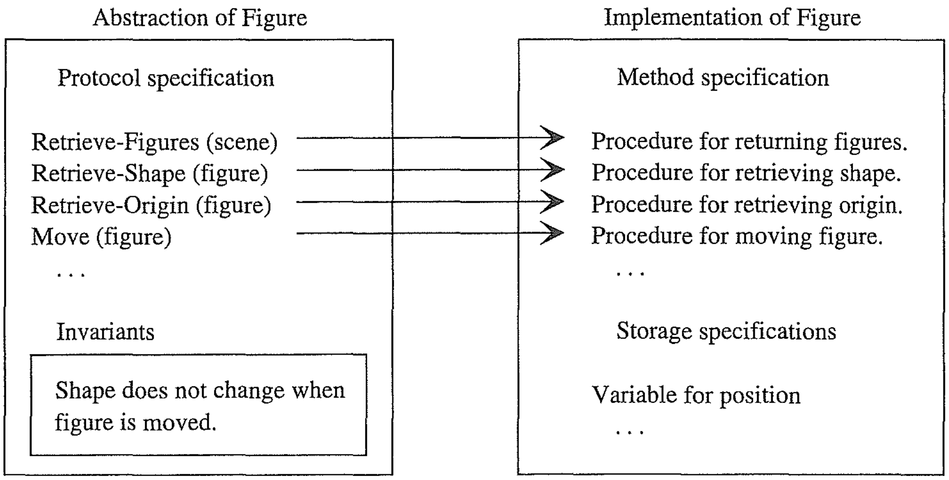


FIGURE 1.28. Abstractions and implementations.

To make the abstraction/implementation distinction concrete we define an abstraction for figures and then compare three implementations of it. Our abstraction of "figure" requires the following specific operations:

- Get-Shape (figure): Retrieve a description of the shape of the figure, one of square, circle, triangle, and spiral.
- Get-Origin (figure): Retrieve the origin of the figure.
- Get-Area (figure): Retrieve the area of the figure.
- Move (figure): Move the figure to a new location.
- Overlap (figure 1, figure 2): Determine whether two figures overlap.
- Get-Overlapping-Figures (figure-set): Find all of the figures in a set that overlap a given figure.

We now consider three alternative data structures for representing a scene with figures. We think of these data structures as alternative implementations. We illustrate these data structures with example representations of a square.

A Binary Array Representation for Figures

Figure 1.29 shows a two-dimensional binary array (bitmap) representation of the square. Each position in the bitmap corresponds to a square picture element or pixel in the scene. A 0 in the bitmap indicates that the position is unoccupied, and a 1 indicates the presence of some object at the position. Movement of an object is represented by appropriately shifting the bits representing the object to new positions in the bitmap.

Figure 1.30 shows how a stack of such bitmaps can represent a scene containing several two-dimensional objects. Each object is represented by a separate bitmap and the composite scene is visualized by sighting through the stack of bitmaps, performing a logical OR of the bits corresponding to the same location on the plane.

```
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 0 0 0 0
0 0 0 0 1 0 0 1 0 0 0 0
0 0 0 0 1 0 0 1 0 0 0 0
0 0 0 0 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
```

FIGURE 1.29. A two-dimensional binary array representation of a square.

Figure 1.31 summarizes how the operations of the figure abstraction could be carried out in this implementation.

This representation has inherent limitations in precision. Consider the program that implements the shape recognizer. For large enough figures it is clear that the imprint on the bitmap will be different for squares, circles, triangles, and spirals. However, for cases where the size of the image is close to the size of a pixel, the images can be indistinguishable. For example, a 2-pixel-by-2-pixel square and a circle with a radius of 1 pixel have the same rendering. Similarly, in deciding whether two figures overlap, protocol must report that they do if they are within one pixel of each other. In a real task, we need to be specific in describing the required precision for the protocols in order to determine whether the bitmap implementation would be adequate.

A Property List Representation for Figures

Figure 1.32 proposes another representation of a square in terms of a coordinate table with property lists. This representation indexes figures through a table according to the coordinates of their origins. For each figure, there is a property list of features indexed through the table. For example, in a square, the value of the shape property would be "square." Also for a square, the value

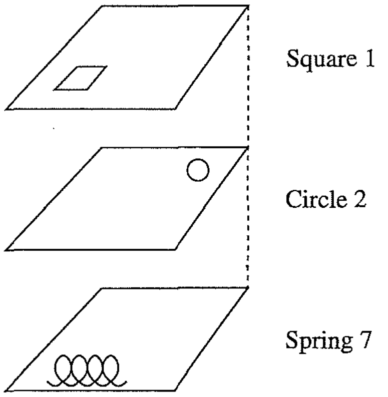


FIGURE 1.30. A stack of bitmaps. The composite scene would be obtained by sighting through the stack.

- Get-Shape: A recognition program analyzes a bit image in a plane to infer a shape and its parameters.
- Get-Origin: A computation is performed on the image. For example, to find the origin of a square the location of the leftmost and bottom-most corner must be determined.
- Get-Area: A computation must be performed on the shape parameters returned by Get-Shape: For a square, the length of a side is multiplied by itself.
- Move: The property list is shifted to a new cell in the array.
- Overlap: A computation must be performed on the two shape descriptions.
- Get-Overlapping-Figures: A search is made through the list of planes, iteratively invoking the Overlap method.

FIGURE 1.31. Implementing the operations for the bit image implementation of a figure.

of the side-length property would be a number indicating the length of the square's side. The origin of the square is determined by the indices of its description in the array. Thus, the lower left corner of the square is located in the scene at position (4, 3) and is indexed in the array through the cell at position (4, 3). If two figures have the same origin, then the coincident cell contains a list of pointers to the figure descriptions.

Figure 1.33 describes one way that the figure operations could be implemented. The operation of Get-Shape is simpler than in the pixel representation since it is only necessary to retrieve the value of the shape property. In contrast, the Overlap operation is made more difficult. Different computational approaches are possible. One approach, equivalent in precision to the bitmap approach, is to project the figure onto a temporary bitmap and then check whether any of the corresponding pixels are both on. An alternative approach is to solve the intersection problem analytically. To determine whether a square intersects a circle, we would need to compute the inter-

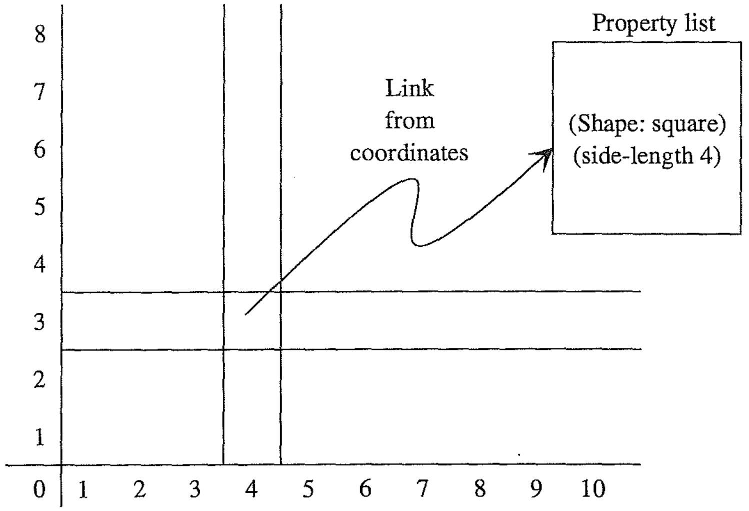


FIGURE 1.32. A feature-oriented representation of a square in a table. The property list description of the square is located at index (4, 3) in the table.

- Get-Shape: Retrieve a token naming the shape from the property list.
- Get-Origin: Return the indices in the array to the property list.
- Get-Area: Perform a computation on the shape parameters. For a square, the length of a side is squared.
- Move: Shift the property list to a new cell in the array.
- Overlap: Perform a computation on the shape. This may be done by intersecting the boundaries of the two figures or by computing a projection for each figure onto a plane and then intersecting those projections.
- Get-Overlapping-Figures: Search through the list of planes for those that overlap the given figure.

FIGURE 1.33. Implementing the operations for the property list implementation of a figure.

section of a circle and a line. If all of the figures are rectangles or polygons, the latter approach can be quite efficient.

A Graph Representation for Figures

Our third candidate representation for figures is a graph representation as in Figure 1.34. This data structure has a unique node called the head node that represents the figure as a whole. In Figure 1.34, the head node has an is-a link to the class Square. The head node has part links to designate relations to each of the four lines that make up the sides, and other nodes representing such information as the position. Moving the square in this representation amounts to changing the position nodes to link to different coordinates. A set of figures is represented as a list of such graph representations.

Figure 1.35 summarizes how the different operations of the figure abstraction could be carried out on the graph representation. As in the case of the property list representation, the shape of the figure is determined by a retrieval operation except that instead of returning the value of a property the procedure must follow an is-a link and return the name of a class. Retrieving the origin of a square requires traversing through the nodes representing the parts of the square to

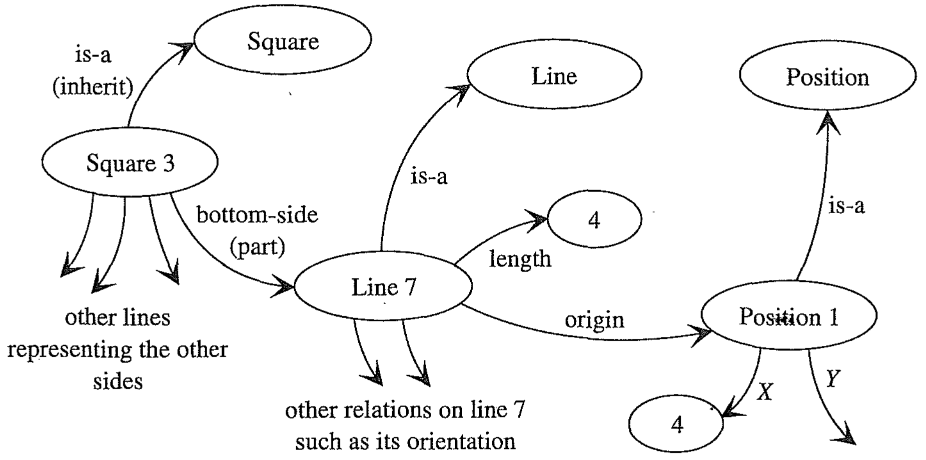


FIGURE 1.34. A graph representation of a square.

- Get-Shape: Starting at the head node, follow the is-a link to its class and return the name of that class.
- Get-Origin: Starting at the head node, follow the bottom-side link to a line, the origin link to a position, and then return the values of the *x* and *y* relations on that position.
- Get-Area: Perform a computation on the shape parameters. For a square, the length of a side is squared.
- Move: Shift the property list to a new cell in the array. All of the positions of all of the lines in the figure are updated to reflect new coordinates
- Overlap: The computation is analogous to the case of the property-list representation.
- Get-Overlapping-Figures: Search through the list of figures to find those that overlap.

FIGURE 1.35. Implementing the operations for a graph representation of a figure.

obtain the origin of the bottom side. The computation of overlap is analogous to the property list case.

Comparing Representations for Figures

The bitmap, property list, and graph all support the same abstraction. We can compare their performance for carrying out the prescribed operations. For example, consider the retrieval of the origin of a square in a scene. In the property-list table, retrieval of the origin requires that the processor be able to determine the indices of the entry in the table containing the square's description. To retrieve the square's origin in the graph representation, the processor must traverse the graph to retrieve the coordinates of the line representing the bottom side of the square. In the bitmap representation, retrieving the origin (at least for a "Manhattan square") involves stepping through the bitmap to locate the lowest and leftmost bit that is 1 and then returning the indices of that bit. Thus, to present the same abstraction, the three implementations must employ radically different processes with different amounts of time and space for the operation. Table 1.4 crudely compares the speeds of the operations on the three representations.

As shown in Table 1.4 there are trade-offs in the choice of implementations. One implementation is better for some operations, and another is better for others. If the objects are rectangles aligned with the coordinate axes, the computation for determining overlap requires only a few arithmetic operations given the positions and sizes of the rectangles. In the absence of special hardware, using bitmaps to decide whether such rectangles overlap would be fast enough for small figures but inefficient for large ones. If, however, the shapes of objects are irregular and nonrectangular and the bitmap logical ANDing operation was limited to tight regions containing

TABLE 1.4. Partial table of trade-offs for the three alternative implementations of a square.

Implementation	*Return Origin*	*Compute Overlap*	*Name Shape*
Pixel arrays	Slow	Fast for small figures	Slow
Property-list table	Fast	Depends on shapes	Fast
List graph representations	Medium	Depends on shapes	Fast

the objects, the bitmap representation could be made simple and efficient for determining the area of overlap. If we want to find the set of figures in a scene that overlap a given figure, then searching through the lists and tables and computing intersections can be a nontrivial computation for both the graph and feature representations.

1.3.3 *Primitive and Derived Propositions*

Reviewing the comparison of representations of figures, it is striking that, for each kind of query we considered, the favored data structure was one in which the required information was directly accessible.

Hector Levesque calls such representations **vivid** (Levesque, 1986). Others have called these analogical and direct. Vivid representations have the following characteristics:

- For every kind of object of interest in the world, there is a type of symbol.
- For every simple relationship of interest in the world, there is a type of connection among the symbols.
- There is a one-to-one correspondence between the symbols and the objects that they designate in the world.
- There is a one-to-one correspondence between the connections and the relations that they designate in the world. That is, the relation holds among the objects in the world if and only if the connection exists among the symbols in the knowledge base.

In considering vividness, we should not treat the notion of a "connection" among symbols too narrowly. Pointers in data structures are the simplest case of connections, but, as Levesque suggests, the notion can be extended to mean that two symbols are connected if they jointly satisfy some predicate that can be computed in bounded time. The tighter that the bounds are on computation, the better.

The terms *vivid*, *analogical*, and *direct* draw on a perceived similarity of the structure of a representation to the structure of the things designated.

An example of a vivid representation is given in Figure 1.36, which shows a line drawing of a chemical structure; Figure 1.37 shows a graph that could be used to represent the same structure. The similarity of both representations is apparent: There is a node for each atom of the molecule and a link for each chemical bond. The graph in Figure 1.37 shows a pair of links for each chemical bond, leading from each participating atom to the other.

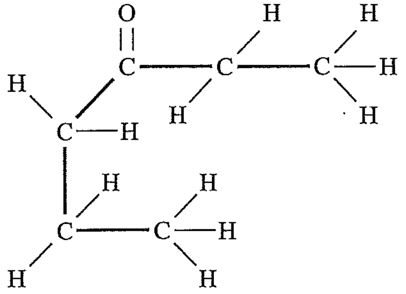


FIGURE 1.36. Line drawing of a chemical molecule.

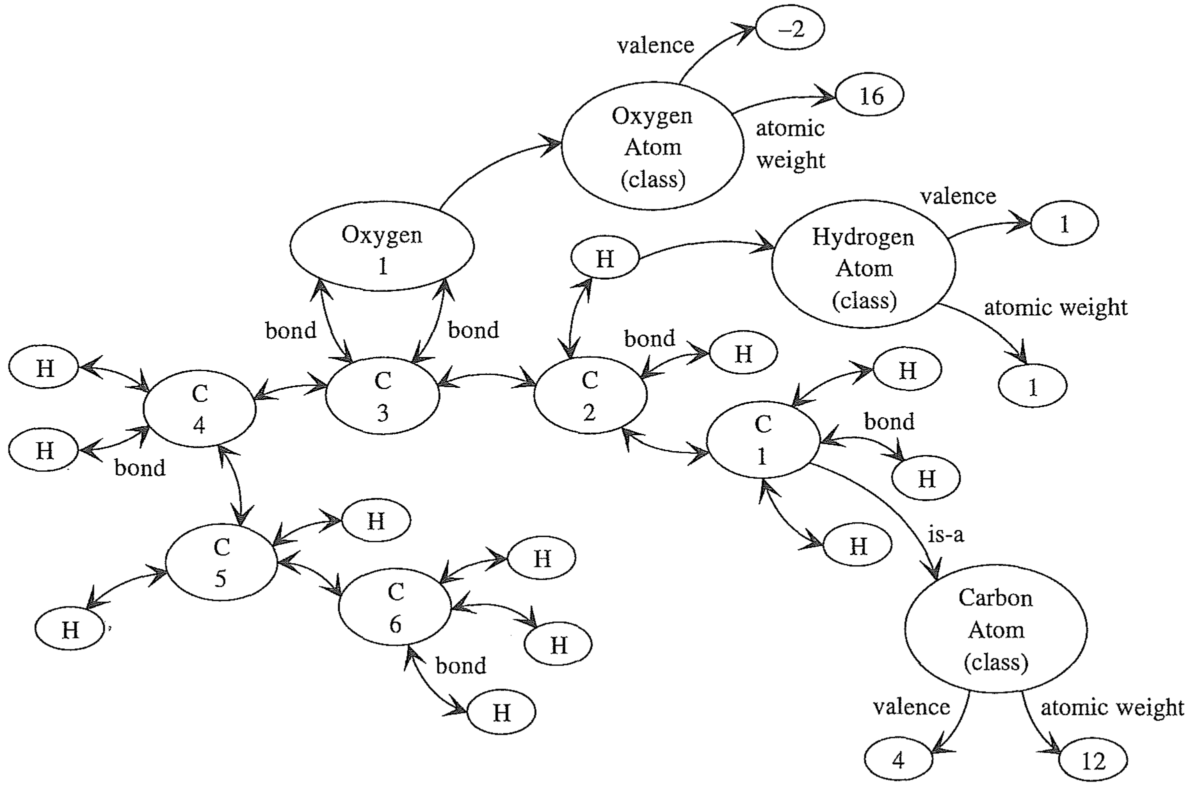


FIGURE 1.37. Semantic network and analogical representation of a molecule.

Analogical representations are like scale models. An often cited example of a direct representation is the representation of spatial relations in a room by maps. A map is called direct because of the similarity between the two-dimensional plane of the paper and the two-dimensional plane of the floor of the room. The paper is a direct homomorph of the room. Map and room have the same sort of structure (two-dimensional Euclidean space) and thereby admit the same sorts of operations such as sliding, rotation, and measurement. Naturally, the map is simpler than the actual room, in that various properties and relations such as texture, color, or a third dimension are missing from the map.

From another perspective, however, these concepts add nothing to the concepts of abstractions and implementations from the previous section. The "real world" does not present itself to us in terms of specific categories, objects, and relations. These concepts are all invented by us and we use them to describe what we perceive. To say that the line drawing of the chemical model is vivid is to say that it shares particular desired properties with the abstraction we want to use. The concepts of atoms and bonds are invented by people. In this regard, the notion of vividness is rather naive. Since designation depends on an observer, the directness of the representation depends on the distinctions already made by the observer. Thus, the world does not present itself to us in terms of objects and relations. Those are properties of how we think about the world, rather than intrinsic properties of the world itself. Thus vividness has to do with the correspondence of a representation with a favored model of the world, rather than a correspondence with the world itself.

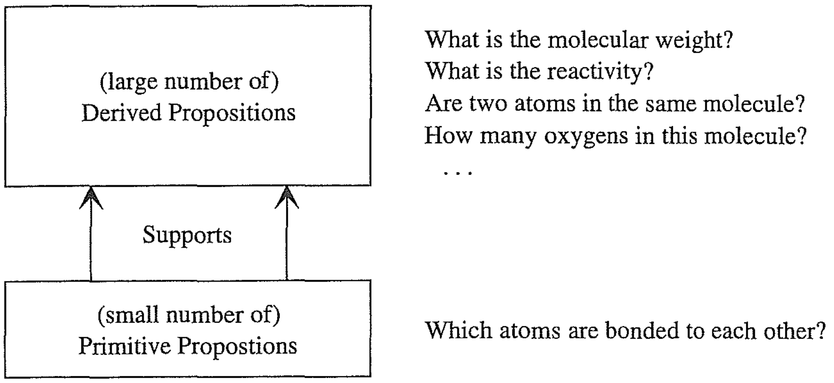

(large number of)
Derived Propositions
What is the molecular weight?
What is the reactivity?
Are two atoms in the same molecule?
How many oxygens in this molecule?
...
Supports
(small number of)
Primitive Propostions
Which atoms are bonded to each other?

FIGURE 1.38. A small number of primitive propositions can support a much larger number of derived ones.

This argument brings us to a different way of understanding and appreciating what is important about vivid representations. Much of the appeal of a vivid representation arises from the astute selection of primitive and derived propositions. **Primitive propositions** are those modeled directly in the representation. **Derived propositions** are other propositions about the objects that can be determined by a computation based on the primitive ones.

The primitive propositions in Figures 1.36 and 1.37 are the representations of atoms and bonds. An example of a derived property is the connectedness of atoms in molecules. To determine whether carbon 1 is in the same molecule as carbon 6, one can simply follow the bond links starting from carbon 1, searching for a path to carbon 6. If no path can be found, the atoms are in separate molecules. Another derived property is the molecular weight, which can be computed by summing the atomic weights of all the connected atoms in a molecule.

Figure 1.38 illustrates a common strategy in the design of knowledge systems that exploits this distinction between primitive and derived propositions. A large number of propositions can be derived from a small number of primitive ones. Derived propositions are computed on demand. In our representation of a chemical molecule, we had explicit representations indicating which atoms are bonded to their neighbors. These bonds were the basis for other computations on the graph, such as computing the molecular weight, computing the number of instances of any particular kind of atom in a molecule, computing the distance of separation of atoms of a particular type in a molecule, and so on.

The disciplined separation of intrinsic and derived properties can simplify the process of verifying that **invariants** are satisfied. To verify invariants in a design that segregates primitive and derived operations, it is enough to check the effects of operations on the primitive propositions. For example, assuming that interactions with air or other environment sources of atoms are properly accounted for, the making and breaking of chemical bonds in organic chemistry does not change the weight of the compounds or the number of atoms present. In building a computational model of chemical reactions, we might demand that our model preserve the same invariants.

To take an example, if the links representing chemical bonds between carbon 3 and carbon 4 are broken, no further changes to the underlying representation are needed to indicate consistently that carbon 1 is no longer part of the same molecule as carbon 6. No further changes are needed to prepare for the computation of the molecular weight of the molecule containing carbon 1. In effect, the "conservation of atoms" that chemists ascribe to the real world is captured faithfully by the "conservation of nodes" during the operations that make and delete links between nodes. For another example, if carbon 1 were bond linked to carbon 6 to form a ring molecule, and then the bond link between carbon 3 and carbon 4 was deleted, then with no further computational work the representation would correctly show that the molecule remains intact. In summary, the design of a vivid representation involves not only deciding which properties to make explicit, but also which properties to compute from a small set of primitive symbol structures, manipulated by the operations of frequent interest.

1.3.4 *Explicit and Implict Representations*

Advocates of early representation languages developed in AI often characterized them as having explicit representations. But what is the difference between an explicit representation and an implicit one and what are the advantages of explicit ones? In this section we define what it means for a representation to be explict and compare some example representations along this dimension.

What It Means for a Representation to be Explicit

A representation is **explicit** or **declarative** to the extent that it has the following properties:

- *Modularity.* The representation is self-contained and autonomous. In other words, there is an identifiable and bounded set of symbol structures that make up the representation, they are distinct and separate from the interpreter programs that use them, and there is a well-defined and narrowly prescribed interface by which other parts of the system access the representations.
- *Semantics.* The representation must have well-understood semantics. As usual, different kinds of semantics are appropriate for different situations. A denotational semantics or interface semantics is required to describe what behaviors correspond to different representations. If the interpreter draws inferences from the representation, the semantics include a reference semantics, a truth semantics, and a proof semantics. When the representation is used in reasoning with limited resources defeasible reasoning, then the semantics must include a reasoning control semantics.
- *Causal Connection.* For the representation to have any effect on the system from which it is separated, there must be a causal connection such that changing the representation causes the system to change its behavior in a way that is appropriate for the change to the representation and its semantics. This causal connection provides the basis by which the representation governs reasoning and behavior of the system.

A representation is said to be **implicit** if it is not explicit.

Explicit representations are important in knowledge systems. Their value does not rest on performance characteristics of the representations, since explicit and implicit representations are the same with regard to performance. Their value rests on the practical leverage that they provide for parameterizing the control of complex knowledge systems.

The importance attributed to explicit representations in knowledge systems reflects the concern of the field to the processes of updating of knowledge bases and reuse of knowledge systems. The combination of modularity, causal connection, and semantic correspondence makes it easier to change explicit representations than implicit ones. The semantics are understood and the interface is narrowly prescribed. These representational properties provide part of the interface to a knowledge system as a computational model.

The success of explicit representations in facilitating reuse and change depends on the extent to which the system's designers correctly anticipate the nature and extent of the different situations in which the system must operate, and which can be provided for by declarations. In this way, the design of explicit representations amounts to a bet about the likely dimensions needed for use and for change.

Comparing the Explicitness of Representations

We now consider two pairs of examples of representations to compare and discuss whether they are explicit. Figure 1.39 shows two approaches for organizing representations of facts about chemical elements in a reasoning system. In both organizations, particular parts of the systems are concerned with reasoning about molecular weights, chemical reactions, and printing reports about chemical structures.

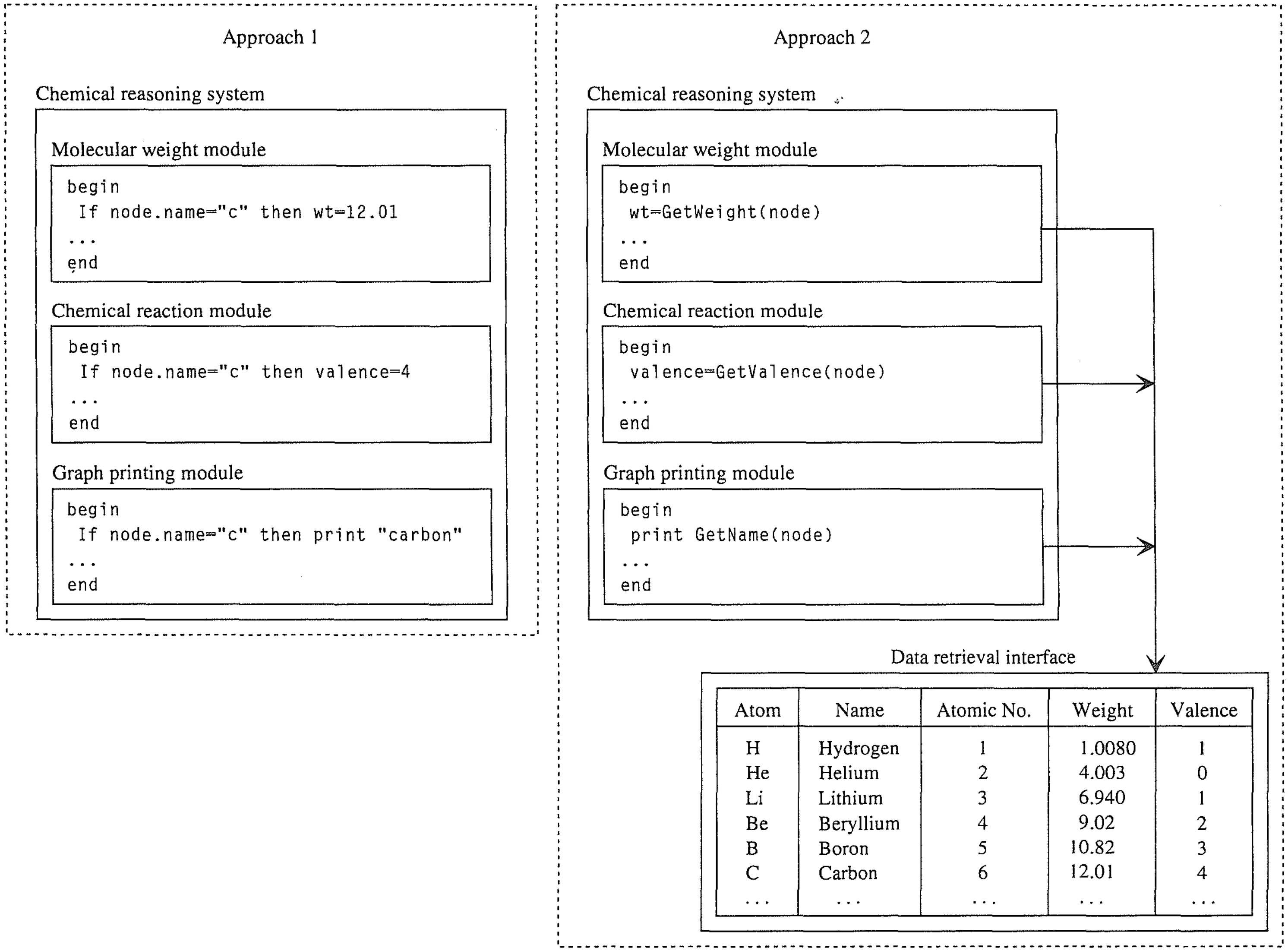
In the first approach, facts about the elements are scattered throughout the system. For example, statements about the weight and valence of the element carbon are located in separate parts of the program, perhaps several times. To add a new chemical element to the system or to change the facts about an existing one requires finding all of the places where it may be encoded.

In the second approach, a single table of chemical facts is stored separately from the reasoning methods. Whenever a fact about the elements is needed, the program invokes a database retrieval interface to retrieve the required information. To update the facts used by the system requires only updating the table.

Which of these approaches use explicit representations? First we consider the requirements for semantics and a causal connections, which the two approaches have in common. Both representations in Figure 1.39 have a semantics described by a characterization of atomic numbers, valence, and molecular weight. These amount to what chemists call a "ball-and-stick" model of molecules or roughly "labeled graphs." The operational semantics of the representation of chemical reactions or the computation of molecular weight are easily described in terms of primitives for making and breaking of bonds and appropriate graph operations and conservation laws. In summary, both representations satisfy the second and third requirements of being an explicit representation.

However, the first approach does not satisfy the modularity requirement. The database interface for the second approach has a quality of modularity that is lacking in the first. The representations of chemical facts in the first approach are scattered throughout the program that uses them. In contrast, the symbols making up the representation in the database are identifiable and separable from their interpreter. The database symbols are stored in a separate database, and are accessible from the program by a retrieval interface to the database. Thus, the database is declarative or explicit, but the first approach is not.

Figure 1.40 shows two approaches for defining the term *father* in a reasoning system. The first approach defines the term *father* using a set of predicate calculus statements. These state-



```
begin
 If node.name="c" then wt=12.01
...
end
```

```
begin
 If node.name="c" then valence=4
...
end
```

```
begin
 If node.name="c" then print "carbon"
...
end
```

```
begin
 wt=GetWeight(node)
...
end
```

```
begin
 valence=GetValence(node)
...
end
```

```
begin
 print GetName(node)
...
end
```

Atom	Name	Atomic No.	Weight	Valence
H	Hydrogen	1	1.0080	1
He	Helium	2	4.003	0
Li	Lithium	3	6.940	1
Be	Beryllium	4	9.02	2
B	Boron	5	10.82	3
C	Carbon	6	12.01	4
...	...	...	...	...

FIGURE 1.39. Two representations of facts about the elements in a chemical reasoning system.

Approach 1

Parent(*x*) => Person(*x*)
Child(*x*) => Perrson(*x*)
Parent(*x*) => HasChild(*x*)
Parent(*x*) => NumChildren(*x*)≥ 1
Father(*x*) => Parent(*x*) ∧ Male(*x*)

Approach 2

Parent
Superclasses: Person
Children: [Value: a Person]
Cardinality: Minimum 1

Father
Superclasses: Parent, Male

FIGURE 1.40. Two representations for defining the term *father* in a reasoning system.

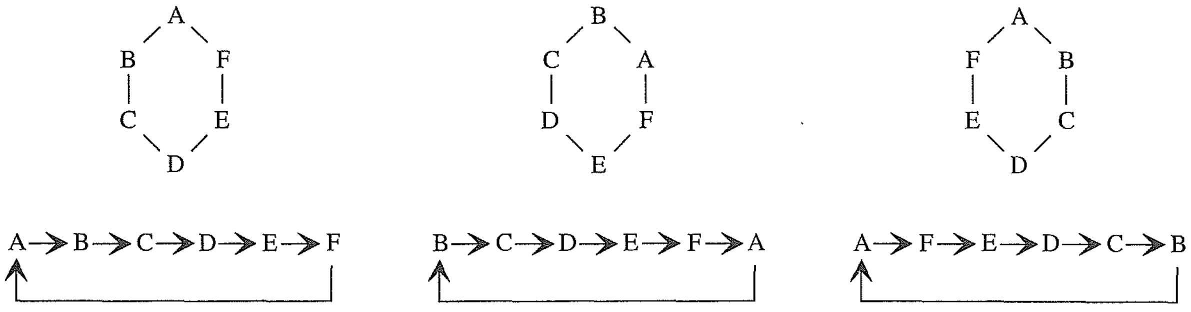
ments enable a reasoning system to answer questions about fathers, noting that fathers have the properties of people, parents, and males and that they have at least one child. In the first approach, the relevant statements can be part of a large base of facts. The second approach in Figure 1.40 draws on concepts from object-oriented and frame-representation languages. A parent is defined as a person who has at least one child. The term *father* inherits the properties from male and parent.

We now consider the explicitness of the two approaches for defining *father*. Again, both approaches have semantics and both approaches have a causal connection. What is at issue is whether either or both representations satisfy the requirement of modularity.

It is sometimes argued that any representation in predicate calculus is explicit because the sentences in a logic language are identifiably separated from their interpreter. However, sentences about growing pumpkins can be mixed up with ones about conducting international affairs. There is nothing in the representation that makes evident the extent of interactions, especially if one assumes a complete theorem prover. Thus, without boundaries in the representation, there is only a limited sense of modularity.

In contrast, the frame representation is designed so that all of the symbols that make up the definition of father are clustered together in a unit. The interpreter of a frame language is limited to making certain kinds of inferences involving inheritance, subsumption, mutual exclusion, and so on. In this way, the interface between different concepts is more narrowly prescribed, according to the rules of the particular frame language. Furthermore, the interaction with and through the interpreter are narrow because the operations of the interpreter in a frame language are fewer and more directed than those in a complete theorem prover.

In summary, explicit representations must be modular, must have well-defined semantics, and must have a causal connection between the representation and the behavior of the system. Explicit representations are used in knowledge systems to provide convenient interfaces for controlling the behavior of the system. A design that makes some representations explicit makes assumptions about what aspects of the systems behavior need to be controlled and parameterized for ease of reuse.



1
(A B C D E F)

2
(B C D E F A)

3
(A F E D C B)

FIGURE 1.41. Three graph representations of a chemical molecule, equivalent under rotation and reflection.

1.3.5 *Representation and Canonical Form*

ADVANCED

The design of symbol structures with structure-building primitives often leads to questions about the equivalence of descriptions. Figure 1.41 shows three graphs representing chemical molecules. Each letter stands for a unique bivalent piece of the molecule, and the links indicate chemical bonds between the pieces. Below each drawing is a picture of a linked list, showing how the descriptions of the pieces might be arranged in a data structure. Thus, in the first molecule, the first element of the linked list is A, the second is B, and so on to F, corresponding to the last piece of the molecule, which is linked back to A.

The three graphs in Figure 1.41 are topologically equivalent; that is, they are made up of the same kinds of pieces connected in exactly the same ways. Graph 2 is like graph 1 except that it has been rotated, so that it starts with B rather than A at the top. Graph 3 is also like graph 1 except that it has been reflected: Reading graph 1 clockwise instead of counterclockwise would yield the linked list shown for graph 3. A computer program that could generate all six of the rotated versions of this data structure, as well as the corresponding six reflections, potentially would have to deal with 12 distinct graph representations for the same kind of molecule.

The graph in Figure 1.42 is made of the same pieces as the three we have considered. However, it designates a distinct class of molecule because it is topologically different. The positions of B and C have been exchanged in the graph. No combination of rotations and reflections will yield a data structure like those in Figure 1.41.

To simplify the testing of equivalence in such representations, it is convenient to define canonical forms and canonical functions. Canonical functions transform expressions into canonical or standard forms. For example, we define the canonical form for these graphs as the list structure that is earliest in an alphabetical order reading left to right.

Given a canonical function, there is a test for deciding whether two expressions are equivalent: Apply the canonical function and test whether the resulting canonical expressions are equal. Thus, a **canonical function** is a procedure *c* that transforms any expression *e* into a unique equivalent expression *c*(*e*) such that, for any two expressions *e1* and *e2*, *e1* is equivalent to *e2* if and only if *c*(*e1*) is equal to *c*(*e2*). The resulting expressions, *c*(*e1*) and *c*(*e2*), are called **canonical forms**. The canonical forms induce equivalence classes on the set of expressions. For each equivalence class, the canonical form is a distinguished representative of the class.

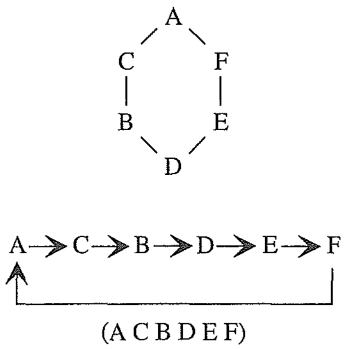


FIGURE 1.42. A data structure representing a molecule that has the same pieces as that in Figure 1.41, but that is not equivalent.

In the circular molecular graphs example, rotation and reflection are equivalence-preserving transformations because they do not change the identity of the graph. Without a canonical function, testing for equivalence would require searching for a chain of equivalence-preserving transformations connecting two expressions. Given a canonical function, we need only compute the corresponding canonical forms and compare them for equality.

For circular lists like these, it is easy to define a canonical form. We need to determine a starting node and an orientation for traversing the graph. One approach is to pick the linear representation that would come earliest in an alphabetic ordering. Thus, the representation (A B C D E F) comes before (B C D E F G) and also before (A F E D C B). Because the linked list for graph 1 in Figure 1.41 is in canonical form, and is different from the linked list in Figure 1.42, which is also in canonical form, the two data structures must designate different kinds of molecules.

The basis of canonical forms comes partly from graph theory and partly from the domain model. Thus, the same issues that we considered in the representation of molecules arise in the representation of electrical circuits. Are two given circuits equivalent? Does a large circuit contain a subcircuit equivalent to a known one? For sufficiently rich domains, the simple assumptions of pattern-matching languages are inadequate for answering these questions.

In many cases, there is no general function for efficiently computing canonical forms. Chapter 2 shows that the issue of equivalence testing arises in the design of systems that systematically generate symbol structures as candidate solutions to a problem. Not paying attention to canonical forms in such systems can result in the unwanted generation of variant and redundant representations of the same solutions. What is important to remember is when our models are rich enough to represent things in more than one way, determining equivalence classes can be a crucial part of designing efficient approaches to reasoning.

1.3.6 *Using Multiple Representations*

All this attention of designing a suitable symbol structure and interpreter as a representation may have left the impression that a knowledge engineer must find a single best representation and stick with it. An important alternative to this is the use of multiple representations. Multiple representations make it possible to combine the advantages of different representational forms. In multiple representations, more than one symbol structure designates something in the environment.

```
Molecule:    A03
Molecular-   Wt: 58 [cached]
Structure:   <pointer to semantic network representation of the molecular
             graph>
```

FIGURE 1.43. Caching molecular weight using multiple representations.

Suppose we had a knowledge system that reasoned about interactions between molecules. Suppose it occasionally reasoned about breaking bonds, but very frequently needed to access the molecular weight. You will recall that in our example of a graphical representation of a molecule, molecular weight was computed by traversing the graph that represents the molecule and summing the individual atomic weights. If molecular weight was used for many inferences, it would be expensive to traverse the graph and recompute it everytime it was needed.

An obvious alternative is to cache the computed molecular weight as part of the representation of the molecule as in Figure 1.43. The cached molecular weight is redundant to the molecular weight computed by traversing the molecular structure.

Whenever multiple representations are used, measures must be taken to keep them synchronized. In this case, whenever any operation is performed that changes the graph representing a molecule, the cached molecular weight could become invalid. This issue is inherent in the use of multiple representations and is called the **stale data problem**. To preclude the accidental use of stale data in a computation, we need to extend our concern with the operations and invariants of abstractions to include multiple representations rather than just primitive propositions. As in the example of caching molecular weight, cached values need to be invalidated whenever the assumptions behind their caching become invalid. Specifically, the cached molecular weight should be invalidated whenever operations make or break chemical bonds.

It is worthwhile to approach the stale data problem in two parts. To preclude inferences using invalid derived data, it is enough to mark the cached and derived representations as invalid whenever the underlying primitive representations are changed. This is known as invalidating the cache. Typically, an additional bit or a special value associated with the cache is used to indicate when the cache is valid. A separate question is when to recompute the derived value. If the primitive values could change several times before the derived value is requested, it is wastefull to recompute the cache every time that the primitive data change. A more efficient alternative is to recompute the derived value only on demand, that is, when the value is requested and the cache is not valid.

For another example of the use of multiple representations, we return to the three implementations of the figure abstraction that we considered earlier. Suppose that a typical scene contained tens of thousands of figures and that a common operation was to find all of the figures in a given region that overlap with a given figure.

Each of the proposed implementations is potentially expensive for this operation. In the pixel representation, we need to compare (or AND) the planes of pixels for every figure. This operation is proportional to nd^2 where where n is the number of figures and d^2 is the number of pixels per plane. For the property-list representation, the table indicates the origins of the figures but not their sizes. Because figures can have arbitrary sizes, figures whose origins are arbitrarily

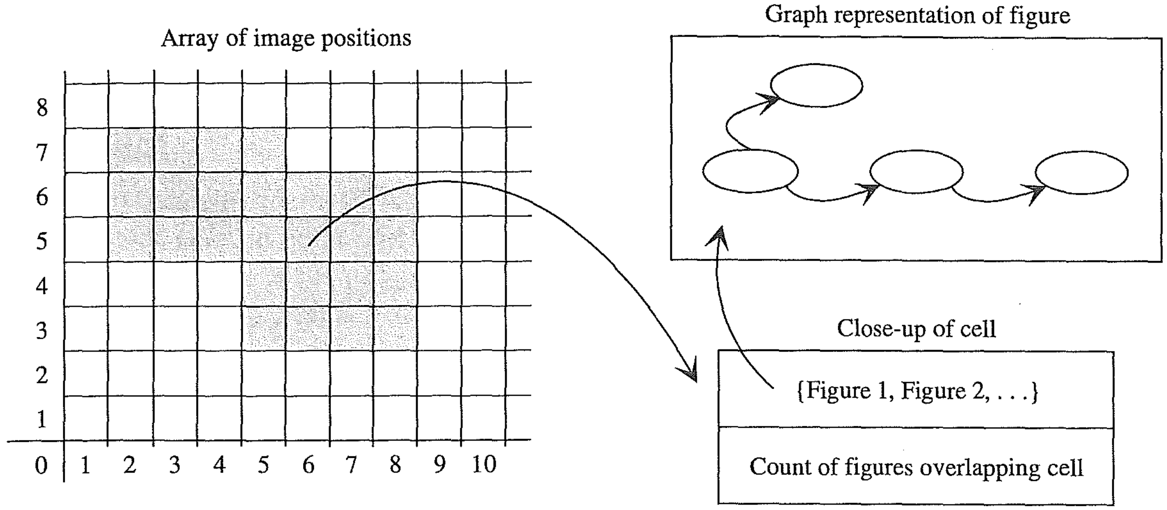

Array of image positions
8
7
6
5
4
3
2
1
0
1 2 3 4 5 6 7 8 9 10
Graph representation of figure
Close-up of cell
{Figure 1, Figure 2, . . .}
Count of figures overlapping cell

FIGURE 1.44. Combining representations.

distant could still overlap. Thus, information about origins does not let us weed out figures that potentially overlap and we need to compute the overlap of the figure of interest with every possible figure. For the graph representation, the computation is similar.

Figure 1.44 shows an approach to making this computation more efficient by combining multiple representations. In this example, each figure is represented in detail by a graph representation. From this graph representation, the coverage on the plane is then projected back onto a coarse map. We call this projection the figure's "shadow." Each cell in this coarse map has a count of the number of figures that cover that cell, wholly or partially, and a list of pointers to those figures.

The advantage of this representation is the way that it provides a coarse but inexpensive filter that makes it unnecessary to perform detailed overlap computations for most figures. To find the figures that overlap a given figure we procede as follows. First we find the shadow for the figure of interest. Then we collect all of the figures which have common elements in the shadow. These are the figures that "potentially" overlap the given figure. This eliminates from detailed consideration all of those figures whose shadows do not overlap. When there are tens of thousands of figures and all but a few of them are eliminated, the savings can be considerable.

As always with multiple representations, we must consider how operations could invalidate the cached values. Thus, when a figure is moved, the projection is recomputed and the old counts in the coarse array of image positions are decremented under its shadow. If any cell has a count of zero, then it's accumulated shadow is zeroed out. Then the origin of the figure is updated and a new projection is computed, the shadow is ORed into the coarse array, and the cell data are updated.

The coarser the shadow array, the less expensive it is to maintain. At any resolution, the shadow overlap is accurate in that it never misses a potential overlapped figure. Precision depends on the coarseness of the map. Lack of precision shows up as a false positive: Two figures may seem to overlap if they actually cover disjoint parts of a cell.

In summary, multiple representations create new options when there are unacceptable trade-offs among fidelity, precision, and efficiency. Whenever multiple representations are used, the concern with consistency and invariants must be extended to include not only primitive propositions but also multiple representations.

1.3.7 *Representation and Parallel Processing* ADVANCED

In a conventional (von Neumann) computer architecture, one processor operates on symbol structures in a passive memory. As alternative computer architectures become available, we should consider the opportunities for parallel and distributed computation. The efficiency of operations on a representation depends both on the data structure and the capabilities of the computer.

Figure 1.45 shows an example of a parallel computer architecture: a simplified picture of a connection machine (Hillis, 1985). A connection machine links together thousands of extremely small processors and memories. The first generation of these machines contains up to 64,000 small processors. The sequence of operations carried out by the small processors is determined by the front-end processor, which broadcasts instructions to them. Figure 1.45 shows communication links from the cells to each of their four neighbors and to the front end processor. In the actual computer there is also a packet switching network that makes it possible to send information simultaneously between arbitrary pairs of processors.

A connection machine can perform discrete simulations of fluid flow, as shown in Figure 1.46. The hexagon-shaped cells correspond to small regions of space. Within a cell, particles move about according to specific interaction rules. For example, a particle in motion will continue to move in the direction it started unless it collides with another particle or with some obstacle in the space. In the figure, an obstacle in a cell is indicated by a solid rectangle and a moving particle is indicated by an arrow.

The wind in a wind tunnel does not blow over one square centimeter of an airplane model at a time. It blows across the whole model at once, showing the engineers how the flow in one

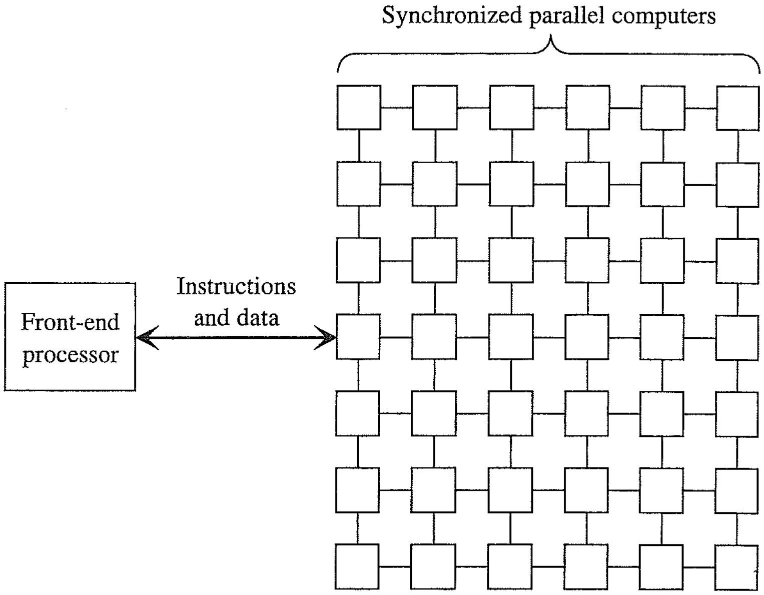


FIGURE 1.45. Architecture of a connection machine.

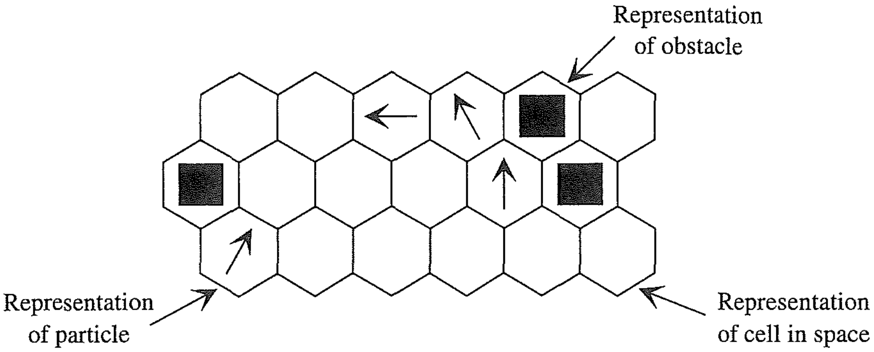

Representation of obstacle
Representation of particle
Representation of cell in space

FIGURE 1.46. Part of a simulation of fluid flow on a connection machine.

section interacts with the flow in another. If we simulate the wind in parallel, that is, simulating the flow of air particles through all of the space at once, the results can be computed more quickly. This approach has been called "data-level parallelism" (TMC, 1986).

With data-level parallelism, the simulation can be carried out by representing each cell with a processor in the connection machine. At each time step, every cell updates its state by checking all of its adjoining cells for particles that are heading in its direction. The cells then update their state according to the interaction rules. A connection machine is capable of modeling approximately 1 billion cell updates per second. These results are averaged to predict macroscopic effects accurately, providing a computational alternative to a wind tunnel.

This approach to simulating fluid flow would be almost unthinkable on a conventional computer. Data-level parallelism makes it practical by providing enough processor power to assign a computer to each region of space. The representation is analogous to a computational view of the universe in which small regions of space are seen as computational elements that communicate with their nearest neighbors. This point of view has some subtle advantages over steady-state modeling. For example, the analogical representation naturally and faithfully predicts the emergence of turbulence; that is, the unstable and chaotic swirls that develop as fluid flows across complex surfaces. This is an important aspect of reality that is not captured by the differential equation models typically used.

The main point of this example is that the efficiency of a representation depends both on the data structure and the capabilities of the computer. When we say that computers are "good with numbers," this is a result of the built-in hardware for efficiently performing arithmetic operations on binary representations of numbers. A connection machine is good for data-level parallelism because of the many built-in and interconnected processors supporting massive parallelism. As new computer architectures become widely available to knowledge engineers, the design of appropriate representations will need to encompass not only the selection of data structures, but also the selection of effective computer architectures.

1.3.8 *Space and Time Complexity* ADVANCED

So far in this chapter we have referred rather loosely to symbol structures and their interpreters as being efficient or not. In this section we present notation and concepts for quantitatively characterizing the complexity of computations.

The amount of storage that an algorithm uses is called its **space complexity** and the amount of time that it uses is called its **time complexity**. In the broadest sense, the efficiency of an algorithm involves all the various computing resources needed for executing it. Time requirements are typically the dominant factor in determining whether or not a particular algorithm is efficient enough to be practical.

Time and space requirements of an algorithm can often be expressed in terms of a single variable, the "size" of the problem instance. For example, the size may be "the number of elements to be sorted" or "the number of cities to be visited." This is convenient because we expect the relative difficulty of problems to vary with their size.

For idealized computations, we assume that accessing any storage element in memory requires unit time, meaning that each access takes the same amount of time. Similarly, in computing time complexity, it is common to count the number of times that the steps are performed, where each step must be executable in constant time. In computing space complexity, it is customary only to account only for space actually accessed by the algorithm. There is no charge for memory that is not referenced during execution. If the algorithm requires that a large data structure be initialized, then we have to be clear about whether we are counting the initialization time and usually we should. Thus, the time complexity of an algorithm is always greater than or equal to its space complexity, since every an instruction can either reuse an element of memory or allocate a new one.

Asymptotic Complexity

In the theory of algorithms, the "big-oh" notation is used to indicate asymptotic measures of complexity. The notation $O(f(n))$ refers to a numeric quantity that is within a constant factor of $f(n)$ where f is a function of the integer n, which is some parameter of the method being analyzed. More precisely, we say that

$$x_n = O(f(n)) \tag{1}$$

when there is a positive constant C such that

$$|x_n| < C\,|f(n)| \tag{2}$$

for all n greater than some initial integer, n_0. For example, if the time complexity of some algorithm is precisely

$$6n^2 + 43n + 16 \tag{3}$$

we could say simply that it was $O(n^2)$ since, for large enough n, the running time is dominated by the squared term. This is called an **asymptotic complexity** because it need only hold for large enough n. In this way, the big-oh notation leaves out the details of a complexity measure while presenting enough information to get the main point of a comparison. It is often the case that minor variations in algorithms do not change the way that they scale, and thus, do not require changes to estimates of asymptotic complexity.

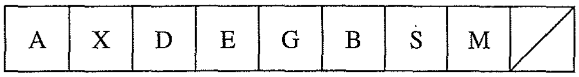


FIGURE 1.47. Entries in an unsorted, linear list. Search time is proportional to the length of the list.

In addition to the big-oh notation, there are two other conventional shorthands—little oh and omega notations—that summarize related asymptotic conditions. These notations are defined in equations (4) through (6).

$$g(n) \text{ is } O(f(n)) \qquad g(n) \le C f(n) \text{ for } n > n_0 \tag{4}$$

$$g(n) \text{ is } o(f(n)) \qquad \lim g(n) / f(n) = 0 \text{ as } n \to \infty \tag{5}$$

$$g(n) \text{ is } \Omega(f(n)) \qquad g(n) > C f(n) \text{ for } n > n_0 \tag{6}$$

Algorithms can take different amounts of time for different classes of starting data. If we talk about the time complexity of the algorithm, we have to specify whether we are referring to the exact time for particular data of size *n*, the worst-case time for data of size *n*, the average time for all possible data of size *n*, or the typical time for data of size *n*. In general, the worst-case times are the easiest to compute and at one time these were the ones most often used to characterize algorithms. However, worst-case times are not always usefully indicative of the performance of algorithms. Today it is considered more appropriate to give several characterizations of the time complexity of an algorithm, including not only worst-case and average times, but also typical times for well-characterized classes of cases.

To give an example of the use of the order notation for comparisons, we will compare three algorithms for searching a symbol structure for an entry. In the first case, the entries are kept in an unsorted linear list, such as that in Figure 1.47. Figure 1.48 presents pseudocode for searching this list. It iterates down the list starting at the beginning, comparing each list element in turn with the target entry. If a matching entry is found, it returns "yes" together with some auxiliary data. If the algorithm encounters the end of the list as indicated by the entry with a slash in it, it returns "no."

If there are *n* items in the list, the algorithm will execute the steps in the loop *n* times. We can treat this loop as a unit of time or otherwise normalize it to the instruction time for the steps in the loop. Of course in doing this, we need to be sure that no step in the loop requires a time that depends on *n*. In such cases, we need to account how much time each step takes.

To search a list:

```
Procedure(target, list)
do begin
      item := pop(list)
      if name(item) = name(target) then return "yes" with data(item);
end until end-of-list (list);

/* Here if target not found. */
return "no"
```

FIGURE 1.48. Method for searching an unsorted, linear list.

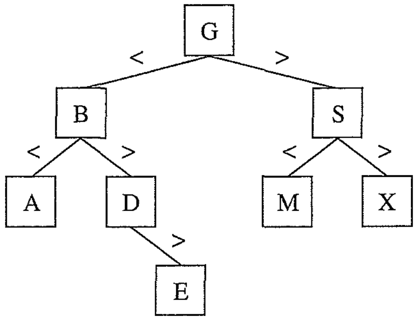


FIGURE 1.49. Entries in a balanced tree. Search time is proportional to the depth of the tree.

How long does our algorithm take to run? In this problem, the worst-case time complexity is $O(n)$, where the algorithm searches to the end of the list and then fails. But what is the average time to find an entry? To answer this question, we need to make some assumptions about the list. Suppose, for example, that that the desired item will almost never be found in the list. In this case, the algorithm usually searches to the end of the list and then fails.

On the other hand, suppose that the target is usually found in the list. Again, to make the estimate we need to make assumptions about the position of the data in the list. Suppose that there is an equal probability that the target will be at each point in the list. It is easy to see that on average the method will search half of the list before matching the target. Since "half" is a constant factor, once again we have that the average time complexity is $O(n)$.

Different symbol structures and algorithms can support the same abstractions but with different efficiencies. Figure 1.49 gives an alternative tree representation of a set to be searched. For now, we assume that each entry in the tree has two links called "more" and "less." The data are arranged so we have a three-way test at each entry. If the entry equals the target, the search is done. Otherwise, if the target is alphabetically less than the entry, the algorithm follows the "less" link and checks the entry there. If the target is alphabetically greater than the entry, the algorithm follows the "more" link and checks the entry there. Eventually, the algorithm either finds the target or reaches the edge of the tree. This algorithm is presented in Figure 1.50.

To search a tree:

```
Procedure(target, tree)
begin
    item := root(tree)
    do begin
        if name(item) = name(target) then return "yes" with data(item)
        elseif lexical(target) < lexical(item) then item := less(item)
        elseif lexical(target) > lexical(item) then item := more(item)
end until null(item);

/* Here if target not found. */
return "no"
```

FIGURE 1.50. Method for searching a tree.

length = 8	A	B	D	E	G	M	S	X

FIGURE 1.51. Entries in a sorted, linear list. Search time is proportional to the length of the list.

Once again, we ask about the time complexity of the algorithm. To analyze the time we need to make assumptions about the shape of the tree. Suppose we know that this is a balanced tree, in the sense that the subtrees on either side of a node are of nearly the same size. As the algorithm proceeds, it marches down a path from the root or top of the tree. The maximum number of times the algorithm must execute the loop is the depth of the tree, that is, the number of steps in a path from the root. The length of such a path for a balanced tree is the ceiling or next integer greater than $\log_2(n)$. Thus, the worst-case time complexity of this method is $O(\lceil \log n \rceil)$. Readers interested in more details about operations on balanced trees can consult a basic textbook on algorithms and data structures, such as Knuth, 1973.

Before leaving this searching example, we note that achieving an $O(\log n)$ time on it does not necessarily require having a tree structure. Figure 1.51 gives a tabular representation of the list where items can be reached by indexing them in array. Unlike the previous list example, this one assumes that the items are kept in the table in alphabetical order. This sorted data structure lends itself to a binary search, where the first item checked is one halfway through the list. If an item does not match, the search then proceeds in a manner analogous to the method in Figure 1.50. It focuses its search to either the elements greater than or less than the item just searched and tries again at the middle of an unsearched portion of the list. At each comparison, the region left to search is approximately halved. A binary search considers the same items in the same order as the tree search. It trades arithmetic operations on indices for link-following operations in a tree. Once again, the average and worst-case time complexities are $O(\log n)$. Updating the table to add or delete an element takes time $O(n)$.

Different algorithms have different asymptotic complexity functions. What is efficient enough depends on the situation at hand. However, one widely recognized distinction is between polynomial time algorithms and exponential time algorithms. A **polynomial time algorithm** is one whose time complexity function is $O(p(n))$ for some polynomial function p, where n is used to denote the input size. An algorithm whose time complexity function is greater than this for all fixed polynomials is called an **exponential time algorithm**.

The distinction between these two types of algorithms is particularly significant when we compare the solution of large problems, since exponential functions grow so much faster than polynomial ones. Because of this growth property, it is common practice to refer to a problem as intractable if there is no polynomial time algorithm for solving it.

Space and Time Complexity for Parallel Algorithms

When multiple or parallel processors are engaged in a computation, it is appropriate to express time complexities in a way that takes into account the number of processors. In some cases, we can assign n processors to problems of size n although this option is often not realistic.

Using k processors can not reduce the asymptotic time complexity of a problem by more than a factor of k. This is not to trivialize the value of multiprocessing. After all, applying 64,000

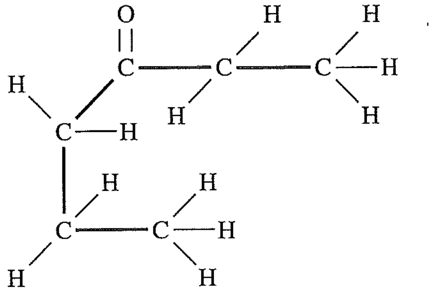


FIGURE 1.52. The weight of a molecule is the sum of the weights of its constituent atoms.

processors to achieve a constant speedup by a constant factor of 64,000 can be very dramatic and practically significant.

Our earlier simulation example of fluid flow showed that when multiple processors are available, there can be an opportunity for radically rethinking an approach to a problem. Discovering different ways to partition problems can be very challenging and does not always lead to dramatic speedups. A recurring issue as parallel processing becomes more available is finding ways to partition problems.

In the fluid flow problem, the dramatic speed-up was largely due to the fact that the problem could be divided into a set of local computations. Each processor represented a small region of space. All of the data required for its computation was contained in the processor itself or could be obtained from its immediate neighbors. Such computations are called **local computations**. This is important because it is often the case in massively parallel computers that communications instructions beyond local neighbors require much more time than other instructions.

In contrast, **global computations** involve combining data from distant processors or aggregating over a large set of processors. In such computations, some serialization of processing is often required. For example, consider our earlier problem of computing the total weight of a molecule given a graph representation of its structure.

Figure 1.52 gives an example of a molecule. Suppose we assign a processor to every atom in the molecule. Each processor has a table of addresses of the processors that represent adjacent or chemically linked atoms in the molecule. An algorithm for computing the total weight must include the weight of each atom exactly once.

Roughly, one way to proceed is to start at one atom and have it ask each of its neighbors in turn to send it their weights for summing. The process iterates until each atom had contacted its previously uncontacted neighbors. The resulting partial sums are accumulated as they approach the starting atom. We can visualize the process in terms of two waves. The first wave is a request for weight sums. It moves out from the processor representing the starting atom. The second wave is the returning wave, containing the partial sums. Unfortunately, in spite of the fact that this approach uses n processors, it still runs in $O(n)$ time in the worst case, such as when the starting atom is at the end of a long linear molecule.

To speed up the process we would need to engage more processors at once. There are many ways to do this and specialized languages for different computer architectures. The details of this are beyond the scope of this section, but one way to understand it is to imagine that we superimpose a summing tree over the molecule as in Figure 1.53. There are 19 atoms in this figure, but they are linked in a tree with only 5 levels. All of the atoms linked together at the first

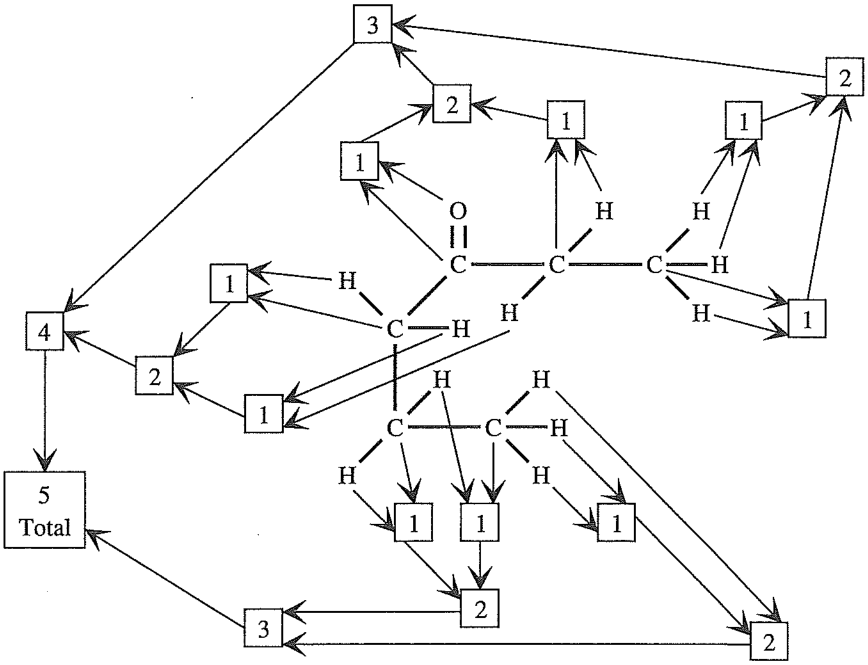


FIGURE 1.53. The weight of a molecule is the sum of the weights of its constituent atoms. Heavy links correspond to chemical bonds. Arrows indicate data flow to summation points for computing the total weight.

level of the tree (labeled 1) are performed on the first cycle. These partial sums are then combined at the second level of the tree on the next cycle. Those partial sums are combined at the next level. After each phase, only half as many processors are active as on the previous phase. The total sum is aggregated in time proportional to the ceiling of the depth of the tree, that is, in $O(\log n)$ time.

Complexity Distributions

It is well known that for some algorithms the worst-case complexity is much worse than the average complexity. But the average case can also be misleading.

Consider the distribution of computation times in Figure 1.54. In this example, there is a population of possible inputs to the algorithm and the time to perform the computation on the input varies drastically. Thus, there is a small set of elements for which the time is $O(x^1)$, and a few more where the complexity is $O(x^2)$. The largest set of elements has complexity $O(x^4)$. The curve drops off so that there are only a few elements where the complexity is $O(x^n)$.

In this example, the worst-case time complexity is $O(x^n)$. Perhaps surprisingly, the mean time complexity is still $O(x^n / n)$. That this is the case can be seen by considering equation (7).

$$\text{mean complexity} = (1/N) \sum_{i=1}^{N} x^i \tag{7}$$

Thus the mean complexity is dominated by the largest term.

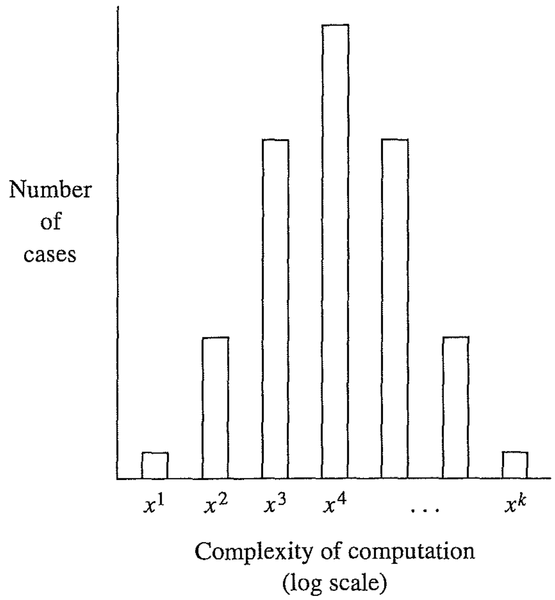


FIGURE 1.54. Distribution of cases with different complexity orders.

What this shows us is that the mean complexity is also not necessarily representative of typical complexity. Referring back to Figure 1.54, most of the elements in the set will result in computations that are $O(x^4)$. Thus, the mode of the distribution is much less than the mean. This example suggests that providing a distribution of complexities may in many cases be a more meaningful way of conveying expectations about running time than either a worst-case or mean-case analysis.

Phases, Phase Transitions, and Threshold Effects

The previous section characterizes expected running times of algorithms in terms of their performance on populations of structures. In this section we consider phenomena that can arise when populations are characterized by possibly critical values of variables. The terminology of these phenomena—**phases**, **phase transitions**, and **threshold effects**—is drawn from physical chemistry. The analyses are drawn from statistical mechanics.

Phase transitions are familiar from everyday experience. Most substances have characteristic melting points and boiling points. As temperature is raised or lowered at constant pressure near these points, materials undergo phase transitions—from solid to liquid or from liquid to gas. Underneath this abrupt, qualitative change at the macroscopic level is a small change at the microscopic level. Temperature is a measure of the vibrations of atoms in a substance. As temperature is increased, these vibrations become more energetic. Increased temperature causes a substance to expand because the vibrations increase the average separations between atoms. Usually small changes in temperature cause small changes in vibration and small changes in the size of an object made from the material.

However, at certain critical points such as the melting point and boiling point, small changes in temperature result in gross changes at the macroscopic level. At these points, the actual energy and separations of the vibrating atoms changes only slightly it but results in a qualitative softening as vibrations become vigorous enought to enable atoms to slide past each other.

To a first order approximation, phase transitions are very sharp. Below the melting point, the substance is a solid and above it is a liquid. Such sharp transition points are called thresholds. If we view the area around the transition point in fine detail, the story is usually more chaotic. As ice forms, sometimes crystals form and then melt and then form again. Liquids sometimes exist in "superchilled" states for periods of time without making the transition. Mathematical models show both phenomena—the relatively sharp macroscopic transition points as well as fine-grained chaotic behavior in the immediate vicinity of the transition.

For a simple example of this phenomenon in the context of algorithmic complexity, we consider again algorithms that operate on trees. Suppose that we have a population of trees characterized by an average branching factor, *b*. We use the term **cluster size** to refer to the expected size of a tree drawn from the population. We will consider several such algorithms in Chapter 2 that search trees. For now it is enough to know that our algorithm visits every node in a cluster.

To determine the time complexity of such an algorithm, it is useful to know how many nodes there are in a cluster. To count the nodes, we start at the root of the tree. On average, there are *b* nodes immediately adjacent to the root. Then there are b^2 nodes immediately adjacent to these one level down. For a tree of infinite extent, the cluster size, $C(b)$, is given by equation (8).

$$C(b) = \sum_{i=0}^{\infty} b^i = 1 / (1 - b) \tag{8}$$

For trees of finite depth, *d*, the cluster size is given by equation (9).

$$C(b) = \sum_{i=0}^{d-1} b^i = (1 - b^d) / (1 - b) \tag{9}$$

Figure 1.55 shows cluster size as a function of the average branching factor for trees of different depth. For a potentially infinite tree, an average branching factor less than one leads to a finite-size cluster. However, when the average branching factor reaches 1, there is a singularity in the graph as number of nodes connected to the root on average suddenly becomes infinite. This singularity is an example of a **threshold**. It marks the existence of a phase transition characterized by an explosive increase in cluster size as the threshold is approached from below. For any fixed polynomial of *b*, the time complexity of our node-visiting algorithm exceeds the polynomial as *b* approaches the critical point.

The signature of the phase transition for potentially infinite trees is still visible for trees of finite depth. The other curves in Figure 1.55 show the expected cluster size for trees with several fixed depths.

Phase transitions show up in many kinds of computations. For example, they show up in the analysis of search methods where the number of nodes explored along a false "garden path" corresponds to the cluster size in this example. They also show up in analyses of algorithms that work over more general graph structures than trees. Again in these cases, the cluster sizes of graphs undergo phase transitions as the number of connections between nodes increases.

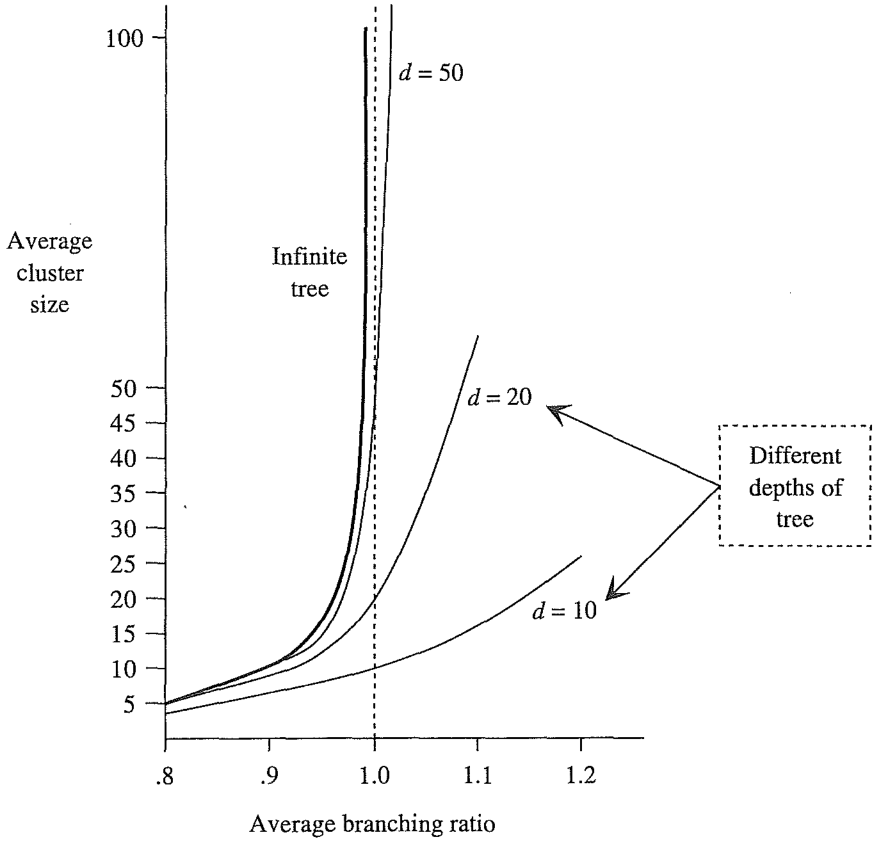


FIGURE 1.55. Cluster size for trees with finite depth, d, as a function of the average branching factor, b. (Adapted fom Huberman, B. A., and Hogg, T. Phase transitions in artificial intelligence systems. *Artificial Intelligence* 33, 1987, pp. 155–171.)

1.3.9 *Structural Complexity* ADVANCED

Space and time complexity are properties of algorithms as applied to populations of different possible data structure inputs. Sometimes the running time of an algorithm over a particular data structure depends a great deal on its particular properties. For example, an algorithm to reverse the order of a list depends on the number of elements in the list. Algorithms that search trees depend on the particular depths and branching factors of the trees.

Measures of the properties of a structure are called **structural complexities**. In this book we use several different measures of structural complexity for different kinds of structures and different purposes, as suggested by Figure 1.56. In this section we develop and study two examples of such measures. First, however, we discuss generally what it is that measures of structural complexity are good for.

Measures of structural complexity provide scales for comparing structures. Suppose that we are given a set of structures $\{S_1, S_2, S_3, \ldots, S_n\}$ and a complexity measure C. If $C(S_1) > C(S_2)$ then we say that structure S_1 is more complex than S_2. We can use the scale to identify the simplest members of a set.

Suppose further that there is a computation, F, that we can perform on various members of the set, and that the time complexity of $F(S_k)$ depends in a known way on $C(S_k)$. In this case, the

Measure	*Remarks*
Number of elements	Same as space complexity.
Depth of tree	Can be used to compute space and time complexities for tree searches.
Branching factor	Can be used together with depth to compute space and time complexities of tree searches.
Hierarchical diversity	Measure of the diversity of structures in unlabeled trees.
Width of constraint graph	Measure of complexity of labeled graphs, such as constraint graphs. Related to time complexity for solving constraint satisfaction problems.
Bandwidth of constraint graph	Measure of complexity of graph structures. Related to time complexity for solving constraint satisfaction problems.
Shannon information entropy	Measure of the sharpness of a distribution of probabilities. Used in diagnostic systems to select probe sites most likely to reduce the number of competing hypotheses.

FIGURE 1.56. Examples of measures of structural complexity.

structural complexity measure can be useful for partitioning the set of structures into families whose time complexities for F are equivalent. For example, suppose that the time complexity relates to the structural complexity as follows:

$$\text{time complexity } (F(S_k)) = O(m^{C(S_K)}) \text{ for some constant } m.$$

That is, the time complexity for computing F on S_k is polynomial in the structural complexity of S_k. In this case, C gives us a means for dividing $\{S_1, S_2, \ldots S_n)$ into easy and hard problems. The more complex the structure, the longer it takes to compute F on it. In this way, measures of structural complexity can provide useful alternatives to nonrepresentative worst-case estimates of time complexity over large populations of data structures. The alternatives are either to measure $C(S_k)$ so as to get a better estimate of the time complexity of $F(S_k)$ or to identify S_k as belonging to a subpopulation where the range of $C(S)$ is much narrower and thereby more representative.

To be concrete, we now consider a particular complexity measure for unlabeled trees. This complexity measure is called **Huberman-Hogg complexity** or **hierarchical diversity** (Huberman & Hogg, 1986). Huberman and Hogg were struck by examples of structures in physics and computational systems. In physics, both crystalline materials and gases are considered to be relatively simple when compared with amorphous structures. Crystalline structures are considered simple because their molecular structures are repetitious. Gases are simple because the molecules are randomly located and do not aggregate. On the other hand, amorphous structures have a mixture of molecular structures of different sizes and are considered complex. Similarly in linguistics, random strings are viewed as simpler than sentences produced by the grammars of formal and natural languages. Huberman and Hogg sought a measure of structural complexity that would reflect these intuitions. They wanted to quantify the regularities and hierarchical structure of trees.

We first consider some examples of hierarchical structures. Figure 1.57 shows three unlabeled trees, $\{T_1, T_2, T_3\}$. Intuitively, what should be the relative complexities of the three trees? Most people would agree that T_1 is the simplest, that is,

$$C(T_1) \leq C(T_2)$$
$$C(T_1) \leq C(T_3)$$

However, what is the relation between $C(T_2)$ and $C(T_3)$. Because T_3 has more nodes than T_2 we might classify it as more complex. On the other hand, T_3 is very symmetric. All of the subtrees of T_3 are identical at each level, making it very simple.

Hierarchical diversity is intended to measure the diversity of a structure and to reflect the number of interactions among nonidentical elements. For this reason, it counts T_2 as the most complex of the three trees in Figure 1.57. In fact, the tree T_3 can be extended to balanced binary trees of arbitrarily large size without increasing its hierarchical diversity.

We are now ready to define hierarchical diversity. The hierarchical diversity of a tree, $D(T)$, is defined recursively. The diversity of a node with no subtrees is just 1. The diversity of a node with subtrees is the made up of two terms. The first term counts the number of nonisomorphic subtrees of the node. We refer to each set of identical subtrees as a cluster. At each level, there are k clusters of identical subtrees. We are interested in counting the number of different interactions at each level, so all identical subtrees count as 1. Specifically, we multiply together the diversities of the clusters at the lower level. This product is then multiplied by N_k, the number of ways that the k different clusters can interact as given in equation (8).

$$N_k = \binom{k}{1} + \binom{k}{2} + \binom{k}{3} + \ldots + \binom{k}{k} = 2^k - 1 \tag{8}$$

Combining these factors we obtain equation (9)

$$D(T) = (2^k - 1) \prod_{j=1}^{k} D(T_j) \tag{9}$$

where j ranges over the k nonisomorphic subtrees.

Figure 1.58 gives examples of computing hierarchical diversity. The number in the box for each level of the tree gives the structural diversity at that node. The diversity of T_1 is trivial, since the node has no subtrees. Because the subtrees of T_3 are identical, there is only one cluster at each level. Thus, the diversity of T_3 is 1 and is less than the diversity of T_2. In T_4 and T_5, the magnitude of the diversity is much higher, because the product of the diversities at lower levels is higher.

It is easy to see that Huberman-Hogg complexity does correspond to the intuitions about physical complexity by comparing the values it yields for different trees. We have already ob-

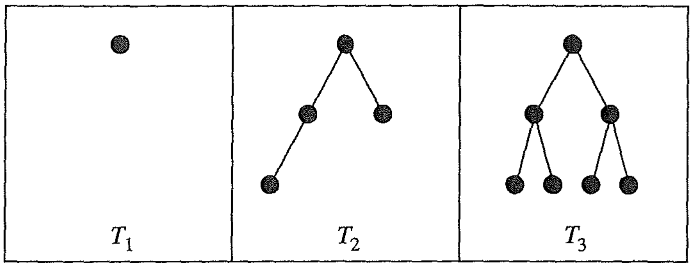


FIGURE 1.57. Some examples of hierarchical structures of differing complexity. By the hierarchical complexity measure, T_2 is the most complex of the three trees.

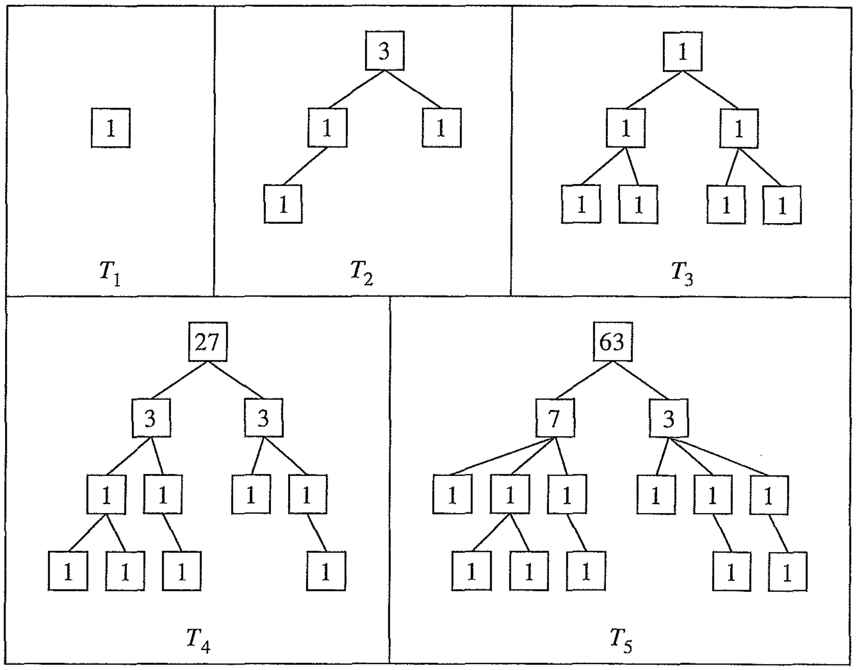


FIGURE 1.58. Examples of computing the hierarchical diversity of trees. The trees T_1, T_2, and T_3 are repeated from Figure 1.57, with the points indicating nodes replaced by boxes containing the diversity at that node in the tree.

served that it has a low value for trees with much symmetry, because the number of nonisomorphic subtrees of a node will be low. Indeed, for complete trees with uniform branching factors like T_3, it is easy to show that $D(T) = 1$. Similarly, hierarchical complexity is low for trees of trivial depth. It is highest for trees like T_4 and T_5, where there is substantial asymmetry. As Huberman and Hogg put it, the measure is optimum at a place midway between order and disorder. Although this goes beyond the scope of this section, Huberman-Hogg complexity and some variants of it are involved in hypotheses about the complexity of systems that are adaptable and are relevant to the study of machine learning systems.

In summary, the point of a structural complexity measure is to summarize intrinsic properties of structures on a scale so that we can compare them. When structural complexity measures are mathematically related to the time complexity of algorithms that run over the data structures, they can be used to partition the space of data structures into cases of graded difficulty.

1.3.10 *Summary and Review*

This section surveys computational issues in the design of representations. It introduces a vocabulary of dimensions for comparing alternative representations for computational models.

We compared symbol structures and their interpreters for representing a square: a pixel array, a property list, and a graph representation. These representations provide alternative implementations of a common abstraction for figures, with operations for such things as retrieving size and position, moving figures, and computing overlap of figures. Different implementations can have dramatically different computational requirements for time and space.

When we design representations, we decide which propositions to make primitive and which derived. This theme pervades our discussion of vivid and analogical representations. Thus directed graphs model the topology of molecules or the connectivity of circuits. Maps, like other scale models, have the same kinds of structures as their domains, that is, two-dimensional Euclidean spaces.

Representations are explict when they are written in a declarative language. This means that they are modular, that they have well-characterized semantics, and that there is a causal connection between the representation and the system that interprets it. The declarative quality of a representation depends not only on the symbols themselves, but also on the people who need to understand it and the complexity of what is described. Explicit representations are widely used because they afford advantages for making changes to systems.

When computational models are sufficiently expressive, they may admit several expressions of the same thing. In such cases we need to consider canonical functions and canonical forms.

It is not necessary to select and use a single representation in knowledge systems. When different implementations offer trade-offs in efficiency, multiple representations can be usefully combined. This requires maintaining an appropriate causal connection between symbol structures, and careful accounting to invalidate and recompute caches at appropriate times.

Parallel computing architectures offer opportunities to include both structure and process in analogical and direct representations. In the wind flow example, not only is the structure of the data similar to structure of the domain, but also the activity of the processors is similar to the activity in spatial regions of the domain.

Finally, we introduced the big-oh notation for summarizing asymptotic space and time complexity. The worst-case complexity is often the easiest to compute but need not be representative of the average complexity. Indeed, even average complexity can be misleading about the expected complexity if the average is skewed by a small number of cases of very high complexity. A more meaningful presentation of expectations is sometimes given by a distribution of complexities over a population of structures. A structural complexity measure is any measure based on the size or organization of a symbol structure. Measures of structural complexity are closely related to measures of time complexity for algorithms that run on the structures.

Exercises for Section 1.3

Ex. 1 [*CD-05*] *Systematic Representations.* Professor Digit was fascinated when he learned about the physical symbol system hypothesis, and dashed off to build several computer programs that he called "minimal symbol systems." He later exhibited these to his colleagues. The first minimal symbol system was a single instruction which compared two bits ("the input bits") in memory and then set a bit ("the output bit") in computer memory from 0 to 1 depending on the results. When his colleagues asked what his program did, he told them that it was an ultrafast multiplication program: When both of the input bits were 0, they represent the numbers 31415926535897932384626 43 and 987654321987654321. In that case, the program sets the output bit to 0, indicating the product of the two numbers. If either of the two input bits is one, the program sets the output bit to 1 meaning that it does not know the input numbers and cannot compute the product. His colleagues were puzzled (flabbergasted?) by this, but he assured them that this his interpretation was reasonable because a bit can designate anything whatsoever.

(a) How many input and output states does his program have?

(b) In what way is Professor Digit's account unsatisfactory? (*Hint*: What do we reasonably require of a computer program that reasons about operations on numbers?)

Ex. 2 [*CP-10*] *Computing about Rectangles.* In this chapter we considered examples of representation where the choice of primitives or symbol structures was intended to make some inferences computationally effective. Manhattan rectangles are rectangles whose sides are parallel to either the *x*- or *y*-axes in the real plane. Such rectangles can be represented as a record with the fields (left, bottom, width, height).

Two rectangles are said to overlap if their areas have at least one point in common. Describe an algorithm for deciding efficiently whether two rectangles overlap. (*Hint*: It is possible to consider area-overlap in terms of *x* and *y* components.)

Ex. 3 [*CP-15*] *More Computing about Rectangles.* The Manhattan separation between rectangles can be defined as the minimum length of a segmented Manhattan line between the closest edges of the rectangles, where the edges of the line are confined to integer coordinates.

Briefly, sketch an algorithm for deciding whether two rectangles are within a Manhattan separation of *k* units.

Ex. 4 [CD-!-*CP-30*] *Parallel Computing about Rectangles.* The claim that bitmaps are inefficient for determining whether two Manhattan rectangles overlap depends on several assumptions about the rectangles and the capabilities of the computer.

Suppose we have a bitmap processing computer (called SAM for "synchronous active memory"). Each processor has 16 bits of addressable memory, a 1-bit accumulator, and 1-bit wide communication lines to each of its four neighboring processors. The instructions can communicate with a host computer. The host computer can select arbitrary combinations of rows and columns of the bitmap processors. All the selected bitmap processors perform the same instruction at the same time, as broadcast by the host computer. The processors have the following instruction set:

Load and store instructions	
`store <bit>`	`<bit>:=Acc`
`load <bit>`	`Acc:=<bit>`
1-bit operations	
`clr–`	`Acc:=0`
`and <bit>`	`Acc :=Acc` *and* `<bit>`
`gt <bit>`	`Acc:=Acc` *greater-than* `<bit>`
`lt <bit>`	`Acc:= Acc` *less-than* `<bit>`
`xor <bit>`	`Acc:=Acc` *xor* `<bit>`
`or <bit>`	`Acc:=Acc` *or* `<bit>`
`nor <bit>`	`Acc:=Acc` *nor* `<bit>`
`cmp <bit>`	`Acc:=Acc` *equals* `<bit>`
`not <bit>`	`Acc:=`*not* `<bit>`
`ge <bit>`	`Acc := Acc` *greater-than-or-equals* `<bit>`
`inv–`	`Acc:=`*not* `Acc`
`le <bit>`	`Acc:=Acc` *less-than-or-equals* `<bit>`
`nand <bit>`	`Acc:=Acc` *nand* `<bit>`
`set–`	`Acc:=1`

Neighbor communication operation

`dot <mask>` $\quad \mathtt{Acc}:=\mathtt{Mask}_0\mathtt{Nbr}_0+\mathtt{Mask}_1\mathtt{Nbr}_1+\mathtt{Mask}_2\mathtt{Nbr}_2$
$\quad +\mathtt{Mask}_3\mathtt{Nbr}_3+\mathtt{Mask}_4\mathtt{Acc}$

Host communication operations

```
read-          Host-Data:=Acc (ored over selected cells)
write-         Acc:=Host-Data (for all selected cells)
```

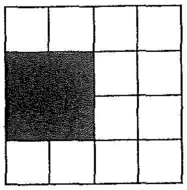
Note: The dot instruction is a kind of dot product. Each mask bit indicates which neighbor's accumulators are included in the result. For example,

dot (0 3) means to OR together the accumulators from neighbors 0 and 3 and save the result in the accumulator.

dot (1 2 4) means to OR together the accumulator values from neighbors 1 and 2 together with the accumulator of self and save the result in the accumulator.

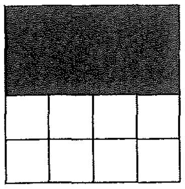
(a) Suppose the initial data in bitplanes 1 and 2 for a 4x4 SAM array are as follows:

Bit 1



Rectangle 1

Bit 2



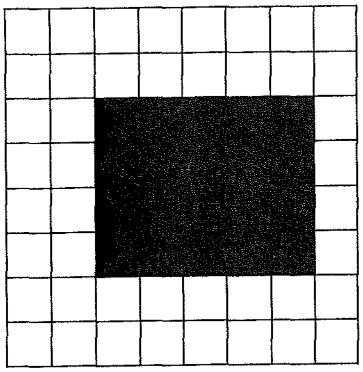
Rectangle 2

The following algorithm computes the intersection of the rectangle stored in bit 1 (across the entire block) with the rectangle stored in bit 2.

```
load 1
and 2
read
```

Illustrate the action of this algorithm using matrices to show the state of bit 1, bit 2, and the accumulator at each step. How does the value read on `Host-Data` depend on the rectangles?

(b) Suppose that bit 0 in an 8x8 SAM memory is initialized with a large filled-in rectangle as follows:



Trace the execution of the following program.

```
load 0
inv
dot (0 1 2 3 4)
inv
xor 0
```

Explain how this is an edge-finding program, finding those points at the outer limits of the rectangle. *Hint*: Consider de Morgan's laws.

Note that the bits in the mask of the dot instruction correspond to the directions of the neighbors with 0=North, 1=East, 2=South, and 3=West. Cells at the edge of the array always return 0 when they try ask for data beyond the array.

(c) Is the computation in part (*b*) an example of a local computation or a global computation? Explain briefly.

(d) Write a SAM program to compute whether rectangles stored in bit 1 and bit 2 are at least 1 unit of Manhattan separation apart. Does your program depend on the shape of the figures (that is, must they be rectangles)?

Ex. 5 [*10*] *Canonical Forms*. Each of the following data structures represents a circular linked structure of six items.

1. (a f b c e d)
2. (b c e d a f)
3. (b f a c e d)
4. (d e c b f a)
5. (c e d a b f)

(a) Suppose the interpreter is supposed to treat rotation and reflection is equivalence-preserving operations. The canonical form for these lists is defined as the list that is earliest "alphabetically" starting with the left-most term. In other words, the leftmost element of the canonical form of the lists is its alphabetically earliest one. In the event of a tie, then the second element is considered following the simplest rules of dictionary order.

Give the canonical form for each structure.

(b) Indicate which of the structures are equivalent to each other. How many distinct structures are represented here?

Ex. 6 [*05*] *Terminology*. Determine whether each of the following statements are true or false. If a statement is ambiguous, explain your answer briefly.

(a) *True or False*. Fidelity and precision refer roughly to the accurateness and detail of a representation in its use in making predictions.

(b) *True or False*. There is usually exactly one best way to implement a given abstraction.

(c) *True or False*. In most inferential systems, a large number of propositions can potentially be derived from a small number of primitive ones.

(d) *True or False*. One of the advantages of using multiple representations is that such systems need not contend with the stale data problem.

(e) *True or False*. In a system that is implemented in multiple layers, the implementation at one level of description may in turn be an abstraction for the next level down.

Ex. 7 [*05*] *More Terminology*. Determine whether each of the following statements is true or false. If a statement is ambiguous, explain your answer briefly.

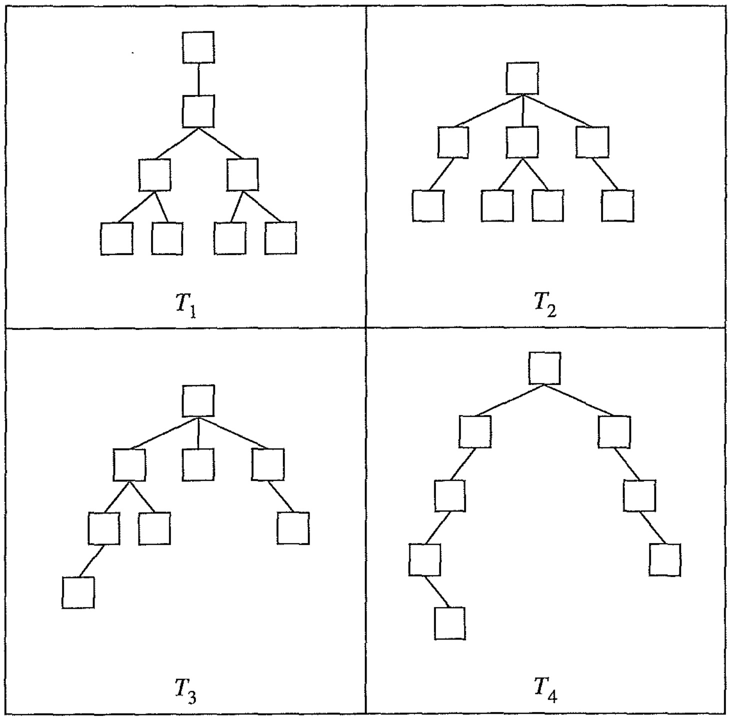


FIGURE 1.59. Three trees.

(a) *True or False.* When a system has two representations for the same thing and one is derived from the other by a long computation and the derived result is referenced more frequently than the primitive one is changed, it is appropriate to cache the derived representation.

(b) *True or False.* To prevent stale data problems, a cache should be invalidated whenever its value is retrieved.

(c) *True or False.* It is appropriate to recompute a derived and cached value whenever an operation changes a primitive value on which it depends.

(d) *True or False.* One problem with the bitmap representation of figures is that for low enough precision or resolution of the bitmap, the shapes of some figures are indistinguishable.

(e) *True or False.* If the running time of a computation is $56m^2n^3 + 37m^3n + 60m^2n + 3n + 30$, then we can characterize its time complexity as $O(m^2n^3)$.

Ex. 8 [*05*] *Even More Terminology.* Determine whether each of the following statements is true or false. If a statement is ambiguous, explain your answer briefly.

(a) *True or False.* Explicit representations offer advantages over implicit ones in knowledge systems that must be modified over time.

(b) *True or False.* An implicit representation is one whose utility rests on unstated assumptions.

(c) *True or False.* A bitmap is an implicit representation of shapes.

(d) *True or False.* An explicit representation usually requires a declarative semantics.

(e) *True or False.* The declarative semantics (including reference semantics, truth semantics, and proof semantics) are the only semantics needed for defining a declarative representation.

Ex. 9 [*05*] *Asymptotic Superlinear Speed-ups.* Is it possible to achieve more than a factor of k order speed-up on a problem by applying k processors instead of one, assuming that all processors are equivalent? If yes, give a brief example. If not, present a brief argument.

Ex. 10 [*05*] *Space versus Time Complexity (True or False).* The asymptotic time complexity of an algorithm for a single processor is always greater or equal to its asymptotic space complexity. Explain briefly.

Ex. 11 [*08*] *Hierarchical Diversity.* Figure 1.59 gives four unlabeled trees of eight nodes each. Compute the hierarchical diversity of each tree.

1.4 *Programs: Patterns, Simplicity, and Expressiveness*

Whenever we build computational models, we express them in programming languages or representation languages. Observably, some languages are better than others for particular programs in that the programs are generally more concise, easier to understand, and easier to extend. But what makes some languages and representations better than others in this way?

In the previous section on representational properties, we considered the use of programming abstractions. Programming abstractions are used to build systems in levels, so that the most abstract level gives a concise description of a program in terms of appropriate primitives.

In this section we consider two additional methods for achieving simplicity of expression: metalevel factoring and the use of epistemological primitives with a large base of shared knowledge. We introduce these concepts through a sequence of increasingly sophisticated examples. Along the way we consider basic ideas widely used in building knowledge systems: production rules, pattern matching, and logic programming.

1.4.1 *Using Rules to Manipulate Symbols*

Our first example is quite elementary for anyone who has written many computer programs. It is a starting point for comparisons.

Figure 1.60 depicts a traffic intersection where a farm road crosses a highway. The right-of-way at this intersection is indicated by a traffic light, which has a sensor to detect traffic entering the highway from the farm road, assuming that traffic drives on the right-hand side of the road.

Figure 1.61 depicts two internal symbol structures for the controller of the traffic-light system. There is a cell or storage element named `farm-road-light` that designates the traffic light facing the farm road. This cell contains one of three symbols `red`, `yellow`, or `green`. Similarly there is another cell, `highway-light`, designating the traffic light facing the highway. It is convenient to think of these cells as program variables, and the symbols in the cells as the values of the variables.

Symbol systems change symbol structures dynamically. Our account of the traffic-light controller is a **simulation**. That is, it uses a running computer program to model another (usually more complicated) system that we call its "world." The values in the program variables correspond to a snapshot of the world at some instant of time. As the program runs, its program actions model world actions that would change it over time. An observer could watch the simulation just as he might watch the world, seeing the traffic light on a highway turn yellow, and then red, and then a light on a farm road turn green.

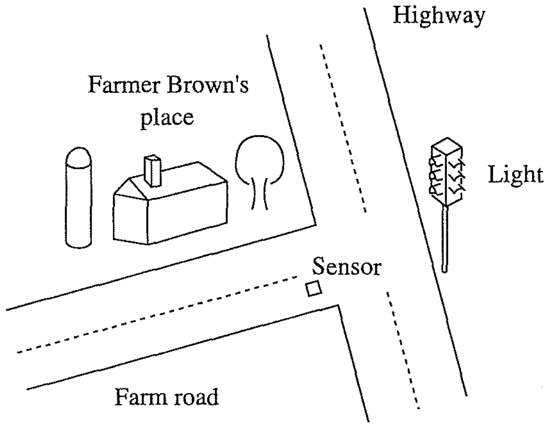


FIGURE 1.60. Traffic light at an intersection of a highway and a farm road.

Figure 1.62 illustrates four computational "rules" that specify the behavior of a traffic-light controller. The particular syntax for the rules in this figure and others in this section is not important. Many variations are possible in programming languages.

The rules in our simulation example make use of two external timers called `light-timer` and `traffic-timer`, which can be started and tested to see whether an allotted time is up, and a sensor, which can be tested to determine whether a car is present at the end of the farm road. The code and variables for the simulated timers and sensor are not shown.

The rules in Figure 1.62 are examples of **production rules**. Each rule has two major parts, called the if-part and the then-part. The if-part of a rule consists of conditions to be tested. In the first rule, the conditions test whether the highway light is green, whether a car is waiting, and whether time is up on the traffic timer. If all of the conditions in the if-part of a rule are true, the actions in the then-part of the rule are carried out. In the idiosyncratic terminology of production rules, we say that the rule is "fired." The then-parts of these rules consist of actions to be taken. In the first rule, these actions turn the highway light to yellow and start the light timer. The parts of a production rule are sometimes called by different names. The if-part is also known as the situation part, the conditional or the antecedent. The then-part is also known as the action part or the consequent.

In our program, after the first rule is tried, the second, third, and fourth rules are tried in order. The program then starts over again with the first rule. The complete simulation program would have representations of cars and rules for modeling traffic. Cars could speed up, slow down, or turn. No two cars could occupy the same space at the same time (we hope!). The simulated sensor would occasionally indicate that a simulated car is present. Then the timers would perform their measurements of simulated elapsed time and the rules of our traffic-light controller

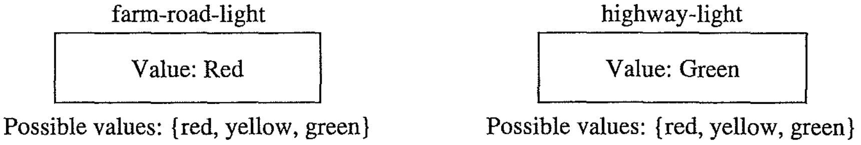

farm-road-light
Value: Red
Possible values: {red, yellow, green}
highway-light
Value: Green
Possible values: {red, yellow, green}

FIGURE 1.61. Symbol structures in the traffic-light controller.

```
do                                      ;infinite loop
begin

   begin                                ;Initially ...
      farm-road-light := 'red              ;turn the farm-road light red
      highway-light := 'green              ;turn the highway light green
      start (light-timer)                  ;start the light timer
      start(traffic-timer)                 ;start the traffic timer
      start (sensor)                       ;start the sensor
   end

  ;rule-1
  if     highway-light = 'green           ;IF the highway light is green and
         and car-waiting? (sensor)        ;and the sensor shows a car waiting
         and time-up?(traffic-timer)      ;and the traffic timer shows time up
  then   highway-light := 'yellow         ;THEN turn the highway light yellow
         start (light-timer)              ;and start the light timer.

  ;rule-2
  if     highway-light = 'yellow          ;IF the highway light is yellow
         and time-up?(light-timer)        ;and the light timer shows time up
  then   highway-light := 'red            ;THEN turn the highway light to red
         farm-road-light := 'green        ;and turn the farm road light to green
         start (traffic-timer)            ;and start the traffic timer.

  ;rule-3
  if     farm-road-light = 'green         ;IF the farm road light is green
         and time-up? (traffic-timer)     ;and the traffic timer shows time up
  then   farm-road-light := 'yellow       ;THEN turn the farm road light to yellow
         start (light-timer)              ;and start the light timer.

  ;rule-4
  if     farm-road-light = 'yellow        ;IF the farm road light is yellow
         and time-up? (light-timer)       ;and the light timer shows time up
  then   farm-road-light := 'red          ;THEN turn the farm road light to red
         highway-light := 'green          ;and turn the highway light to green
         start (traffic-timer)            ;and start the traffic timer.

end
```

FIGURE 1.62 Rules in the traffic-light controller.

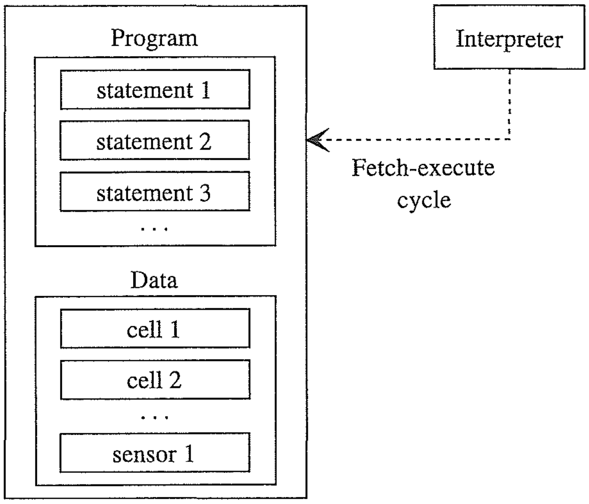


FIGURE 1.63. Program and interpreter for the traffic-light controller.

would cycle around, switching the lights to control the simulated flow of traffic at the intersection.

Production rules take many different forms. The rules shown in Figure 1.62 are similar to IF statements available in most programming languages. In this example, the selection of which rule to fire next is quite trivial. The program specifies exactly what to do, step by step.

Variations on production rules can admit many kinds of additional information, different syntaxes, and different methods for controlling execution. They are used to model simulation and also to model reasoning. Later in this section we will consider production rules that are based on pattern matching. Production rules are an important representation in knowledge-based systems because they directly represent how actions depend on conditions. Production rules are sometimes called **situation-action rules**.

Figure 1.63 shows the organization of the traffic-light controller system. There is a program made up of a set of statements—the rules from Figure 1.63. The statements in this program are retrieved and executed by an interpreter. This architecture is essentially the same as that of a von Neumann computer where the program is represented as instructions in computer memory and the interpreter is the processor which carries out a fetch-execute cycle. The difference in this case is that the statements are rules and the interpreter executes the rules. Nonetheless, the interpreter is still quite simple, in that the rules are executed in sequential order. The bottom part of the figure corresponds to data read or written by the program. The data are the program variables that store the state of `farm-road-light` and `highway-light`. The reading from the sensor is also included in data. These data are read and sometimes written as the program runs. By design, the program never directs its interpreter to decode the data as instructions.

1.4.2 *Treating Programs as Data*

The representation of programs as symbol structures is related to the **stored program concept**: the storing of computer instructions in memory in the same manner as data. Different codes are recognized by the processor as different instructions. Storing programs in memory has the practi-

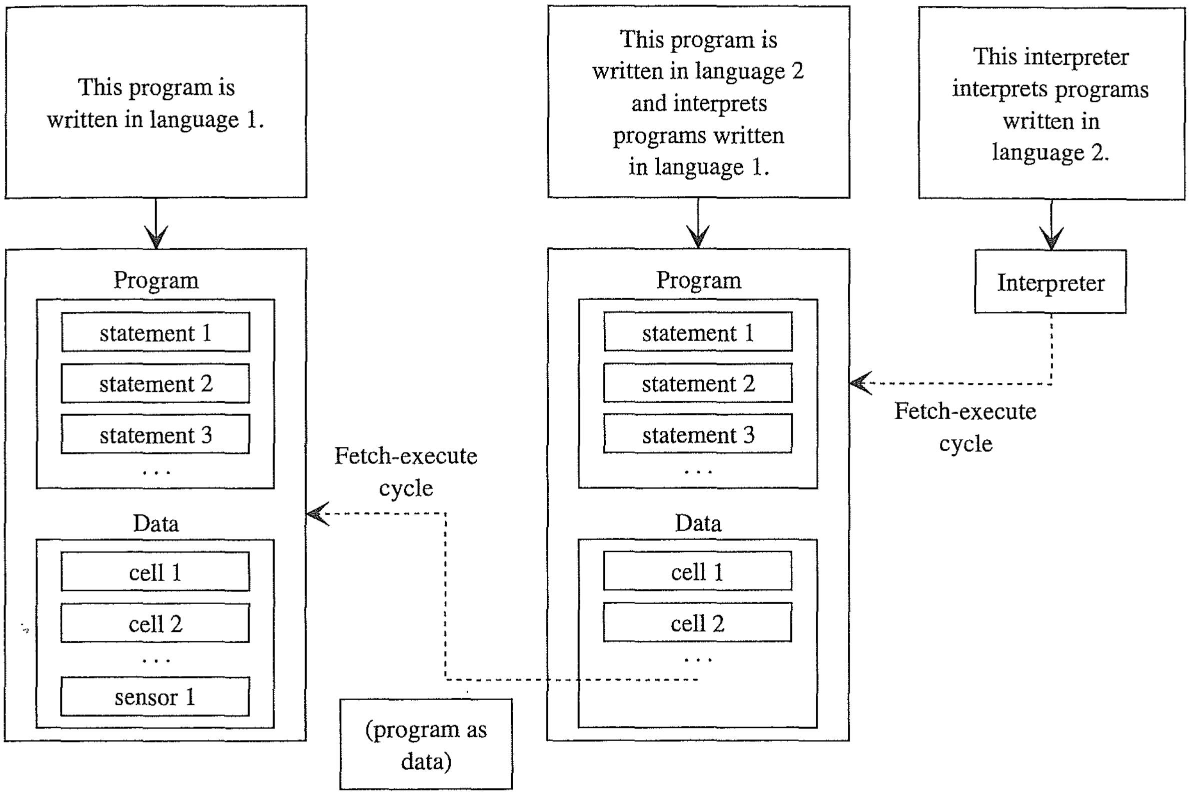


FIGURE 1.64. Universal computation, in which one computational process emulates another by interpreting its instructions as data.

cal advantage of making it easier to arrange for computers to run different programs. This is the major contribution of the von Neumann architecture for computers.

That programs on one computer can interpret symbolic instructions of a different computer is the **universal computation** concept. The "universal" part of this refers to the ability to simulate the instructions of *any* computer. A universal computer can be programmed to carry out the instructions of a different computer, even though its instruction codes are different. The basic idea as shown in Figure 1.64 is that the universal computer maintains a table to look up sets of instructions in its memory that simulate the instructions written for the other computer.

Universal computation places few constraints on the representations that are used in the two computers. The instruction sets on the two computers could be identical or very different. The symbols can have structural similarities or be very different. Let $\{s_1, s_2, \ldots, s_k\}$be symbols on the simulated computer, representing either instructions or data, and $\{S^1, S_2, \ldots, S_k\}$ be symbols on the simulating computer. There must be a mapping, M, between the two sets of symbols, stated as M: $\{s_k\} \rightarrow \{S_k\}$. Similarly there must be a mapping, P, between the processing steps or instructions on the two computers. Suppose that $\{i_1, i_2, i_3, \ldots, i_n\}$ is a sequence of descriptions of the symbols in the memory of the simulated computer when its interpreter is run. Universal computation requires that when the simulating computer runs, that it yields a sequence of states, such that M holds between corresponding symbols in each of the states in order.

As suggested by Figure 1.64, universal computation divides an interpreter into layers. This layered approach is not limited, say, to simulating obsolete computers that are no longer manufactured. It is widely used in knowledge systems to partition programs into simpler layers. At each layer, interpreters make much use of symbol manipulation operations for recognizing patterns, extracting parameters, and carrying out parameterized action.

In the following sections, we see concrete examples of how symbol manipulation languages provide symbol manipulation primitives and thereby layers of interpretation that can simplify the writing of programs.

1.4.3 *Manipulating Expressions for Different Purposes*

The layperson's view of computers is that they are good with numbers, that is, they are able to do arithmetic rapidly. In contrast to arithmetic, algebra and calculus involve much richer symbol manipulation. For example, a typical exercise in an introductory calculus course is to differentiate a polynomial like that in expression (1) to yield its derivative (2).

$$\text{function: } f(x) = (2/3)x^3 + 7x^2 + 12x - 8 \tag{1}$$
$$\text{derivative: } f'(x) = 2x^2 + 14x + 12 \tag{2}$$

The function in (1) can be represented indifferent ways, such as the LISP computer language, as shown in (3). The advantage of LISP in the following examples is that the patterns are simpler because the program syntax requires no parsing.

```
(defun f (x) (+ (* (/ 2 3) (expt x 3))
                (* 7 (expt x 2))
                (* 12 x)
                (- 8)))                                 (3)
```

Equation (1) could also be expressed in a graph notation, as shown in Figure 1.65. In either case, a symbol system (such as a standard LISP interpreter) can be made to compute the value of the polynomial given a value for *x*. A LISP interpreter would evaluate the equation in (1) to yield that *f*(0) = –8, *f*(1) = 11.666667, and *f*(.508517) = 0. A different program, a *differentiator*, could accept the definition of *f* as input and produce as output a symbolic expression *f1* representing its derivative:

```
(defun f1 (x) (+ (* 2 (expt x 2))
                 (* 14 x)
                 12))                                   (4)
```

This expression (4) is a full-fledged LISP program, as is the original function definition in (3). It could be executed or further differentiated.

Looking at the equivalent semantic network representation in Figure 1.65, it is easy to see how a differentiation program could work, that is, how it could symbolically differentiate a polynomial. A differentiation program needs to create a second linked structure of terms, computing the coefficient and exponent relations of the new terms from corresponding relations on the old terms. For each term, it creates a new term node, linking it to a new coefficient node, a new exponent node, and a new next-term. Each new coefficient is computed by multiplying the old co-

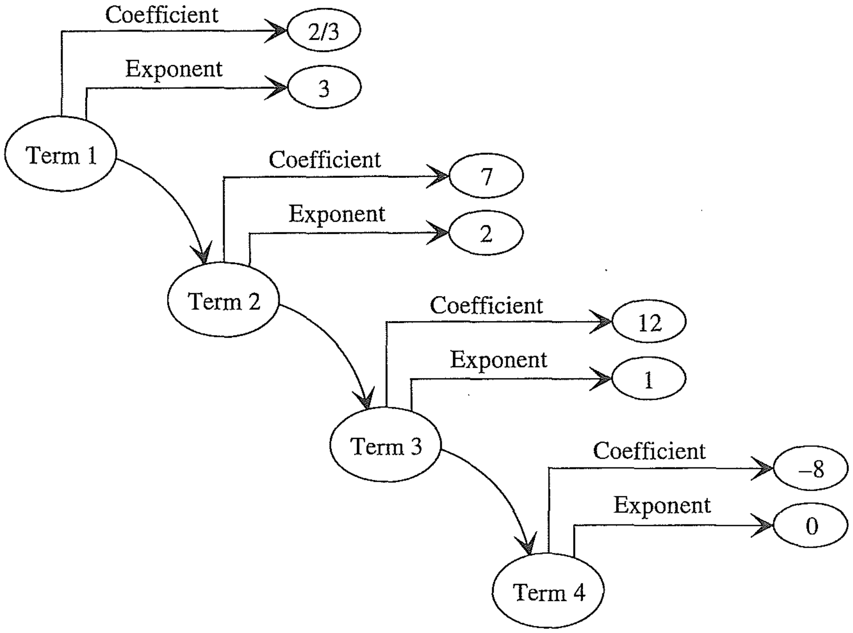


FIGURE 1.65. Graph representation of a polynomial.

efficient by the old exponent; the new exponent is the old exponent minus one. The new terms represent the new polynomial with the values as shown in (4). Thus, the differentiation process builds a new polynomial representation as it traverses the original polynomial representation, computing the parameters of each new term from the parameters of the old terms. It must walk through the graph and create a new graph whose properties are computed from those of the original graph.

Furthermore, different computations can be done on a single representation. Another symbolic manipulation program, a quadratic root finder, could be written to take the roots of a second-order polynomial such as (4). For this, we view a polynomial as fitting the following pattern

$$f(x) = ax^2 + bx + c \tag{5}$$

and then apply the quadratic equations

$$\begin{aligned} \text{root1} &= (-b + \text{sqrt}(b^2 - 4ac))/2a \\ \text{root2} &= (-b - \text{sqrt}(b^2 - 4ac))/2a \end{aligned} \tag{6}$$

A quadratic root finder steps through the terms of a polynomial and extracts values for the parameters a, b, and c. Given these parameter values, computing the roots as in (5) and (6) requires another modest procedure, not much more complicated than the preceding polynomial functions.

In summary, radically different computations can be performed on a representation. The polynomial representation can be used to yield specific evaluations for different values of x. Alternatively, a differentiator program operate on the same representation to produce derivatives.

Similarly, a root finder can use the representation as a template for extracting parameters for computing quadratic roots.

1.4.4 *Pattern Matching*

In the examples of the last section, the symbol-manipulating operations included traversing graphs, extracting information, and creating new expressions. Because structure recognition and parameter extraction are so fundamental, people have designed computer languages that provide these operations as primitive computational mechanisms. Such mechanisms are widely used in production-rule and logic programming languages. This section uses these operations to give concrete examples of how appropriate symbol manipulation primitives can simplify the expression of programs.

There are two main variations on pattern matching, notably one-way pattern matching and two-way pattern matching (unification). In one-way pattern matching, a pattern with variables is matched against an expression containing constants. At the end of the operation, the variables in the pattern may take on values corresponding to parts of the constant structure. In **unification**, patterns with variables are matched against other patterns with variables. Variables in both patterns may take on values at the end of the matching operation.

Figure 1.66 shows a simple example of a pattern and the results of matching it against an expression. A key element in the syntax of pattern matching is the specification of **pattern variables**. In the following, there are three pattern variables to be matched: *a*, *b*, and *c*. In the following examples, we indicate that terms are variables to be matched by preceding them with question marks at the parts of the pattern where matching is to take place. Other terms are assumed to be constants in the matching process. When a pattern variable appears again later in a pattern-matching rule it need not be preceded by a question mark. The value of the variable is required to match the corresponding datum in the matching expression.

Matching a pattern against an expression requires aligning the corresponding parts of the pattern with the expression and then extracting values for the pattern variables. Thus, the pattern matching process assigns values $a = 2$, $b = 14$, and $c = 12$.

Matching the pattern

```
    (+ (* ?a (expt x 2))
       (* ?b x)
       ?c)
```

to the expression

```
    (+ (* 2 (expt x 2))
       (* 14 x)
       12))
```

yields values for the pattern variables

```
    a = 2, b = 14, c = 12
```

FIGURE 1.66. An example of pattern matching.

if-part: *{pattern to be matched}*

```
    (if (match test-expression                   ;IF the test expression matches
         (+ (* ?a (expt x 2))                    ;a quadratic polynomial
         (*    ?b x)
               ?c))
```

then-part: *{actions to be carried out}*

```
    (then (setf root (/ (+ (- b)                  ;THEN compute a root
                           (sqrt (- (* b b)
                                    (* 4 a c))
                        (* 2 a))
```

FIGURE 1.67. Pattern-matching rule for computing the discriminant of a second-order polynomial.

Figure 1.67 shows an example of how pattern matching can be used with a production rule. If the pattern matches the test expression, then the parameter values are extracted and a root of the polynomial can be computed.

To simplify programming, many production-rule languages and logic-programming languages provide an automatic way of searching for candidate expressions. The job of the interpreter is deciding how to traverse a database to find candidate expressions for possible matching. For example, the following three production rules do most of the work for differentiating polynomials:

```
rule 1: (deriv (* ?a (expt x ?n))) -> (* n a (expt x (n-1)))
rule 2: (deriv (+ ?u ?v)) -> (+ (deriv u) (deriv v))
rule 3: (deriv (- ?u ?v)) -> (- (deriv u) (deriv v))
```

These rules are examples of a rewrite or **substitution language**. In such a language, there is a working memory containing one or more expressions. Patterns are matched against these expressions. If they match, then the rule replaces the expression with a new one. Multiple rules correspond to different possible patterns in memory, treated as distinct cases. For rules in this format, the if-part is often called the **left side** and the then-part is called the **right side**.

Figure 1.68 dissects the first rule for our analysis. Given an appropriate interpreter, this rule specifies how to differentiate a single term in a polynomial: Create a new term such that the coefficient is the coefficient of the matching term times its exponent, and the exponent is the exponent of the matched term minus one.

if-part: *{pattern to be matched}*

```
    (deriv (* ?a (expt x ?n)))      ;IF there is an expression matching ?a x^?n
```

then-part: *{rewrite actions to be carried out}*

```
    (* n a (expt x (n-1)))          ;THEN replace it with na x^(n-1)
```

FIGURE 1.68. Rewrite production rules combine pattern matching and action with search.

Initial expression

```
    (deriv (+ (* (/ 2 3)(expt x 3))
              (* 7 (expt x 2))
              (* 12 x)
              (- 8)))
```

Step 1: Apply rule 2 to the sum of terms

```
    (+ (deriv (* (/ 2 3)(expt x 3)))
       (deriv (* 7 (expt x 2)))
       (deriv (* 12 x))
       (deriv (-8)))
```

Step 2: Apply rule 1 to the first term

```
    (+ (* 2 (expt x 2))
       (deriv (* 7 (expt x 2)))
       (deriv (* 12 x))
       (deriv (-8)))
```

Step 3: Apply rule 1 to the second term

```
    (+ (* 2 (expt x 2))
       (* 14 x)
       (deriv (* 12 x))
       (deriv (-8)))
```

After a few more steps we have the derivative

```
    (+ (* 2 (expt x 2))
       (* 14 x)
       12)
```

FIGURE 1.69. Rule applications in the differentiation of a polynomial.

Figure 1.69 shows how these rules could be applied in the differentiation of a polynomial. In the first step, rule 2 is applied to change the derivative of a sum to the sum of some derivatives. Our single-step application of the rule operates on all of the terms at once. Alternatively, rules could be written that convert a sum of *n* terms into one term plus the sum of the remaining terms. In the second step, rule 1 is applied to differentiate the first term. This process continues until no more rules match and the polynomial is completely differentiated. For simplicity we have omitted some intermediate rules and steps for simplifying terms. For example, in addition to the steps in Figure 1.69, `(expt x 1)` is reduced to `x`, and `(* 2 7)` is reduced to `14`.

Pattern-matching languages enable us to write very short programs. The first advantage is that the program does not need to specify in detail how to extract information. It need only specify the patterns to be matched and the use to be made of the extracted information. Much of the programmatic detail is supplied by the language interpretation program, which searches the graph, matches patterns against expressions, extracts parameters, and builds new structures with substituted information.

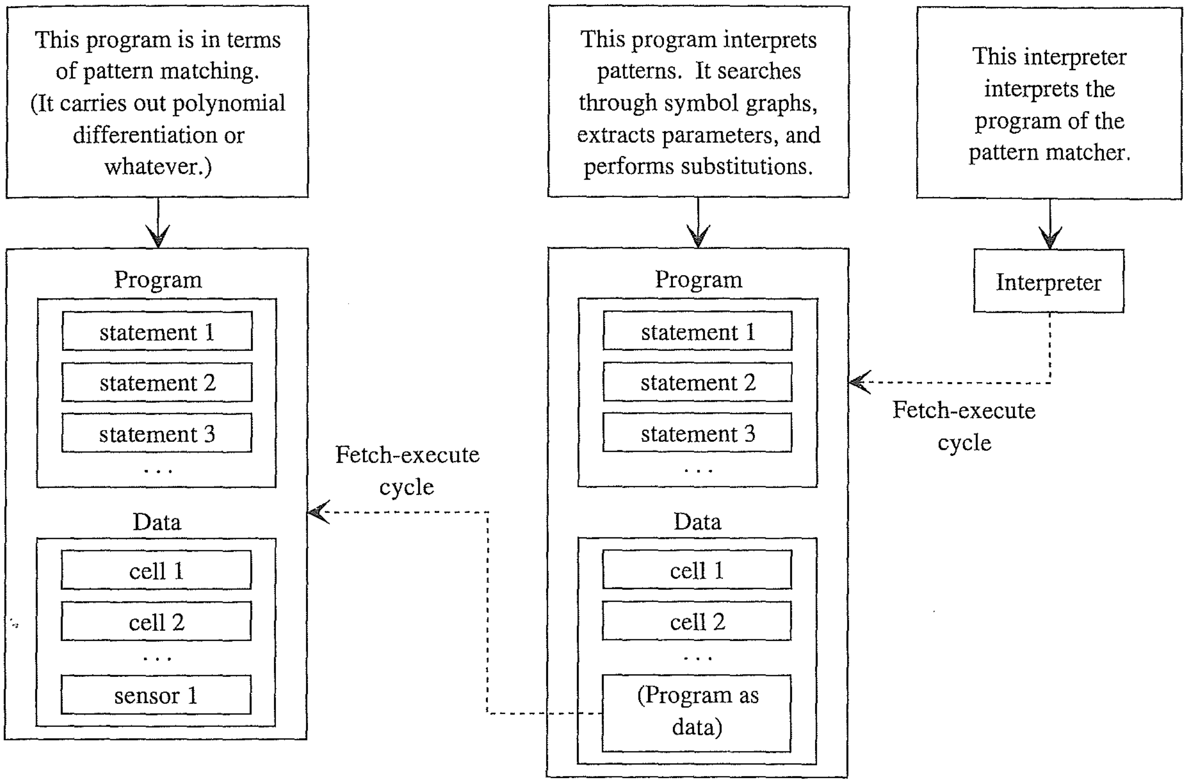

This program is in terms of pattern matching. (It carries out polynomial differentiation or whatever.)
Program
statement 1
statement 2
statement 3
...
Data
cell 1
cell 2
...
sensor 1
Fetch-execute cycle
This program interprets patterns. It searches through symbol graphs, extracts parameters, and performs substitutions.
Program
statement 1
statement 2
statement 3
...
Data
cell 1
cell 2
...
(Program as data)
This interpreter interprets the program of the pattern matcher.
Interpreter
Fetch-execute cycle

FIGURE 1.70. An example of metalevel factoring. In pattern-matching languages, programs can be terse because much of the work of graph search, matching, parameter extraction, and substitution is done automatically.

This approach in pattern-matching language shows an important way that conciseness can be achieved in programming: metalevel factoring. This conciseness is possible due to the partitioning of interpretation into two levels. As illustrated in Figure 1.70, one level of interpretation is carried out by the rules in pattern matching. This process differentiates polynomials, finds roots, and so on. The symbolic description of this program is simple, largely, because so much of the work is done by the next interpreter: the processor that interprets the pattern-matching rule program. That processor selects which rule to fire, locates candidate symbolic expressions to consider, tests candidate expressions to determine whether they match the patterns given in the if-parts of rules, extracts the values of pattern variables, and rewrites expressions according to the actions in the then-parts of the rules.

1.4.5 *Expressiveness, Defaults, and Epistemological Primitives* ADVANCED

The preceding discussion of symbol manipulation languages emphasizes simplicity of expression. In this section, we extend the discussion from programming languages to representation languages. The boundary between programming and representation languages is not sharp since many ideas that started out as part of representation languages are now part of programming languages and conversely. Roughly, representation languages represent facts in a way that enables their use for many different possible purposes. A **representation system** includes a database containing facts expressed in the representation language plus an interpreter that provides answers to queries. In this section, we consider what it means for representation languages and query languages to be expressive.

Our perception of the simplicity of computational models depends on being able to describe them in simple terms and on being able to leave many things unsaid. We begin with a characterization of representation in terms of predicate calculus and then consider other well-known languages.

In 1985, Hector Levesque and Ronald Brachman (Levesque & Brachman, 1985) called attention to what they described as a tradeoff between tractability and expressiveness. They asked: What is the appropriate service to be provided by a knowledge representation system? They offered a strawman proposal. A knowledge representation (KR) system manages a knowledge base and presents a picture of the world or an environment based on what it represents. Given a knowledge base, they required that a KR system be able to determine, given a sentence, Z, whether that sentence is believed to be true. More precisely, assuming that the knowledge base is a finite set of sentences and that KB stands for their conjunction, they wanted to know whether KB => Z. Determining this generally requires not just retrieval capabilities but also inference.

Their strawman proposal was to use the first-order predicate calculus as the exemplary representation language. This has the advantage that

$$KB \models Z \qquad \text{iff} \qquad \vdash (KB \Rightarrow Z)$$

That is, if Z is true then it is derivable from the sentences in the knowledge base. Unfortunately, this requirement also has the effect of ensuring that the strawman service as described cannot be provided.

The difficulty of providing this service follows from its generality. No theorem prover can always determine the answer for arbitrary sentences in reasonable time. Completeness requires generality and brings a price of intractability. This is not to say that finding a proof or counterexample is intractable for every sentence. For most theorem provers there are classes of problems that are easy. Many theorem provers also have worst cases that are much more time-consuming than the average cases and the easy ones. If theorem provers are complete then it follows that some classes of problems are intractable.

Knowledge systems use a wide range of representations with different expressiveness. One dimension along which representations differ is the degree to which they can leave things *unsaid* about a model. There are different ways that a language can be incomplete in what it can say.

Consider the logic sentences in Figure 1.71. The first sentence says John is not a female without saying what he is. The second sentence says that either Jack or Monica is a parent of Steve, but does not specify which. Nor does it say that some other person might not be a parent of Steve, since nothing has been said about a person having exactly two biological parents. The third sentence says that Steve has at least one male parent but does not say who that parent is. The fourth sentence says that all of Steve's neighbors own pets without saying who those neighbors are, who their pets are or even whether there are any neighbors. These examples show that first-order logic can leave many kinds of details unspecified. It avoids representing details that may not be known. Indeed, this ability to not specify details is quite remarkable. It reflects the original intended use of predicate calculus to formalize infinite collections of mathematical enti-

$\neg$ Female (John)
Parent (Steve, Jack) $\vee$ Parent (Steve, Monica)
$\exists$ x Parent (Steve, x) $\wedge$ Male (x)
$\forall$ x Neighbor (Steve, x) => $\exists$ y Owns-Pet (x y)

FIGURE 1.71. Example sentences with different kinds of incomplete information.

ties. This use emphasizes different requirements from typical applications of knowledge systems where domains are often finite and many of their details are explicitly represented.

Following the exposition in Levesque and Brachman, 1985, we consider incomplete knowledge and inference for a relational database, a logic program, and a semantic network. As we consider KR systems based on these languages, we will see not only that they cannot perform the strawman KR service, but also that they necessarily must perform different though related services. Our goal in the following is not to give a thorough characterization of relational databases, logic programming, or semantic networks. Rather, it is to sketch them briefly and compare them to reveal general issues about the expressivenesss of representations.

Representation and Inference Using a Relational Database

Databases store facts. In this section we consider examples from a relational database to show limitations that the format imposes on the kinds of incompleteness in represented facts. In principle, it might be argued that a database imposes no limitations on information because it is a general storage medium. In this argument, the interpretation of symbols is entirely open-ended. For example, we could use a database to build exactly one table that would just be addresses and memory contents for an arbitrary computer program. However, this misses the intent of the mechanisms provided with databases.

Databases have mechanisms for matching and retrieval and these mechanisms are used in a well-defined way by the query language. These are similar in purpose to the pattern-matching operations we discussed earlier for symbol-manipulation tasks. When we speak of using a database and having a "natural" or "standard" interpretation, we presume to make the most use of the query language and the retrieval mechanisms.

Figure 1.72 lists some entries from a relational database.

To characterize in first-order logic the information in these sentences, we can use a collection of function-free atomic sentences such as the following:

Course(CS131) Course(CS112) Enrollmen (CS131, 60)
Days(CS112,TTh) Days(CS228A, TTh) Cross-Listin (CS275, Ling120)

The expression of incompleteness is much more limited than in our earlier examples from predicate calculus. For example, we have no sentences like the following.

$\neg$ Time(CS228A, 11) Days(CS112, TTh)$\vee$ Days(CS112, MWF)

Course	*Name*	*Days*	*Time*	*Enrollment*	*Instructor*	*Cross-Listing*
CS131	Applied Linear Algebra	MW	11	60	Golub	
CS275	Computational Linguistics II	TTh	3	10	Kay	Linguistics 120
CS112	Computer Organization	TTh	11	40	Gupta	
CS228A	Intro to Knowledge Systems	TTh	4	50	Stephic	
CS237A	Advanced Numerical Analysis	MWF	10	40	Staff	
CS260	Concrete Mathematics	MWF	9	35	Knuth	
	. . .					

FIGURE 1.72. Example entries from a relational database.

Furthermore, the standard interpretation of a database contains more information than is typical of a collection of sentences from first-order logic. Suppose we were to ask the question,

"How many courses are offered at 11 o'clock?" (7)

Assuming that the database includes only courses in the Computer Science Department, question (7) might be answered by a query (8) such as

Count c in Course where $c.\mathit{Time} = 11$ (8)

This query asks about the entries in the database. The relation between those entries and the world is in the mind of the questioner. It is also supported by various assumptions that the query system makes in its processing.

Returning to question (7), a logical account of what the query should return must use information not explicitly stored in the database. For example, it must determine which literal atoms correspond to distinct course names. In the simplest case, unique atom names correspond to unique courses. To indicate this with theorems would require us to add a statement expressing explicit inequality for every pair of constants. Such theorems are not represented explicitly in the database. However, even if we did that, it would not quite work in this case. Course CS275 is cross-listed between the departments of Linguistics and Computer Science, resulting in two course numbers for the same course. Thus, the rule for determining unique courses would need to be expanded to something like "unique atoms declared to be courses correspond to unique courses unless declared otherwise by an explicit Cross-Listing relation." Another important assumption used in answering query (8) to the database is that all of the courses that are offered are listed in the database. This is an example of a **closed-world assumption** (CWA). By embodying these assumptions, the query system is able to answer a question about the world using a computation on its database.

An important issue in processing queries is in finding answers quickly. In the context of databases this topic is called query optimization. Suppose we wanted to answer the following question:

"How many courses in the Computer Science Department are offered at 11 o'clock
by graduates of MIT who also teach in the medical school?" (9)

In searching for an answer to such a query there are several choices about how to search the database. Should the system first identify all of the instructors of 11 o'clock classes in the Computer Science Department? Alternatively, it could identify all instructors who graduated from MIT or all instructors who teach both at the medical school and in the Computer Science Department. Query optimization employs a computational model of a database based on the times required by different database operations. Query optimization is concerned with the order that different fields are indexed and the size of sets on which various operations are performed. Although a discussion of the methods of query optimization is beyond the purposes of this section, we notice that such concern with the efficiency of answering a particular query is an important practical concern. Unfortunately, it is also outside the scope of the strawman KR service. Knowledge about query optimization is another example of knowledge we like a knowledge system to have but that we would not need to specify. In this case, the knowledge is unsaid and is unsayable in the knowledge base because it is handled by the query optimizer, which is a separate program. Beyond trivial problems, knowledge about how to make inferences efficiently is crucial.

In summary, inference is carried out by database operations and accessed through a query language. Normally, a query language makes certain assumptions about the way that symbols are used to represent the world. Knowledge for making inferences efficiently is held in the query system.

Representation and Inference Using Logic Programming

In this section we give a short introduction to logic programming to show how it fits in this sequence of representations with different expressiveness.

In logic programming a knowledge base is a collection of first-order sentences. They have the form:

$$\forall\ x_1\ x_2\ \ldots\ x_n\ [P1 \wedge \ldots \wedge Pm \Rightarrow P_m+1]$$

where each P_i is atomic and $m \geq 0$. (10)

Sentences in this form are called **Horn clauses**. Horn clauses are more general than sentences in a relational database. In the case where $m = 0$ and the arguments to the predicates are all constants, this reduces to the form of a relational database.

Figure 1.73 gives some examples of sentences in PROLOG. Each sentence ends with a period. The first six sentences are function-free atomic sentences as with the relational database examples. They say that "Paige is female" and "Morgan is male" and so on. The next four sentences are similiar except that they involve a 3-ary predicate.

The second to the last sentence defines `brother` as a **derived relation**. Roughly, it says that "someone is my brother if he has the same parents that I have and he is male and he is not me." In comparison with the previous syntax for Horn clauses, PROLOG elides the universal quantification symbols, replaces the conjunction symbols with commas, and replaces the implication symbol with :-. The \= means "roughly not equal," although we will discuss some subtleties of this later. A PROLOG rule consists of a head and a body, connected by :-. The head of the last rule in Figure 1.73 is `brother(Self, Sibling)`. The head describes what the rule is intended to define. The body of the rule is the implied conjunction `parents(Self, Mom,`

```
                    female(paige).
                    female(mabel).
                    female(ellen).
                    male(morgan).
                    male(stan).
                    male(mark).

                    parents(mabel, paula, peter).
                    parents(stan, paula, peter).
                    parents(paige, ellen, mark).
                    parents(morgan, ellen, mark).

brother(Self, Sibling) :- parents(Self, Mom, Dad), parents(Sibling, Mom,
       Dad), male(Sibling), Sibling \= Self.

likes(polar, Person) :- has_food(Person), pats(Person, polar).
```

FIGURE 1.73. Example sentences from PROLOG.

`Dad), parents(Sibling, Mom, Dad), male(Sibling), Sibling \= Self`. The body describes a conjunction of goals that must be satisfied, one after the other, for the head to be true. The body and head in PROLOG are in the reverse order that we defined them for the Horn clause in (10).

In PROLOG, atoms that begin with capital letters are taken to be variables and atoms that begin with lowercase letters are taken to be constants. The last sentence means roughly that "Polar likes anyone who pats him and has food." In this sentence, `polar` is a constant corresponding to my neighbor's dog and `Person` is a variable. (Curiously, English conventions for capitalizing names is at odds with PROLOG syntax in this example.)

The analog in PROLOG to processing a database query is interpreting a PROLOG program. Figure 1.74 illustrates some steps in processing the statement `brother(paige, Sibling)`, which translates roughly as "Who is Paige's brother?" The process begins by retrieving the rule that defines `brother` and binding `Self` to `paige`. This value for the `Self` variable is kept for the subsequent matches in the sentence. The interpreter then steps through the clauses of the rule. In the first clause, unification binds `Mom` to `ellen` and `Dad` to `mark`. After a few more steps, a consistent binding is found and the answer `morgan` is returned.

This example is a very simple case. Only one step of backtracking was required, when the matching routine binds both `Self` and `Sibling` to `paige`.

Figure 1.75 gives another example of running the same program, except that a different query is answered because a different argument is made constant. In this case, the query `brother(Self, morgan)` translates roughly as "Who is Morgan the brother of?" This ability to use definitions in more than one kind of computation is one of the interesting features of logic programming. It exemplifies what is meant when people say that logic programming makes it possible to say *what* a program should compute without saying *how* it should compute it. This is

A Query

```
   ?- brother(paige, Sibling)
```

Trace of the matching process
Match `brother(paige, Sibling)` against the database.
This retrieves the definition of brother:

```
   parents(Self, Mom, Dad), parents(Sibling, Mom, Dad), male(Sibling), Sibling
       \= Self.
```

bind `Self` to `paige`.
Match `parents(paige, Mom, Dad)` against the database.
bind `Mom` to `ellen`.
bind `Dad` to `mark`.

Match parents`(Sibling, ellen, mark)` against the database.
This first match is with `parents(paige, ellen, mark)`
bind `Sibling` to `paige`.

Match `male(paige)` against the database.
This fails and there is no method for proving male. So the matcher starts to unwind.

It looks for another match to `(Sibling, ellen, mark)` in the database.
It matches parents `(morgan, ellen, mark)`
bind `Sibling` to morgan.

Match `male(morgan)` against the database.

Test that `morgan /= paige`.

Return: `morgan`, that is, `brother(paige, morgan)`

FIGURE 1.74. Processing a PROLOG query using the program in Figure 1.73.

also called a "declarative reading" of a logic program. For this query, the program returns the answer `paige`.

However, reading and executing a program in more than one way is not without costs. A PROLOG interpreter proceeds through its generation and backtracking process in a predetermined order. In logic programming languages, common practice is to write programs to be efficient for one expected use.

In our example of the `brother` predicate, it is assumed that the most common computation is to find the brother of `Self`. The way the predicate is written prescribes how the computation in that case is carried out as follows. First the `Mom` and `Dad` of `Self` are determined. Then the system generates candidates in the database having the same `Mom` and `Dad` as parents. Then candidates that are not male are eliminated. Finally, a check is made that the candidate is not equal to `Self`. The backtracking interpreter enables the program to answer other questions, but at a cost of more search. For example, when `brother` is asked how to determine whose brother `Sibling`

Another Query

```
    ?- brother(Self, morgan)
```

Trace of the matching process

Match `brother(Self, morgan)` against the database.
This retrieves the definition of brother:
`parents(Self, Mom, Dad), parents(Sibling, Mom, Dad), male(Sibling).`
bind `Sibling` to `morgan`.

Match `parents(Self, Mom, Dad)` against the database.
The first match is with `parents (mabel, paula, peter).`
bind `Self` to `mabel.`
bind `Mom` to `paula.`
bind `Dad` to `peter.`

Match `parents (Sibling=Morgan, Mom=paula, Dad=peter)` against the database.
This fails. The interpreter backtracks.

Match `parents (Self, Mom, Dad)` against the database.
The next match is `parents (stan, paula, peter).`
This fails in the same manner as the previous.

Match `parents (Self, Mom, Dad)` against the database.
The next match is `parents (paige, ellen, mark).`
bind `Self` to `paige.`
bind `Mom` to `ellen.`
bind `Dad` to `mark.`

Match `parents(morgan, ellen, mark)` against the database.
This succeeds.

Match `male(morgan)` against the database.
This succeeds.

Test that `morgan /= paige.`
This succeeds.

Return: `paige`, that is, `brother(paige, morgan)`

FIGURE 1.75. Another PROLOG query: "Who is Morgan the brother of?"

is, it begins by generating candidates for `Self` that have a `Mom` and `Dad`. The process is sound, but inefficient.

Logic programming extends the capability for determining whether a sentence is true beyond that of a relational database because it includes instructions for performing inferences that can determine the truth of derived relations. Thus, unlike the case of a relational database, some of the knowledge about the order in which to try inferential steps is in the knowledge base (database) and under the control of the programmer as expressed by the ordering of clauses.

In performing its inferences, the PROLOG interpreter makes several assumptions. It assumes that unique atoms refer to unique individuals. This assumption is reflected in the way that unification works. Obviously, the only constants it considers are those in the database. This is the closed-world assumption. It also assumes that something it cannot prove must be false. This explains the rationale behind the backtracks in Figures 1.74 and 1.75. This is called a "failure-as-negation" (FAN) assumption.

PROLOG is not a complete theorem prover. It is based on a resolution theorem prover for Horn clauses. The particular strategy that it uses is a form of linear input resolution. The inference process is split into two parts: a retrieval part that extracts atomic facts from a database by pattern-matching and a search part that tries to use non-atomic Horn sentences to form the inferences. There are many details that could be discussed about how PROLOG goes about deciding what clause to consider next and how this relates to more general theorem provers, but discussing them would take us beyond our current purposes.

However, it is interesting to note that the kinds of incompleteness of knowledge that logic programming can represent are similar to those in relational databases. For example, there are no clauses like the following:

```
¬ male (paige)        parents (paige, ellen, mark) ∨ parents (paige, ozma, wizard)
```

PROLOG has predicates for NOT and disjunction. However the predicate NOT(B) does not declare that B is false. Rather, it tests whether the negated clause *can be proven*. It succeeds if the negated clause cannot be proven. The status of this kind of statement is different from the others that we have discussed. For example, its effect depends not only on the contents of the knowledge base but also on the power of the theorem prover.

On one hand, including NOT in a language undermines its proof semantics. Characterizing what inferences are true becomes quite difficult. On the other hand, statements involving not can be quite useful for modeling certain kinds of human reasoning about defaults that resist formalization using standard semantics of logic. They allow the language to say "assume x if you cannot prove otherwise."

Within PROLOG, negation and disjunction can be used in rules, but they cannot be included as ground statements since all ground statements must be atomic. Similarly, there are restrictions on the ≠ predicate. Different versions of PROLOG treat it differently, but in most common ones it can only be used if the terms have already been bound. It succeeds only if the terms are different.

In summary, logical languages express several things. Like a relational database, they describe a space of possibilities. This is reflected in focusing on the terms in the database coupled with the closed world assumption. We can find all possible candidates for Paige's brother in the database. Restricting the kinds of incomplete knowledge in the language limits the work required of the inference machinery. Unlike a relational database, logical programming makes it possible to represent derived propositions. At the same time, the way that the statements for deriving propositions are expressed determines the efficiency of the search strategy for each kind of query. Clauses are considered in a predefined order so that logic programs implicitly embody expectations about what queries will be asked in terms of how a computation is performed. In

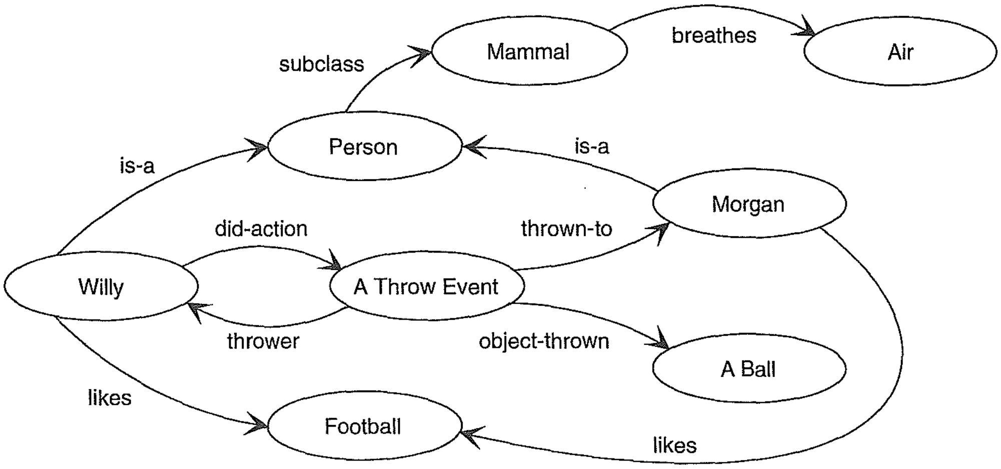


FIGURE 1.76. A simple semantic network.

other words, the way that a program is written makes some inferences more efficient than others. Finally, the inclusion of facilities like NOT can express metalogical knowledge. They make it possible to advise the inferential process about how to carry out default reasoning, a concept that is not part of standard logic.

Representation and Inference Using Frame Languages

We now revisit semantic networks as introduced earlier. Since there is no single language of semantic networks, we will concern ourselves with some features that have been common to a wide range of frame languages.

Figure 1.76 presents a semantic network representation similar to ones we considered earlier in this chapter.

One of the things to notice is that a semantic network provides different ways to handle predicates of different arity. Unary relations are called types or classes. They are organized in a taxonomy. For example, the relation

Person(Willy)

is represented using the is-a link between the nodes Willy and Person. Binary predicates are represented by arcs. For example, in Figure 1.76, the binary relation meaning "Willy likes football," represented in logic by

Likes(Willy, Football)

shows up in the semantic network as the arc labeled likes connecting the nodes Willy and Football. Binary relations are sometimes called attributes. Higher-order relations are represented by creating special objects. For example, the predicate

Threw(Willy, Ball, Morgan)

is represented in Figure 1.76 with a special kind of object called a "throw-event."

Inferences in frame languages about classes take several forms. In the simplest case, all members of a class have some common property. The characterization of the property *A* is given at the level of the class class, but is interpreted as being a property of its instances.

$$\forall x \text{ (is-a } (x \text{ class}) \Rightarrow A(x))$$

We called such reasoning from classes to instances inheritance reasoning. In Figure 1.76, we infer that "Willy breathes air" from the facts that Willy is a person, person is a subclass of mammal, and all mammals breath air. By placing an attribute high in a taxonomy, one can allow many instances below to inherit that property. This kind of reasoning lends itself to a graph search where one starts with the node in question and searches up the taxonomy to find a node defining the property. It also lends itself to certain intuitive approaches for default reasoning. For example, if nodes at different levels of a taxonomy indicate different values to be inherited, then the search process can be used to either specialize or override the values of attributes of nodes at different levels.

There are other variations on information about inheritance. Predicate calculus provides no knowledge or epistemological primitives for organizing a knowledge base. In contrast, many frame languages provide primitives for expressing certain commonly used relations for organizing the terminology of a KS. These include such relations as subclasses, specifications for defaults, and primitives for defining the structure of objects in terms of their parts. These primitives are a beginning of a theory of semantics.

A simple example suffices to give the flavor of such expressions as they relate to the inheritance relations discussed so far. Figure 1.77 defines Plant-Part as a class for large structural elements of growing plants.

One of the properties of a plant part is that it has a color. In a frame language, we can indicate that all parts have some color by establishing a description involving a color link from the class. This is the link labeled "value specification" in Figure 1.77. That link indicates a closed-world specification. In particular, it indicates that the color of an instance of Plant-Part must be some instance of the Color node and that in the absence of other information we can assume that the color of a plant part is brown. In frame languages, such an instance of Color must be a **filler** of this role in an instance of Plant-Part.

Another node in the figure is the Tree node. The Tree node illustrates some notions for structuring inheritance relations. Frame languages incorporating such notions are called **structured inheritance languages**.

The Tree node defines a property of trees called foliage. The notation in Figure 1.77 is supposed to indicate the following: Trees have foliage. Foliage is a relation indicating a part of a tree. The value or filler in a foliage relation of an instance of a tree must be an instance of a plant part. Like all plant parts, tree foliage has a color. Its color, however, is limited to red and green rather than the full range of colors. Furthermore, the default color of foliage is green rather than brown. The dashed links in Figure 1.77 are intended to suggest this web of relations, so that the plant-part description for foliage of trees is related to the general description of plant part and overrides some of its elements.

This richer set of links conveys an important shift in style of representation. This style recognizes the awkwardness of representing objects, especially objects with structure, in terms of flat propositions associated with a single node. In this style, an individual is represented by a

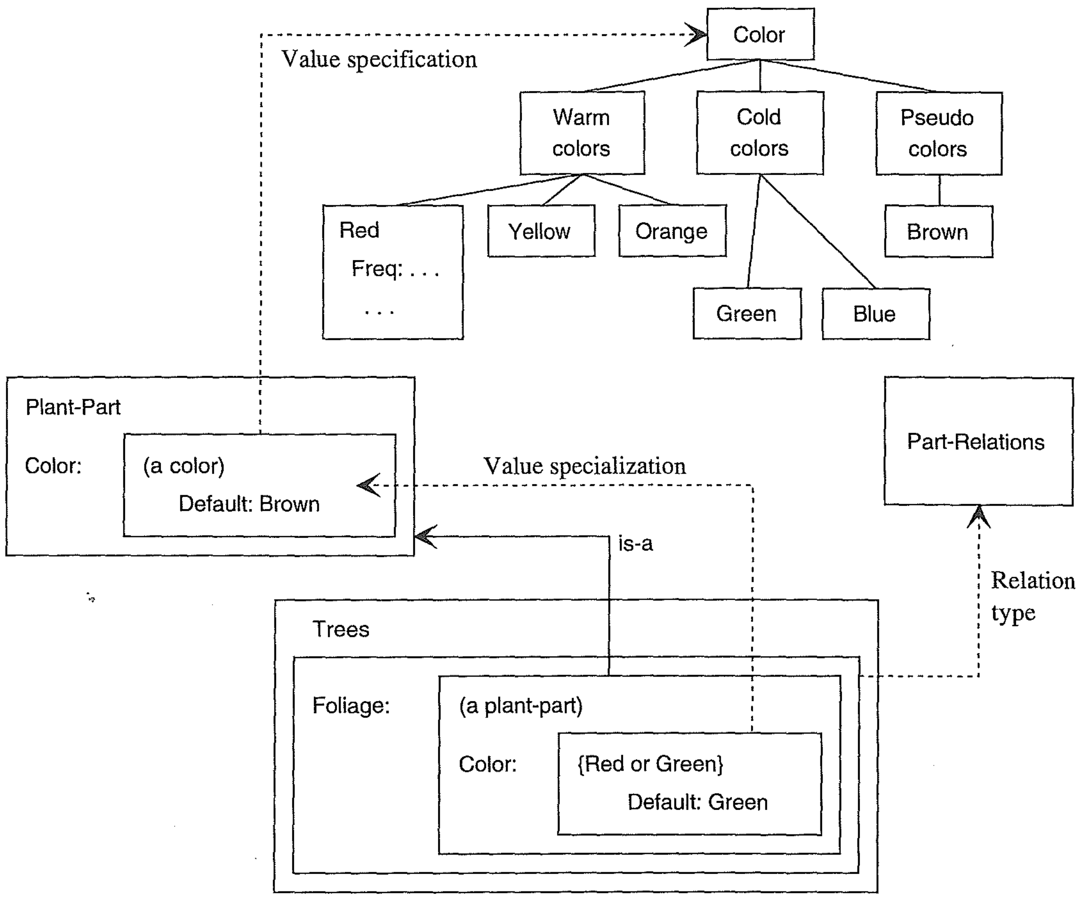


FIGURE 1.77. Example of structured inheritance in a frame-based representation language.

cluster of nodes where some nodes and links, primitive to the interpreter, are used to define the mutual roles. Within such representations, single is-a links are inadequate for capturing the many relations between the bundled parts. The richer set of connections opens up the possibility of specializing subconcepts of generic ones by restricting some of the subparts of the description embodied by the generic concept.

With the usual but important caveats about efficiency of inference and reasoning with incomplete knowledge, the meaning of everything we have said here about frame languages is similar to what we said about relational databases and logic programming in that it can be characterized by predicate calculus statements. What is extra is the set of knowledge primitives and guidelines for representation. Such guidelines are sometimes called the epistemological level for characterizing frame languages because they introduce primitives for knowledge used by the interpreter. In practical knowledge base tools, which emphasize the process of building KSs incrementally, such characterizations are also useful for limited forms of type checking as new concepts are added to the KS.

The appeal of the graphical nature of semantic networks has led to many forms of reasoning that are not well understood. Historically, developers of semantic networks have been lax in characterizing the semantics of the representations. As many formalists have lamented, it is unfortunately much easier to develop algorithms that seem to reason over structures of a certain kind than it is to characterize the behavior of those algorithms carefully or to justify the reasoning that is carried out.

Referring back to our earlier examples of a relational database and logic programming, frame languages are intermediate in terms of how they embody inference. As with relational databases, inference is carried out by an external interpreter that makes various assumptions about the network. Unlike logic programming, there is no way to express derived propositions. The main example of inference for semantic networks is reasoning about inheritance of properties. As in the case of NOT in logic programming, computations of default values have provided examples of reasoning that are said to model human reasoning but that are not characterized easily in terms of truth or proof semantics.

1.4.6 *The Symbol Level and the Knowledge Level*

The *what-to-how* spectrum ranks programming languages in terms of expressiveness. Programs written in languages at the "how" end of the spectrum describe what to do in many, simple steps. At the "what" end of the spectrum, languages provide integrated sets of powerful primitives. Because many of the details of what to do are represented in the interpreter, the programs written in these languages are shorter.

A representation system is expressive to the extent that it makes it possible to leave things unsaid, both in the language and in queries to the representation system. A striking aspect about this characterization of expressiveness is that the prescription of what an interpreter can bring to bear to process either a program statement or a query is extremely open-ended. Interpreters can be categorized according to what they bring to bear in processing queries.

In the design of knowledge systems, it is useful to distinguish between two levels of analysis known as the symbol level and the knowledge level. The **symbol level** is concerned with manipulating patterns of symbols. It is not concerned with what the symbols designate in an external world, that is, it is not concerned with their reference semantics. The **knowledge level** is concerned with particular tasks and, for that reason, with what symbols mean. At the knowledge level we are concerned with what knowledge is needed to carry out a task and how that knowledge can be used.

Most of the examples of expressiveness in this section have been about the symbol level. Analysis at the symbol level is concerned with the structural complexity of symbol structures. It is concerned with space and time complexity of algorithms over symbol structures, independent of what the symbols mean.

The advantages of expressiveness provided by pattern-matching languages are entirely due to operations at the symbol level. That is, the knowledge that pattern-matching interpreters use to make programs simpler is all about the symbols themselves. The interpreters know about pattern matching, they know about the extraction of parameters during matches, they know about substituting constants for variables, and they know about backtracking search. A theorem prover, a pattern matcher, or a query-processing program could be said to have knowledge, but what they know is about the structure of the symbols that they manipulate. They know about efficient indexing of their tables. They know how to reliably traverse a graph. They can carry out a search for solutions using such information.

When we analyze the complexity of various kinds of algorithms over graphs and relate their running time to properties of the graphs, we are working at the symbol level. Symbol-level analyses are of broad interest in computer science. Similar computational phenomena arise when similar procedures run over similar symbol structures.

The knowledge that an interpreter may use to interpret a statement in a program, representation language, or query, however, can also include knowledge and assumptions about the world. In our examples of the expressiveness of representation languages, we referred to closed-world assumptions and failure as negation assumptions. These examples are at the knowledge level because they are concerned with what symbols mean.

More broadly, an interpreter can gain further expressive leverage in query processing by effectively solving problems. In this way it moves beyond what we call a representation system toward what we call a knowledge system. The knowledge employed includes methods for finding solutions and knowledge for guiding that search. Chapter 2 discusses problem solving and the use of search methods as ways to find solutions. An analysis of a task at the knowledge level characterizes the use of world knowledge in terms of its effect in guiding the search of a space of possible solutions.

Looking further ahead, Chapter 3 discusses fundamentally what knowledge is and where it comes from. It extends our discussion about the construction of knowledge systems beyond the symbol-level manipulations in the computer to the social situations in which knowledge systems are created and used.

1.4.7 *Summary and Review*

In this section we considered the expressiveness of programming and representation languages. One indicator of expressiveness is what things need not be stated explicitly. In pattern-matching languages, the simplicity of the programs results because some operations are performed automatically by the interpreter of the language. Some of the complexity is factored out of the program and placed in the interpreter. In several example programs we saw how the expression of the differentiation program was made simpler when many of the details of graph search, parameter extraction, and substitution were handled by the interpreter of the pattern-matching language. This approach is called metalevel factoring.

Representations differ what they can express easily and the things they express at all. For example, all of the representations we considered were less capable than predicate calculus in expressing certain kinds of incompleteness of knowledge, such as ground literals involving negation and disjunction.

In the relational database example, the language is equivalent to function-free terms with constant arguments and the interpreter is the query system. We saw that a practical query language actually depends on additional assumptions, such as closed-world assumptions, and also that it needs to be concerned about the efficiency of query processing.

In the logic programming example, the representation is in terms of Horn clauses and the interpreter is that for the language. We required closed-world assumptions as well as failure-as-negation assumptions for use in the search for solutions.

In the structured inheritance networks, we saw that epistemological primitives can be used to organize clusters of nodes into larger coherent structures. These languages provide prescriptions for organizing representations, including specialized closed-world assumptions and knowledge about defaults. This idea has been extended to the design of specialized representation shells for specific tasks and to the approach of building large knowledge bases with many potentially reusable representations and compositional primitives. Such approaches are on the edge of current research.

Exercises for Section 1.4

Ex. 1 [*00*] *Pattern Matching.* Assume the following symbol structures:

```
(and (likes Fido Bonzo)(feather-brained Fido))
(and (likes Rover Lucky)(feather-brained Lucky))
(and (likes Spot Sperald)(feather-brained Lucky))
(or (likes Sport Dizzy)(feather-brained Dizzy))
```

What values would be assigned to the pattern variables by matching the following pattern against each of the symbol structures?

```
(and (likes ?dog ?person)(feather-brained ?person))
```

Hint: Some of the expressions do not match at all.

■ **Ex. 2** [*15*] *Common Sense and the Database.* Consider a database of family relationships. Suppose there are predicates for brother, father, mother, sister, and surname. For example, the database may contain several thousand facts like the following:

```
(brother Joe Mike)      (brother Mike Azel)      (brother Ed Joe)
(father Joe Pete)       (father Fred Sam)        (father Andy Nick)
(surname Mike Smith)    (surname Sam Stevens)
```

You may assume for the purposes of this exercise that the personal names used in these relations are unique identifiers for individuals. In addition, there are some rules for inferring surnames:

Rule 1: "A father and a son always have the same surname."

```
If (father ?fath-1 ?son-1) and (surname ?fath-1 ?sur),
then (assert (surname ?son-1 ?sur))
```

Rule 2: "Two brothers always have the same surname."

```
If (brother ?bro-1 ?bro-2) and (surname ?bro-1 ?sur),
then (assert (surname ?bro-2 ?sur))
```

(a) In what way is rule 1 weaker than the common-sense meaning of the corresponding natural language statement? Discuss briefly.
(b) Does rule 2 have the same problems as rule 1? How does the answer to this question depend on the properties of the pattern-matching process of the rule interpreter?
(c) Give two additional rules similar to the ones above that would extend the situations for inferring the surnames of brothers, fathers, or sons. Briefly discuss any relevant properties of the matching process of the rule interpreter.
(d) Would it be appropriate to add a notation saying that the brother relation is commutative? (*Hint*: This is a trick question. The given database is missing entries for some crucial kinds of people in families.)

Ex. 3 [*CD-00*] *Simulation versus Deduction.* After staring at several production-rule programs, Professor Digit exclaimed, "Eureka! Simulation and deduction are just the same thing! All of these programs use production rules and some control structure or other! It's all just if–then or pattern matching." Do you agree with Professor Digit? Explain briefly.

Ex. 4 [*L-05*] *Representation Using Predicate Calculus.* Sometimes the naive translation of statements into the predicate calculus can be misleading. Consider the following statement:

```
Every person knows someone who has a feather-brained dog.
```

(*a*) Represent this statement in predicate calculus.
(*b*) Is the logical interpretation of the statement realistic? Give an example of what else a program would need to know to make realistic use of such a statement.

■ **Ex. 5** [*!-15*] *Control in Production Systems.* In production-rule systems, it often happens that more than one rule matches the data in the working memory. When this happens and it is desired to fire only one rule, some means must be found for selecting among the rules. The set of rules that match the data at a given cycle is called the **conflict set**. A method for choosing which rules in a conflict set to fire is called a **conflict-resolution strategy**.

Suppose the following set of rules is used for recommending choices of food for patrons in a restaurant. The left sides are conjunctions of terms that must be in the memory, and the right sides list foods to be recommended.

For example, rule 3 can be translated roughly as "If it is breakfast time and the patron is overweight, then recommend cereal."

```
rule-1: breakfast-time -> (eggs bacon)
rule-2: breakfast-time hungry -> (eggs bacon waffles)
rule-3: breakfast-time overweight -> (cereal)
rule-4: breakfast-time hurried -> (juice toast jelly)
rule-5: breakfast-time overweight hungry -> (fruit cereal)
rule-6: breakfast-time kids -> (milk pancakes)
rule-7: breakfast-time runner -> (yogurt fruit)
rule-8: breakfast-time kids hungry -> (pancakes eggs juice)
rule-9: breakfast-time hungry runner -> (fruit cereal)
rule-10: kids -> (pizza)
rule-11: lunch-time kids -> (burgers fries shakes)
rule-12: lunch-time overweight -> (salad)
rule-13: runner -> (sportDrink)
```

(a) A rule is in the conflict set if all the terms on the left side of the rule are present in the working memory. What is the conflict set for each of the following cases?

```
working memory-1: kids
working memory-2: hungry runner
working memory-3: lunch-time
working memory-4: lunch-time kids
```

Note: The conflict set can be large or small, depending on how many rules match. Remember that every element on the left side of the rule must match an element in working memory.
(b) Suppose instead that the interpreter uses a "most specific rule" strategy. Again, the conflict set is made up of all of the rules that match the data. The winning rule is the one that matches the most terms in the data. In the event of a tie, the earliest rule in the

sequence dominates. For each of the following cases, give the conflict sets, the match scores, and the winning rules.

```
working memory-1: kids
working memory-2: breakfast-time hungry kids
working memory-3: hungry runner
working memory-4: breakfast-time seniors
```

(c) Suppose that we change the interpreter of the rules so each cycle proceeds in two passes. First all of the rules are matched against all of the data. Then the most specific rule/data match (determined by the maximum number of matching terms) is identified and that rule is fired. After a rule fires, the data it matched are removed from the working memory and the process is repeated. In the event of a tie, the rule with the lowest index numbers is done first. Conflict sets are recomputed at each cycle. Briefly, describe the actions of the interpreter given the following initial working memory.

```
working memory: breakfast-time kids hungry runner
```

Ex. 6 [*10*] *Rule Interpretation Example.* Use the traffic-light controller rules in Figure 1.62.
(a) Describe the behavior of the traffic-light system when there is no traffic on the farm road. (Focus on the case where there has been no traffic there for a long time.)
(b) Describe the behavior of the traffic-light system when the farm road and the highway are both crowded with traffic.
(c) Describe the behavior of the traffic-light system if the sensor is jammed, so that it always indicates "car-waiting" even when there is no traffic on the farm road.

■ Ex. 7 [!-*20*] *Forward and Backward Chaining.* When the firing of rules is driven by matching the left sides of the rules against data, we say that the rules are driven by **forward chaining**. Consider the following rules:

```
(Rule-1
  (if (mother ?mom ?kid))
  (then (parent mom kid)))

(Rule-2
  (if (father ?dad ?kid))
  (then (parent dad kid)))

(Rule-3
  (if (and (parent ?gran ?par)
           (parent par ?kid)))
  (then (grandparent gran kid)))

(Rule-4
  (if (and (brother ?unc ?mom)
           (mother mom ?kid)))
  (then (uncle unc kid)))

(Rule-5
  (if (and (brother ?unc ?dad)
           (father dad ?kid)))
  (then (uncle unc kid)))
```

```
(Rule-6
  (if (and (sister ?an ?mom)
           (mother mom ?kid)))
  (then (aunt an kid)))

(Rule-7
  (if (and (sister ?an ?dad)
           (father dad ?kid)))
  (then (aunt an kid)))
```

(a) Suppose the interpreter fires the rules using forward chaining. The rules are matched against the data in the working memory. When more than one rule matches, the first rule is fired first. The interpreter uses an audit table to keep track of which rules have been fired on which data. Each rule is fired exactly once on each combination of matching data. More specifically, every combination of data is considered for each rule and no rule is fired more than once on the same data. When a rule is fired, the then-part adds new data to the working memory which may then be processed by other rules. The process stops when no more rules match any new data.

Write down the resulting data given that the working memory initially contains the following:

```
(mother Mary Ellen)
(father Virgil Ellen)
(brother Karl Ellen)
(sister Paula Ellen)
(sister Julie Ellen)
(mother Ellen Paige)
(mother Ellen Morgan)
(father Mark Paige)
(father Mark Morgan)
(brother Eric Mark)
(brother Mike Mark)
```

(b) Considering what you observed in part (a), is the answer sensitive to the order in which the rules are fired? Explain briefly.

(c) When the firing of rules is driven by matching the right sides of the rules against goals, we say that the rules are driven by **backward chaining**. For example, we could use backward chaining on the rules to find all of Morgan's aunts. We would first find the rules that conclude that somebody is an aunt, in this case, rules 6 and 7. Proceeding first with rule 6, we examine the left side. If the rule is satisfied, we can apply the rule at once. Otherwise, we can make new goals from the left side and try again. In this case, we can infer that Paula and Julie are Morgan's aunts.

Show how backward chaining can be used to determine Paige's uncles starting from the initial state of the working memory.

(d) Give the trace of backward chaining for determining Paige's grandparents.

(e) In backward chaining to compute Paige's grandparents, and expanding the clauses in rule 3,

```
(and (parent ?gran ?par)
     (parent par Paige)))
```

why is it probably more efficient to consider the clause about Paige's parents first?

(f) Briefly, what would happen if we use this approach to determine Ellen's grandparents?

Ex. 8 [*!-10*] *From Explicit Representation to Machine Learning*. Upon understanding the operation of a universal Turing machine and the von Neumann architecture, Professor Digit was struck by a sense of profundity. He gathered his graduate students together and announced that there were three main principles that should forever change the way they view computation and machine learning:

□ Symbol structures are dynamic and changeable in physical symbol systems.

□ Programs are prescriptions for changing symbol structures.

□ Programs themselves are represented by symbol structures and can be interpreted by other programs.

From these, Professor Digit concluded that we have the key for building learning systems. Programs could just reason for a while, see how they are doing, and then use symbol processing to modify themselves to be better. Programs could be self-organizing, using the powerful pattern-matching ideas discussed in this section. For example, when a pattern in a rule (the manipulating rule) matched part of another is that simple!

Do you agree? Explain briefly.

Ex. 9 [*CD-CP-16*] *"Predicate Calculus as Assembly Language."* John Sowa sometimes remarks that predicate calculus (PC) is to a representation language (RL) what assembly language (AL) is to a high-level programming language (HL). Noting that the analogy is not exact, he observes that characterizing what a representation language means in terms of predicate calculus requires an increase in verbosity not unlike the increase in the size of a program description when a program in a high-level language is compiled. This exercise explores some questions inspired by this analogy.

In the following, RL, AL, and HL all correspond to classes of languages. For the purposes of discussion, choose whatever instances of these classes are appropriate.

(a) What semantics is relevant for comparing PC and RL? What semantics is relevant for comparing AL and HL?

(b) What expectations does the analogy about HL:AL create about the relative computational efficiencies of PC and RL? Briefly explain how the analogy is misleading with regard to efficiency.

(c) What things can you express in AL that you cannot say in HL? What kinds of things can you say in HL that you cannot say in AL?

(d) What things can you express in PC that you cannot say in RL? What kinds of things can you express in RL that you cannot say in PC?

Ex. 10 [*05*] *Basic Concepts*. Determine whether each of the following statements are true or false. If a statement is ambiguous, explain your answer briefly.

(a) *True or False*. Pattern-matching languages tend to provide simple expressions of programs because the detailed operations of matching, parameter extraction, substitution, and backtracking need not be explicitly described in the program.

(b) *True or False*. Requiring that it be possible to tell whether a sentence follows from a database of sentences is intractable even if we limit the class of sentences that can be expressed and tested. (*Note*: The sentence tested and the sentences in the database are *not* arbitary but are limited to some restricted class. The question is whether proof must be intractable even so.)

(c) *True or False.* Failure as negation is seldom used as a policy in logic programming because there is no fixed upper limit on the amount of time needed to construct a proof.
(d) *True or False.* Horn clauses are more general and expressive than the atomic sentences used in a relational database.
(e) *True or False.* Compared with logic programming languages, structured inheritance languages tend to be strong on expressing inference and weak on providing epistemological primitives.

1.5 *Quandaries and Open Issues* ADVANCED

This chapter concentrated on symbols, symbol structures, and physical symbol systems as fundamental concepts for understanding knowledge systems. In this section, we step back from this broad development to ask how our attitudes about symbols and traditional formal systems can be misleading. We consider results and speculations from cognitive science and philosophy. We briefly discuss theories of cognition that build upward from nerves and others that build downward from intelligent agents. These theories enrich our perspectives about the role of symbols in both computation and communication.

The Physical Symbol System Hypothesis, Again

We begin our consideration of open issues with the hypothesis with which we opened this chapter, the physical symbol system hypothesis by Newell and Simon (1975).

> **The Physical Symbol System Hypothesis.** A physical symbol system has the necessary and sufficient means for general intelligent action.

By this, Newell and Simon meant that an analysis of any system exhibiting general intelligence would show that the system is a physical symbol system and that any physical symbol system of sufficient complexity could be organized to exhibit general intelligence. By *general intelligent action*, they meant the same order of intelligent and purposeful activity that we see in people, including activities such as planning, speaking, reading books, or composing music.

The hypothesis proposes that intelligence follows from the organization of physical systems and that it obeys natural laws. It also suggests that human intelligence follows directly from our organization as physical symbol systems and that, in principle, it is possible to build artificially intelligent systems by creating symbol structures that have the right properties.

Throughout most of this book, we skirt around the question of what constitutes "intelligence." This issue leads to many debates within artificial intelligence, but the arguments have tended to be rather sterile in the context of knowledge systems, where the focus is on the construction of systems with task-specific performance criteria.

Within AI the hypothesis has received a mixed reception. Some researchers consider it obvious and tautological, given that physical symbol systems are capable of manipulating and interpreting symbols. They believe that **mind** is an emergent phenomenon from the right kind of computation. Others find the hypothesis fundamental, but not obvious and potentially wrong. Another argument is that the physical symbol hypothesis is vague and trivial. It begs the important questions of just what kinds of organization are necessary for intelligence, and what kinds of mechanisms are needed for processing the symbols. Supporters of this view argue that the essence of intelligence is in the details. They argue that vague hypotheses are of little use scientifically because they are not testable by experiment. Still others consider the hypothesis circular, turning on definitions of symbol, action, and intelligence that preclude normal kinds of scientific testing.

One of the most extreme positions on this was presented by Rodney Brooks when he argued that intelligent systems need not have representations (Brooks, 1991). Brooks' paper was based on ideas explored building simple robotic creatures called "insects," "mobots," or "animats." He argued that AI has relied too much on the study of representations. He proposed an approach wherein increases in functionality come from a layering of systems, each of which connects perceptors to effectors without symbolic intermediaries. In the same issue, Kirsh (1991) argues that the potential of Brooks' seemingly symbol-free approach is overstated and that the example systems embody symbols anyway.

The status of the physical symbol system hypothesis now is like the status of the axiom of choice before set theory was made rigorous. For many years, set theory was informal. Its theorems were considered obvious and not worthy of careful attention. Then, some puzzling examples were found in strangely constructed infinite sets. This led to the formalization of the seemingly ingenuous axiom of choice in mathematics. This axiom says that the Cartesian product of a nonempty family of nonempty sets is nonempty. Restated more simply, given a set, it is possible to select an element from it. At first nobody recognized that the axiom was needed. Then, after the axiom was made explicity, it was not at all clear what its consequences were. Set theory now includes branches of study with and without this axiom. In just this way, research on artificial intelligence now debates whether symbols are necessary for intelligence.

The debate about the role of symbols takes place in the context of models of mind. Since the time when he first proposed the physical symbol system hypothesis, Newell and others have gone on to develop much more elaborate hypotheses and models of mind. Although reviewing these is beyond the scope of this section, the interested reader is referred to Newell, 1991. The next few sections show how these debates are concerned both with the nature of intelligence and the nature of symbols.

Symbols in Natural Minds

Symbols in computer languages and memories are compact, discrete markings. Critics of computational models of cognition have argued that because symbols in computer systems are digital, they are irrelevant to the operation of memories in living brains. Our concern is with the converse. How do studies of brains or memory offer insights about the design of knowledge systems?

At the time of this writing, progress in understanding how memory in a brain works has been quite limited. Experiments on the biological mechanisms of memory have focused on animals with extremely simple nervous systems. These experiments study small parts of small nervous systems and raise many new questions. Are the mechanisms for memory in one part of the nervous system the same as mechanisms for other parts? What storage and retrieval mechanisms are universal across different species? Are the same mechanisms used for short- and long-term memories?

Beyond the issue of how memories work are larger questions about how minds and brains work. The organization of brains and nervous systems is being studied on many fronts. Within that context, the most crucial property of symbols is their use in causing action at a distance in space or time. Many kinds of actions are possible, such as triggering a specific external activity by the symbol system or evoking larger symbol structures. Symbols can cause action at a distance in space because they can be copied and transmitted to distant processors. Whenever they are presented, symbols cause a processor to carry out a reproducible action. Symbols can cause action at different times because they can be stored in memories and recalled for later use. This "action at a distance" property explains how memories stored in one part of a brain can be used to cause actions, controlled by a distant part of the brain.

The biological memories studied so far employ local, chemical, and physical changes during learning. Chemical traces of brain activity also provide data on how different areas of the brain have specialized functions; detailed timing studies of linguistic and problem-solving activities provide data on how much parallelism must be employed for various mental tasks. Combining these kinds of data for a unified understanding of brain function is a long way off.

Connectionism, Signals, and Symbols

Most of our discussion of the operation of physical symbol systems was based on architectural concepts from Turing and von Neumann machines, in which the processor is separate and distinct from the memory that retrieves symbols, interprets them, and causes operations to be carried out. In these models memory is a passive structure, capable of storage and retrieval but little else. Inspired by models of neural networks, there is an active and vigorous school of thought in cognitive science called connectionism that challenges these basic assumptions about symbols and information processing. Connectionists argue that von Neumann computational models are irrelevant to the operation of a brain.

Connectionist systems are networks of large numbers of simple but highly interconnected units. Each unit is assumed to receive signals along its input lines, either excitatory or inhibitory or both. Typically the individual units do little more than add the signals, perhaps combining them with an internal state. The output of a unit is a simple, nonlinear function, such as a threshold function of the sum. The connectionist framework gives us "distributed symbols," colorfully described as "symbols among the neurons." These distributed symbols have many of the essential properties of the symbols described in this chapter: They are material patterns, recognizable by complicated processors. However, connectionism mixes memories and processors together so much that one cannot draw a neat boundary between them. Connectionism challenges the notion that memory and processors are separable.

One mystery in cognitive models is how it is that slow nerves can compute so quickly. The massively parallel connectionist models have much appeal in explaining this. Related mysteries potentially drawing on massive parallelism include how people can recognize enormous numbers of patterns and how brain function continues without catastrophic failure even when the brain has sustained damage.

How do new symbols and expressions acquire their meanings? Connectionists look toward repeated patterns in the orderliness of the world and in the repeated structure of routine tasks. In this view, the meaning of symbol structures is intimately tied to their creation and use. From the beginning, symbols are linked to perception and action.

Agre and Chapman (1988) have argued that information-processing theories of intelligence presuppose a substantial and implausible amount of mental processing machinery. For example, they propose an "indexical functional" theory of representation rather than having unique symbols that stand for unique objects in the world. Illustrating their ideas in a video game player, they would have an indexical symbol standing for "the bee on the other side of the block in the direction I am moving" rather than having a `bee-064` symbol. Roughly, the patterns of interaction between the symbols reflects the patterns of interaction between the observer and the environment. This moves much of the inferential load into the representation and even into the environment. By defining terms indexically, it may also reduce the number of symbols needed.

Fodor and Pylyshyn (1988) argue that a crucial point of difference between connectionist and classical models of mind is that the meaning of a unit in a connectionist account is not a rule-based composition of the meanings of the units to which it is connected. In this chapter we saw that this reductionist view that the meaning of an expression is composed from the meanings of its parts is a property of truth semantics of most representation languages such as the predicate calculus. It is also a property of denotational semantics that describe the operation of programming language, and it is useful in explaining concepts such as recursion.

In a connectionist graph, a link between unit *x* and unit *y* means that states of node *x* causally affect states of unit *y*. Fodor and Pylyshyn argue that this particular aspect of connectionism disqualifies it from providing a complete basis for a theory of cognition. They favor a classical information-processing model. For example, they observe that people can understand sentences that they have never heard before and that are structured quite differently. Similarly, people carry out wide classes of inferences that they have never made before. In the information-processing model, much of the power comes from pattern matching and **recursive processing** of symbol structures. In the grammar case, rules of grammar and rules of interpretation can be applied recursively in processing long (previously unheard of) sentences. Such recursive processing seems essential to the process and is missing from the connectionist account.

For Fodor and Pylyshyn, the connectionist models are more compelling for explaining low-level perceptual processes than high-level symbol manipulation processes in intelligent systems. From the perspective of knowledge systems, we would like to understand the computational limitations of models built in either way. This is an area in which much basic work remains to be done.

Although the physiological elements of a brain can be reduced to smaller elements, the basic notion that the meaning of the whole is a function of the meanings of the parts becomes less tenable as the parts become as small as individual nerve cells. More important in this realm are patterns of interactions. Marvin Minsky addresses this issue extensively in his book *The Society of Mind.* He describes many kinds of mental phenomena, processes, and possible constructs. One interesting aspect of Minsky's theory is that its elements range from being implementational in nature to being psychological in nature. In this vein, Minsky sees "symbolness" as a matter of degree rather than as a sharp issue. Comparing connectionist and symbolic formalisms for computing in terms of their capabilities and computational resources, Minsky (1990) argues that mental architectures need both kinds.

This notion that representations can have different degrees of a symbol-like character is also in line with computational models of perception. One of the ubiquitous concepts in research on perception is the **signal**. Examples of signals include acoustic waveforms, visual images, or the output of various other sensor arrays. Signals have extents in space and time. We can talk about changes in a signal over a second and can analyze the structure of a signal across both large and small intervals. Signal processing includes developing analyses of signals and various abstractions of them. For example, we can say that a signal has a particular frequency in some interval, or that its amplitude is increasing, or that it is periodic. Analyses of one-dimensional signals on time can attribute particular properties to arbitrary points in time; analyses of two-dimensional images can attribute abstractions to arbitrary points in a plane. When an abstraction is attributed to all of the points of a signal, the abstraction itself is said to be a signal-like representation. We can reasonably ask whether mental representations are more like signals or more like symbols.

Within architectures of cognition, it is now common to refer to subsymbolic processing. As experimentation with neural networks continues, such terminology and methodology will probably find its place in the design of knowledge systems.

Cognition and Levels

It would be nice to have theories of cognition that explain mental activities all the way from the firing of nerves to emotion and reason. Returning to our discussion of the ongoing dialog between information processing psychology and connectionism, Fodor and Pylyshyn suggest that instead of providing a comprehensive architecture for cognition, connectionism may provide a computational account of how nerves work, or rather the physical mechanisms of memory and low-level computation. Thus, although connectionist architectures may be unsuitable as a complete basis for explaining cognition, they are appropriate for *implementing* other levels of cognition. Many cognitive scientists suggest that systems should be understood in terms of **levels** of cognition and representation. Levels also lead to insights about different kinds of symbols, what they are used for, and how they fit into different kinds of theories.

Cognitive scientists characterize two levels of description above the raw physical encodings of memory. The memory itself is called either the **physical level** or the biological level. Above that Newell distinguishes the **symbol level** from the **knowledge level** (Newell, 1982); other cognitive scientists have used the corresponding terms *functional level* and *semantic level* (Pylyshyn, 1984), respectively. The symbol level is a description in terms of symbols (tokens and terms), expressions, and the deterministic interpretation of them. A symbol theory does not refer to the physical properties of a system but only to the way that the system operates. It is concerned with how the behavior of a system can be explained in terms of processes on symbols. In representational theories of mind, a system's behavior is explained not only in terms of the sensory inputs from its immediate environment, but also in terms of its internal state encoded in symbols. The symbol level is concerned with the manipulation of these symbols.

In theories of intelligence, the knowledge level is concerned with the representational content of the symbols. The knowledge level describes systems as agents, having goals, actions, and physical embeddings. An agent selects actions to achieve its goals. Newell's knowledge level is intended for predicting and understanding behavior without having an operational model of the processing actually done by the agent at the symbol level and below. To predict the behavior of an agent at the knowledge level, an observer ascribes to the agent principles of **rationality**. These principles provide constraints on the role and interpretation of symbols, but they are a much less complete description of behavior than a problem-solving process. More detailed accounts of behavior are possible at the symbol level. For example, at the symbol level one could predict which state-to-state transitions are likely to occur in a system, corresponding to rational decisions. Methodological approaches based on taking protocols of people solving problems and comparing these with computer traces (Newell, Simon, & Shaw, 1963) have shown how computer programs can predict and model in considerable detail the steps that human problem solvers take in solving problems. Such experiments provide evidence supporting the validity both of computational models of problem solving and for representational theories of mind.

The terms *symbol level* and *knowledge level* are used somewhat differently for knowledge systems. In the context of knowledge systems, we use the term *knowledge level* to refer to analyses of a task in terms of the knowledge that is needed for a task and how it is used. We use the term *symbol level* to refer to physical representations.

The Semantics of Existential Quantifiers

Terms in natural language routinely refer to states of affairs that are contrary to fact. The following phrases are all problematic in the analysis of their designations: "the current king of France," "the Wizard of Oz," "justice," "the common cold," "the unexplored regions of Africa," "the way things could have been," and "the average American." A well-known aphorism, attributed to Korzibski, comes to mind: "The map is not the territory." One should not confuse symbols with their designations. Linguists collect such odd examples of referring expressions. These examples reflect issues that a coherent theory for reference semantics must deal with. A larger set of such examples gathered from many sources can be found in Chierchia and McConnell-Ginet, 1990.

Graeme Hirst (1989) has analyzed such examples in a challenge to the semantics of existential and universal quantifiers in logic as used to represent the meanings of sentences in natural language. His examples range from sentences about things that aren't there ("I don't own a dog"), events that never happened, existence ("the existence of carnivorous cows"), fictional characters, things at different times, and things that might have been. His examples show that if we want a reasonable account of common sentences about existence, nonexistence, and nonexistent objects then we need more than one notion of existence.

Hirst's work makes it possible for us to see more clearly the assumptions behind our formulations of truth and proof semantics. In our discussion in this chapter about different kinds of semantics, we saw how some linguists use formal language semantics to represent the meaning of sentences. Hirst turns this around. He notices that most logics and knowledge-representation languages base their semantics of universal and existential quantifiers on the ontological assumptions of Russell and Quine. These examples show that these semantics are inadequate for capturing subtleties of meaning in natural language.

How Is Communication Possible?

The interaction between natural language and thought has been a topic of interest of many years. In the 1930s, Benjamin Whorf, an insurance company fire inspector, teamed up with an anthropologist Edward Sapir to explore the influence of language on thought and culture.

According to legend, Whorf developed his interest in language when he saw how frequently verbal misunderstandings led to fires. For example, he noted that people smoke and then thoughtlessly toss their spent matches into "empty" gasoline drums. Because gasoline fumes are highly flammable, empty does not mean safe.

The Sapir–Whorf hypothesis holds that language molds the form and texture of thought. As Whorf puts it, "We dissect nature along lines laid down by our native tongues. . . . We cut nature up, organize it into concepts, and suscribe significances as we do, largely because we are parties to an agreement to organize it this way—an agreement that holds throughout our speech community and is codified in the patterns of our language."

Language influences thought in several ways. It provides the words we use for expression, it determines what is routinely included in sentences, and it determines what is easy and difficult to express. This highlights the role of the social context in an analysis of the meaning of language. Children both learn about the environment and learn language in a social setting. Language shapes learning and perception and arguably thought.

When we communicate with each other through language, we routinely leave things out in the interest of efficiency in speech. This is why communication requires more than translation to internal symbols. To communicate effectively one needs not only only to translate or change the form, but also to fill in the missing information, to make plausible inferences, to integrate new knowledge with previous knowledge, and to signal understandings and possible misunderstandings with the other communicants.

There are deep questions about how these processes might actually work. What is really happening when two people believe they have achieved mutual understanding about the meaning of a word and how do they create a basis for convergence? Faced with such questions, Lakoff and Johnson studied examples of ordinary conversation. Lakoff is a linguist who was struck by the pervasive use of metaphor in everyday language and thought. Johnson is a philosopher who was struck that traditional philosophical views permitted little role for metaphor in understanding the world or mental life. Their book (Lakoff & Johnson, 1980) is a rich source of examples of how metaphors enable us to comprehend one aspect of a concept in terms of another. Consider the sentences "It's hard to *get* that idea *across* to him" and "Your reasons *came through* to us." Both sentences make use of a *conduit metaphor* for communication and meaning. "That boosted my spirits." "I'm feeling up." These sentences use an up/down metaphor, where happy is up and sad is down. Perhaps this is based on common human experiences related to posture. Drooping posture typically goes along with sadness and an erect posture with a positive emotional state. Lakoff and Johnson believe that metaphor enables new symbols and statements to draw on presumably familiar situations, imbuing symbol structures with meaning that relates to commonly shared experiences. Metaphors carry hints about the construction of meaning. This work is exciting because it suggests basic ways that communications can carry meaning, enabling a listener to construct meaning about experiences that he or she did not have with his own senses.

Pieces of Mind

Imagine that we are observing a team of people working together. As outside observers, we could try to model the group as a single symbol system. Of course, there are some immediate quibbles about the persistence of symbols. When someone speaks, the "speech symbols" are heard, recorded, and processed by the participants. One point of view is that the group of people functions as a single physical symbol system, or as an organization as having a **collective intelligence** or group mind. This attribution of agency to a group is not uncommon. Committees, companies, and even nations are often described anthropomorphically as having goals, personalities, strengths, and emotions.

Minsky (1986) has proposed modeling a mind as a society of agents, composed of simpler agents all the way down. In this model, new agents are created and specialized as a mind develops. In comparison with connectionism, which tries to extend upward from essentially nerve models, Minsky's **society of mind** tries to extend downward from powerful abstract agents to simpler ones. Symbols enable remote access. Symbols correspond to activations of "*k*-lines" that cause various parts of the brain to become active. There are many things yet to be understood about the power of this model of cognition. Indeed, it is difficult to tell at this stage whether various models of cognition are distinguishable experimentally. As a challenge, try to design an experimental psychological test under which the society-of-mind model and some different one would yield different signatures.

The society of mind blurs together all of the issues about communication and computation. Suppose we as observers have a wonderful "symbol-scope" for looking into the activities of a brain housing a mind. In a society model the interactions among high-level agents may be more akin to **communication** between separate beings while symbolic interactions between low-level agents may be more akin to **computation** or message passing as in a programming language. We can imagine looking at a robot who is looking at a scene. Suppose the robot is reaching for a block. As outside observers, we may see early visual processors creating scene descriptions. The early-perception agents may reduce these symbols to descriptions passed along to scene-interpretation agents, which have access to their previous interpretation of the scene a few moments ago, as well as to other scenes. They may identify changing parts of the picture and update an indexical representation of the "current scene." Elsewhere in the mind, we may see some high-level agent describing the place to put the block in terms of other constructions it has made before, possibly using "metaphor" in its communications. Of course, this scenario is very speculative. We just don't know how all this processing is done in the human brain. We can now build robots to carry out parts of it.

Minsky has been a critic of the assumption that there is a useful and crisp division between symbolic and nonsymbolic systems. The society model challenges another sharp distinction, between symbols used externally between agents (natural language) and symbols used internally between agents that form a mind.

Active Documents

People build knowledge bases collaboratively. They discuss and agree about the meanings of symbols in the knowledge bases. They design the behavior of the systems around the agreed upon meanings of the symbols. Viewed this way, knowledge systems are like blackboards or paper. They are a place for writing symbols. They augment human memory and processing with external memory and processing. Like scratch pads or calculators, knowledge systems augment our short-term memories. Like books in a library, they augment long-term memory. Like mail, speech, and blackboards, they augment various media for human communication. As we talk about this, it is curious how we begin to mix cognitive and document perspectives. In any social situation in which knowledge systems are built and serve a group of people, they necessarily must function as **electronic documents**, used for communication among those people.

Knowledge systems are not just passive media for recording and retrieving writings. We expect them to carry out rational processes using the symbols. This brings us back to the cognitive perspective on the meaning of symbols. In knowledge systems, we often use the same language to serve both cognition and documentation. For example, the same production rule language may be used for inference and as elements of explanations of the behavior of a system; the concepts acquired in knowledge acquisition are the same as the concepts used in problem solving. In programming terms, this is akin to the advantage of a "source language debugger," which makes it unnecessary to know a lower-level machine language (the target language for the compiler) when debugging a program. There can be several layers of symbol interpretation and compilation between the symbols that users see and the ones that are used in the rational processes of symbol manipulation. The advantage is that computations can be more efficient when the source language is compiled. To have it both ways, systems need to be able to translate back and forth between external (source) and internal (compiled) languages.

Returning to the discussion of the role of symbols, knowledge systems straddle the distinction between internal and external symbols. The psychological models of symbols are useful for describing the reasoning processes that knowledge systems engage in. The communications models for symbols are most useful for describing how separate knowledge system can interact, and for describing how knowledge systems can interact with people.

About Foundations

Although speculations about evolution, communication, and rationality are fascinating, they are far beyond the concepts that are applied routinely in knowledge engineering. Brian Smith (1986) wants to develop a theory of correspondence explaining the intricate relations among representation, specification, implementation, communication, and computation. Understanding these relations may provide insights about building knowledge systems. Symbols are deeply rooted not just in the methods of knowledge engineering and AI, but also in psychology, mathematics, computer science, and logic. The open issues in this section show that we have not yet heard the last word.

Knowledge systems are not built entirely on firm and securely established foundations. As is the case with even such fields as physics, astronomy, and mathematics, the foundations are subject to inspection, reexamination, and occasional challenges. We make progress in the absence of entirely satisfactory answers. Practitioners depend more on their native ability to communicate than on having a fundamental grasp of the connections between communication and computation. They draw insights from cognitive architectures without insisting that knowledge systems be modeled closely after human minds; they draw on insights from logic without insisting that a knowledge system use strictly logical principles.

It is important to adopt an appropriate attitude toward foundations. In physics and astronomy, new theories of elementary particles and cosmology provide insights about foundations. But only slowly does this work yield knowledge that changes what applied physicists do. In mathematics there have been revolutions in theories of measure and sets, but basic mathematics stays the same. For example, there have been no changes in the tables of integration and differentiation used in calculus, even as the foundations of measure theory have shifted. The subject matter of these fields depends on stable properties that are emergent from the properties of the foundations. For example, a sense of connection and coherence about topics in biology is mostly independent of foundations in chemistry, and a sense of connection and coherence in chemistry is mostly independent of foundations in physics. Although knowledge engineering is newer, there is a sense of connection and coherence about our theories of symbols, knowledge, and search that we expect to persist even as the field expands at its edges.

In considering the foundations of mathematics, Bertrand Russell once remarked that we judge the veracity of our axioms by their implications for our theorems, not our theorems by the axioms. We might say that we judge our symbols by what we can compute with them. This reflects a confident experimental and flexible attitude about foundations: They are important, but subject to change. The details become less relevant with increasing distance. This section conveys that spirit, as well as an engineering at titude about the use of symbols in the design of knowledge systems.

Many important problems require the use of special case knowledge to solve them efficiently. This chapter shows how this phenomenon arises, why it seems counterintuitive, and what the implications are for creating knowledge systems.

3

Knowledge and Software Engineering

In the summer of 1973, Patrick Winston gave the Computers and Thought Award lecture at the International Joint Conference on Artificial Intelligence at Stanford University. In this lecture he spoke of his experience teaching MIT undergraduates about the workings of an AI program, MACSYMA, that could integrate and differentiate expressions from the real calculus. The problems that the program solved were the same kinds of problems that are given in first-year calculus courses. Sample problems for the program included the differentiation and integration of trigonometric functions, polynomials, exponentials, and compositions of them.

According to Winston, the pedagogical sequence usually worked out in the same way. The students were familiar with numerical approximation methods for integration and differentiation and with computer programs that used these methods. They were surprised and impressed, however, that a program could perform integration and differentiation *symbolically*. In their view, such a program had a pretty good claim on being intelligent. The next part of the course explored how the program worked. The students learned that the program was organized around search. It had a collection of rules for differentiation and integration. It matched these rules against situations to decide which ones to apply. When the students understood that the AI program used the basic differentiation and integration rules taught in an introductory calculus course, their perception changed. A typical response was: "That program is not so smart! It integrates the same way I do."

In retrospect, it is not clear what Winston's students expected to find inside the integration program beyond knowledge (the integration and differentiation rules) and the search methods. Their surprise at finding these ingredients at the core of the system suggests that they were prepared culturally for something quite different, something awesome and mysterious. In all fairness, the students' surprise when they understood the workings of the integration program was tracking a shift that was spreading throughout the field of AI at the same time. AI was shifting from the pursuit of powerful search and reasoning methods toward a recognition of the role of special case knowledge—in this case the transforms of calculus. As Goldstein and Papert (1977) put it four years later in an often-quoted paper, there had been a shift of paradigm in AI from a technique-oriented theory of intelligence to a knowledge-oriented theory of intelligence:

> The fundamental problem of understanding intelligence is not the identification of a few powerful techniques, but rather the question of how to represent large amounts of knowledge in a fashion that permits their effective use and interaction. (p. 85)

This chapter is about the formulation and formalization of knowledge for knowledge systems. It attempts to bridge the gap that separates our everyday human understanding of how we discover, use, and articulate knowledge from our technical understanding of how we can incorporate knowledge in computational models.

3.1 *Understanding Knowledge Systems in Context*

We begin by discussing terminology that is widely used for describing knowledge systems. To illustrate issues in developing knowledge sytems, we then consider a sequence of settings in which knowledge systems are developed and used.

3.1.1 *The Terminology of Knowledge Systems and Expertise*

Since the 1970s when the term *expert system* came into use, terminology has shifted to reflect a deeper understanding of issues and distinctions. This section discusses the terminology, the shifts, and the reasons for the shifts.

What Knowledge Is, According to the Dictionary

Since the beginning of knowledge engineering, the term *knowledge* has been controversial when it is used as a description of something that computers can represent and use. Much of the struggle in making sense of knowledge engineering is in dealing with this word. In ordinary usage, the term *knowledge* is used imprecisely and sometimes synonymously with other words such as *data*, *information*, and *truth*. Within technical literature, however, there is potential for confusion and controversy. To illustrate this, consider the simpler word *information*. The term *information theory* is routinely applied to concepts for encoding bits for efficient transmission over communication channels. Information theory is about noisy signals, compact encodings, redundancy, bandwidth, and so on. Perhaps surprisingly, information theory has nothing to do with what information *means*. This is confusing because in common discourse the term *information* is used broadly, leading us to expect much more from a theory with that name.

Philosophers distinguish several kinds of knowledge, such as knowledge about what we perceive. Mathematical theorems are arguably formalized knowledge. There is knowledge about natural laws. There is knowledge about social, legal, and regulatory constraints. There is knowledge about effective reasoning in particular contexts.

Webster's New Twentieth Century Dictionary of the English Language (second edition) provides evidence of the broad usage of the term *knowledge*. This dictionary was published in

1968, very near the time at which knowledge engineering was getting started. This dictionary definition of knowledge does not say explicitly that knowledge can be held only by people, but it does not mention computers or even books. It offers the following seven meanings relevant to our purposes:

1. a clear and certain *perception* of something; the act, fact, or state of knowing; understanding.
2. *learning*; all that has been perceived or grasped by the mind.
3. *practical experience*; skill; as a knowledge of seamanship.
4. *acquaintance* or familiarity such as with a fact or place.
5. *cognizance*; recognition.
6. *information*; the body of facts accumulated by mankind.
7. *acquaintance with facts*; range of awareness, or understanding.

We begin with meaning 1, about perception. The term *perception* connotes the certainty provided by reliable sensory perception as portrayed by the expression "seeing is believing." Perception provides evidence about an external reality. All models of knowledge formulation rely on external evidence somehow. Knowledge system development includes feedback loops that ultimately involve the sensory abilities of the system developers and others. However, a strict reliance on perception is too confining and overstates the reliability of the senses. What did the magician really do? The senses can be fooled. Such strict reliance also ignores the utility of communication for conveying knowledge of distant events. For example, we all know a bit about great historical figures from the past even though none of us has seen them; nor have we seen the integers of mathematics. From a perspective of knowledge engineering, perception is a *basis* for knowledge. Perception plus prior knowledge plus rationality gives a basis for action. But perception should not be confused with knowledge itself.

Meaning 2 is about learning. This can refer either to academic learning, as in references to a learned scholar or to more mundane forms of everyday learning. ("Everyone makes mistakes, kid. Now tell me, what you have learned about playing baseball near windows?") The academic meaning is too restricted for knowledge engineering because academic concerns at any given time are only a subset of human concerns. The more mundane interpretation of learning encompasses methods for acquiring information through processes of abstraction, generalization, and model building. In knowledge engineering, machine-learning techniques formulate experience as knowledge. But computers can represent and use knowledge obtained from other agents, without having direct experiences themselves and without generalizing from cases themselves.

Meaning 3 is about practical experience. Experience is what knowledge is about and is essential for the creation of knowledge. But one must reflect on the experience to gain knowledge. The seamanship example conveys the idea that to be considered knowledgeable, a person must have a breadth of practical experience. The implication is that a person who has been at sea often enough will probably have encountered enough situations to acquire whatever he needs to know.

Meaning 4 refers to acquaintance or familiarity with a fact. Colloquially, we contrast someone who has "book knowledge" with others who have practical experience. A medical intern with book learning may be a riskier candidate for treating a patient than a seasoned doctor. The former's experience is less complete than that of the latter. From a perspective of knowledge engineering, acquaintance and familiarity refer to degrees of knowledge, but should not be confused with the nature of knowledge.

Meaning 5, about recognition, refers to a shallow degree of knowledge. If we recognize a face but cannot remember much about the person, we "know" the person but not very well. Recognition is often thought to be easier than generation, as in the case of people who can roughly understand a foreign language without being able to speak it. This meaning of the term *knowledge* is similar in status to meaning 4.

Meaning 6, about information accumulated by humankind, suggests that knowledge can be accumulated. This meaning of information is not the same as in information theory. It suggests that knowledge is somehow encoded.

Meaning 7 is about acquaintance with facts. We attribute knowledge to those who demonstrate broad competence. This meaning has some of the same force and limitations as meaning 3. It suggests that part of the work of knowing something is being able to apply that knowledge to a range of situations. We must reason in new situations using what we have acquired in specific ones. This implies an ability to infer and to generalize. For example, if an automobile driver knows that he is driving in a school neighborhood where children are playing, he is expected to drive his vehicle slowly and cautiously. Suppose something unusual happens in a driving situation, such as a wagon load of children's toys rolling into the street. The driver's handbook probably does not mention this precise situation. Nonetheless, if a driver fails to slow down and ultimately injures someone, a court of law will not accept excuses about the "incompleteness of a mental theorem prover" or claims that the driver never had exactly that experience before. As in meaning 3, having knowledge implies broad competence.

Looking back over these dictionary meanings of knowledge—perception, recognition, learning, experience, competence—it is noteworthy that they are all about relations and processes involving agents and their environments. Knowledge is not characterized by such properties as weight or extent. Knowledge is not a substance or a quantifiable property. It is not simply an encoding. What seems to matter and what we are inclined to describe when we characterize knowledge are the expectations it creates between environments, agents, and their rational actions. This stance on the meaning of the term *knowledge* is consistent with usage of the term in technical discussions about knowledge systems.

Defining Knowledge in Terms of Situations, Action, and Agents

Knowledge, as the word is used for knowledge systems, refers to the **codified experience** of agents. The experience is the source of the information for solving problems. By *codified*, we mean that the knowledge has been formulated, recorded, and made ready for use. This statement connects the practical intuitions of those who build knowledge systems, the theoretical foundations of knowledge as embedded representations to guide action, and the issues and problems that are driving the development of the field. The formulation as codified experience acknowledges that such experience is generally hard-won and valued.

Thinking about experience and isolating what is new in it is hard work. Experience must be articulated to become explicit knowledge, and in our sense, that requires more than just a listing of facts, such as the price of eggs or the postal addresses of the founders of AI. Codified experience must be organized and generalized to guide future action. We can formalize the process of knowledge creation in terms of the scientific method. Knowledge is that which is justified by our experience, or more formally, it is what we have learned from our experiments.

It is easy but misleading to overlook the roles of agents with respect to knowledge. When we refer to an "experience," some agent must interact with the world to have the experience. Usually this agent is assumed to be a person. When we refer to the codification of experience in symbols, some agent must conceive the symbols. We can say that books contain knowledge, but when we do so, we tacitly assume that there are people who can make sense of the writings in the books.

Agents are also involved when we consider written representations of knowledge. One of the insights in the past few years about the nature of knowledge and knowledge systems is that when meaning is attributed to systems and symbols there is necessarily an **agent**. As discussed in Chapter 1, symbols do not have meanings inherently; they are assigned meanings by an observer/agent and the assignment of meaning is in the mind of the observer.

If we say that "Computer system X knows that meningitis causes fever," we imply that from the perspective of some observers (such as ourselves), the computer system has representations of meningitis and fever; has representations of the relations between them; can form judgments about this situation based on generalizations of other situations; and can render and communicate its judgments involving the situations, meningitis as an infectious agent, and fever.

Thus, knowledge cannot be isolated from its creation or use. Experience involves agents in particular situations working on particular tasks with their own background assumptions. For example, a doctor seeing a patient with particular symptoms may quickly propose an explanation of why the patient is sick, knowing that there has recently been well-water contamination in the area. Such knowledge is particular to experiences of the time and place.

Knowledge as a "Transportable Substance"

Within AI and knowledge engineering, the term **knowledge acquisition** refers to any technique by which computer systems can gain the knowledge they need to perform their tasks. For newcomers and also those who have thought deeply about knowledge, the term is problematic. The term *acquisition* is odd because it suggests that knowledge is like a substance. In this view, knowledge is delivered using a transportation process from someone or somewhere and the crucial step in building a knowledge system is to "get" the knowledge.

The metaphor of knowledge as a "transportable substance" has long been used in informal English conversation. For example, books and teaching are said to "convey knowledge"; elders "pass along" knowledge to the next generation; knowledge and secret information sometimes "leak out." These metaphors reflect a truism. During education, knowledge becomes known by additional people. Knowledge systems themselves are sometimes thought of as knowledge distribution systems. However the process of spreading and knowing involves more than transportation.

The substance and transportation metaphor implies that knowledge starts out in a domain expert's head and that the object is to get the knowledge into a computer. This sounds like a simple matter that involves asking a few questions of an expert followed by some programming. Unfortunately, on this simplistic line of thought, many knowledge systems projects have floundered in months of misguided attempts at knowledge acquisition that never converge. Some projects fail utterly.

The transportation metaphor ignores the processes of knowledge creation and use. It does not consider that knowledge must somehow originate. It does not consider that knowledge can be implicit in the interactions of an agent with an environment. It does not consider that communication is at least two-way, involving the cooperative activity of agents that need to agree on spoken or written symbols and behaviors.

To emphasize the constructive and comparative aspects of building knowledge bases we prefer the term **knowledge formulation**. The formulation of knowledge includes identifying it, describing it, and explaining what it means. To emphasize aspects of writing and the choice of representation, we say that knowledge is **codified**. This suggests that besides transportation, the process of building a knowledge base also involves articulating essential elements of experience and preserving them in writing.

Knowledge often goes through stages of refinement in which it becomes increasingly formal and precise. Part of this process involves identifying the conditions under which the knowledge is applicable, and any exceptional conditions. It also involves organizing the representations so they can be used by a problem solver or other interpreter. To convey these meanings, the term **knowledge formalization** is used. This term helps us to characterize what happens as we progress from informal notes to formal computational models.

As knowledge systems are built, it is often the case that participants achieve greater insights about how to carry out the task. These insights come from the careful examination of particular cases and the search for better generalizations. This suggests that the knowledge that goes into knowledge systems does not simply originate from finished mental representations that already express all the information. Experts sometimes can give only vague accounts of their thinking. But they do know how to make their thinking more precise through experiments. In this experimentation process, knowledge is often created or discovered.

In summary, the process of developing a knowledge base involves more than transportation of facts. Historically, the process has been called knowledge acquisition. Recognizing other dimensions of what is needed helps us to make sense of the process and the methods by which people approach it.

Expert Systems and Knowledge Systems

The terms **expert system** and **knowledge system** were invented in the context of artificial intelligence research and partly in reaction to it. The term *intelligence* has always been problematic because there is no widespread agreement about exactly what intelligence is.

In this context, the developers of the first expert systems were interested in what they saw as practical problems. They took scientific reasoning and practical problem solving as their central examples of intelligent behavior. They adopted interview techniques, protocol analysis, and other methods of information-processing psychology to investigate the nature of these task domains. As they built computational models, they were struck by the amount of special case knowledge that was brought to bear in solving the problems efficiently. This difference dominated the character of the programs and was the context for the "knowledge and expertise" terminology.

An **expert system** is a computer program whose performance is guided by specific, expert knowledge in solving problems. The problem-solving focus is crucial in this characterization. The knowledge of central interest in expert systems is that which can guide a search for solutions. The term *expert* connotes both narrow specialization and substantial competence. Having a narrow focus is essential to the feasibility of a system. Although the term *expert* has been loosely applied in some cases, it is intended to describe systems that solve problems that are otherwise solved by people having substantial training and exceptional skill. Thus the standard of performance for expert systems is in human terms, by comparison with people carrying out a particular kind of task.

The first expert systems were characterized as **expert-level advisors** or **consultants**. Expert systems are now being used in a wide range of different interactive roles. To the consultant metaphor, we now add other metaphors describing the role and interactivity of a system: the smart spreadsheet, the intelligent patient monitor, the financial advisor, the scheduling assistant, the therapy critic, the cognitive coprocessor. These terms suggest different ways of thinking about the split of initiative and responsibility between knowledge systems and their users. In whatever role we employ expert systems, those systems require knowledge to be competent.

Even in the most successful applications where expert systems outperform human experts in their reliability and consistency of results, expert systems have less breadth and flexibility than human experts. This has created confusion about the suitability of the term *expert*, often resulting in unproductive arguments about the quality and boundaries of expertise.

The term **knowledge system** is a shorthand for the term *knowledge-based system*. A knowledge system is a computer system that represents and uses knowledge to carry out a task. As the applications for the technology have broadened, the more general term *knowledge system* has become preferred by some people over *expert system* because it focuses attention on the knowledge that the systems carry, rather than on the question of whether or not such knowledge constitutes expertise.

The Parts of a Knowledge System

The classical formulation of a knowledge system is shown in Figure 3.1. This formulation has become somewhat frayed around the edges, and we offer alternative formulations shortly. Nonetheless, familiarity with it is worthwhile because the terminology is still widely used.

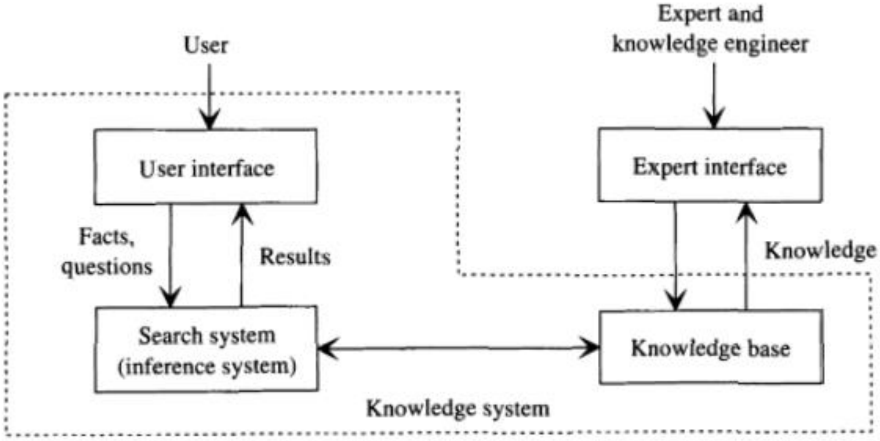


FIGURE 3.1. The classical characterization of a knowledge system.

The classical formulation includes a knowledge base, two human-computer interfaces, and a search or inference subsystem. The **knowledge base** is the repository for the knowledge used by the system—the rules and hints for guiding the search for solutions. The knowledge base is updated periodically to reflect changes or extensions to the domain knowledge.

The **user interface** is the part of a knowledge system that interacts with the system's primary users. It is responsible for all communications in the system's role as a consultant for solving problems in its domain, such as asking questions about the problem at hand, answering queries, and displaying results.

The second human-computer interface in the classical formulation is the **expert interface**. This is the interface by which knowledge is entered into the system. The expert interface is used by a knowledge acquisition team consisting of an expert and a knowledge engineer. The expert interface provides access for updating, testing, and debugging a knowledge base, including tools for examining the contents of the knowledge base. By implication, not all users can update a system's knowledge base. That responsibility is assigned to a smaller group that oversees and approves changes and takes into account effects across a wide range of situations. In personal knowledge systems or in knowledge systems that allow personalizing a knowledge base, the responsibilities and interfaces may be partitioned differently.

The inference subsystem is the part that reasons its way to solutions of problems, with its search guided by the contents of the knowledge base. Traditionally, colorfully, and colloquially, this part has been called the **inference engine**. This part of a knowledge system must include provisions for setting goals, representing and recording intermediate results, and managing memory and computational resources.

These terms—*knowledge base, search system, expert* and *user interfaces*—all name parts of a program or computer system. However, an inventory of program parts is not the best way to understand a knowledge system. Systems for different purposes and based on different approaches look too much alike by these top-level categories. Nor does further decomposition into smaller parts necessarily help us to understand how different systems work or what their limitations are.

Another problem with this classical formulation of expert systems in terms of their parts is that the formulation does not say precisely what the roles of the different parts are. Consider the knowledge base. In some systems the knowledge-base subsystem does no more than manage a collection of data structures, perhaps providing some search facilities. In others, the knowledge-base subsystem performs inference on the representations. With such variations in the architectural components, the terms give us little guidance for developing or understanding systems.

To make sense of knowledge systems and the knowledge that they carry, we need to step back and observe more than the apparent organizational structures of the programs. We need to look at a knowledge system both in its problem-solving context and in the social and organizational contexts in which knowledge is created, communicated, and reinterpreted. We begin by defining some terms about knowledge and work settings. In the next section, we consider a sequence of scenarios that provide contexts for understanding the processes by which knowledge and knowledge systems can be developed. First, we introduce terms about knowledge and activity.

Bodies of Knowledge

A **domain** is a body of knowledge. The subject matter of a domain must be recorded or carried somehow. It may be recorded in written literature or carried by people and conveyed verbally or by apprenticeship training. The term *domain* does not connote anything about the amount of knowledge included. Often a domain is either a field of academic study or a professional area. Internal medicine, health care, diets for diabetics, agriculture, law, civil engineering, and force transducer design are all examples of domains. Different domains have different degrees of specialization. For example, estrogen chemistry is more specialized than organic chemistry, which is more specialized than general chemistry.

A **task** is a kind of job that is done. In the context of knowledge systems, a task involves solving a kind of problem. Tasks can be described at different levels of generality. Very general tasks include such things as diagnosis, configuration, design, planning, and scheduling. An example of a very specialized task would be "inferring DNA segmentation structure from restriction enzyme digest data."

When we refer to **domain knowledge**, we mean the general terminology and facts of a domain without a focus on a particular task. For example, we might refer to a general corpus of facts about chemistry without considering whether the task was to plan a chemical synthesis or to interpret the data from an instrument. When we use the term **task knowledge**, we refer to the terminology, computational models, and facts associated with carrying *out* a kind of task, without necessarily focusing on a particular domain. For example, we might consider the knowledge and models about a scheduling task, without regard to whether it was job-shop scheduling for a manufacturing plant or course and classroom scheduling for a university.

The term **task domain** combines these ideas. A task domain is the knowledge, assumptions, and requirements associated with doing a particular task in a particular domain. The term is usually used in the context of a particular set of people doing a specialized task. "Diagnosing infectious diseases" is an example of a task domain. A **problem** is a particular instance of a task. It consists of a particular set of input data together with the accepted answers. A **case** is an example of a problem that includes the given information, the methods used, and the results obtained.

3.1.2 *Knowledge Systems and Document Systems: Five Scenarios*

Putting knowledge into computers raises many foundational questions. What is knowledge? Where does it come from? How is it created? How is it held by computers? In the context of particular knowledge systems, these questions tend to have concrete and immediate answers. We now present five scenarios in which knowledge systems are developed and used.

Personal Knowledge Systems

The first scenario is a **personal knowledge system** as illustrated in Figure 3.2. In this scenario a person has a sample situation or device in mind. It might be a radio, an electronic circuit, or a billing and inventory system. The task might be to fix the radio, to design the circuit, or to develop a purchasing schedule that reduces warehouse costs. Carrying out the task is framed in terms of a search for solutions. We refer to the person who builds the knowledge system as the system builder.

There are several motivations for creating personal knowledge systems. For example, the system builder may want the knowledge system to solve routine problems or routine parts of problems so he can focus on harder and more interesting ones. Alternatively, he may want to use the knowledge system to improve his productivity, check his work, or extend his methods to larger problems than he can solve by hand. He can rely on the tirelessness of an automatic system

Configuration tasks select and arrange instances of parts from a set. Computational models for configuration are used for build-to-order manufacturing tasks and also for tasks such as planning and therapy recommendation. Incremental decision-making methods for configuration must cope with threshold effects and horizon effects.

8

Configuration

8.1 Introduction

A **configuration** is an arrangement of parts. A **configuration task** is a problem-solving activity that selects and arranges combinations of parts to satisfy given specifications. Configuration tasks are ubiquitous. Kitchen designers configure cabinets from catalogs of modular cupboards, counters, closets, drawers, cutting boards, racks, and other elements. Salespeople for home entertainment systems must design configurations from tape decks, optical disk players, receivers, turntables, and other units that can play recordings from different media, as well as from video screens, speakers, and other units that can present performances to people. Computer engineers and salespeople configure computer systems from various computing devices, memory devices, input and output devices, buses, and software. Dieticians configure diets from different combinations of foods. Configuration problems begin with general specifications and end with detailed specifications of what parts are needed and how they are to be arranged.

A definitional characteristic of configuration, as we shall use the term, is that it instantiates components from a predefined, finite set. This characterization of configuration does not create new parts or modify old ones. Configuration tasks are similar to classification tasks in that instantiation includes the *selection* of classes. However, to solve a classification problem is merely to select one (or perhaps a few) from the predefined set of classes. To solve a configuration problem is to instantiate a potentially large *subset* of the predefined classes. The search space of possible solutions to a configuration problem is the *set of all possible subsets* of components. This is the *power set* of the predefined classes. Furthermore, multiple instantiations of the same part may be used in a solution, and different *arrangements* of parts count as different solutions. Configuration problems are usually much harder than classification problems.

8.1.1 Configuration Models and Configuration Tasks

Custom manufacturing and mass manufacturing are often seen as opposites. Custom manufacturing tailors products to the specific and different needs of individual customers; mass manufacturing achieves economies of scale when high-volume manufacturing techniques make it possible to reduce the unit costs of nearly identical products. An industrial goal common to many kinds of manufacturing is to combine the flexibility of custom manufacturing with the economics of mass production. An important strategy for this is to create lines of products that can be tailored to a customer's requirements by configuring customer-specific systems from a catalog of mass-produced parts. This approach has been called an a la carte or build-to-order marketing and manufacturing strategy.

A la carte manufacturing requires special capabilities of an organization. For production, the required capabilities include the efficient means for managing of an inventory of parts, for bringing the parts together that are required for each order, for assembling the parts into products, and for testing the quality of the different assembled products. A well-recognized logistic requirement for this process is to accurately understand the customer's needs and to develop a product specification that meets those needs. This step, which actually precedes the others, is the configuration step. The success and efficiency of the entire enterprise depends on configuration.

Since flexible product lines can involve hundreds or thousands of different, configurable parts, there are many possibilities for errors in the configuration process. Errors in a configuration can create costly delays when they are discovered at assembly time, at testing time, or by a customer. An effective configuration process reduces costs by reducing level of inventory, reducing the amount of idle manufacturing space, reducing the delay before customers' bills are paid, and better ensuring customer satisfaction.

As these factors have been recognized, companies have sought automated and semi-automated configuration systems for routinely creating, validating, and pricing configurations to meet the differing requirements of their customers. One of the first and best known knowledge-based configuration systems is the XCON system for configuring VAX computer systems (McDermott, 1981). Knowledge-based systems for configuration tasks are being built by many organizations.

Configuration is a special case of design. In design the elements of the designed artifact are not constrained to come from a predefined set. They are subject only to the constraints of the manufacturing methods and the properties of the raw materials. The knowledge necessary for configuration tasks is more bounded than the knowledge for general design tasks. In general design tasks, the solution space is more open-ended. For example, designing a pickup truck from scratch is more open-ended than choosing options for a new one. In configuration tasks, much of the "design work" goes into defining and characterizing the set of possible parts. The set of parts must be designed so they can be combined systematically and so they cover the desired range of possible functions. Each configuration task depends on the success of the earlier task of designing configurable components.

From a perspective of knowledge engineering, configuration tasks are interesting because they are synthetic or constructive tasks. They are tasks for which combinatorics preclude the pre-enumeration of complete solutions. Just as not all classification tasks are the same, not all configuration tasks are the same. Configuration models can be used as a basis for different kinds of tasks, not necessarily involving manufacturing and physical parts. For example, "configuring" therapies can be a part of a larger diagnosis and therapy task. Configuring alternative combinations of steps for a plan can be an important part of a planning process. In such cases we refer to a **configuration model** for a task.

Knowledge-engineering techniques are suitable for classification tasks and configuration tasks because most of the complexity in specifying how to carry out the task is in the domain knowledge rather than the methods themselves. The knowledge-level methods define roles for different kinds of knowledge and these roles help to control the way that the bodies of knowledge interact and guide the search for solutions.

8.1.2 *Defining Configuration*

Selecting and arranging parts is the core of solving a configuration problem. Figure 8.1 presents an example of a configuration that is a solution to a configuration problem. In this example, the configuration has two top-level parts: Part-1 and Part-2. Each of these parts has subparts.

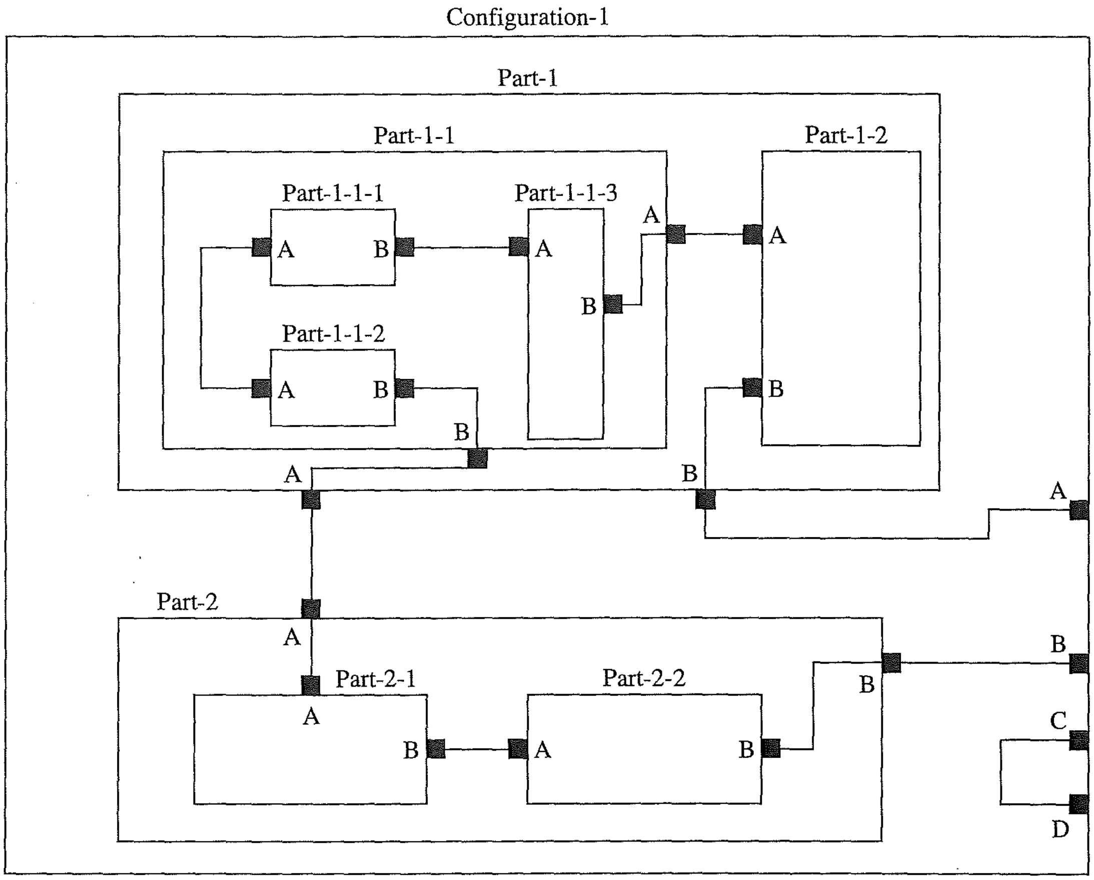


FIGURE 8.1. An example of a configuration using a port-and-connector model. In this figure, all components are shown as boxes and the top-level component is Configuration-1. It has two parts, Part-1 and Part-2. The arrangement of the configuration is indicated by the part-of hierarchy, displayed graphically by the containment of boxes and by the interconnections among parts. Each part has a number of ports, labeled A,B,C, or D. For example, the B port of Part-1-1-1 is connected to the A port of Part-1-1-3.

Configuration domains differ in their representations of arrangements. For example, Figure 8.1 represents arrangements using a **port-and-connector model**. The ports on each part correspond to the different roles that subparts can have with respect to each other. Like all models governing arrangements, a port-and-connector model constrains the ways things can be connected. Components can be connected only in predefined ways. Ports may have type specifications indicating that they can be connected only to objects of a particular kind. Each port can carry only one connection.

Variations in the search requirements and the available domain knowledge for different configuration tasks lead to variations in the methods. Nonetheless, certain regular patterns of knowledge use appear across many different configuration tasks. Configuration methods work through recognizable phases. They map from user specifications to abstract descriptions of a configuration, and they refine abstract solutions to detailed configurations specifying arrangements and further requirements. We distinguish between solution expansion and solution refinement phases, not to imply that they are independent, but to emphasize the distinct meanings of part requirement hierarchies, functional abstraction hierarchies, and physical containment hierarchies. Addition of required components is often a specialized process and can involve goals for both correcting and completing a specification. Refinements of specifications include the consideration of alternative arrangements as well as selection among alternative implementations. Configuration methods must mix and balance several kinds of concerns and different bodies of knowledge.

Figure 8.2 shows the relevant spaces for defining configuration problems. It is a useful starting framework from which we shall consider variations in the following sections. In some domains there is no separate specification language: Specifications are given in the language of

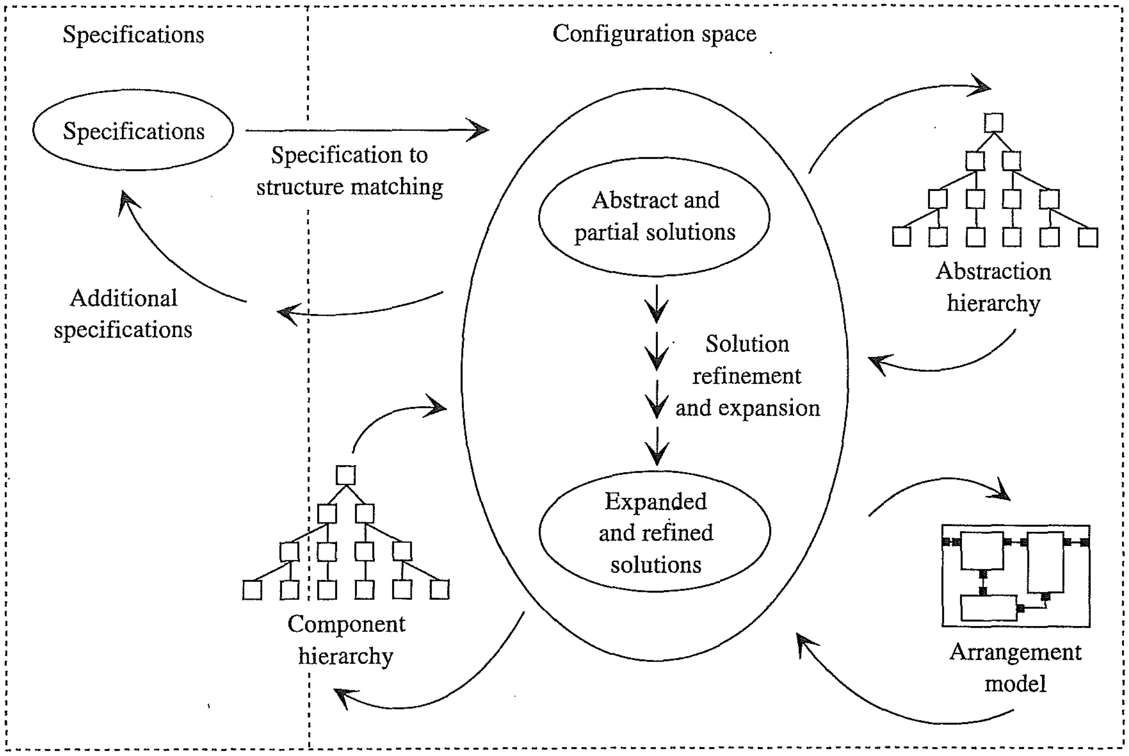

Specifications
Configuration space
Specifications
Specification to structure matching
Additional specifications
Abstract and partial solutions
Solution refinement and expansion
Expanded and refined solutions
Abstraction hierarchy
Component hierarchy
Arrangement model

FIGURE 8.2. Spaces in configuration tasks.

parts and arrangements and the task begins with a partial configuration. In that case, we omit the boundary in the figure between the specification space and the configuration space. The outer oval in the configuration space signifies that the representations for partial and expanded solutions are mixed together. Most configuration methods work incrementally on candidates, expanding the candidates to include more parts and refining the specifications to narrow the choices.

By analogy with names for classification methods, the pattern of knowledge use suggested by Figure 8.2 could be called **heuristic configuration** to emphasize the use of heuristic knowledge to guide the generation and testing of possible configurations. It could also be called **hierarchical configuration** to emphasize the importance of reasoning by levels, both component hierarchies and abstraction hierarchies. For simplicity, we just use the term *configuration.*

8.2 *Models for Configuration Domains*

This section discusses what configuration is and what makes it difficult.

8.2.1 *Computational Models of Configuration*

To understand configuration tasks we need a computational model of the search spaces and the knowledge that is used. Our model has the following elements:

- A specification language
- A submodel for selecting parts and determining their mutual requirements
- A submodel for arranging parts
- A submodel for sharing parts across multiple uses

We begin by describing these elements generally and simply. We then introduce the "widget" domain as a concrete instantiation of a configuration model. We use this domain to illustrate reasoning and computational phenomena in configuration tasks. Afterward, we draw on the domains from the previous section to extract examples of kinds of knowledge and the use of knowledge.

A **specification language** for a configuration task describes the requirements that configurations must satisfy. These requirements reflect the environment in which the configured product must function and the uses to which it will be put. A specification may also indicate which optimizing or due-process criteria should be used to guide the search, such as minimizing cost or space or preferring some possibilities over others. (See the exercises.)

A functional specification describes capabilities for desired behaviors. For example, rather than saying that a computer configuration needs a "printer," a functional specification might say that the computer needs to be able to print copies. The printing function may be further specialized with qualifiers about speed, resolution, directionality, character sets, sizes of paper, color or black and white, and so on. One advantage of describing systems in terms of functions rather than in terms of specific classes of manufactured parts is that functions can anticipate and simplify the addition of new classes of components to a catalog. When a functional specification language is used, the configuration process must have a means for mapping from function to structure.

A specification language is not defined independently of the other submodels. Some configuration models use the same language for describing specifications and for describing configurations. This approach avoids the function-to-structure mapping by associating each part with a list of key functions. The most common approach is called the **key-component approach**. This approach has two parts: (1) For every major function there is some key component, and (2) all the key components (or abstract versions of them) are included in the initial partial configuration that specifies a configuration. A specification is assumed to include all the key parts. To complete a configuration, a configuration system satisfies the prerequisites of all the key parts it starts with as well as the parts it needs to add to the specification, and it also determines a suitable arrangement for all the included parts.

A **submodel for parts** specifies the kinds of parts that can be selected for a configuration and the requirements that parts have for other parts. Some parts require other parts for their correct functioning or use. For example, computer printed circuit boards may require power supplies, controllers, cables, and cabinets. A part model defines required-component relations, so that when a configuration process considers a part description, it can determine what additional parts are needed. Required parts can be named explicitly or described using the specification language. Descriptions indicate which parts are compatible with each other and what parts can be substituted for each other under particular circumstances.

A **submodel for spatial arrangements** provides a vocabulary for describing the placement of parts and specifies what arrangements of parts are possible. Together, the arrangement model and the specification language form a basis for describing which arrangements are acceptable and preferred. They make it possible to determine such things as where a part could be located in an arrangement, whether there is room for another part given a set of arranged parts, and which parts in a set could be added to an arrangement. Arrangement models constrain the set of possible configurations because they govern the consumption of resources, such as space, adjacency, and connections. These resources are limited and are consumed differently by different arrangements. Arrangement models are a major source of constraints and complexity in configuration domains.

A **submodel for sharing** expresses the conditions under which individual parts can be used to satisfy more than one set of requirements. In the simplest case, part use is mutually exclusive. For example, a cable for one printer cannot be used simultaneously to connect to another printer. For some applications, mutual exclusion is too restrictive, such as with software packages that may use the same memory, but at different times. The following exemplifies a range of categories of use and sharing for configuration systems:

- *Exclusive use*: Components are allocated for unique uses.
- *Limited sharing:* Parts can be shared between certain functions but not across others.
- *Unlimited sharing:* A component can be allocated for as many different purposes as desired.
- *Serial reusability:* A component can be allocated for several different purposes, but only for one purpose at a time.
- *Measured capacity:* Each use of a component uses up a fraction of its capacity. The component can be shared as long as the total use does not exceed the total capacity. The electrical power provided by a supply is an example of a resource that is sometimes modeled this way.

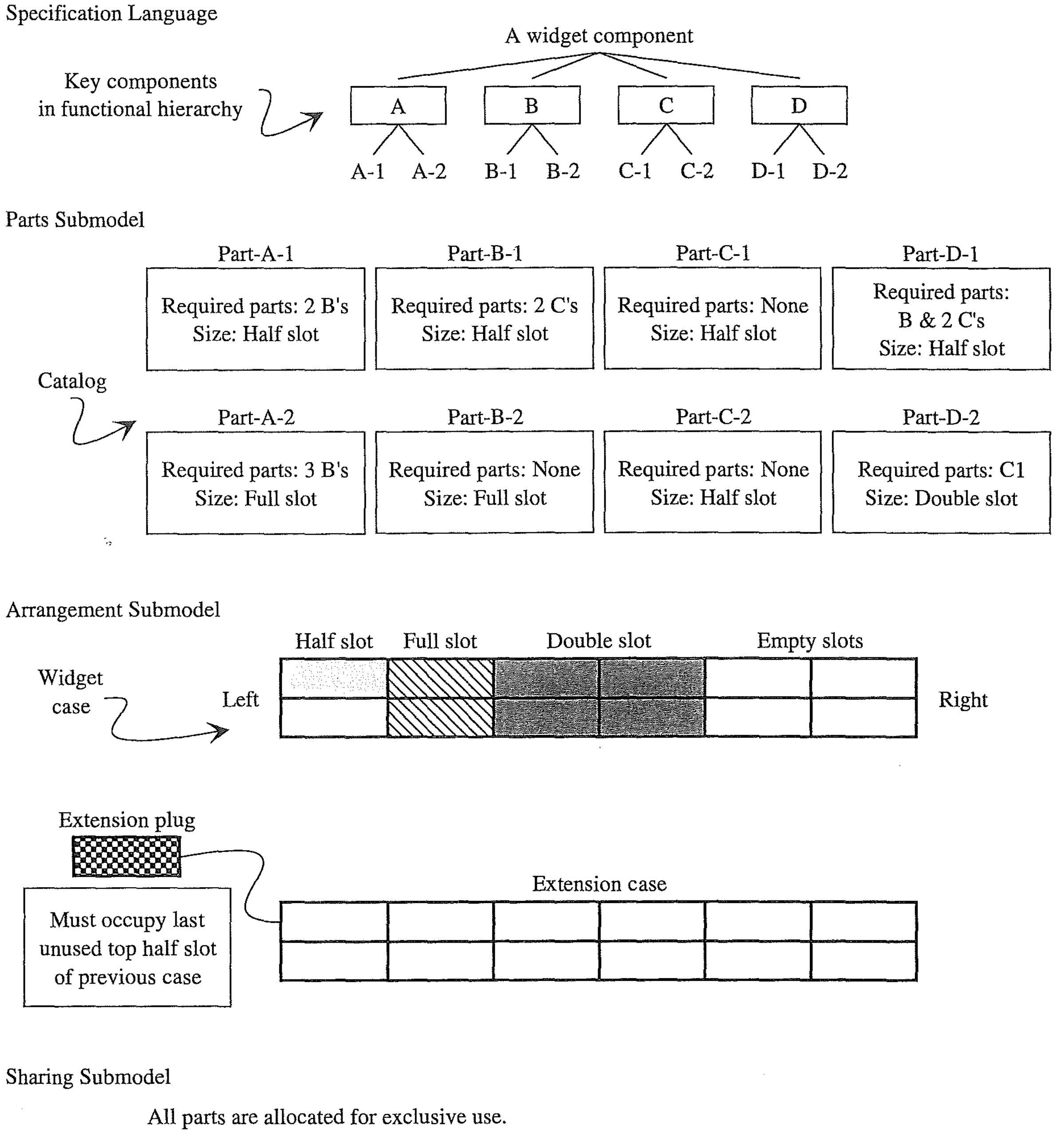


FIGURE 8.3. The widget model, W-1, illustrates phenomena in configuration problems.

A particular domain model may incorporate several of these distinctions, assigning different characteristics to different parts. Sometimes these distinctions can be combined. One way to model how different software packages could share memory would be to post capacity constraints indicating how much memory each package needs. The system allocation for serial usage would be determined as the maximum capacity required for any software package. This approach mixes serial reusability with measured capacity.

In summary, our framework for computational models of configuration has four submodels: a specification language, a submodel for parts and requirements, a submodel for arrangement, and a submodel for component sharing. We return to these models later to describe variations on them and to identify the ways in which they use domain knowledge. In the next section, we instantiate our configuration model for the widget domain and use it to illustrate important phenomena in configuration tasks.

8.2.2 *Phenomena in Configuration Problems*

This section describes reasoning phenomena that arise in configuration problems. These phenomena are illustrated in the context of the sample domain in Figure 8.3. Figure 8.3 presents W-1, a model for configuring widgets. We use W-1 to illustrate configuration phenomena in the following discussion and in the exercises at the end of this section.

Widget requirements are specified in terms of the key generic components A, B, C, and D. The W-1 parts submodel provides a catalog with two choices of selectable components for each generic component. Thus, generic component A can be implemented by either A-1 or A-2, B can be implemented by B-1 or B-2, and so on. Each component requires either a half slot, a full slot, or a double slot. Figure 8.4 presents the rules for the widget model.

When components are arranged in the widget model, there are sometimes unavoidable gaps. When configurations are combined, these gaps are sometimes complementary. For this reason it is not always adequate to characterize the amount of space that a configuration takes up with a single number. Figure 8.5 defines several terms related to measures of space in the widget model. **Occupied space** refers to the number of slots occupied by components, including any required extension plugs. **Trapped space** refers to any unoccupied slots to the left of other slots, including slots to the left of or below an extension plug. **Space** (or total space) is the sum of occupied space and trapped space. The **minimum space** for a specification is the minimum amount of space required by any possible configuration that satisfies the specification.

Required Parts

Each part in a configuration must have all of its required parts, as shown in Figure 8.3.

Arrangement

Components must be arranged in alphabetical order (left to right and top to bottom). Parts with the same letter (e.g., C-1 and C-2) can be mixed in any order.

All identical components (components with the same letter and number) must be in the same widget case.

A full-slot component must occupy a single vertical slot. It cannot be split across slots. A double-slot component must occupy two adjacent full slots in the same widget case.

An upper half slot must be filled before the corresponding lower half slot can be filled.

No full-slot gaps are allowed, except at the right end of a case.

If a case is full, an extension case can be added but it requires using an extension plug in the last upper half slot of the previous case.

When an extension plug is used, no component can be placed in the half slot below the plug.

Sharing

The W-1 sharing submodel requires that distinct parts be allocated to satisfy each requirement.

FIGURE 8.4. Further rules governing the W-1 widget model.

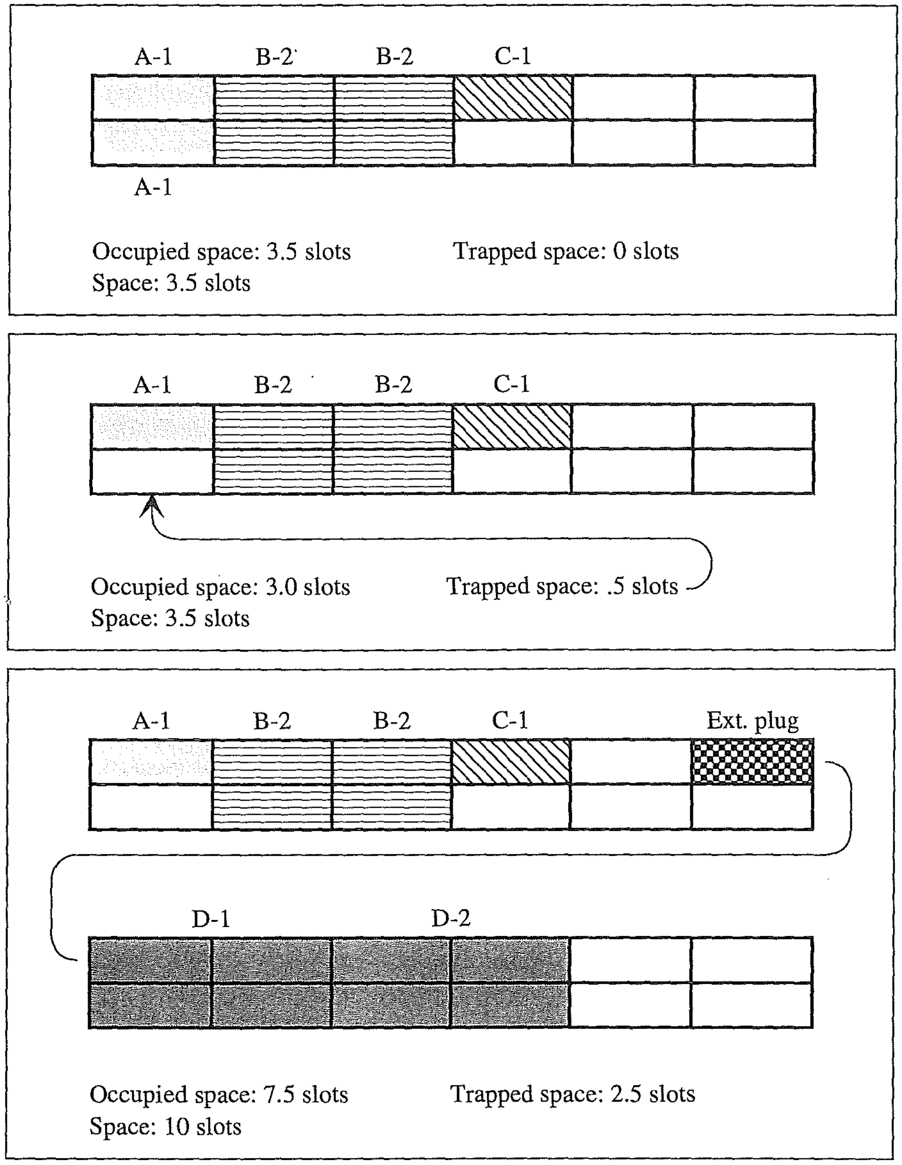


FIGURE 8.5. Examples illustrating the terms total space, occupied space, and trapped space. For most problems, the key factor is the total space, which is called simply "space."

Configuration decisions are made incrementally. Figure 8.6 illustrates stages in a solution process, starting with the initial specifications {A, D}. Under the part-expansion phase, alternatives for required parts for A and D are considered. For requirement A, the candidates are the selectable parts A-1 and A-2. However, A-1 and A-2 both have further requirements. A-1 expands to two B's, which could be implemented as either B-1's or B-2's. If B-1's are chosen, these require C's, which themselves can be implemented as either C-1's or C-2's. This expansion of requirements shows why we cannot use a simple constraint-satisfaction method with a fixed number of variables and domains to solve the configuration task for the top-level parts. Depending on which alternatives are chosen, different required parts will be needed, and different fur-

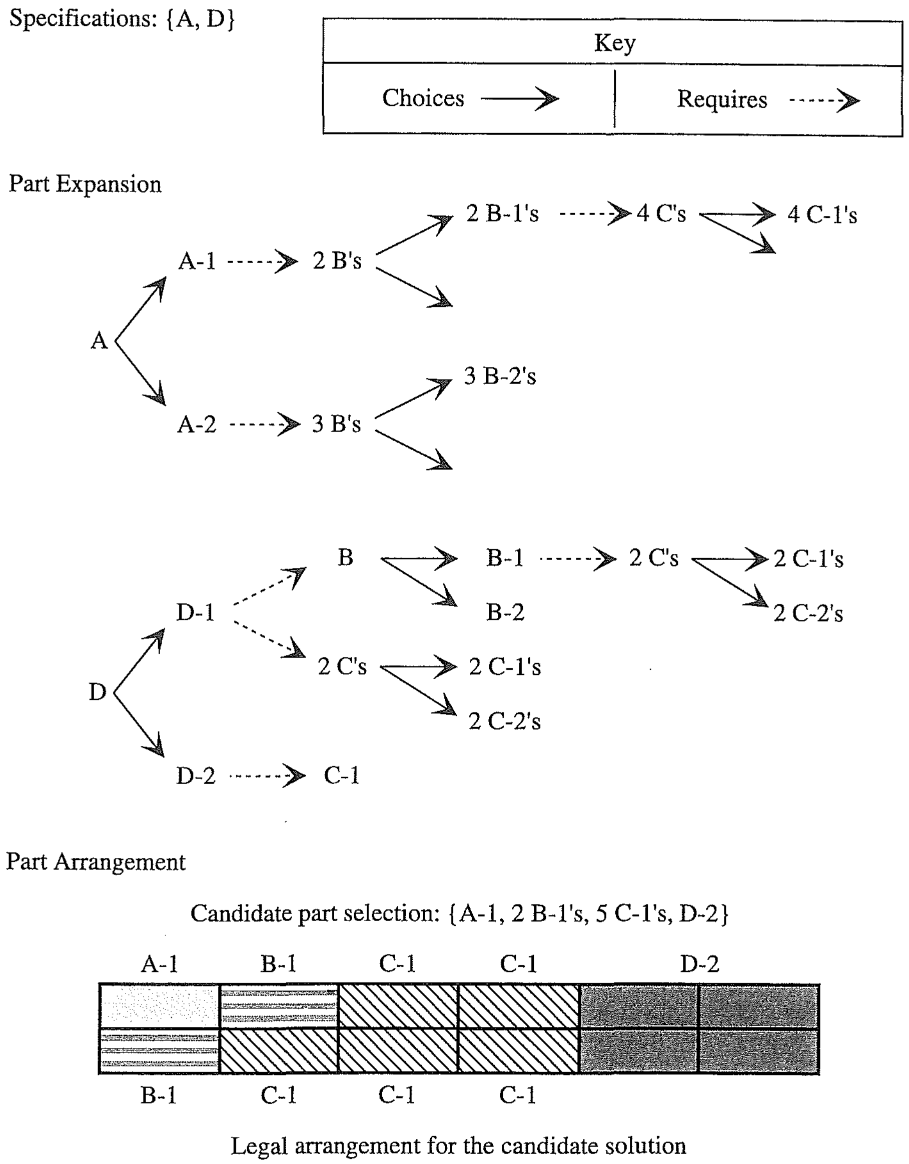


FIGURE 8.6. The dynamic nature of configuration problems. In this example, the initial specifications are {A, D}. The solid arrows indicate where there are alternatives for implementing a part. The dashed arrows indicate places where a part requires further parts. There are several possible solutions for the given specifications. One solution is shown at the bottom. This one fits within a single widget case.

ther requirements will be noted. This dynamic aspect of the problem suggests the use of dynamic and hierarchical constraint methods.

The arrangement model for W-1 lends itself to a sequential layout process, which first places the required A components and then the B components, C components, and D components in left-to-right order. If at any point all of the components of the same type cannot be fitted into the space remaining in a widget case, an extension case is required. If the number of required components of a given type is more than fits into a single case, then the candidate cannot be arranged according to the rules. Furthermore, when more than one case is required, there must be an available slot in the previous case for its extension plug. At the bottom of Figure 8.6 is one solution to the sample problem, which fits within a single widget case.

Threshold Effects

Configuration problems exhibit **threshold effects**. These effects occur when a small change in a specification causes candidates to exceed certain fixed thresholds, requiring discontinuous and potentially widespread changes to the candidate. For example, in computer configuration domains, overflowing the capacity of a cabinet requires the addition of not only an extension cabinet, but also various components for extending the bus and additional power supplies. In applications where parts have parameters to be determined by configuration, requirements for upgrading the size of components can have ripple effects, so that further parts need to be changed to accommodate changes in weight or size, resulting in shifts to different motor models. Depending on how the decision making is organized in a configuration system, threshold effects can occur in the middle of solving a problem if requirements discovered partway through the process exceed initial estimates of required capacity. In such cases, many early decisions may need to be revised (see Exercise 4).

Figure 8.7 illustrates a threshold being exceeded when the widget specification is changed from {A, D} to {A, 2 D's}. In the two solutions shown, the additional required components exceed the capacity of a single widget case. Furthermore, in the second candidate solution, some of the implementations for a C part were switched to C-2 (rather than C-1) to avoid violating the rule that identical parts must be placed in the same case and to avoid overflowing the extension case. The capacity of a case is a resource. Threshold effects can occur for different kinds of resources. Other arrangement resources include order and adjacency.

Global resources that can be exceeded in different configuration domains include size, power, weight, and cost. These resources present special challenges for allocation and backtracking. They are called **global resources** because they are consumed or required by components throughout a configuration rather than by just one component. In the cases cited, the consumption of the resource is **additive consumption**, meaning that requirements can be characterized in terms of numeric quantities. Requirements from different parts are summed together.

Horizon Effects

When configuration decisions are made incrementally, their evaluation criteria are typically applied in a local context, meaning that they recommend the best solution limiting the evaluation to a small number of factors. This due-process approach is called myopic because it does not try to take into account all of the entailments of a decision caused by the propagation of effects in the model. This leads to a phenomenon called the **horizon effect**. In nautical experience, the curvature of the earth defines a horizon beyond which things are hidden. To see things farther away, you have to get closer to them. In reasoning systems, the "visible horizon" corresponds to things that have been inferred so far. Partway through a configuration process, you cannot see all the consequences of the choices so far because some inferences have not yet been made and because some other choices have yet to be made. At best, you can make estimates from the decisions made so far.

In the idealized case where configuration choices are independent, locally optimal decisions yield globally optimal solutions. For example, if the total cost of a solution is the sum of

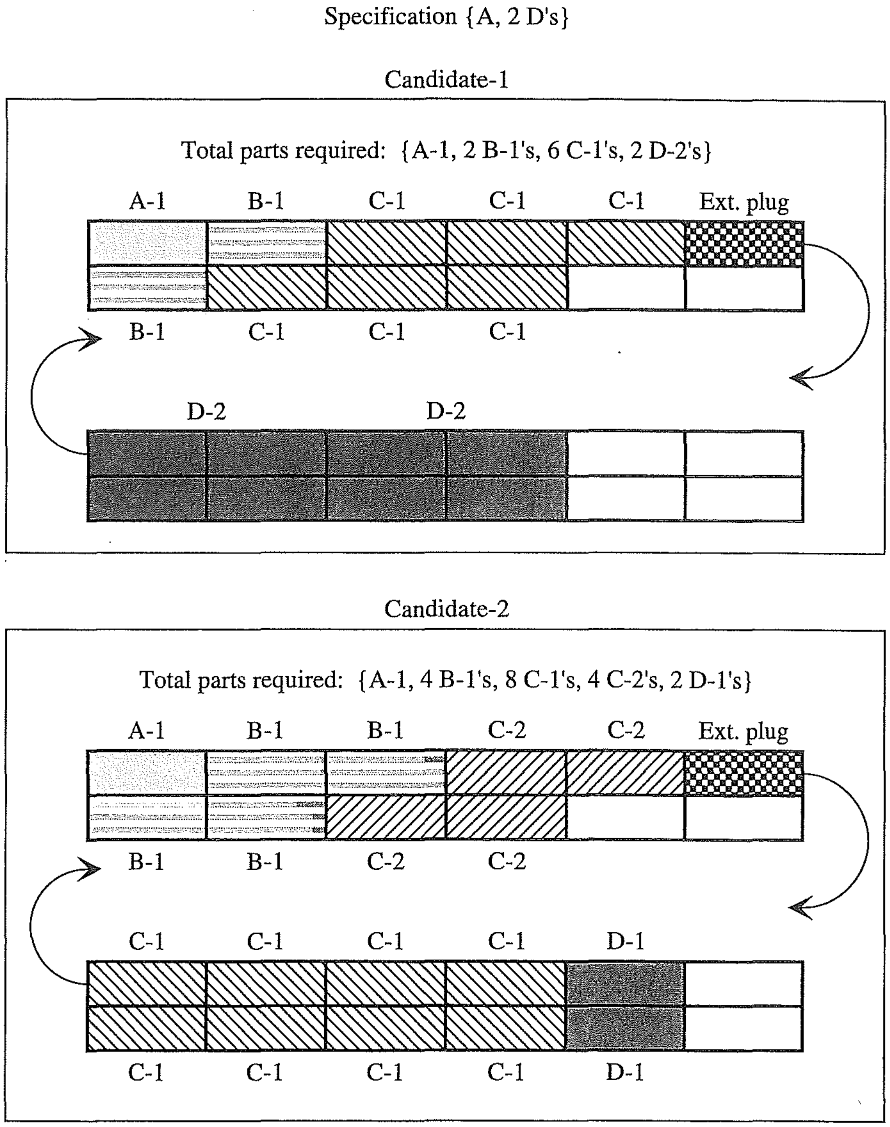


FIGURE 8.7. Threshold effects. This figure illustrates two possible solutions to a widget configuration problem with the specifications {A, 2 D's}. Both candidate solutions exceed the capacity of a single widget case. In more involved examples, exceeding one threshold can lead to a ripple effect of exceeding other thresholds as well, leading to widespread changes in a configuration.

the costs of the independently determined parts, then picking the cheapest part in each case will lead to the cheapest overall solution. If the choice of the next component can be determined by combining the known cost of a partial solution with an estimate of the remaining cost, and the estimate is guaranteed never to be too high, then the A* algorithm can be used as discussed in Section 2.3.

However, in many configuration situations and domains these conditions do not hold and myopic best-first search does not necessarily lead to a globally optimal solution. In each of the following cases, both part A and part B satisfy the local requirements. Part A costs less than part

B and would be chosen by an incremental best-first search that is guided by cost of the part. These cases show why a choice that is locally optimal need not be globally optimal.

Case 1. *Undetected required parts.* The other parts required by part A are much more expensive than the parts required by part B, so the total cost of a complete solution is higher if A is chosen. The further requirements for parts A and B may involve constraints not yet articulated, choices dependent on situations not yet determined, or design variables not yet introduced.

Case 2. *Unanticipated arrangement conflicts.* The parts required by part A consume slightly more of some resource than the parts required by part B. If A is chosen, a threshold will be crossed by later decisions leading to additional expensive cabinets, cases, or whatever. If B had been chosen, the threshold would not have been crossed and the overall solution would have been cheaper (see Exercise 4).

Case 3. *Undetected sharability.* There are requirements not yet considered for a part that is compatible with B but not with A. If B is a sharable component, then choosing B at this stage will make it unnecessary to add parts later on, leading to an overall savings. This phenomenon is especially likely to occur in domains that include multifunctional components, such as multifunctional computer boards that incorporate several independent but commonly required functions (see Exercise 5).

All three cases exhibit the horizon effect. The local evaluation fails to lead to an optimal solution because the indicators fall past the "observable horizon" defined by the commitments and inferences made so far.

If computational complexity was not an issue, the effects of the first case could be mitigated by determining the problematic effects of the required components for a part before committing to a part. Thus, no commitment would be made to choose A until the effects of all of its required components are known. The second case could be mitigated by considering the effects on all resources consumed by a component. The effects in the third case could be mitigated by considering all possible sharings of a component before committing to the selection.

To summarize (with tongue in cheek), the effects of incremental decision making and myopic optimization can be mitigated by performing decision making as far past the immediate horizon as needed. Unfortunately, exponential computational resources are required to look so far ahead. In the following knowledge and symbol-level analyses, we discuss more realistic ways to cope with threshold and horizon effects.

8.2.3 Summary and Review

This section models configuration in terms of four elements: a specification language, a model for selecting parts and determining their mutual requirements, a model for arranging parts, and a model for sharing parts across multiple functional requirements. Parts models can be simple catalogs of parts. They can also include hierarchical relations based on functionality, required parts, and bundled packaging. Arrangement models express what physical arrangements of parts are possible and how some arrangements of parts preclude other arrangements. Arrangement models range from complex spatial models to simple named resource models where parts fit in any of a finite number of separately accounted spaces. Sharing models indicate when parts chosen to satisfy one requirement can be used to satisfy others as well. Variations in sharing include exclusive use, open use, metered use, and serial reuse.

Search spaces for configuration domains can be large. Configurations themselves are represented in terms of fairly complex symbol structures that indicate the selection, arrangement, and parameterization of their parts. Most configuration tasks are carried out incrementally, building up descriptions of candidate solutions from descriptions of their parts.

Although configuration domains differ in the complexity of their submodels and the degree to which decisions in each submodel are dependent on decisions in others, they typically exhibit threshold effects where small changes in local requirements can cause discontinuous effects when overall capacities are exceeded. These threshold effects can arise in any of the submodels. Threshold effects and horizon effects for large interconnected problems create a challenge in ordering and managing configuration decisions. The next section presents case studies of configuration systems.

Exercises for Section 8.2

Ex. 1 [*!-05*] *Global Resources.* Decisions about global resources are often problematic in configuration tasks.

(a) Why? What are some examples of global resources from the configuration domains in this chapter?

(b) Briefly describe some alternative approaches for making decisions about global resources.

Ex. 2 [*10*] *Skeletal Configurations.* Professor Digit is a consultant to a company that configures computer systems for its customers. He argues that the number of possible computer configurations may be high but that the number of "really different" configurations sought by customers is actually quite low. He proposes that a knowledge system for configuration would be simpler if it proceeded as follows:

1. Search a library of configurations and find the one closest to the initial configuration proposed for the customer, taking into account which components are key components.
2. Add new key components from the customer's initial configuration that are missing, and remove any extra key components and their required parts from the matching library configuration.
3. Revise the configuration to accommodate the customer's needs.

(a) Briefly describe the main assumptions about the configuration task on which the viability of Professor Digit's proposal depends.

(b) Sam, a graduate student working with Professor Digit, says this proposal creates a tension in the solution criteria for cases in the library between conservative and perhaps inefficient configurations that are easy to extend and ones that are minimal for a fixed purpose. Do you agree? Explain briefly.

(c) Suppose the computer configuration domain is subject to many threshold effects. Briefly, what are the implications for Professor Digit's proposal?

■ **Ex. 3** [*25*] *Configuring Widgets.* The text defines a configuration model, W-1, for widgets. We assume that the following cost figures apply for widget parts.

Part	*Unit Cost*
A-1	$50
A-2	$20
B-1	$20
B-2	$40
C-1	$ 5
C-2	$10
D-1	$40
D-2	$20

(a) Find a minimal space configuration for the specification {A, B}. Use the standard widget model in which there is no sharing of parts across functions. If there is more than one configuration with the minimum space, choose the one with the lowest cost. Draw the final configuration, following packing constraints.
(b) Find a minimal cost configuration for the specification {A, B}, ignoring case costs. If there is more than one configuration with the minimum cost, choose the one with the lowest space requirement. Explain your reasoning. Draw the final configuration.
(c) Parts a and b use the same solution criteria but in a different order. Are the results the same? Why?

■ **Ex. 4** [*30*] *Threshold Effects in Local Cost Evaluations.* This exercise considers threshold effects in a configuration task. It uses the same cost figures as Exercise 3.
(a) Complete the following table showing the configurations for the minimum costs and minimum space solutions for the specifications {A}, {B}, and {C}. Include trapped space in your computation of minimum space.

Specification	***Minimum Cost***	***Minimum Space***
{C}	$5 {C–1}	.5 {C–1} or {C–2}
{B}		
{A}		

(b) Find a minimal cost configuration for the specification {A, 2 B's}. Explain your reasoning. Ignore case costs.
(c) Suppose we are given the following costs for cases:

Widget case	$60
Extension case	$120

Find a minimal cost configuration for the specification {A, 2 B's} including the cost of the case(s). Draw your configuration and explain your reasoning.

■ **Ex. 5** [*15*] *Estimates That Aggregate Space and Cost Constraints.* Professor Digit proposes that a simple and appropriate way to account for case costs would be to add to the cost of each component a fractional cost corresponding to the fraction of the slots that are required. He offers this as a method of reflecting case costs back into the decision making about individual parts. The purpose of this exercise is to evaluate this approach.

We are given the following costs for cases:

Widget case	$60
Extension case	$120

(a) Show the adjusted component costs by filling in the missing entries in the accompanying table. The case increment is the additional cost of the part accounting for the space the part requires in terms of a fraction of the case cost.

Part	*Part Cost*	*Case Increment*	*Case-Adjusted Part Cost*
A-1	$50	$ 5	$55
A-2	$20	$10	$30
B-1	$20		
B-2	$40		
C-1	$ 5		
C-2	$10		
D-1	$40		
D-2	$20		

(b) Using Professor Digit's adjusted case costs, complete the following table, showing the configurations with the minimum costs.

Specification	***Minimum Cost Configuration***
{C}	$10 {C-1}
{B}	$45 {B-1, 2 C-1's}
{A}	

(c) Briefly explain why Professor Digit's proposal is both fair and inadequate for guiding a search for a minimal cost solution.

(d) Using the above table as an evaluation function to guide the search for a minimum cost solution, what would be the minimum cost configuration that Digit's approach would yield for {A, 2 B's}? What is its estimated cost? What is its real cost? Explain briefly.

(e) What is the real minimum cost solution? What is its estimated cost and its real cost?

■ **Ex. 6** [*40*] *Shared Parts.* In this exercise we develop the modified widget model W-S to enable limited sharing of parts. Suppose the functions for {A} and for {D} are never needed at the same time. We define A* to be the set of components including an A and all of its required parts. Similarly D* is the set including a D and all of its required parts. In W-S, sharing is permitted between A* and any D*, which means a single component can satisfy a requirement both for A and for D. No other part sharing is allowed, such as between two different D*'s.

(a) Fill in the following table of configurations, focusing on those requiring minimal space. For simplicity, leave specifications for C unrefined to C-1 or C-2 where possible.

You can use the results of previous exercises. You need only fill in the minimal space partial configurations.

Hint: You may find it useful in solving the later parts of this exercise to include other "near minimal space" partial configurations in your table as well.

Specification	*Minimum Space Configurations*			
	Configuration	*Space*	*Occupied Space*	*Trapped Space*
	{A-1, 2 B-2} {A-1, B-1, B-2, 2 C}	3	2.5	.5
{D}	{B-2, 2 C, D-1} . . . more		2.5	
{2 D}	{2 B-2, 4 C, 2 D-1} . . . more	5		

(b) Without sharing, what are all the minimal space configurations for {A, 2D}? Draw the configurations you find. Briefly explain your reasoning.
(c) With sharing, what are the minimal space configurations for {A, 2D}? Draw the configurations you find. Briefly explain your reasoning.
(d) What do the results of parts b and c tell us about optimizing in shared part models? Briefly describe the computational phenomena introduced to the optimization process when sharing is permitted.

Ex. 7 [*05*] *Widget Thresholds*. In the example of the widget configuration domain, how does the rule that all identical parts must be in the same case contribute to a threshold effect?

Ex. 8 [*05*] *Special Cases for Spatial Reasoning*. The following paragraph describes some requirements for spatial relationships in configurations of elevators:

> The counterweight is a piece of equipment made up of metal plates that are always 1 inch thick. The plates are stacked to get the total load needed to balance the weight of the elevator car, the car's load, attached cables, and other components. The plates can be ordered in varying lengths and widths. They are usually hung from the machine sheave that provides the traction that moves the elevator. The placement of the sheave and its angle of contact with the cable are critical to get enough traction to make the elevator move. Safety codes regulate the minimum clearance between the elevator car and the counterweight, between the counterweight and the shaft walls on its three other sides, and between the counterweight and the floor in its lowest position and the ceiling in its highest position. These code clearances, in turn, are calculated using values such as the weight supported, length of cables, height of the building, and other factors. The configuration must specify the position of the sheave, the length of hoist cable, the size of the counterweight in three dimensions, and actual clearances of the counterweight and any obstacle it might run into all along the elevator shaft.

(a) Is a general approach to spatial reasoning important for this example? If not, why not?
(b) Why was it appropriate to use a fixed structure for its parameter network to capture the reasoning above?

8.3 *Case Studies of Configuration Systems*

In this section we consider several knowledge systems and show how they solve configuration problems. Our purpose is to consider the search requirements of the task, the kinds of knowledge that are needed, the organization of the knowledge, and examples of implementation. The cases illustrate a range of approaches to configuration tasks.

8.3.1 *Configuration in XCON*

XCON (also called R1) is a well-known knowledge system for configuring computer systems (McDermott, 1982). XCON has a place among configuration systems analogous to MYCIN's place among classification systems. XCON was the first well-known knowledge system for configuration, and it was initially characterized in terms of symbol-level issues and basic search methods. Work on both MYCIN and XCON continued for several years and resulted in successor systems. Retrospective analyses of these systems have uncovered new ways of looking at them in knowledge-level terms. For MYCIN, the analysis has shifted from an account of backward chaining at the symbol level to models for classification at the knowledge level. For XCON, the analysis has shifted from forward chaining and match at the symbol level to search spaces and models for configuration at the knowledge level. In both cases insights from the analysis have led to new versions of the systems. For MYCIN the new system is called NEOMYCIN (Clancey & Letsinger, 1981); for XCON it is XCON-RIME (Barker & O'Connor, 1989).

We consider XCON together with its companion program, XSEL. Together, the two systems test the correctness of configurations and complete orders of Digital Equipment Company computers. Different parts of the configuration task are carried out by the two systems. XSEL performs the first step in the process: acquiring initial specifications. Neither XSEL nor XCON has a separate "specification language," but XSEL acquires a customer's order interactively in terms of abstract components. XSEL checks the completeness of an order, adding and suggesting required components. It also checks software compatibility and prerequisites. The output of XSEL is the input to XCON.

XCON checks orders to greater detail than XSEL. XCON adds components to complete orders. It also determines a spatial arrangement for the components, power requirements, cabling requirements, and other things. It considers central processors, memory, boxes, backplanes, cabinets, power supplies, disks, tapes, printers, and other devices. It checks prerequisites and marketing restrictions. It also assigns memory addresses and vectors for input and output devices. It checks requirements for cabling and for power. Figure 8.8 illustrates the knowledge and inference patterns for XSEL and XCON. For simplicity in the following, we refer to the combined system simply as XCON.

Although XCON does not have a functional specification to satisfy, it still needs to decide when an order is complete. It needs to detect when necessary components have been omitted, but it would not be appropriate for it to pad a customer's order with extra components, routinely inflating the price and the functionality of the configured system. Instead, XCON just adds what-

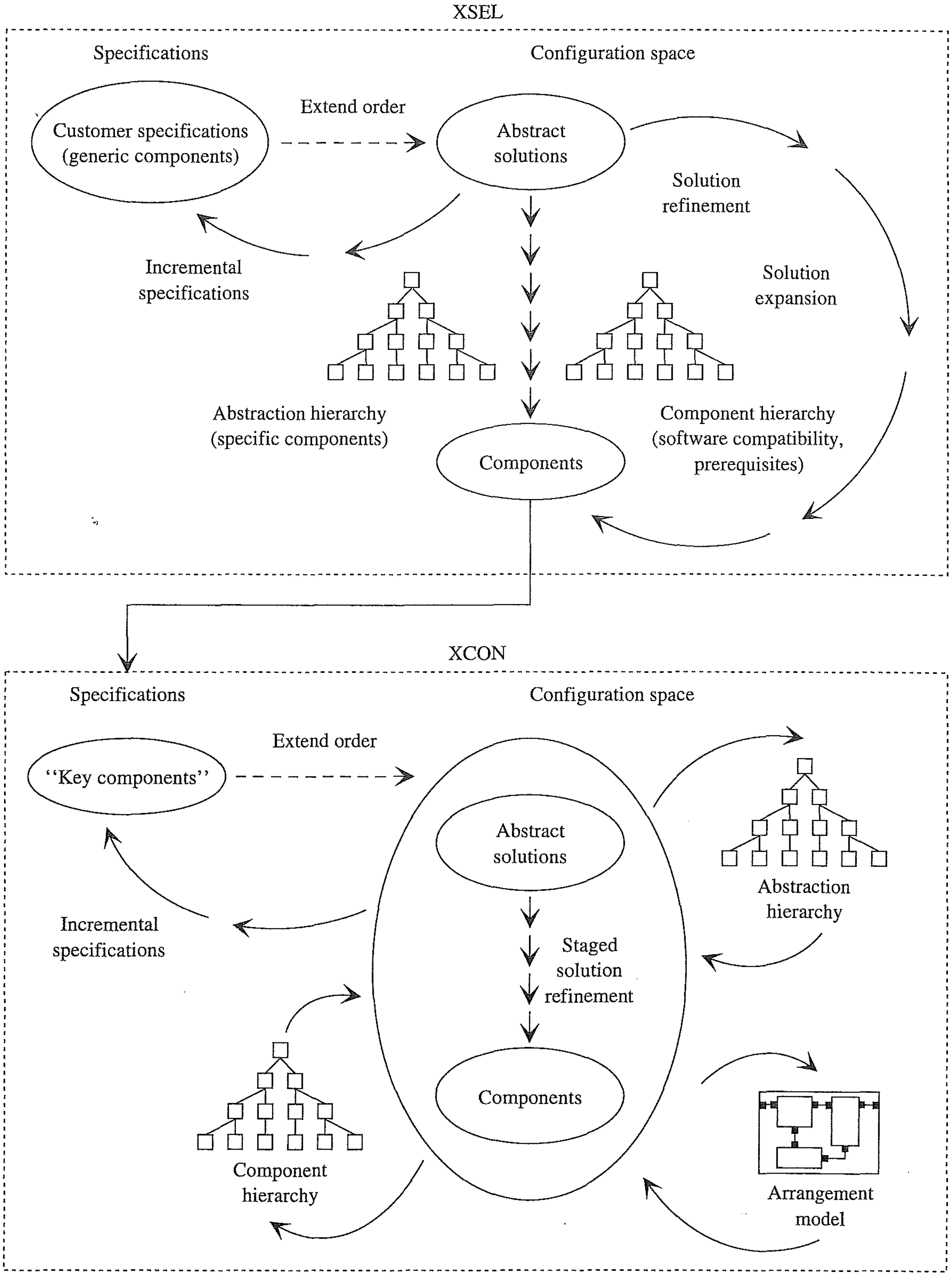


FIGURE 8.8. The pattern of knowledge use for XCON and XSEL. The systems split the responsibility, with XSEL performing the first steps and XCON performing a more detailed job. XSEL is concerned mainly with the completeness of a customer's order with regard to major system components ("key components") and software compatibility and licensing. XCON focuses on hardware configuration and considers power requirements, bus connections, cabling, and arrangement of components in the cabinets.

ever parts are necessary for the operation of the parts that have been specified already. XCON uses a key-component approach.

Crucial to the operation of XCON is its component database. This database has grown over time and in 1989 was reported to contain information on more than 31,000 parts. There were 40 types of parts and an average of 40 attributes per part. Figure 8.9 gives an abbreviated example of two component descriptions. The interpretation of the information in Figure 8.9 is approximately as follows:

> The RK711-EA is a bundle of components. It contains a 25-foot cable (70-12292-25), a disk drive (RK07-EA*), and another bundle of components (RK611). The RK611 consists of three boards (G727), a unibus jumper cable (M9202), a backplane (70-12412-00), and a disk drive controller (RK611*). Because of its interrupt priority and data transfer rate, the RK611* is typically located toward the front of the unibus. The module is comprised of five hex boards, each of which will start in lateral position "A." It draws 15.0 amps at +5 volts, 0.175 amps at -15 volts, and .4 amps at +15 volts. It generates one unibus load and can support up to eight disk drives. It is connected to the first of these with a cable (070-12292).

Other information includes requirements for boards and power and information about communications that will influence how the devices are connected to the computer's buses.

In addition to the component database, there is a container-template database that describes how parts can physically contain other parts. These templates enable XCON to keep track of what container space is available at any point in the configuration process and what the options are for arranging the components. They also provide coordinate information so XCON can assign components to locations in a cabinet. In XCON, arrangement includes electrical connections, panel space, and internal cabinet locations. XCON uses a model for spatial reasoning in which the possible locations are described in terms of "slots," places where boards can plug in, and nexuses, places where devices can be attached in cabinets. Figure 8.10 gives an example of a template. Part of the interpretation of this template follows:

> The components that may be ordered for the CPU cabinet are SBI modules, power supplies, and an SBI device. Up to six bus interface modules fit into the cabinet. The cabinet contains a central processor module and some memory. There are three slots for options that occupy 4 inches of space and one slot for an option that occupies 3 inches of space. The description "CPU nexus-2 (3 5 23 30)" indicates that the central processor module must be associated with nexus 2 of the bus interface. The numbers in parentheses indicate the top left and bottom right coordinates of the space that can be occupied by a CPU module.

Customer orders to XCON are highly variable. Although two system configurations may need most of the same kinds of actions to occur, even slight differences in the configurations may have broad consequences because of complex interactions between components. The knowledge for driving the configuration task in XCON is represented mostly using production rules. In the following we examine some rules from XCON and consider the kinds of knowledge they represent.

```
RK711-EA
CLASS:                       BUNDLE
TYPE:                        DISK DRIVE
SUPPORTED:                   YES
COMPONENT LIST:              1       070-12292-25
                             1       RK07-EA*
                             1       RK611

RK611*
CLASS:                       UNIBUS MODULE
TYPE:                        DISK DRIVE
SUPPORTED:                   YES
PRIORITY LEVEL:              BUFFERED NPR
TRANSFER RATE:               212
SYSTEM UNITS:                2
SLOTS REQUIRED:              6 RK611 (4 TO 9)
BOARD LIST:                  (HEX A M7904)
                             (HEX A M7903)
                             (HEX A M7902)
                             (HEX A M7901)
                             (HEX A M7900)
DC POWER DRAWN:              15.0 .175 .4
UNIBUS LOAD:                 1
UNIBUS DEVICES SUPPORTED:            8
CABLE TYPE REQUIRED:         1       070-12292 FROM A DISK DRIVE UNIBUS DEVICE
```

FIGURE 8.9. Two component descriptions from XCON's database. The attributes of a component are used in the configuration process to determine when other parts are needed, such as subsystems and cabling. They also describe parameters that influence the arrangement of parts and the power requirements. (Adapted from McDermott, 1982a, Figure 2.2, pp. 46–47.)

Parts require other parts for their correct operation. The initial specifications to XCON refer explicitly to key components, but not necessarily all the other components required to enable a configuration to function correctly. Figure 8.11 gives an example of a rule for adding a component.

Parts require other parts for a variety of reasons. For example, disk drives require disk controllers to pass along commands from the central processor. One disk controller can service several drives. All of these devices require cables to interconnect them. They also require power supplies to power them and cabinets to house them. The interlocking requirements are represented in XCON by a combination of rules and the databases about components and templates. Requirements for additional components are expressed locally, which means each description of a component includes specifications for other components that it requires. When XCON adds a component to a configuration, it needs to check whether the new component introduces requirements for other components.

XCON uses knowledge to govern the automated arrangement of components, as demonstrated in Figure 8.12. Several kinds of choices must be made about arrangements in XCON. A

CPU CABINET	
CLASS:	CABINET
HEIGHT:	60 INCHES
WIDTH:	52 INCHES
DEPTH:	30 INCHES
SBI MODULE SPACE:	CPU NEXUS-2 (3 5 23 30)
	4-INCH-OPTION-SLOT 1 NEXUS-3 (23 5 27 30)
	MEMORY NEXUS-4 (27 5 38 30)
	4-INCH-OPTION-SLOT 2 NEXUS-5 (38 5 42 30)
	4-INCH-OPTION-SLOT 3 NEXUS-5 (42 5 46 30)
	3-INCH-OPTION-SLOT NEXUS-6 (46 5 49 30)
POWER SUPPLY SPACE:	FPA NEXUS-1 (2 32 10 40)
	CPU NEXUS-2 (10 32 18 40)
	4-INCH-OPTION-SLOT 1 NEXUS-3 (18 32 26 40)
	MEMORY NEXUS-4 (26 32 34 40)
	4-INCH-OPTION-SLOT 2 NEXUS-5 (34 32 42 40)
	CLOCK-BATTERY (2 49 26 52)
	MEMORY-BATTERY (2 46 26 49)
SBI DEVICE SPACE:	IO (2 52 50 56)

FIGURE 8.10. An example container template from XCON. (Adapted from McDermott, 1982a, Figure 2.3, page 48.)

distinction is made between the CPU cabinet, the CPU extension cabinets, and the unibus cabinets. The CPU is the central processing unit and SBI modules are bus interface modules. Components whose class is SBI module are located in the CPU cabinet. There is only a limited amount of space in these cabinets, as indicated by the templates. XCON assigns components to locations incrementally. In placing objects in the unibus cabinet, XCON needs to account for both panel space and interior space. Modules that support each other need to be located either in the same backplane or in the same box. Another arrangement issue is the electrical order by which objects are located on the bus. This order affects system performance. Other arrangement decisions concern the allocation of floor space to cabinets, taking into account any known obstructions at the customer site.

Rule for adding parts

```
   IF:      The most current active context is assigning a power supply
            and a unibus adapter has been put in a cabinet
            and the position it occupies in the cabinet (its nexus) is known
            and there is space available in the cabinet for a power supply
            for that nexus
            and there is an available power supply
            and there is no H7101 regulator available
   THEN:    Add an H7101 regulator to the order.
```

FIGURE 8.11. A rule from XCON for adding required parts to a configuration.

Rule for arranging parts

```
IF:      The most current active context is assigning a power supply
         and a unibus adapter has been put in a cabinet
         and the position it occupies in the cabinet (its nexus) is known
         and there is space available in the cabinet for a power supply
         for that nexus
         and there is an available power supply
         and there is an H7101 regulator available
THEN:    Put the power supply and the regulator in the cabinet in the
         available space.
```

FIGURE 8.12. A rule from XCON for arranging parts in a configuration.

Other rules add specifications to further guide the configuration process. The XCON rule in Figure 8.13 is an example of this. In other cases in XCON, the rules propose changes to the given specifications. For example, if there are components in the order that have incompatible voltage or frequency requirements, XCON tries to identify a minority set of components having the "wrong" voltage or frequency and replaces them with components of the right voltage and frequency.

Choices also must be made about implementation and arrangement. Some combinations of choices are incompatible with others. For example, there may be a tension between having optimal performance of the bus and still packing as many components into a single cabinet as possible. XCON can fall back to delivering a configuration that works without guaranteeing that performance is optimal.

The strategy taken by the designers of XCON was to organize it to make decisions in an order that would enable it to proceed without undoing the decisions it has made so far. XCON's decisions are organized into stages and subtasks. Stages correspond to large groups of subtasks,

Rule for extending the specifications

```
IF:      The most current active context is checking for unibus jumper
         cable changes in some box
         and the box is the second box in some cabinet on some unibus
         and there is an unconfigured box assigned to that unibus
         and the jumper cable that has been assigned to the last
         backplane in the box is not a BC11A-10
         and there is a BC11A-10 available and the current length of the
         unibus is known
THEN:    Mark the jumper cable assigned to the backplane as not assigned
         and assign the BC11A-10 to the backplane
         and increment the current length of the unibus by ten feet.
```

FIGURE 8.13. An example of a rule for adding or changing specifications.

Stage 1 Determine whether anything is grossly wrong with the order, such as mismatched items or missing prerequisites.
Stage 2 Put the appropriate components in the CPU and CPU expansion cabinets.
Stage 3 Put boxes in the unibus expansion cabinets and put the appropriate components in those boxes.
Stage 4 Put panels in the unibus expansion cabinets.
Stage 5 Create the floor layout of the system.
Stage 6 Add necessary cables to the configuration.

FIGURE 8.14. Stages for the XCON system. The grouping and ordering of decisions in these stages is arranged so that, for the most part, later decisions depend only the results of earlier decisions.

and subtasks can involve as little as a single rule. There are six stages and several hundred subtasks. We first discuss gross interactions among the stages and then the more complex interactions among subtasks.

XCON's stages are shown in Figure 8.14. These stages were not determined through a rigorous analysis of the configuration problem. Rather, the ordering was originally designed to reflect the way human experts performed the task. The ordering has evolved since then for XCON, based on the insights and measurements that have become possible with an explicit knowledge base. Nonetheless, the basic rationale of XCON's ordering of stages is straightforward. Component selection and component placement in boxes determines what cables are needed, and floor layout must be known to determine the length of cables. This is why stage 6 comes after stages 1, 2, 3, and 5. Any arrangement of components made before all the required components are known is likely to require rearrangements, because prerequisite components often need to be co-located. This is why stages 2 and 3 should come after stage 1. Any attempt to locate front panels for unibus devices on cabinets before it is known how many cabinets are being used and where devices are located would probably need some redoing. This is why stage 4 comes after stage 3. Ordered in this way, the later stages implicitly assume that the previous stages have been completed successfully.

The first stage adds missing but required components to the specification. The component descriptions characterize some components as requiring others for their operation and also characterize some components as coming in "bundles," which means they are packaged together for either marketing or manufacturing reasons. Adding required components is potentially recursive, since some required components require still other components. The first stage does not arrange components. In addition to adding components, the first stage detects any mismatched components in the order, such as components with mismatched power requirements as discussed already.

The second and third stages arrange components. The second stage locates SBI modules in the CPU cabinet. This stage can introduce additional components, but not ones that could themselves require revisiting decisions in the first stage or this stage. For example, stage 2 may need to add CPU extension cabinets if there are too many components to fit in the one containing the central processor. When all the relevant components have been placed, stage 2 puts a bus terminator in the appropriate place in the last cabinet and adds dummy or "simulator" modules to fill out any leftover slots corresponding to unordered options. Until these cabinets have been configured, it is not known whether these additional parts are needed. Because these parts do not com-pete for space with other components and because they have no further requirements, adding them does not trigger requirements for further parts.

The third stage is the most complex. This stage determines the parameters of three kinds of arrangements: spatial arrangement of boxes in cabinets, spatial arrangement of components inside boxes, and electrical arrangement of components on the unibus. For communication on the unibus, the relevant factors are interrupt priority level and transfer rate. In an optimal order, the components would be arranged in descending order of priority level (or more precisely, in nonincreasing order). Within components of the same priority, they should be ordered by decreasing transfer rate. XCON defines an "(almost) optimal unibus order" as one in which no pairs of components at the same priority level are connected so that the one with lower transfer rate comes before the one with the higher transfer rate. Below that, XCON characterizes any other ordering as suboptimal. The third stage uses an iterative approach in configuring the unibus expansion cabinets. It first tries to arrange the components in an optimal unibus order within the space available. If that fails, it then tries other rearrangements of the components that compromise the optimal unibus order but save space. Depending on what it can achieve, XCON may need to allocate new cabinets and try again. This leads to ordering further parts in addition to the cabinets, needed for extending the unibus across the cabinets. Within this stage XCON must sometimes backtrack to consider all the alternatives.

The fifth and sixth stages for XCON introduce no new theoretical issues. The fifth stage generates a linear floor plan for the computer system. The last stage generates cable assignments, using the interdevice distances that have been determined by the earlier stages. It is interesting to note that XCON's method does more than satisfice. Especially in stage 3, its ranking and iterative development of alternatives is a **due-process approach** that attempts to optimize the layouts using essentially a best-first generate-and-test method.

The next level down from stages in XCON is subtasks, implemented as one or more rules. In contrast to the simple and fixed ordering of stages, the determination of the order of performing subtasks is complex, situation specific, and governed by conditionals. XCON employs extensive look-ahead to detect exceptional situations and to check for possible future conflict before making decisions. Early subtasks check for interactions with later ones. One difficulty in XCON is that there are many potential long-range interactions. Restated, in spite of the task ordering, there are still many possible interactions among decisions that are scheduled in early tasks and those that are scheduled in later ones. Anticipating these interactions with conditional subtasks has yielded a knowledge base that has proven quite complex to maintain. In 1989, XCON had more than 31,000 components and approximately 17,500 rules (Barker & O'Connor, 1989). The knowledge base changes at the rate of about 40 percent per year, meaning that modifications are made to that fraction of it.

This maintenance challenge has led to the development of a programming methodology for XCON called RIME (Bachant, 1988; van de Brug, Bachant, & McDermott, 1986), which provides guidelines intended to simplify the continuous development of the system. RIME provides structuring concepts for rule-based programming. From the perspective of configuration problems, it institutionalizes some of the concepts we have seen already in the organization of XCON, especially in the third stage.

RIME advocates an approach based on concepts called "deliberate decision making" and "propose-and-apply." Deliberate decision making amounts to defining configuration in terms of small formal subtasks that are represented as explicit, separate processes. Examples of these processes in the RIME implementation of XCON include selecting a device, selecting a container for a component, and selecting a location within a container to place a component. These processes were distributed and sometimes duplicated among the rules of XCON. By naming these processes explicitly and representing them uniquely, RIME brings to XCON-RIME a concern for modularity at the symbol level.

Another part of the RIME methodology, called propose-and-apply, organizes subtask descriptions as a sequence of explicit stages that include proposing operators, eliminating the less desirable ones, selecting an operator, and then performing the operation specified. In the context of configuration tasks, this separates each subtask into three parts: looking ahead for interactions in the situation, selecting a best choice for part selection or arrangement, and then executing that choice. This bundling institutionalizes the way that XCON-RIME considers interlocked combinations of choices and backtracks on generating alternatives but not on making commitments.

Sometimes configuration choices cannot be linearly ordered without backtracking because the optimization requires a simultaneous balance of several different kinds of decisions. In XCON, this occurs most noticeably in the third stage, which resolves this issue by bundling decisions and generating combinations of decisions in an iterative approach. In particular, by generating composite alternatives in a preferred order, this stage generates some combinations of choices before committing to any of the individual choices about unibus order and spatial arrangement.

In summary, XCON is a knowledge system for configuring computer systems. Together with XSEL, it accepts customer orders interactively in terms of abstract parts. Using the key component assumption, functionality is specified in terms of a partial configuration. Decisions about the selection and arrangement of a part often interact with decisions about distant parts. XCON organizes its decision making into an ordered set of stages, but the detailed set of decisions is still highly conditional and must account for potential interactions. RIME is a programming methodology for a new implementation of XCON that organizes the configuration process in terms of small, explicit subtasks with explicit modules for look-ahead, generation of alternatives, and commitment.

8.3.2 *Configuration in M1/MICON*

M1 is an experimental knowledge system that configures single-board computers given a high-level specification of board functions and other design constraints. M1 is part of a larger single-board computer design system called MICON (Birmingham, Brennan, Gupta, & Sieworek, 1988). Single-board computer systems are used for embedded applications, in which computer systems are integrated in larger pieces of equipment. The limitation to single-board systems means M1 does not contend with spatial layout issues for components inside boxes and cabinets.

M1 configures systems that use a single microprocessor, memory, input and output devices, circuitry to support testing, and circuitry to enhance reliability. The output of M1 is a list of components for a design, the list of how these components are interconnected, an address map that defines where input and output devices are in memory, and some graphs detailing options and costs for enhancing reliability and fault coverage. Figure 8.15 sketches the main search

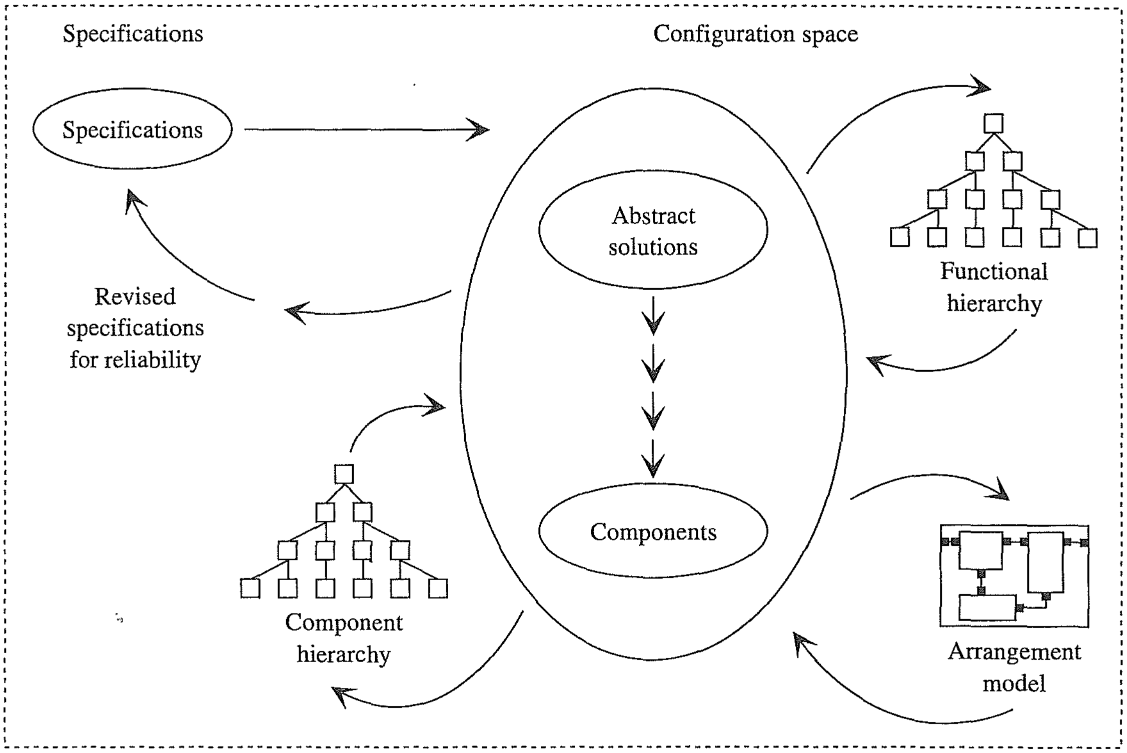


FIGURE 8.15. The specification language for M1 is a list of key components, using M1's functional hierarchy. M1 also has a component hierarchy that describes required parts, which are supporting circuitry. Its arrangement model is based on ports and connections. Not shown in this figure is M1's knowledge for reliability analysis.

knowledge for M1. Like our exemplar of configuration systems, M1 has knowledge for mapping specifications to abstract solutions; knowledge for refining abstract solutions by selecting, adding, and arranging parts; and knowledge for adding specifications.

M1 represents a functional hierarchy of components, as shown in Figure 8.16. This hierarchy includes both abstract parts and physical parts. The abstract parts are developed by generalizing the functions of a class of parts. For example, the abstract part PIO-0 embodies a set of functions common to all the devices in M1's database for parallel input and output. An important purpose of abstract components is that they define a standard wiring interface called a "functional boundary."

M1 begins with an interactive session with a user to acquire specifications for the system to be configured. This session leads to a number of specifications about power, area, and the desired functionality of the system. Like XCON, M1 uses a key-component assumption for specifying functionality. The key components used by M1 correspond to the third level (the lowest abstract level) of its component hierarchy. For example, M1 asks the user whether a serial input/output device is required and represents what it knows about serial input/output devices in its description of the abstract class SIO-0, as shown in Figure 8.16.

Each abstract part has a set of specifications that characterize its function and performance. In 1989, the parts database described 574 parts, of which 235 were physical parts and the rest were abstract parts. The user's requirements for these are entered as a set of values. For example, the specifications for a serial input/output device allow the user to specify, among other things,

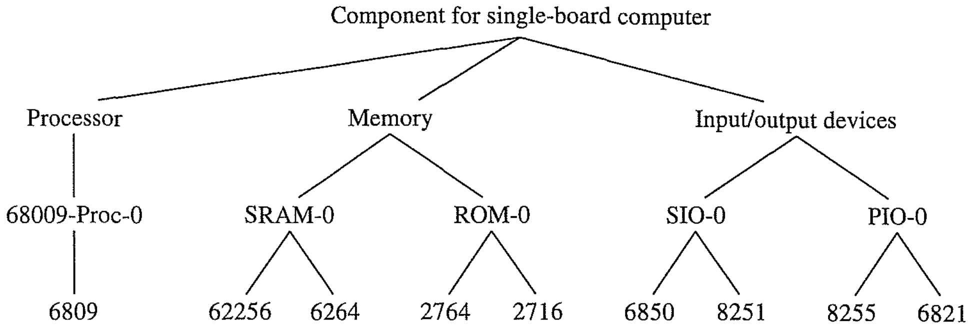


FIGURE 8.16. A functional hierarchy used in the M1 system. Some parts in this hierarchy represent abstract parts. These parts are developed by generalizing the functions of a class of physical parts. For example, the abstract part PIO-0 embodies a set of functions common to all parallel input/output chips in M1's database. SRAM-0 represents static random-access memory. ROM stands for read-only memory. SIO-0 and PIO-0 represent serial and parallel input/output devices, respectively. Parts like SIO-0 and PIO-0, at the third level of M1's component hierarchy, correspond to key components and are used for specifying functionality. They also define functional boundaries, which provide a framework for specifying how the more detailed parts should be connected. (Adapted from Birmingham, Brennan, Gupta, & Sieworek, 1988, Figure 2, page 38)

how many ports are needed, a chip type if known, an address, a baud rate, and whether the device should be compatible with RS232 output. The user requirements then become constraints to guide the initial selection of parts.

Once abstract parts are selected, M1 moves on to instantiate these parts and to arrange them. The main arrangement decisions for M1 are the electrical connections between the components. In contrast with the spatial arrangement subtasks in XCON, M1 produces a spatially uncommitted network list that is passed to commercial programs for layout and wire routing.

During the phase of part instantiation, M1 adds supporting circuitry. Figure 8.17 shows that a baud rate generator and RS232 driver are instantiated as parts of a serial input/output device. The templates are represented procedurally as rules, as suggested by the abbreviated English rule for connections in SIO-0 in Figure 8.18. In some cases the part expansion involves substantial expansion into individual parts. For example, individual memory chips are organized into an array to provide the required word width and storage capacity.

The next step in M1 is an optional analysis of the reliability of the design. If the analysis is requested, the user is asked about several more characteristics as well as a selection of reliability metrics such as mean time to failure and coverage of error detection. The user is also asked to provide relative weights for trading off total board area and monetary cost. M1 has a number of rules for determining sources of failure. These rules also guide it in making design changes, such as substituting parts, changing their packaging, or adding error-correcting circuitry and redundant circuitry. M1's method is summarized in Figure 8.19.

In summary, M1 has one of the simplest configuration tasks of the applications we will consider. Specifications include parameters about size and power as well as function. Functionality is represented in terms of key components, where key components are expressed as abstract parts in a component hierarchy. A weighted evaluation function is used to select among competing parts when there is no clear dominance. Part expansion is driven by templates represented as rules. No conflicts arise in expanding or arranging parts. Other than recomputations required

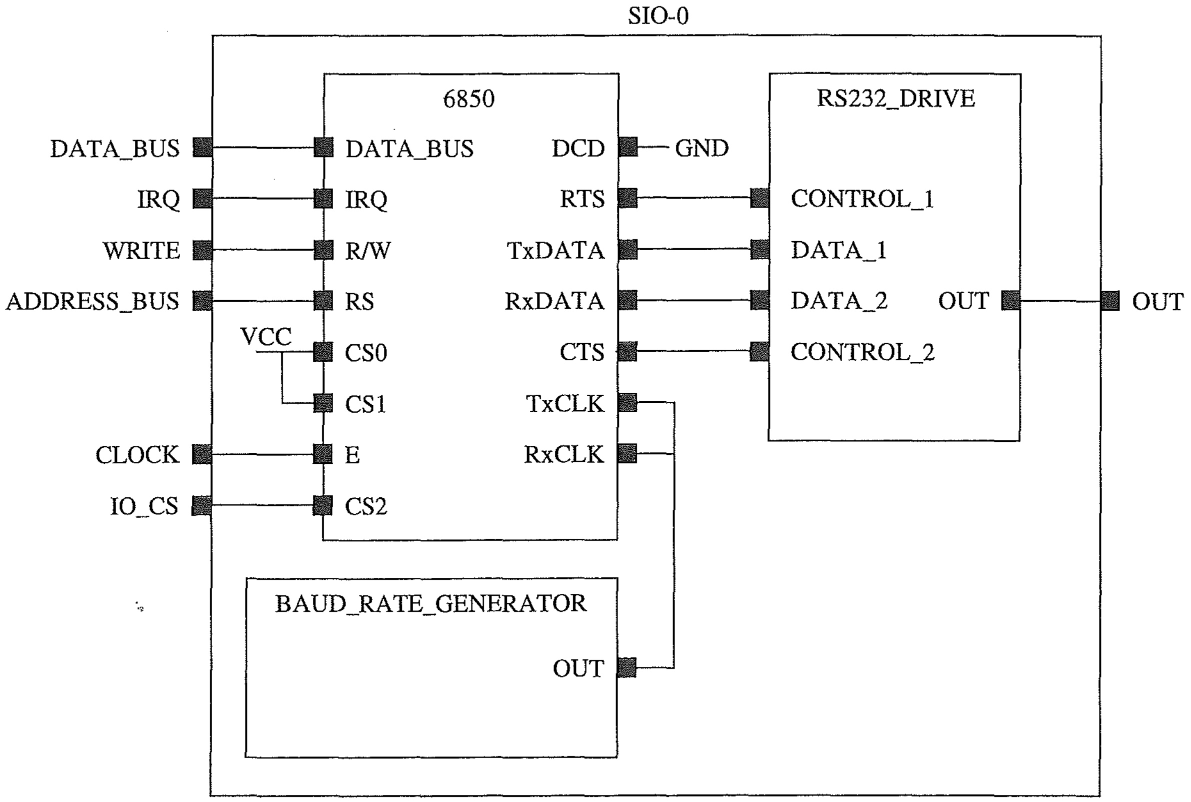


FIGURE 8.17. M1 uses templates and a port-and-connector model for describing the space of possible interconnections. The template in this figure is for the serial input/output device SIO-0. (Adapted from Birmingham, Brennan, Gupta, & Siewiorek, 1988, Figure 4, page 37)

after the reliability analysis, M1 performs no backtracking. M1 uses a port-and-connector model for arrangements. The arrangement process does not compute an optimal layout. All topological arrangements are achievable, so no revision of component choices or specifications is required on the basis of connectivity constraints.

```
IF      the goal is assert_template and
        the part_name is 6850 and
        the abstract_part_name is 6850
THEN    get_part(RS232_DRIVER)
        get_part(BAUD_RATE_GENERATOR)
        connect_net(6850, d0, SIO_0, DATA_BUS)
        connect_net_VCC(6850, CS0)
        connect_net_GND(6850, DCD)
        connect_net(6850, TxCLK, BAUD_RATE_GENERATOR, OUT)
        ... (more)
```

FIGURE 8.18. Sample rule representation for template knowledge, specifying connections between the internal parts that make up an abstract component and the outer ports of the abstract part. These outer ports are called a functional boundary.

To perform M1's method for configuring single-board computer systems:

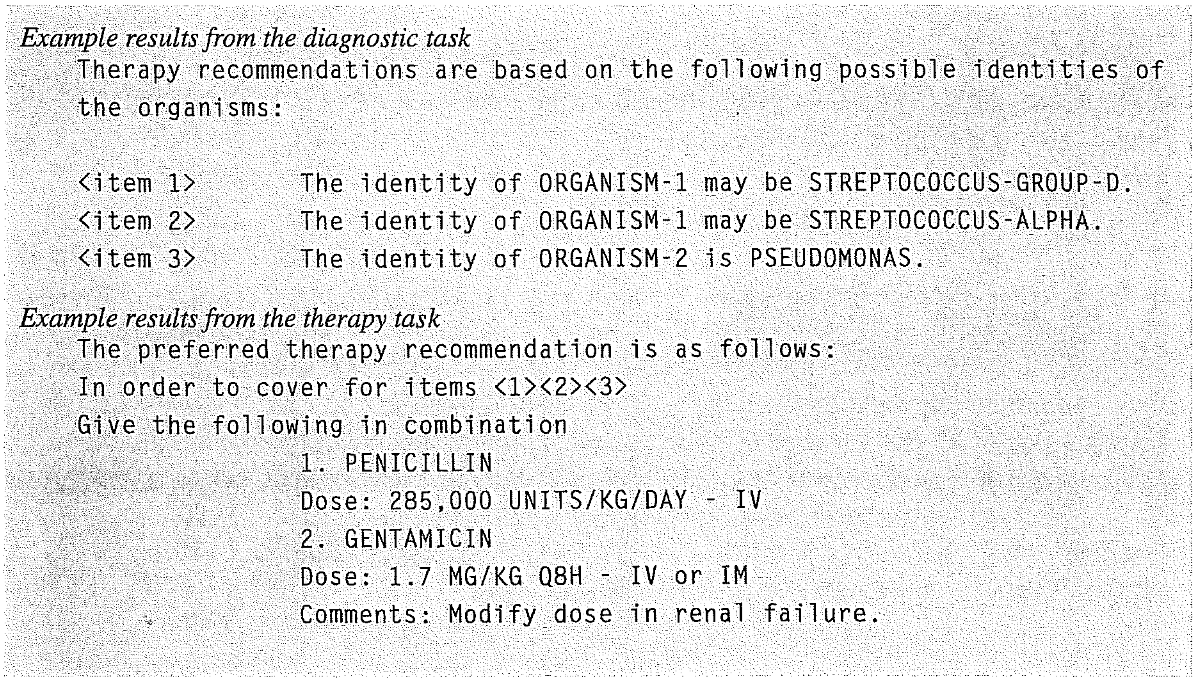
```
 1.  Get specifications.

     /* Initial part selection. */
 2.  Repeat through step 4 for each key function of the specifications.
 3.        Collect parts that satisfy the specification.
 4.        Select the part having the highest rating according to M1's
           feature-weighted evaluation function.

     /* Perform a reliability analysis if requested and make part
     substitutions. */
 5.  If a reliability analysis is requested, then repeat through step 9
     until the reliability specifications are satisfied.
 6.        Use an external program to predict mean time to failure, error-
           detection coverage, and reliability of individual components.
 7.        Report on reliability and acquire revised specifications.
 8.        Acquire weighting factors from the user to guide trade-offs
           between board area and cost.
 9.        Select parts, substituting more reliable components and adding
           redundant and error-detection circuitry as needed.

     /* Expand design for cascaded components. */
10.  Repeat through step 11 for each cascadable part.
11.        Invoke associated procedure to cascade the parts to the
           required sizes. (Generate arrays for memories or trees for
           priority encoding and carry look-ahead.)

     /* Connect parts using structural templates. */
12.  Repeat through step 14 for each unconnected part.
13.        Retrieve the template rule for this component.
14.        Apply the rule to retrieve parts for ancillary circuitry and
           connect the ports of the parts as required.
```

FIGURE 8.19. The search method used by M1.

8.3.3 *Configuration in MYCIN's Therapy Task*

MYCIN is well known as a knowledge system for diagnosing infectious diseases. Its diagnostic task has been so heavily emphasized that its second task of therapy recommendation is often overlooked. The computational model for therapy recommendation is different from that for diagnosis. This section explains how therapy recommendation in MYCIN is based on a configuration model.

The term *therapy* refers to medical actions that are taken to treat a patient's disease. The most powerful treatments for infectious diseases are usually drugs, such as antibiotics. In



Example results from the diagnostic task

```
Therapy recommendations are based on the following possible identities of
the organisms:

<item 1>      The identity of ORGANISM-1 may be STREPTOCOCCUS-GROUP-D.
<item 2>      The identity of ORGANISM-1 may be STREPTOCOCCUS-ALPHA.
<item 3>      The identity of ORGANISM-2 is PSEUDOMONAS.
```

Example results from the therapy task

```
The preferred therapy recommendation is as follows:
In order to cover for items <1><2><3>
Give the following in combination
              1. PENICILLIN
              Dose: 285,000 UNITS/KG/DAY - IV
              2. GENTAMICIN
              Dose: 1.7 MG/KG Q8H - IV or IM
              Comments: Modify dose in renal failure.
```

FIGURE 8.20. Therapy recommendation in MYCIN. (From Shortliffe, 1984, pp. 123, 126.)

MYCIN, a therapy is a selected combination of drugs used to cover a diagnosis. Figure 8.20 presents an example of diagnostic results and a therapy recommendation. The results are characterized in terms of items. The first two items correspond to alternative and competing identifications of the first organism. Medical practice usually requires that the therapy be chosen to cover both possibilities. A possible therapy is shown at the bottom of the figure, recommending a combination of drugs. In some cases, MYCIN ranks alternative therapies, each of which is a combination of drugs. By analogy with more typical configuration problems, the organisms that must be covered by the treatment correspond to specifications, drugs correspond to parts, and therapies correspond to configurations.

It would have been simpler in MYCIN if the same computational model could have been used for both diagnosis and therapy. However, there are several phenomena and considerations that arise in therapy recommendations that violate assumptions of the classification model that MYCIN used for diagnosis. The most basic issue is that the number of possible solutions based on combinations of drugs to cover combinations of diseases is too large to pre-enumerate. More problematic than this, the selection of drugs to cover one item is not independent of the selection of drugs to cover another. This follows from competing concerns in therapy. On the one hand, it is desirable to select therapeutic drugs that are the most effective for treating particular diseases. For the most part, this decision is based on drug sensitivities of different organisms. Tests are run periodically in microbiology laboratories to determine the changing resistance patterns of microorganisms to different drugs. On the other hand, drugs sometimes interfere with each other and it is usually desired to keep as low as possible the number of drugs simultaneously administered to a patient. Thus, the best combination of drugs to cover two different diagnoses might be different from the union of the best drugs to treat them separately. For example, it might be best to treat both possible infections with the same drug, even if a different drug might be selected by itself for either one of them.

Sometimes the risks associated with illness require that a doctor prescribe a treatment before time-consuming laboratory tests can be completed. For example, culture growths would narrow the range of possible diagnoses but may delay treatment for several hours. Other complicating factors include drug interactions, toxic side-effects, cost, and ecological considerations. An example of an ecological factor is when physicians decide to reserve certain drugs for use only on very serious diseases. In some cases, this measure is used to slow the development of drug-resistant strains of organisms, which is accelerated when a drug is widely prescribed. In medical jargon, all negative factors for a therapy are called "contra-indications." Patient-specific contra-indications include allergies, the patient's age, and pregnancy.

In the initial therapy algorithm for MYCIN (Shortliffe, 1984), therapy recommendation was done in two stages. A list of potential therapies was generated for each item based on sensitivity information. Then the combination of drugs was selected from the list, considering all contra-indications. This approach was later revised (Clancey, 1984) to make more explicit the mediation between optimal coverage for individual organisms and minimization of the number of drugs prescribed.

The revised implementation is guided by the decomposition of the decision into so-called local and global factors. The local factors are the item-specific factors, such as the sensitivity of the organism to different drugs, costs, toxicity, and ecological considerations. Global factors deal with the entire recommendation, such as minimizing the number of drugs or avoiding combinations that include multiple drugs from the same family, such as the aminoglycosides family. One variation is to divide the hypothesized organisms into likely and unlikely diagnoses and to search for combinations of two drugs that treat all of the likely ones. There are sometimes trade-offs in evaluating candidates. One candidate might use better drugs for the most likely organisms but cover fewer of the less likely organisms. In such cases, the evaluation function in the tester needs to rank the candidates. Close calls are highlighted in the presentation to a physician.

Figure 8.21 shows MYCIN's revised therapy method. Local factors are taken into account in the "plan" phase. Global factors are taken into account in the generate-and-test loop. Thus, the

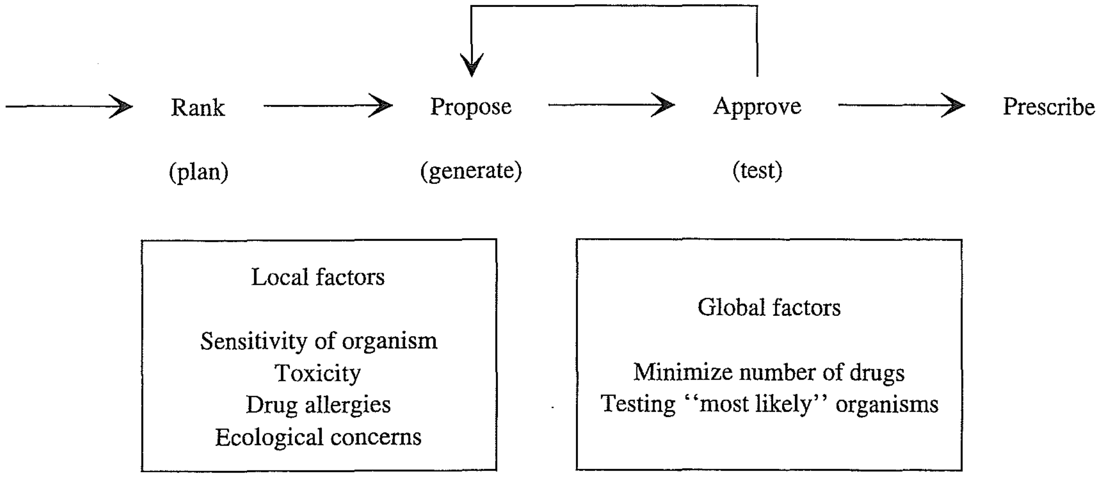


FIGURE 8.21. Therapy recommendation viewed as a plan, generate, and test process. (Adapted from Clancey, 1984, Figure 6-1, page 135.)

test step considers an entire recommendation. Patient-specific contra-indications are also taken into account at this stage.

In summary, MYCIN is a knowledge-based system that performs diagnosis and therapy tasks. The diagnosis task is based on a classification model and is implemented using production rules. The therapy task is based on a configuration model and is also implemented using production rules. A solution to the therapy task is a selected subset of drugs that meet both local conditions for effectively treating for organisms as well as global optimality conditions such as minimizing the number of drugs being used. MYCIN's configuration model involves selection but not arrangement of elements. As in all configuration problems, small variations in local considerations can effect the scoring of an entire solution.

8.3.4 *Configuration in VT*

VT is a knowledge system that configures elevator systems at the Westinghouse Elevator Company (Marcus, 1988; Marcus & McDermott, 1989; Marcus, Stout, & McDermott, 1988). VT searches for solutions by proposing initial solutions and then refining them. This is called a "propose-and-refine" approach. The search spaces for VT are shown in Figure 8.22.

Interwoven with VT's model of parts is a parametric model of elevator systems. Components are described in terms of their parameters. For example, a hoist cable may have a parameter for HOIST-CABLE-WEIGHT. Parameters can also describe aspects of an elevator system not associated with just one of its parts. For example, the parameters TRACTION-RATIO and MACHINE-SHEAVE-ANGLE depend on other parameters associated with multiple parts.

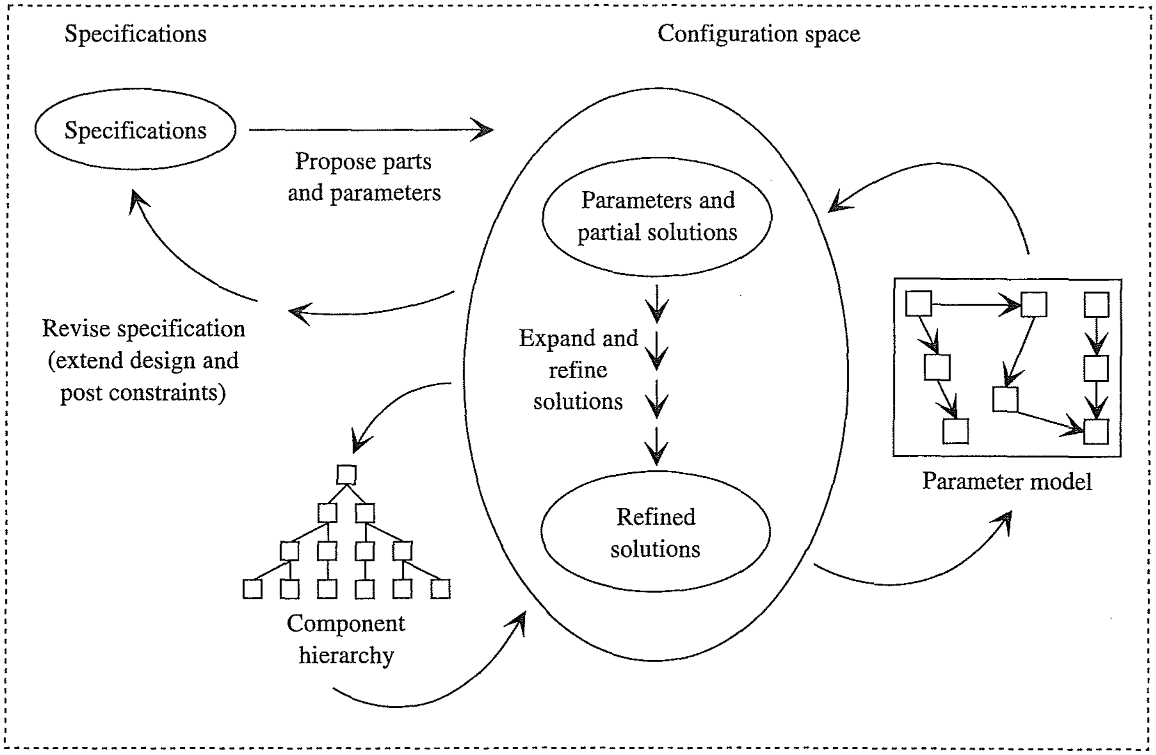

Specifications
Configuration space
Specifications
Propose parts and parameters
Parameters and partial solutions
Revise specification (extend design and post constraints)
Expand and refine solutions
Refined solutions
Parameter model
Component hierarchy

FIGURE 8.22. VT uses a propose-and-refine approach.

Design methods involving such parametric models are sometimes called **parametric design**. Collectively, VT's parameters provide an abstract description of a solution.

VT starts with specifications of elevator performance, architectural constraints, and design drawings. Specifications of elevator performance include such things as the carrying capacity and the travel speed of the elevator. Architectural constraints include such things as the dimensions of the elevator shaft. VT's task is to select the necessary equipment for the elevator, including routine modifications to standard equipment, and to design the elevator layout in the hoistway. VT's design must meet engineering safety-code and system performance requirements. VT also calculates building load and generates reports for installing the elevator and having it approved by regional safety-code authorities. Data for VT come from several documents provided by regional sales and installation offices. There are documents describing customer requirements, documents describing the building where the elevator will be installed, and other design drawings. VT also accepts information about the use of the elevator, such as whether it is mainly to be used for passengers or freight, the power supply available, the capacity, the speed, the shaft length, the required width and depth of the platform, and the type and relative location of the machine.

VT's approach involves two separate phases. The first phase is for knowledge acquisition and analysis. During this phase VT uses a knowledge acquisition tool called SALT to develop a general plan for solving elevator configuration problems. SALT is not concerned with the specifics of a particular case, but rather with the acquisition and organization of knowledge for the range of cases that VT needs to solve. The concepts and analyses of SALT are discussed in the following. The second phase for VT is the solution of particular cases. This is where the propose-and-refine approach is applied. Roughly, VT follows a predetermined plan using knowledge organized by SALT to guide its decisions. VT constructs an elevator configuration incrementally by proposing values for design parameters, identifying constraints on the design parameters, and revising decisions in response to constraint violations in the proposal.

To explain the operation of VT we begin in the middle of the story, with its parameter network. This network is generated by SALT and used by VT on particular cases. Figure 8.23 gives an example of part of a parameter network. Each box represents a design parameter. The arrows show a partial order of inferences in which values of parameters at the tail of an arrow determine values of parameters at the head of an arrow. Solid arrows indicate how one parameter is used to compute values for another. Dashed lines indicate cases where values for parameters establish constraints on values for other parameters. We use this network to establish a vocabulary about kinds of knowledge used in VT's propose-and-refine approach. Afterward, we consider the analysis used by SALT to organize such knowledge.

Within a parameter network, knowledge can be expressed in one of three forms: procedures for computing a parameter value, constraints that specify limits on parameter values, and fixes that specify what to do if a constraint is violated. Figure 8.23 shows some of VT's knowledge for extending a configuration by proposing values for parameters. SALT expects to have a procedure for every parameter. The first rule in Figure 8.24 computes a value for a design parameter from other parameters. A rule like this is shown in a parameter network by arrows between boxes. In Figure 8.23, the rightmost pair of boxes connected by a downward arrow depicts this relation.

The second rule in Figure 8.24 selects a part from the database. Like M1 and XCON, VT has a database of parts that describes the pieces of equipment and machinery VT configures.

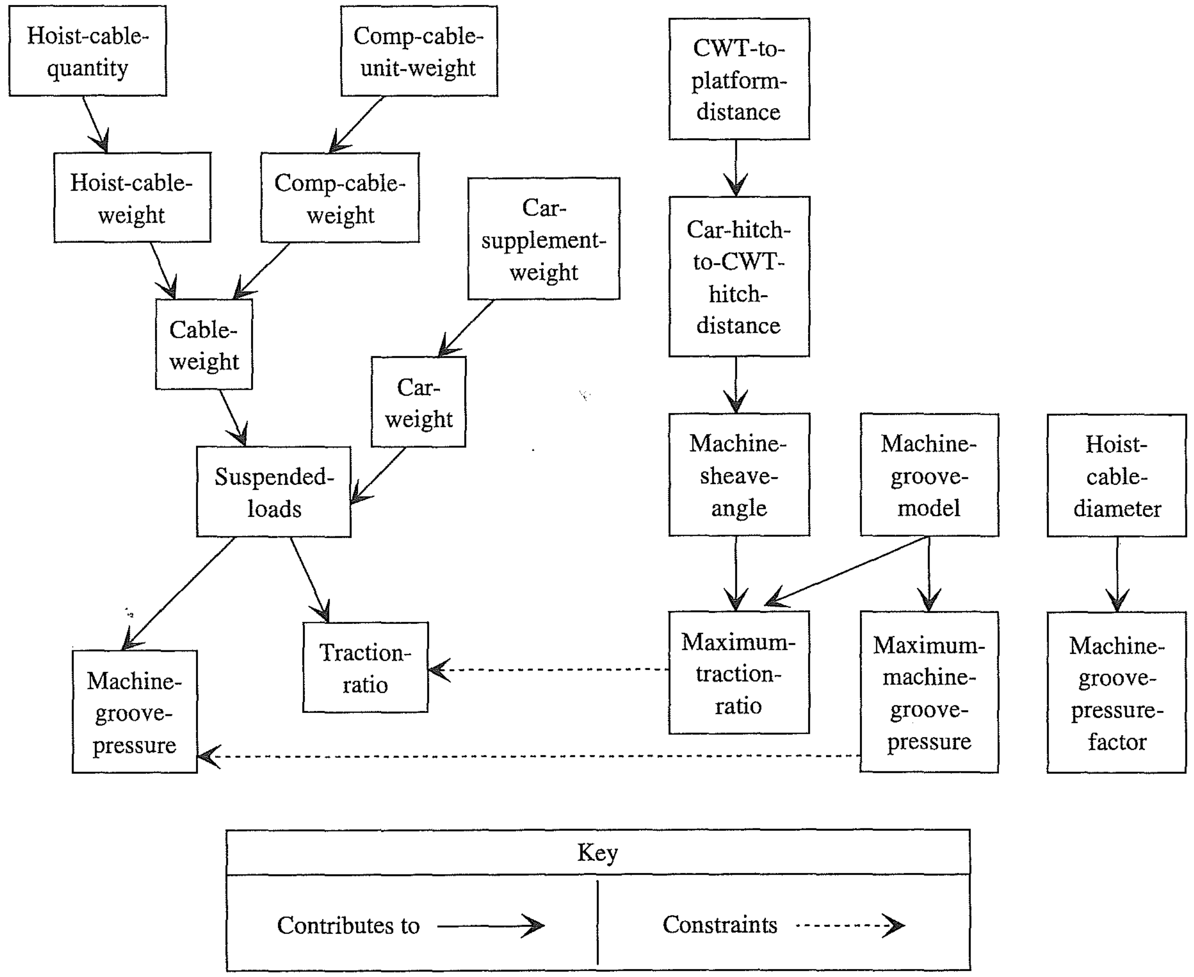


FIGURE 8.23. Segment of VT's knowledge base showing relations among parameters. Each box represents one of VT's design parameters. Arrows between the boxes indicate a partial order for determining values for parameters. (Adapted from Marcus, 1988, Figure 4-5, page 102.)

There is a table of attributes for each kind of part, such as motors and machine models. The attributes describe functional restrictions, such as the maximum elevator speed or the maximum load, and other attributes, such as height or weight. Roughly speaking, VT first establishes values for certain key design parameters and then selects appropriate parts from the database.

Within VT's representational framework, the type of a part is itself just another parameter. The third rule in Figure 8.24 uses attributes from a database entry about a selected part to set values for design parameters. The domain for a given design parameter is the set of values obtainable from the relevant fields of the parts database.

Besides proposing values for design parameters, VT can post constraints on design values. Figure 8.25 gives an example of a constraint, expressed both in a tabular form and as a production rule. The precondition or conditional part of a constraint determines when the constraint is applicable, which means when it will be posted. VT uses its constraints to influence values in a directional way. The constraint in Figure 8.25 constrains a value of the parameter CAR-JAMB-

Computing a value for a parameter

```
IF:        A value has been generated for HOIST-CABLE-DIAMETER, and
           there is no value for MACHINE-GROOVE-PRESSURE-FACTOR
THEN:      Set MACHINE-GROOVE-PRESSURE-FACTOR to be 2 * HOIST-CABLE-DIAMETER.
```

Selecting a part from the component database

```
IF:        A value has been generated for SUSPENDED-LOAD, and
           there is no value for MACHINE-MODEL
THEN:      Search the database in the MACHINE table and retrieve the value
           of the MACHINE-MODEL field for the entry with the SMALLEST WEIGHT
           whose field value for MAX-LOAD is greater than the value of the
           SUSPENDED-LOAD parameter.
JUSTIFICATION:       Taken from Standards Manual IIIA, page 139.
```

Using attributes from a selected part to set other parameters

```
IF:        A value has been generated for MACHINE-MODEL, and
           there is no value for MACHINE-SHEAVE-DIAMETER
THEN:      Search the database in the MACHINE table and retrieve the value
           for MACHINE-SHEAVE-DIAMETER from the value of the SHEAVE-DIAMETER
           field in the machine entry whose field value for MODEL is the
           same as the value for the MACHINE-MODEL parameter.
```

FIGURE 8.24. Examples of rules representing knowledge for extending a configuration. All three rules propose values for parameters. (Adapted with permission from Marcus, Stout, & McDermott, 1988, Figures 7 and 8, page 100.)

Tabular form

```
   constrained value:   CAR-JAMB-RETURN
   constraint type:     MAXIMUM
   constraint name:     MAXIMUM-CAR-JAMB-RETURN
   PRECONDITION:        DOOR-OPENING = SIDE
   PROCEDURE:           CALCULATION
   FORMULA:             PANEL-WIDTH * STRINGER-QUANTITY
   JUSTIFICATION:       This procedure is taken from Installation Manual I,
                        page 12b.
```

Rule form

```
   IF:      DOOR-OPENING = SIDE
   THEN:    CAR-JAMB-RETURN MUST BE <= PANEL-WIDTH * STRINGER-QUANTITY
```

FIGURE 8.25. An example of a constraint from VT's knowledge base. (From Marcus, 1988, page 88.)

```
IF:     There has been a violation of the MAXIMUM-MACHINE-GROOVE-PRESSURE
        constraint
THEN:   (1) Try a DOWNGRADE for MACHINE-GROOVE-MODEL. (Level 1 effect:
        Causes no problem.)
        (2) Alternatively, try an INCREASE BY-STEP of 1 of HOIST-CABLE-
        QUANTITY. (Level 4 effect: Changes minor equipment sizing.)
```

FIGURE 8.26. An example of a rule representing knowledge for selecting a part by performing a search of the database when enough of the parameters are known. When there is more than one possible fix for a problem, VT relies on an ordered set of effect levels, trying more drastic fixes only after the less drastic ones have failed.

RETURN, depending on values of the parameters PANEL-WIDTH and STRINGER-QUANTITY.

Figure 8.26 gives an example of the third form of knowledge used by VT, knowledge for revising a configuration after a violation has occurred. In this example, two alternatives are proposed to guide the choice of revisions when the constraint is violated. Constraint violations can be understood as conflicts over priorities. Viewed in the larger context of the entire elevator system, every procedure for proposing parameter values is a myopic best-first search. If the solution to the overall elevator configuration problem were made up of independent subproblems, then each parameter subproblem could be solved or optimized separately. Constraint violations correspond to antagonistic interactions between subproblems. To arbitrate in these cases, VT uses a scale for comparing the effects of changes. VT recognizes several different kinds of effects for making a change and establishes preferences among them. These effects are summarized in Figure 8.27. According to this list, the least desirable kind of fix is one that compromises the perfor-

Effect level	*Effect of fix*
1	Causes no problem
2	Increases maintenance requirements
3	Makes installation difficult
4	Changes minor equipment sizing
5	Violates minor equipment constraint
6	Changes minor contract specifications
7	Requires special part design
8	Changes major equipment sizing
9	Changes building dimensions
10	Changes major contract specifications
11	Compromises system performance

FIGURE 8.27. Categories of effects from making a change in VT. When VT detects that a constraint has been violated and there are several different possible fixes, it prefers to make fixes with the least drastic consequences. This list enumerates the kinds of fixes from least drastic to most drastic. Every "fix" rule is assigned an effect level corresponding to an entry in this list.

mance of the elevator system. In weighing the costs of different changes, one-time difficulties in installation (#3) are considered less costly than ongoing increases in maintenance costs (#11).

To review briefly, VT distinguishes three main forms of knowledge about elevator design: knowledge for proposing parameter values, knowledge constraining parameter values, and knowledge about what revisions to make when constraints are violated. We next consider some of the analyses and considerations that go into VT's organization of such knowledge. The core of SALT's operation is the analysis and structuring of parameter networks.

Much of SALT's analysis is concerned with checking paths through the network. In the ideal case, there are unique paths through the network whereby all the elevator system parameters can be computed for any combination of input parameters. In the simplest case, VT would need only to invoke knowledge for proposing parameter values, without using knowledge for constraining parameter values or fixing values after constraint violations. The partial ordering represented by the network would allow VT to compute other parameter values from the given ones, continuing in a data-driven fashion until the elevator was configured. Such an ideally structured computation has not proven workable in practice. Analogous to RIME, SALT can be understood as a structured programming environment for building parameter networks that approach this ideal. It uses the equivalent of structured programming in the parameter network to yield behaviors that converge to optimal configurations given a wide range of input parameters.

SALT's analysis includes checking for a number of undesirable patterns in the parameter network, including ones involving loops. These patterns correspond to such things as race conditions, unreachable paths, and potential deadlocks as described in the following. For example, if one procedure is applicable for computing motor torque for speeds less than 300 and another is applicable for speeds greater than 200, then there would be two applicable procedures for the speed 250. This is a race condition because either procedure could be executed depending on the order of traversal used by VT's interpreter. Similarly, there might be combinations of conditions such that no procedure is applicable for some combination of parameters.

A deadlock condition is recognized when there are procedures for computing two parameters such that each procedure requires the value of the other parameter before it can be run. In a data-driven interpreter, this would result in an infinite delay while each procedure waits for the other. When it detects the possibility of a deadlock, SALT guides the builder of a knowledge base in adding procedures that estimate initial parameter values. When this results in multiple procedures proposing a value, SALT guides the user in converting some of the procedures to post constraints on values and in adding procedures that propose fixes when there is a conflict. In this way, SALT incrementally builds revision cycles into the parameter network. These revision cycles correspond to programming idioms. For example, a revision loop can incrementally increase the value of a parameter up to some limit. That idiom, implemented in terms of constraints and fixes, corresponds in function to an iterative looping construct in procedural programming languages.

SALT assumes that the procedures for proposing parameter values correspond to an underconstrained case reflecting local optimality. Potential fixes should be less preferred (locally) than the value originally proposed. The default strategy in VT is to try values in a best-first order. If the fixes for one constraint violation have no effect on other constraint violations, then this strategy guarantees that the first configuration found will be the most preferred. However, it is possible that fixes selected for one constraint violation may interact with values for other parameters in the network. A particularly pathological case of feedback between conflicting fixes is called thrashing. Thrashing occurs in scenarios of different complexity. In a simple scenario, a fix for one parameter value causes a new value to be computed for a second variable, which then violates a constraint, causing a fix to a third variable, which causes an opposing change in the first variable, violating the original constraint again. The term **thrashing** refers to any persistent looping behavior that results when antagonistic fixes undo and redo a repeating sequence of changes.

SALT uses syntactic techniques to analyze a parameter network for the possibility of these behaviors. A detailed consideration of SALT's syntactic analysis methods and the expressive power of the language is beyond the scope of this section. The tests built into SALT are intended to detect common errors of omission by builders of a VT knowledge base.

Given a SALT-created network, VT works on a particular case as follows. It starts with a forward-chaining or data-driven phase in which procedures extend the configuration by proposing parameter values. As it extends the design, VT records which parameter values were used to derive other ones using a truth maintenance system (TMS). Constraint violations are detected by demons, which invoke procedures to revise the network. The fix procedures then propose changes to values of individual parameters or to combinations of parameters.

The effects of the proposed change are investigated with limited look-ahead as follows. First VT verifies that the revised value violates no constraints on the changed parameter. Then it works through the consequences, verifying that there are no antagonistic changes to the proposed fix using a TMS. If a proposed change violates constraints, then the next alternative fix is considered. The TMS keeps track of combinations of revisions it has tried so as not to repeat any. This process is a due-process approach. The prioritized list of effects in Figure 8.27 guides backtracking to approximate a best-first search.

From a network perspective like that in Figure 8.23, many kinds of interactions in the search space can be visualized. The most common interaction among design parameters in VT is a rippling effect that follows when a constraint violation causes one piece of equipment to be upgraded, or increased in size. In this case, other pieces of related equipment may be affected and upgraded as well. Depending on circumstances, the initial fix may trigger a sequence of changes. For example, requiring more hoist cables than fit on the machine model selected may result ultimately in selecting a larger machine model and using larger sheaves.

The 1988 version of VT recognized 54 different kinds of constraint violations, of which 37 had only one known fix, meaning one parameter that might be revised. The remaining constraint violations also had a small number of fixes. Each possible fix provides directions of where to look next in VT's search for solutions. On an average case, however, VT needs to consider only a small number of fixes. In a typical run, VT makes between 1,000 and 2,000 design extensions and only about one-hundredth as many fixes to a design. In a typical case, VT detects 10 or 20 constraint violations and averages just slightly more than 1 fix per violation.

In summary, VT constructs an approximation to a solution and then refines it. VT's knowledge base is made up predominantly of three kinds of knowledge: knowledge for proposing design extensions, knowledge for posting constraints, and knowledge for revising the configuration when constraint violations are detected. VT's rules are applied in a data-driven fashion with limited look-ahead. VT's performance using this approach depends on the prior analysis by SALT, which identifies undesirable patterns and missing information in the network. In representative runs, VT detects very few constraint violations, and the first fix tried almost always works.

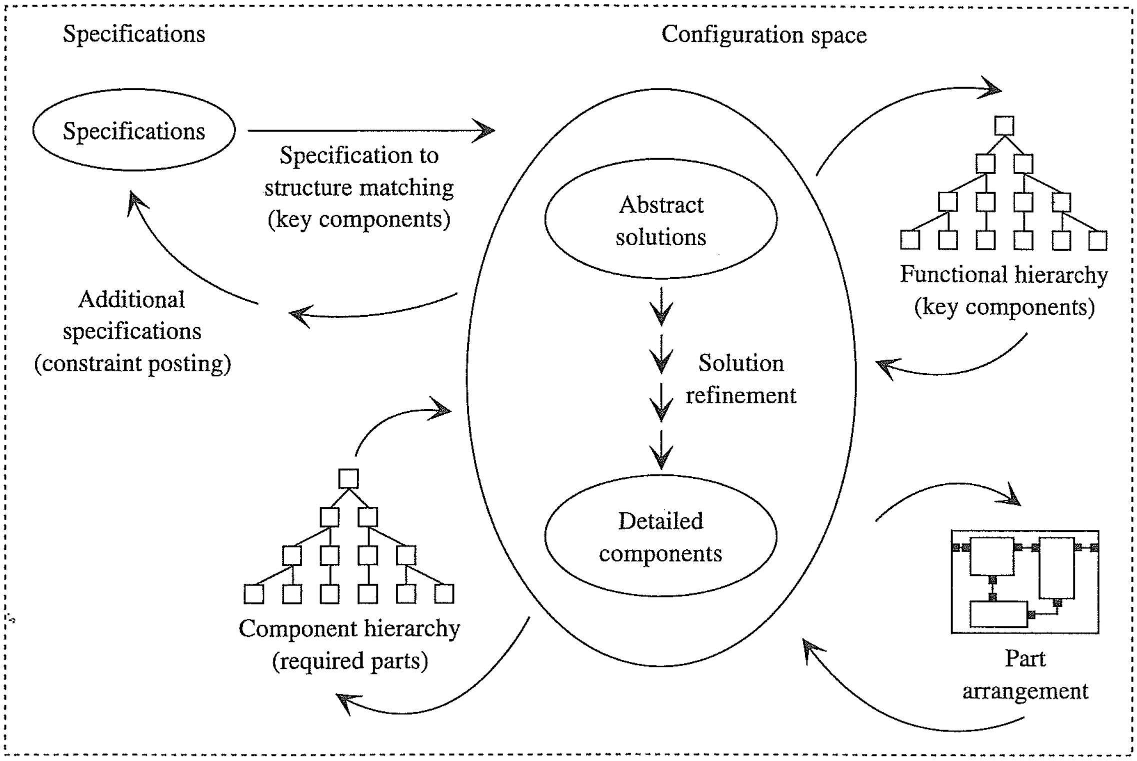

Specifications
Configuration space
Specifications
Specification to structure matching (key components)
Additional specifications (constraint posting)
Abstract solutions
Solution refinement
Detailed components
Functional hierarchy (key components)
Component hierarchy (required parts)
Part arrangement

FIGURE 8.28. COSSACK begins with user specifications, in terms of evaluation criteria and functional specifications. The evaluation criteria are used to order candidates in component selection. Functionality specifications are represented using a key-component approach, where key components may be either hardware or software components.

8.3.5 *Configuration in COSSACK*

COSSACK (Mittal & Frayman, 1989) is a knowledge system for configuring personal computers that was developed at Xerox Corporation. Because Xerox stopped selling personal computers shortly after COSSACK became operational, the system never received substantial field testing. However, COSSACK is an interesting system from a technical point of view especially in its use of a **partial-choice strategy**.

Like the other computer configuration systems we have considered so far, COSSACK's process starts with customer requirements. These include evaluation criteria, such as minimum cost, expandability, early delivery requirements, and functionality requirements. COSSACK performs a best-first search, using the evaluation criteria to order candidates during component selection. Figure 8.28 shows the elements involved in COSSACK's search. Specifications are mapped onto abstract solutions. Abstract solutions are incrementally refined, making choices about component selection and arrangement.

COSSACK uses key components to express functionality specifications. It creates requirements in terms of specific components, such as an "Epson-LX80 printer," and also in terms of abstract component descriptions, such as a "letter-quality printer." Figure 8.29 presents a portion of COSSACKS's functional hierarchy. In this class hierarchy, each node represents a sub-

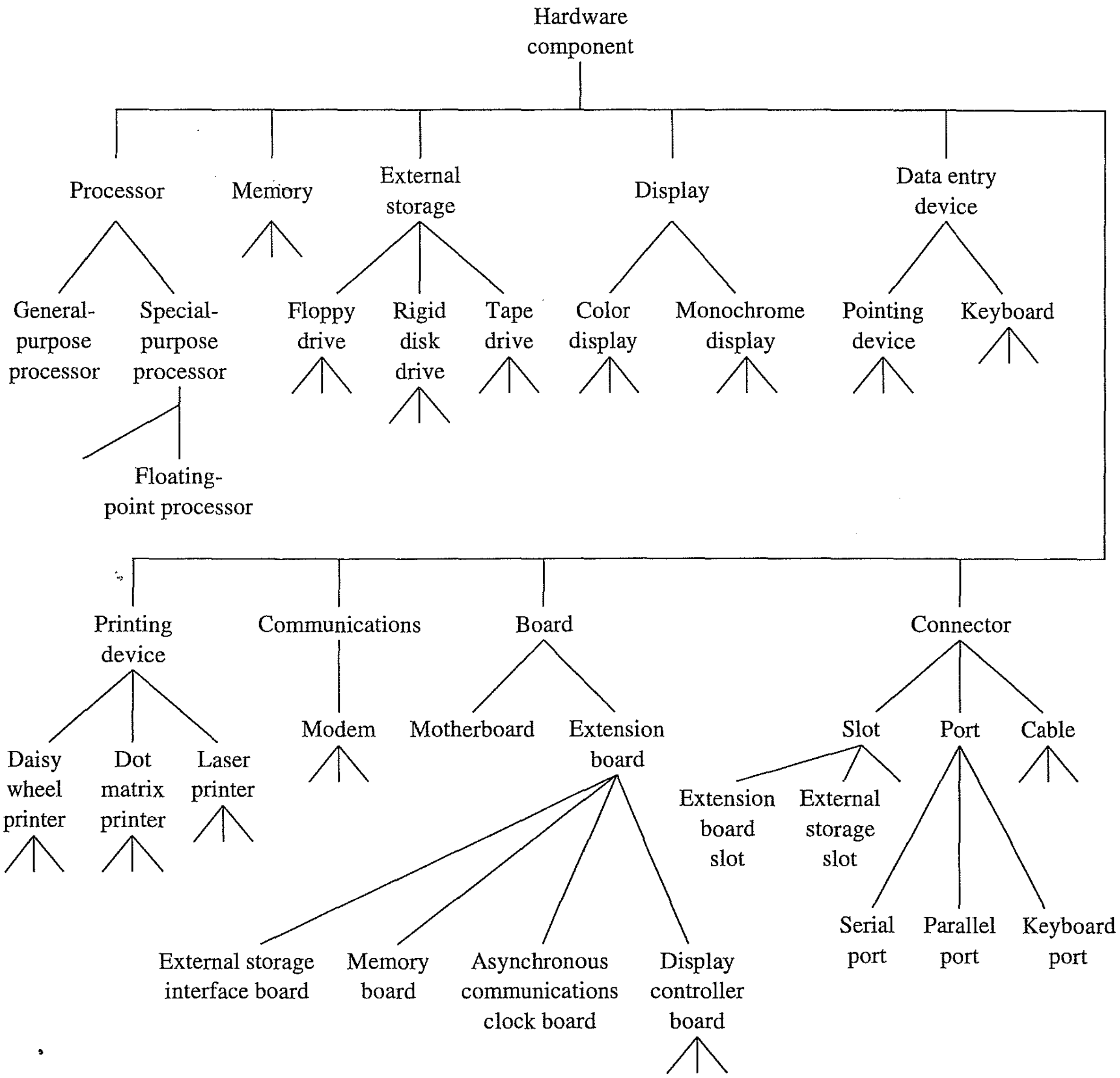
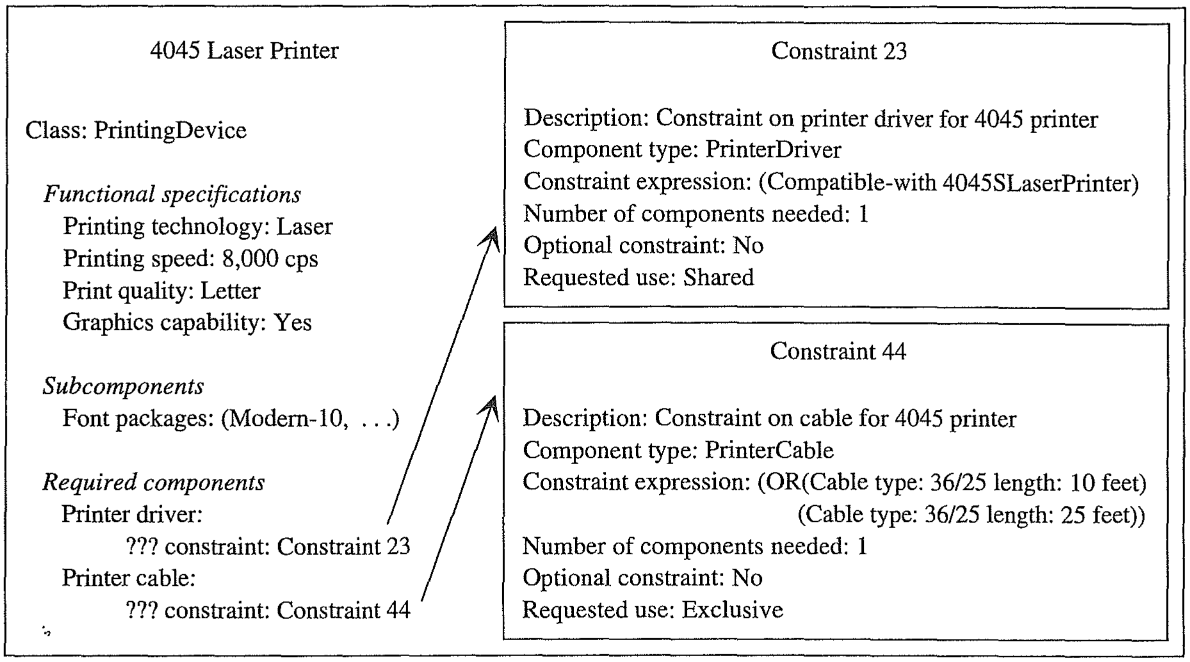


FIGURE 8.29. Portion of the functional hierarchy for components in COSSACK. Any component below the root can be a key component. COSSACK also considers software components in its configuration and has classes for word processors, accounting programs, spreadsheets, and operating systems.

class of its parent node. Any node lower than hardware component, the root, can be used to express a key component. Besides the classes of hardware components shown, COSSACK represents software components as well. In January 1987, COSSACK represented a total of 65 different hardware components and 24 different software components.

As in other configuration tasks, specifying a key component often leads to requirements for other components. COSSACK distinguishes two main relations for this, `subcomponent` and `required component`. The `subcomponent` relation indicates that other components are always included with the key component. This can reflect a manufacturing decision, as when multiple components are assembled as part of the same board. For example, the manufacturing



4045 Laser Printer

Class: PrintingDevice

Functional specifications
Printing technology: Laser
Printing speed: 8,000 cps
Print quality: Letter
Graphics capability: Yes

Subcomponents
Font packages: (Modern-10, . . .)

Required components
Printer driver:
??? constraint: Constraint 23
Printer cable:
??? constraint: Constraint 44

Constraint 23

Description: Constraint on printer driver for 4045 printer
Component type: PrinterDriver
Constraint expression: (Compatible-with 4045SLaserPrinter)
Number of components needed: 1
Optional constraint: No
Requested use: Shared

Constraint 44

Description: Constraint on cable for 4045 printer
Component type: PrinterCable
Constraint expression: (OR(Cable type: 36/25 length: 10 feet)
(Cable type: 36/25 length: 25 feet))
Number of components needed: 1
Optional constraint: No
Requested use: Exclusive

FIGURE 8.30. Frame representation of a component in COSSACK's functional hierarchy. This frame represents a 4045 laser printer. Frame slots are organized into groups: functional specifications, subcomponents, required components, and processing information. The functional specifications are used when matching this description against specifications using key words. Slots in subcomponents refer to other parts, both hardware and software, that are included or bundled with this component. The required-component slots describe other components, not bundled with this one, that are needed for correct operation. Constraints associated with those slots are used to narrow the choices.

group may package a clock, memory, and modem on one multifunctional board. Parts can also be bundled for marketing reasons. The `required component` relation indicates that other parts are necessary for the correct functioning of a part. As with XCON and M1, this indicates a range of requirements for auxiliary parts and cables. `Required component` relations are also useful for software components. For example, these relations can indicate how much memory or external storage capacity a particular software package needs to run.

Figure 8.30 gives an example of a frame representation of a component from COSSACK's knowledge base. The key component in this case is a 4045 laser printer. The slots grouped under "Functional specifications" are used for matching. For example, this printer would satisfy a specification for a letter-quality printer. The subcomponents include various other parts that are necessarily included. This example shows that various font packages are routinely bundled with a printer purchase.

Required components for the 4045 printer include a software driver for the printer and a cable. Like VT, COSSACK posts constraints to guide future decisions. In Figure 8.30 Constraint 23 indicates that any driver can be used that is compatible with the 4045 printer. Constraint 44 indicates that the cable must be of type 36/25 and can have a length of either 10 or 25 feet.

The two constraints in this figure illustrate a representational issue not dealt with in the previous configuration systems that we considered: the reuse or sharability of components across different requirements. When one component description presents a requirement for another component, can the second component be used to satisfy any other requirements? In Figure 8.30, the software module for the printer driver can be shared, meaning that, in principle, the driver can be used for other components as well, such as when a configuration includes multiple printers. Similar issues come up for required components for hardware. For example, can a modem and a printer use the same RS232 port? In contrast, constraint 44 indicates that the printer cable cannot be shared. It must be dedicated to the exclusive use of one printer.

Like VT, COSSACK does not organize its design subtasks in a fixed order for all subproblems. Also like VT, it has knowledge for extending a design by making choices and for posting constraints. COSSACK is able to retract component decisions, revisit preference-guided choices, and remove constraints that were posted as requirements when the retracted components were originally selected. Unlike VT, COSSACK does not use domain-specific knowledge for modifying a configuration when constraint violations are detected, but rather employs a blind search.

COSSACK uses a partial-commitment strategy to reduce backtracking. Figure 8.31 shows a simplified example of a case where partial commitment makes guessing and backtracking unnecessary. In this example, both the operating system and an accounting package are specified as key components. Neither component has any immediate constraints bearing on the other. We assume that no other preferences are known to guide the choice. At this point, a simple generate-and-test approach could arbitrarily select candidates for either one. Suppose it arbitrarily selected Accounting Package #1 and OS Version B. Then later, when it expanded the required components for Acounting Package #1, it would pick a rigid disk drive and discover that none of the satisfactory disk drives are compatible with OS Version B, leading to backtracking. The partial-choice approach enables it to generalize the requirements held in common by Accounting Package #1 and Accounting Package #2. This would result in the generation of a constraint specifying that whatever rigid disk was chosen, its operating system must be OS Version A.

Arrangement issues in COSSACK are not as complex as in XCON and are represented using constraints on a port-and-connector model. The main arrangement conflict is that personal computers have a limited number of slots. Components consume both slots and ports.

In summary, COSSACK is a knowledge-based system for configuring personal computers. It uses a key-component model for functionality, a preference model for selecting components in a best-first search, constraint posting to add specifications for related decisions, and partial commitment to reduce backtracking. It relaxes the notion that required components need to be dedicated to a single use by representing "sharable" components.

8.3.6 *Summary and Review*

A configuration is an arrangement of parts. A configuration task instantiates a combination of parts from a predefined set and arranges them to satisfy given specifications. The large size of configuration search spaces precludes pre-enumeration of complete solutions. Knowledge-based systems for configuration are important to manufacturers that customize their products for individual customers by configuring customer-specific systems from a catalog of mass-produced parts.

XCON is a well-known knowledge system for configuring computer systems at Digital Equipment Company. Together with its companion program, XSEL, XCON tests the correctness

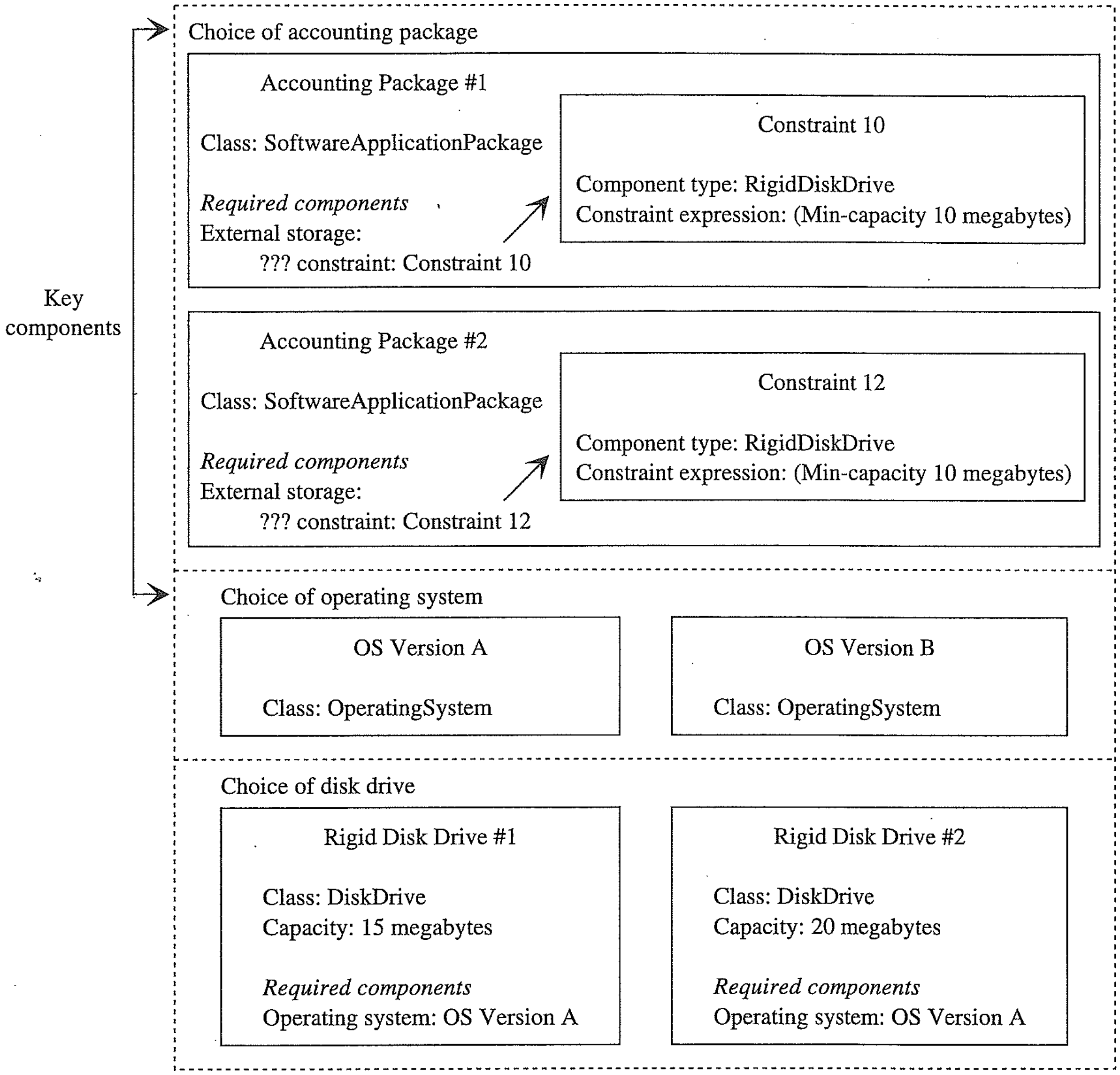

Key components
Choice of accounting package
Accounting Package #1
Class: SoftwareApplicationPackage
Required components
External storage:
??? constraint: Constraint 10
Constraint 10
Component type: RigidDiskDrive
Constraint expression: (Min-capacity 10 megabytes)
Accounting Package #2
Class: SoftwareApplicationPackage
Required components
External storage:
??? constraint: Constraint 12
Constraint 12
Component type: RigidDiskDrive
Constraint expression: (Min-capacity 10 megabytes)
Choice of operating system
OS Version A
Class: OperatingSystem
OS Version B
Class: OperatingSystem
Choice of disk drive
Rigid Disk Drive #1
Class: DiskDrive
Capacity: 15 megabytes
Required components
Operating system: OS Version A
Rigid Disk Drive #2
Class: DiskDrive
Capacity: 20 megabytes
Required components
Operating system: OS Version A

FIGURE 8.31. Simplified partial commitment example from COSSACK. In this example, we assume that an accounting package and an operating system have been specified as key components. COSSACK notices that whatever rigid disk drive it chooses, the required operating system is OS Version A.

of configurations and completes orders for computers. Like all configuration systems, XCON relies on a component database and a body of rules that describes what combinations of parts work correctly together and how they should be arranged. The challenge of maintaining a large database about parts and a body of rules about their interactions has led to the development of a programming methodology for XCON called RIME. RIME organizes configuration knowledge in terms of small recurring subtasks such as selecting a device, selecting a container for a component, and selecting a location within a container to place a component. To a first approximation, these subtasks are local, explicit, and separate processes. To represent knowledge about the interactions among these subtasks, RIME organizes the subtask descriptions as a sequence of explicit stages that look ahead for interactions in the situation and select a best choice for part selection or arrangement before executing that choice.

M1 is an experimental knowledge system that configures single-board computers given a high-level specification of board functions and other design constraints. M1 represents components in a functional hierarchy. Like XCON, M1 relies on the use of key components to specify system function. It selects abstract parts and then instantiates them. Spatial layout is carried out by commercial programs for layout and wire routing.

In MYCIN, a therapy is a selected combination of drugs used to treat a patient for infectious diseases. This example shows that configuration can be used as a model for a task not involving manufacturing. The heuristic classification model that MYCIN uses for diagnosis is inadequate for characterizing therapy because the number of possible solutions based on combinations of drugs to cover combinations of diseases is too large to pre-enumerate. MYCIN's configuration model separates consideration of local factors, such as the sensitivity of the organisms to different drugs, from global factors that deal with the entire recommendation, such as minimizing the number of drugs.

VT is a knowledge system that configures elevator systems at the Westinghouse Elevator Company. Interwoven with VT's model of parts is a parametric model of elevator systems. VT's approach involves separate phases for knowledge acquisition and analysis and for solving particular cases. The analysis phase includes checking for race conditions, unreachable paths, and potential deadlocks. The case phase uses data-driven procedures to extend the configuration by proposing parameter values. VT's knowledge is organized within a parameter network in one of three forms: procedures for computing a parameter value, constraints that specify limits on parameter values, and fixes that specify what to do if a constraint is violated. Analogous to RIME, VT's analysis and knowledge acquisition facility can be viewed as a structured programming environment for building parameter networks.

COSSACK is a knowledge system for configuring personal computers. It begins with user specifications, in terms of evaluation criteria and functional specifications. The evaluation criteria are used to order candidates in component selection. Functionality specifications are represented in terms of key components. In describing how components require each other, COSSACK distinguishes between subcomponents that are always included and ones that must be added. COSSACK also distinguishes between different ways that components may be shared to satisfy multiple requirements. COSSACK uses a partial-commitment strategy.

Challenges in configuration tasks arise from the size of the search spaces and the complexity of individual solutions. Most configuration systems build descriptions of solutions incrementally. However, the acceptability of a configuration depends not only on local considerations. Following the threshold effect and horizon effect, interactions can propagate and be far-reaching. The next section considers how different computational methods used in these cases try to cope with these effects.

Exercises for Section 8.3

Ex. 1 [*05*] *Prefigured Configurations.* Professor Digit argues that very few of the possible VAX computer configurations are actually ever built. By his own estimate, no more than 1 in 1,000 of the configurations that are possible for VAXes will ever be ordered. He believes that all the effort in building knowledge-based configuration programs like XCON is unnecessary. He recommends instead that a table recording all actual configurations be kept and used to avoid refiguring them. This table would contain the input/output pairs from the history of configuration. It would show what configuration was used for every VAX system that has been ordered. The first column of the table would contain the specifications and the second column would describe the configuration.

(a) Briefly discuss the merits of Professor Digit's proposal. Is a reduction by a factor of 1,000 enough to make the table size manageable?

(b) Discuss advantages for building a knowledge-based configuration system over a "table system" even in cases where the determination of configurations is not computationally challenging.

Ex. 2 [*10*] *Due Process in Configuration Problems*. Several of the descriptions of the configuration systems mentioned optimization and best-first search.

(a) Which of the configuration systems discussed in this section perform exhaustive searches for solutions?

(b) Briefly explain the meaning of the term *due-process search* in the context of configuration problems, relating it to ideas for best-first search and optimization as discussed in this section. What are the implications of a high rate of turnover of knowledge in configuration domains?

Ex. 3 [*05*] *"Parts Is Parts."* This exercise considers limitations of a simple port-and-connector model for configuration as shown in Figure 8.32.

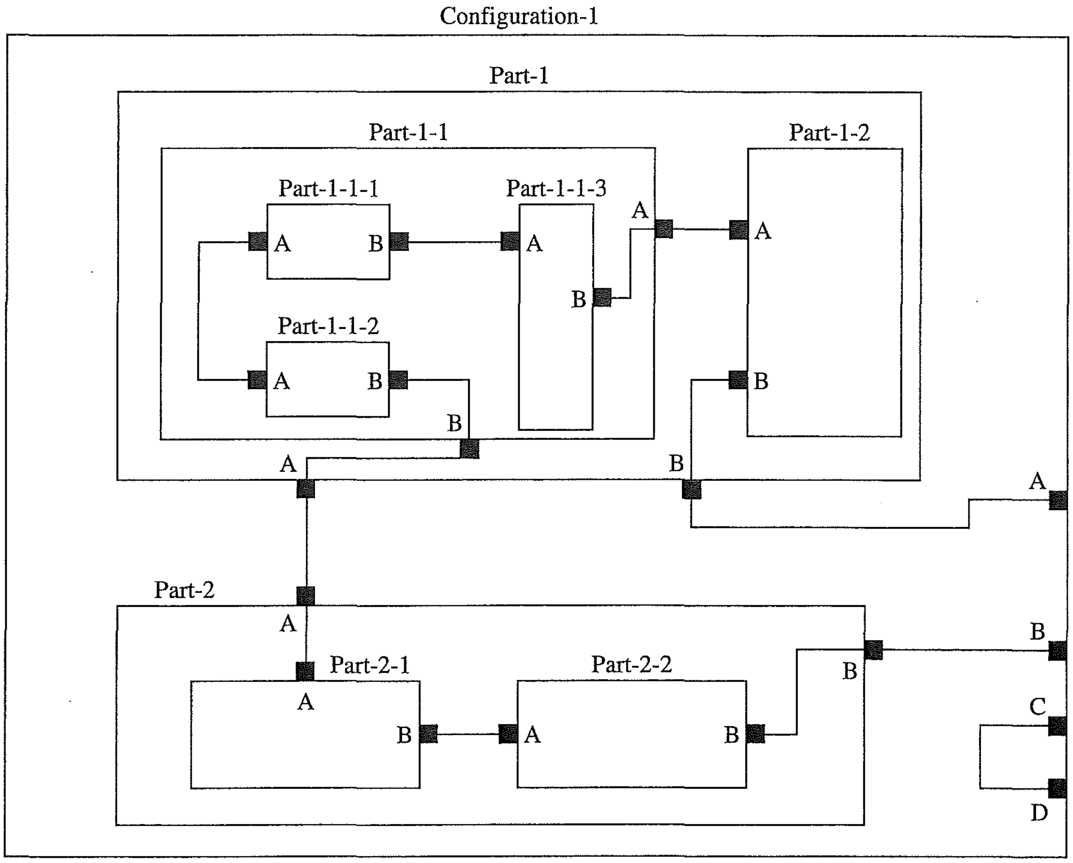


FIGURE 8.32. Ports and connectors.

(a) How many top-level parts are there in configuration-1? How many detailed parts are shown?

(b) Why is there a difference between subparts and required parts in configuration problems?

(c) Are cables parts? A port-and-connector model represents cables as mere connections between ports. Briefly, what are some of the practical issues in representing cables as parts? How could a port-and-connector model be augmented to accommodate this?

Ex. 4 [*05*] *Key Components.* The key-component approach is the technique of representing the functionality of a configuration in terms of essential components or abstract components in a partial configuration.

(a) Briefly, what representational issues does this approach seek to side-step?

(b) What is an advantage of this for the user interface in systems like XCON?

(c) Does the configuration model for therapy in MYCIN use a key-component approach?

■ **Ex. 5** [*!-10*] *Thrashing in Antagonistic Subproblems.* Thrashing occurs in VT when fixes for constraint violations of one parameter lead to constraint violations of another parameter, whose fixes lead ultimately to further violations of the first parameter.

Here is an example of thrashing behavior for the parameter network in Figure 8.33: VT derives a value for machine-groove-pressure and maximum-machine-groove-pressure and finds that machine-groove-pressure is greater than the maximum. This triggers a fix that decreases car-supplement-weight. This decreases car-weight, which in turn decreases suspended-load. This decreases machine-groove-pressure, the desired effect, but also increases traction-ratio. An increase in traction-ratio makes it more likely for it to exceed its maximum. A violation of maximum-traction-ratio leads to the fix of increasing comp-cable-unit-weight, which in turn increases comp-cable-weight, cable-weight, and suspended-load. Increasing suspended-load increases machine-groove-pressure. Thrashing occurs if this scenario repeats.

(a) Annotate Figure 8.33 with directed arcs to show the operation of the fix operations to revise parameter values. Also annotate it with + and – to show the subsequent increases and decreases in parameter values.

(b) An antagonistic interaction is one where there are simultaneous contributions from different arcs in opposite directions to change the value of a parameter. At which nodes in the thrashing scenario are there antagonistic interactions? Briefly, what measures can be taken in systems like VT to deal with such interactions?

Ex. 6 [*10*] *Structured Programming Methodologies for Configuration Knowledge.* Several of the configuration projects discussed in this section have needed to face the classical knowledge engineering issue of providing support for acquiring, organizing, and maintaining a large knowledge base. In two of the cases that were described (XCON/RIME and VT/SALT), special languages or subsystems were developed for this. This exercise briefly compares the approaches taken for RIME and SALT.

(a) Both XCON and VT build descriptions of solutions incrementally. How is this reflected in the forms of knowledge that they expect?

(b) What is the distinction between local and global interactions in configuration problems? Why is this distinction important?

(c) Briefly compare propose-and-apply with propose-and-revise.

Ex. 7 [*10*] *Configuration Grammars.* Professor Digit says the knowledge representations used in configuration systems are excessively ad hoc. In particular, he advocates developing gram-

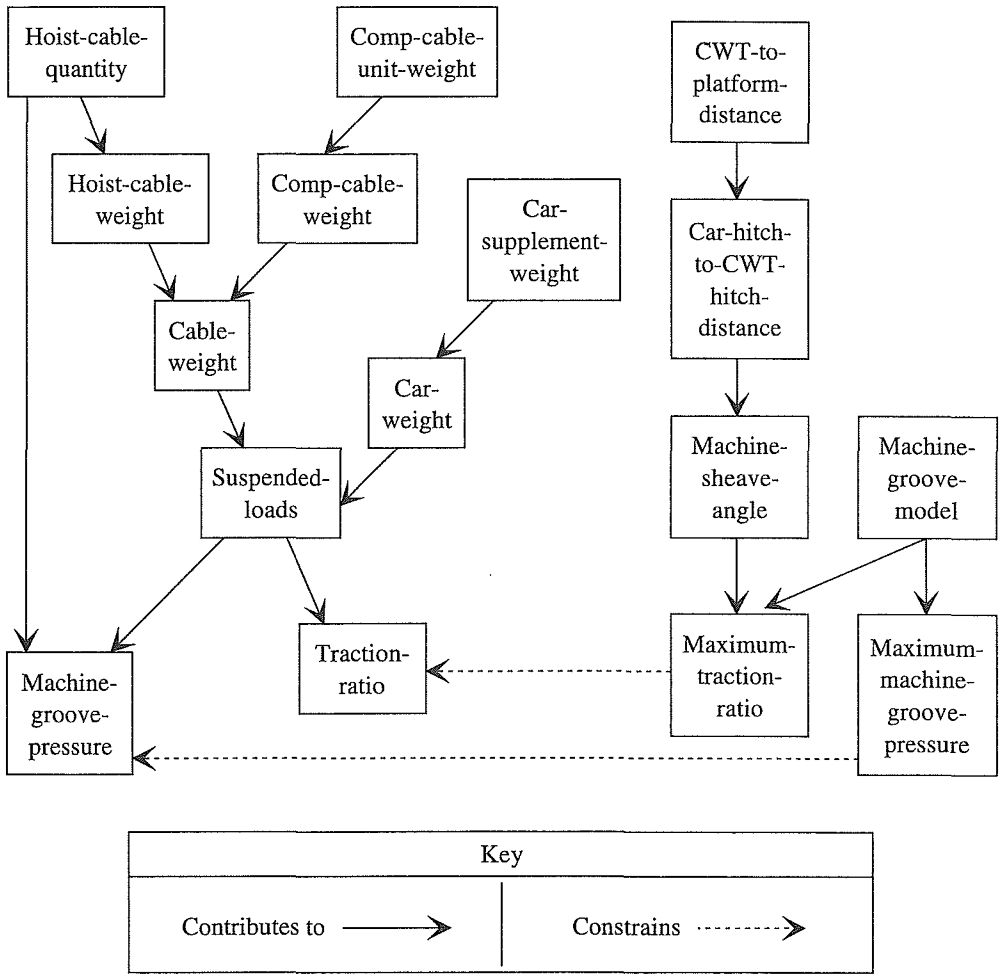


FIGURE 8.33. Interactions between antagonistic subproblems. (Adapted from Marcus & McDermott, 1989, Figure 5, page 18.)

mars for describing valid configurations analogous to those used for describing valid sentences in computer languages. He argues that standard algorithms from the parsing literature could then be used to carry out configuration tasks.

(a) Briefly list some of the kinds of domain knowledge needed for configuration tasks. What are some of the main problems with Professor Digit's proposal for using grammars to represent this knowledge?

(b) Is configuration essentially a parsing task? Explain briefly.

Ex. 8 [05] *Levels of Description.* Professor Digit can't decide whether XCON, the well-known rule-based computer configuration program, reasons forward or backward. On the one hand, the production rules used by XCON are interpreted by forward chaining. On the other hand, he has seen XCON described as working backward by setting up subproblems. Briefly, how can the apparent terminological confusion be resolved? Are either of these characterizations useful in understanding the main computational phenomena that arise in XCON's configuration task?

8.4 *Methods for Configuration Problems*

In the preceding sections we considered example knowledge systems for configuration tasks and developed a computational model to compare and analyze configuration domains. This section revisits our model of configuration to discuss knowledge-level and symbol-level analysis and then presents sketches of methods for configuration. Our goal in this section is not to develop an "ultimate" configuration approach, but rather to show how variations in the requirements of configuration tasks can be accommodated by changes in knowledge and method.

As in the case of classification tasks, most of the complexity in configuration tasks is in the domain-specific knowledge rather than in the search methods. Furthermore, most of the remaining complexity in the methods is in implementations of general search techniques. In this section the methods are stripped down to basics so we can see the assumptions they depend on in the configuration domain.

8.4.1 *Knowledge-Level and Symbol-Level Analysis of Configuration Domains*

Knowledge about configuration is used by the submodels of our model. We begin our knowledge-level analysis by considering variations in the knowledge from the domains of our case studies. The submodels define major categories of knowledge for configuration in terms of its content, form, and use.

The Parts Submodel: Knowledge about Function and Structure

The parts submodel contains representations and knowledge about what the available parts are, what specifications they satisfy, and what requirements they have to carry out their functions. The simplest parts submodel is a catalog, which is a predetermined set of fixed parts. However, in most configuration domains the set of parts includes abstractions of them, organized in an abstract component hierarchy also called a functional hierarchy. Such hierarchies are used in mapping from initial specifications to partial configurations and in mapping from partial configurations to additional required parts. In most configuration domains these hierarchies represent functional groupings, where branches in the hierarchy correspond to specializations of function. In the last section we saw several examples of functional hierarchies in the computer configuration applications for XCON, M1, and COSSACK.

At the knowledge level an abstraction hierarchy guides problem solving, usually from the abstract to the specific. At the symbol level a parts hierarachy can be implemented as an index into the database of parts. There are several other relations on parts that are useful for indexing the database.

The determination of required parts is a major inference cycle in configuration. This process expands the set of selected parts and consumes global resources. The **required-parts relation** indicates what additional parts are required to enable a component to perform its role in a configuration. For example, a disk drive requires a disk controller and a power supply. These parts, in turn, may require others. These relations correspond to further requirements, not to specializations of function. In contrast, power supplies are required parts for many components with widely varying functions. Power supplies perform the same function without regard to the function of the components they serve.

We say that parts are **bundled** when they are necessarily selected together as a group. Often parts are bundled because they are manufactured as a unit. Sometimes parts are manufactured together because they are required together to support a common function. For example, a set of computer clocks may be made more economically by sharing parts. In other cases parts are made together for marketing reasons. Similarly, bundling components may reduce requirements for some resource. For example, communication and printing interfaces may be manufactured on a single board to save slots.

In some domains, parts are bundled because they are logically or stylistically used together. In configuring kitchen cabinets, the style of knobs and hinges across all cabinets is usually determined by a single choice. Parts can also be bundled for reasons not related to function or structure. For example, a marketing group may dictate a sales policy that certain parts are always sold together.

Some part submodels distinguish special categories of "spanning" or "dummy" parts. Examples of spanning parts from the domain of kitchen cabinets are the space fillers used in places where modular cabinets do not exactly fit the dimensions of a room. Examples of filler parts from computer systems are the conductor boards and bus terminators that are needed to compensate electrical loads when some common component is not used. Examples from configuring automobiles include the cover plates to fill dashboard holes where optional instruments were not ordered. Dummy and filler parts are often added at the end of a configuration process to satisfy modest integrity requirements.

Another variation is to admit parameterized parts, which are parts whose features are determined by the given values of parameters. A modular kitchen cabinet is an example of a parameterized part. Parameters for the cabinet could include the choice of kind of wood (cherry, maple, oak, or alder), the choice of finish (natural, red, golden, or laminate), and the choice of dimensions (any multiple of 3 inches from 9 inches to 27 inches).

Finally, parameterization can be used to describe properties of parts and subsystems, as in the VT system. Global properties of a configuration such as power requirements, weight, and cost are usually treated as parameters because they depend on wide-ranging decisions. In the VT system, essentially all the selectable features of the elevator system and many intermediate properties are represented in terms of parameters.

The Arrangement Submodel: Knowledge about Structure and Placement

The arrangement submodel expresses connectivity and spatial requirements. There are many variations in the requirements for arrangement submodels. Some configuration domains do not require complex spatial reasoning. For example, VT's elevator configurations all use minor variations of a single template for vertical transport systems. M1 uses a port-and-connector model for logical connections but its configurations are spatially uncommitted. Spatial arrangement of antibiotics is not relevant for MYCIN therapy recommendations.

Arrangements do not always require distance metrics or spatial models. In configuring the simplest personal computers, the slots that are used for different optional parts are referenced and accounted for by discrete names or indexes. A configuration task need only keep track of which slots are used without bothering about where they are.

Figure 8.34 illustrates several examples of specialized arrangement models. The simplest of these is the port-and-connector model. This model was used in the M1 and COSSACK appli-

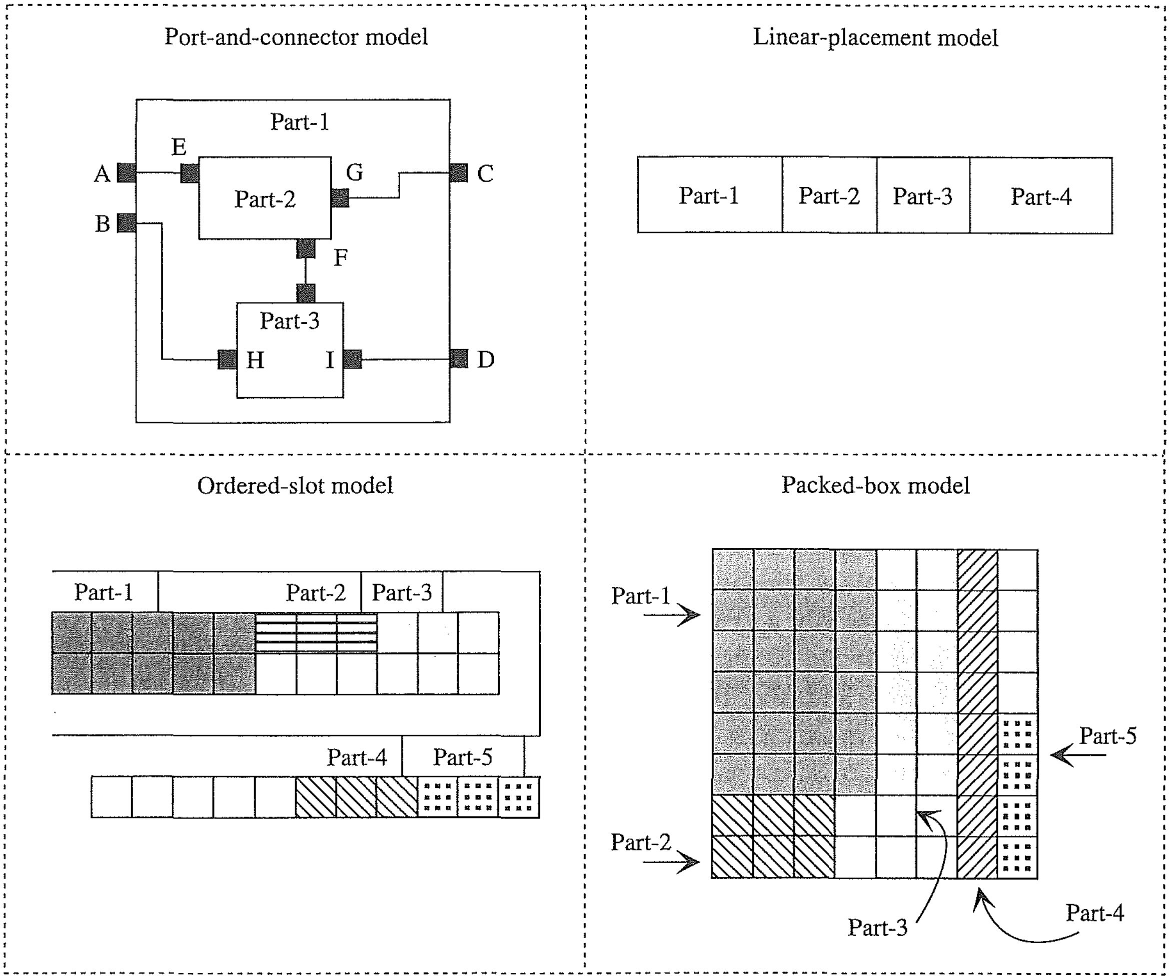


FIGURE 8.34. Alternative models of arrangement for configuration problems. Different arrangement models lend themselves to different resource allocation strategies.

cations discussed earlier. In this model, ports are defined for all the configurable parts. Ports have types so that different kinds of ports have different properties. Ports are resources than can be assigned to at most one connector. COSSACK implemented a port-and-connector model in a frame language, combining its use for specifying the arrangement of parts with indexing on constraints that specified what parts were compatible or required. The linear-placement model organizes parts in a sequence. A specialization of this model was used in XCON for specifying the order of electrical connections on a computer bus. In that application, roughly speaking, it is desirable to locate parts with a high interrupt priority or a high data rate nearer the front of the bus. Location on the bus was a resource that was consumed as parts were arranged. The ordered-slot model combines a linear ordering with a spatial requirement. In the version in Figure 8.34, parts are ordered along a bus, starting with Part-1 and ending with Part-4. Parts in the first row can be up to two units deep, and parts in the second row can be only one unit deep. In this example, both space and sequence are resources that get consumed as parts are arranged. The packed-box model emphasizes the efficient packing of parts. In this model, space is a consumable

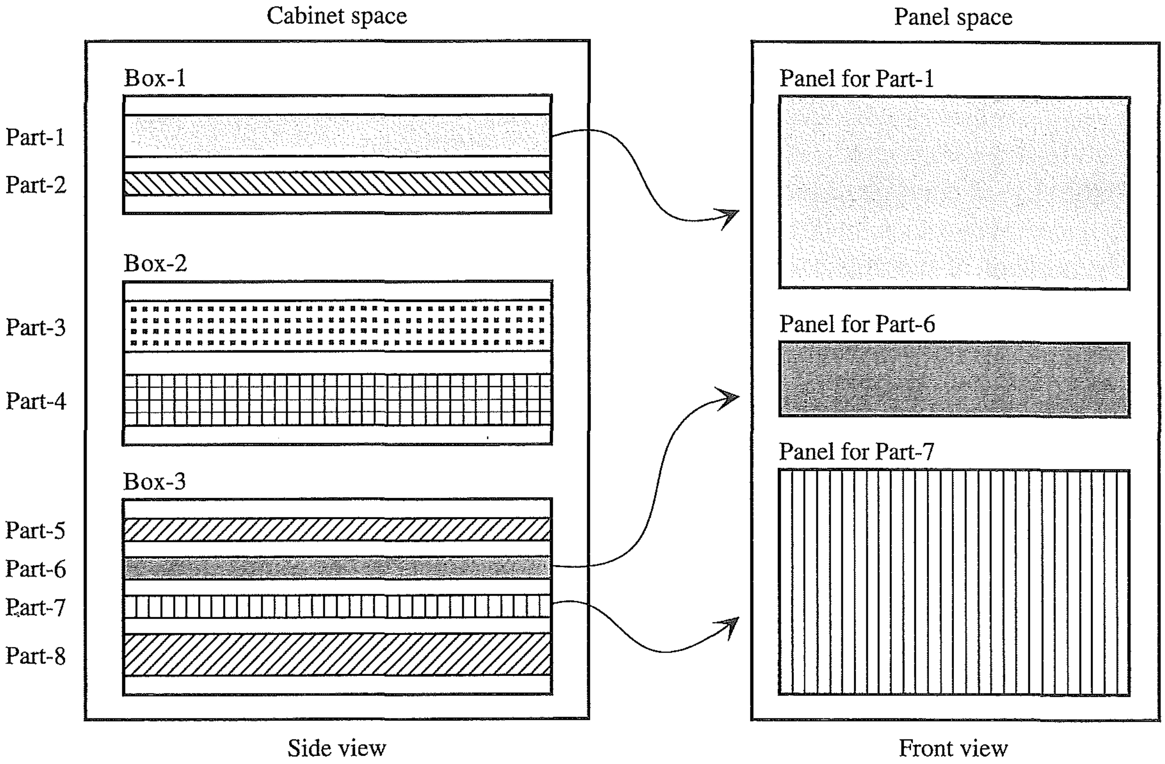


FIGURE 8.35. Another variation on models of arrangement for configuration problems. In this model, parts are arranged in boxes and boxes are arranged in cabinets. In addition, some parts require control panels and there is a limited amount of panel space. Panels for parts must be located on the cabinet that contains the parts. In this model, space in boxes, space in cabinets, and space for panels are all resources consumed during configuration. Because of the connections between parts and panels, the resource allocation processes are interdependent.

resource. Constraints on packing may also include requirements about certain parts being adjacent to one another.

All the variations in Figure 8.34 represent physical containment. **Physical-containment relations** form a hierarchy whose branches indicate which parts are physically contained in others. For example, a cabinet may contain a circuit board, which in turn may contain a processor and some memory. Contained parts do not necessarily carry out specializations of their container's function, nor are they necessarily required parts for their container.

COSSACK uses a port-and-connector model to account for the use of slots. It also keeps track of memory resources. XCON's arrangement models are the most complex in our example systems. Figure 8.35 illustrates another complication in arrangement models. In this example, there are three interlinked tasks in arrangement: the location of boards inside boxes, the location of boxes inside cabinets, and the location of panels on the front of cabinets. The tasks are interlinked because the control panels for particular boards need to be located on the cabinet containing the corresponding boards.

XCON also admits complex arrangement conditions and interactions with nonspatial resources. For example, a rule for allocating slots inside boxes says: "a box can accommodate five 4-slot backplanes; the exception is that a 9-slot backplane may not occupy the space reserved for the second and third 4-slot backplanes." This example combines electrical requirements with space allocation.

Knowledge about Decision Ordering and Interactions

The difficulty of configuration problems arises from a three-horned dilemma, where the three horns are combinatorics, horizon effects, and threshold effects. The combinatoric difficulty is that the number of possible configurations increases multiplicatively with the number of parts and arrangements. In the configuration domains discussed in this chapter, there are far too many possible configurations to instantiate them all in complete detail. Most decisions need to be made on the basis of partial information about the possible configurations. This is where the horizon effect comes in. At each stage of reasoning, some interactions are not yet visible either because not enough inferences have been made or because not enough commitments have been made. When information is partial, a natural approach is to rely on estimates. This is where the threshold effect comes in. A small difference in specifications or descriptions can lead to large, widespread changes. Because of their approximate nature, estimation functions can be off by small amounts. If these small errors are near critical thresholds, then the estimates may not be trustworthy.

To reiterate the argument in short form, there are too many configurations to develop them in detail. Making choices on the basis of partial information is subject to horizon effects. Estimates at the horizon based on partical information are subject to threshold effects.

The approaches that we consider for coping with these effects assume that there are predictable or typical patterns of interaction and that knowledge about these patterns is available. The following discussion is concerned with the dimensions of these patterns.

The first horn of the dilemma, combinatorics, is dealt with by two major techniques: abstraction and decision ordering. Thus, the use of functional abstractions in reasoning about configuration amounts to hierarchical search and has the usual beneficial effects. Controlling the order of decisions can reduce the amount of backtracking in search. In the context of configuration, knowledge-level justifications of decision ordering complement the symbol-level graph analyses. In XCON and VT, the reduction of backtracking is characterized in terms of dependent and independent decisions, where each decision is carried out only after completing other decisions on which it depends.

For example, XCON defers cabling decisions to its last subtask. Three assumptions rationalize this late placement of cabling decisions. The first assumption is that for any configuration, cabling is always possible between two components. This assumption ensures that the configuration process will not fail catastrophically due to an inability to run a cable. The second assumption is that all cables work satisfactorily, for any configuration. Thus, a computer system will work correctly no matter how long the cables are. The third assumption is that cable costs do not matter. Cable costs are so low when compared with active components that they will not influence an overall cost significantly. As long as these assumptions hold, XCON would never need to backtrack (even if it could!) from a cable decision and there is no need to consider cable issues any earlier in XCON's process. If there were some special cases where cables beyond a certain length were unavailable or would prevent system operations, a fix in terms of other component choices or arrangements would need to be determined and XCON would be augmented with look-ahead rules to anticipate that case.

Much of the point of the SALT analysis in the elevator domain is to identify patterns of interactions. VT's data-driven approach is a least-commitment approach and is analogous to constraint satisfaction methods that reason about which variable to assign values to next. Least commitment precludes guessing on some decisions when refinements on other decisions can be made without new commitments. Deadlock loops correspond to situations where no decision would be made. To handle cases where decisions are inherently intertwined so there is no ordering based on linear dependencies, SALT introduces reasoning loops in the parameter network that search through local preferences toward a global optimum.

The partial-commitment approach extends least commitment by abstracting what is common to the set of remaining alternatives and then using that abstraction to constrain other decisions. VT narrows values on some parameters, sometimes on independent grounds, so that components are partially specified at various stages of the configuration process.

In VT, such reasoning can amount to partial commitment to the extent that further inferences are drawn as the descriptions are narrowed.

All efforts to cope with the combinatorics of configuration encounter horizon effects. Since uniform farsightedness is too expensive, an alternative is to have limited but very focused farsightedness. Domain knowledge is used to anticipate and identify certain key interactions, which require a deeper analysis than is usual. This approach is used in XCON, when it employs its look-ahead rules to anticipate possible future decisions. The required conditions for look-ahead can be very complex. In XCON, heuristics are available for information gathering that can tell approximately whether particular commitments are feasible. The heuristics make look-ahead less expensive.

Finally, we come to techniques for coping with the threshold effect. The issue is not just that there are thresholds, but rather that the factors contributing to the threshold variables are widely distributed. Sometimes this distribution can be controlled by bringing together all the decisions that contribute. This helps to avoid later surprises. Knowledge about thresholds and contributing decisions can be used in guiding the ordering of decisions. In general, however, there is no single order that clusters all related decisions. One approach is to anticipate thresholds with estimators and to make conservative judgments. For example, one could anticipate the need for a larger cabinet. This conservative approach works against optimization. In XCON, many of the constraints whose satisfaction cannot be anticipated are soft. This means that although there may be violations, the softness of the constraints reduces the bad consequences. Thus, the extra part may require adding a cabinet, but at least it is possible to add new cabinets.

In summary, domain knowledge is the key to coping with the complexity of configuration. Knowledge about abstractions makes it possible to search hierarchically. Knowledge about key interactions makes it possible to focus the computations for look-ahead to specific points beyond the horizon. Knowledge about soft constraints reduces the effects of threshold violations.

8.4.2 *MCF-1: Expand and Arrange*

In this section and the following, we consider methods for configuration. In principle, we could begin with a method based on a simple generate-and-test approach. By now, however, it should be clear how to write such a method and also why it would be of little practical value. We begin with a slightly more complex method.

Figure 8.36 presents an extremely simple method for configuration, called MCF-1. MCF-1 acquires its specifications in terms of key components, expands the configuration to include all required parts, and then arranges the parts. Although this method is very simple it is also appropriate for some domains. MCF-1 is M1's method, albeit in skeletal form and leaving out the secondary cycle for reliability analysis and revision. It is practical for those applications where the selection decisions do not depend on the arrangement decisions.

MCF-1 has independent auxiliary methods get-requirements, get-best-parts, and arrange-parts. The auxiliary method get-requirements returns all the requirements for a part and returns the initial set of requirements given the token initial-specifications. Given a set of requirements, there may be different alternative candidate sets of parts for meeting a requirement. The method get-best-parts considers the candidate sets of new parts and employs a domain-specific evaluation function to select the best ones. Finally, given a list of parts, the method arrange-parts determines their best arrangement. A recursive subroutine, add-required-parts, adds all the parts required by a component.

MCF-1 relies on several properties of its configuration domain as follows:

- Specifications can be expressed in terms of a list of key components.
- There is a domain-specific evaluation function for choosing the best parts to meet requirements.
- Satisfactory part arrangements are always possible, given a set of components.
- Parts are not shared. Each requirement is filled using unique parts.

The first assumption is quite common in configuration tasks and has little bearing on the method. The second assumption says there is a way to select required parts without backtracking. An evaluation function is called from inside get-best-parts. The third assumption says that arrangement concerns for configurations involve no unsatisfiable constraints, even in combination. This is not unusual for configuration tasks that use a port-and-connector model for arrangements. In such domains, one can always find some arrangement and the arrangement does not affect the quality of the configuration. This assumption is crucial to the organization of MCF-1 because it makes it possible to first select all the required parts before arranging any of them. It implies further that the arrangement process never requires the addition of new parts. A somewhat weaker assumption is actually strong enough for a simple variation of the method: that part arrangement never introduces any significant parts that would force backtracking on selection or arrangement decisions. The last assumption means the system need not check whether an existing part can be reused for a second function. This method would need to be modified to accommodate multifunctional parts.

8.4.3 *MCF-2: Staged Subtasks with Look-Ahead*

Method MCF-1 is explicit about the order that configuration knowledge is used: first, key components are identified; then required parts are selected; and finally, parts are arranged. As discussed in our analysis of configuration, these decisions may need to be more tightly interwoven. Sometimes an analysis of a domain will reveal that certain groups of configuration decisions are tightly coupled and that some groups of decisions can be performed before others. When this is the case, the configuration process can be organized in terms of subtasks, where each subtask

To perform configuration using MCF-1:

```
/* Initialize and determine the key components from the
specifications. */
Set parts-list to nil.

/* Get-requirements returns specifications for required parts given
a part. */
Set requirements to get-requirements(initial-specifications).
Set key-components to get-best-parts(requirements).

/* Add all the required parts to the parts-list. */
For each key component do
begin
      Push the key component onto the parts-list.
      Add-required-parts(key-component).
end

/* Arrange parts in the configuration. */
Arrange parts using arrange-parts(parts-list).
Return the solution.


/* Recursive subroutine to add parts required by a given
part. When there are multiple candidates it selects the best one
get-best-parts. */
Add-required-parts(part)
begin
      Set requirements to get-requirements(part).
      If there are some requirements, then
      begin
            Set new-parts to get-best-parts(requirements).
            For each of the new-parts do
            begin
                  Push the new-part onto the parts-list.
                  Add-required-parts(new-part).
            end
      end
end
```

FIGURE 8.36. MCF-1, a simple method for configuration problems in which there is no interaction between selection and arrangement of required parts.

performs a combination of closely related decisions to refine, select, and arrange components. Subtasks cluster decisions so that most interactions among decisions within a subtask are much greater than interactions with decisions outside of it.

In some domains the partitioning of decisions into subtask clusters does not necessarily yield a decision ordering in which all later decisions depend only on early ones. To restate this in constraint satisfaction terms, there may be no ordering of variables such that values can be assigned to all variables in order without having to backtrack. To address this, knowledge for selective look-ahead is used to anticipate interactions. Such knowledge is domain specific and may involve conservative estimates or approximations that anticipate possible threshold effects. This approach can be satisfactory even in cases where the look-ahead is not foolproof, if the failures involve soft constraints.

This approach is followed in MCF-2. The method description given in Figure 8.37 says very little about configuration in general. Nonetheless, MCF-2 is the method used by XCON, albeit with its domain-specific particulars abstracted.

To perform configuration using MCF-2:

```
/* Initialize and get the key components from the specifications. */
Set parts-list to nil.
Set requirements to get-requirements(initial-specifications).
Set key-components to get-best-parts(requirements).

/* Conditionally invoke Subtasks. */
While there are pending subtasks do
begin
        Test conditionals for invoking subtasks and choose the best
        one.
        Invoke a specialized method for the subtask.
end
Return the solution.

/* Each specialized method performs a subset of the task. It is
responsible for refining some specifications, adding some parts, and
arranging some parts. It incorporates whatever look-ahead is needed.
*/
Method-for-Subtask-1:

/* Subtask-1: Look ahead as needed and expand some parts. */
/* Subtask-2: Look ahead as needed and arrange some parts. */

Return.
...
Method-for-Stage-3:
...
```

FIGURE 8.37. MCF-2, a method for configuration problems in which part selection and arrangement decisions can be organized in a fixed sequence of subtasks. The method depends crucially on the properties of the domain. This method tells us very little about configuration in general.

This method is vague. We might caricature it as advising us to "write a configuration system as a smart program using modularity as needed" or as "an agenda interpreter that applies subtasks in a best-first order." Such symbol-level characterizations miss the point that the decision structure follows from an analysis of relations at the knowledge level. This observation is similar to the case of methods for classification. The particulars of the domain knowledge make all the difference. The burden in using MCF-2 is in analyzing a particular domain, identifying common patterns of interaction, and partitioning the decisions into subtasks that select and arrange parts while employing suitable look-ahead.

It is interesting to compare MCF-2 to the general purpose constraint satisfaction methods. One important result is that when the variables of a constraint satisfaction problem (CSP) are block ordered, it is possible to find a solution to the CSP with backtracking limited to the size of the largest block, which is depth-limited backtracking. Although there are important differences between the discrete CSP problems and the configuration problems, the basic idea is quite similar. Subtasks correspond roughly to blocks. When subtasks are solved in the right order, little or no backtracking is needed. (See the exercises for a more complete discussion of this analogy.)

As in MCF-1, MCF-2 does not specify an arrangement submodel, a part submodel, or a sharing submodel. Each domain needs its specially tailored submodels. In summary, MCF-2 employs specialized methods for the subtasks to do special-case look-ahead, to compensate for known cases where the fixed order of decision rules fails to anticipate some crucial interaction leading to a failure. The hard work in using this method is in the analysis of the domain to partition it into appropriate subtasks.

8.4.4 *MCF-3: Propose-and-Revise*

In MCF-2, there are no provisions in the bookkeeping for carrying alternative solutions, or for backtracking when previously unnoticed conflicts become evident. Figure 8.38 gives MCF-3, which is based loosely on the mechanisms used in VT, using the propose-and-revise approach. Like MCF-2, we can view MCF-3 as an outline for an interpreter of knowledge about configuration. It is loosely based on the ideas of VT and the interpretation of a parameter network. It depends crucially on the structure of the partial configuration and the arrangement of knowledge for proposing values, proposing constraints, noticing constraint violations, and making fixes.

The beauty of MCF-3 is that the system makes whatever configuration decisions it can by following the opportunities noticed by its data-driven interpreter. In comparison with MCF-2, MCF-3 is does not rely so much on heuristic look-ahead, but has provisions for noticing conflicts, backtracking, and revising.

8.4.5 *Summary and Review*

This section sketched several different methods for configuration, based loosely on the example applications we considered earlier. We started with a method that performed all selection decisions before any arrangement decisions. Although this is practical for some configuration tasks, it is not adequate when arrangements are difficult or are an important determinant of costs. The second method relied on look-ahead. Our third method triggered subtasks according to knowl-

To perform configuration using MCF-3:

```
/* Initialize and obtain requirements. */
Initialize the list of parts to empty.
Set requirements to get-requirements(initial-specifications).
Set key-components to get-best-parts(requirements).

/* Apply the domain knowledge to extend and revise the configuration.
*/
While there are open decisions and failure has not been signaled do
begin
      Select the next-node in the partial configuration for which a
      decision can be made.
      If the decision is a design extension, then invoke method
      propose-design-extensions(next-node).
      If the decision is to post a constraint, then invoke method
      post-design-constraints().
      If there are violated-constraints, then invoke revise-design().
end
Report the solutions (or failure).
```

FIGURE 8.38. MCF-3, a method based on the propose-and-revise model.

edge about when the subtasks were ready to be applied and used constraints to test for violations when choices mattered for more than one subtask.

Exercises for Section 8.4

Ex. 1 [*10*] *MYCIN's Method.* How does MYCIN's method for therapy recommendation fit into the set of methods discussed in this section? Is it the same as one of them? Explain briefly.

■ **Ex. 2** [*R*] *Subtask Ordering.* After studying the XCON system and reading about constraint satisfaction methods, Professor Digit called his students together. "Eureka!" he said. "MCF-2 is really a special case of an approach to constraint satisfaction that we already know: block ordering. In the future, pay no attention to MCF-2. Just make a tree of 'constraint blocks' corresponding to the configuration subtasks and solve the tasks in a depth-first order of the tree of blocks."

(a) Briefly sketch the important similarities and differences between CSP problems and configuration problems.

(b) Briefly, is there any merit to Professor Digit's proposal and his suggestion that there is a relation between traversing a tree of blocks in a CSP problem and solving a set of staged subtasks in a configuration problem?

(c) Briefly relate the maximum depth of backtracking necessary in a CSP to the "no backtracking" goal of systems like XCON.

Ex. 3 [*10*] *Methods and Special Cases.* The methods for configuration are all simple, in that the complexity of configuration tasks is manifest in the domain-specific knowledge rather than in the methods. The methods, however, form a progression of sorts, in that they can be seen as providing increased flexibility.

(a) In what way is MCF-1 a special case of MCF-2?

(b) In what way is MCF-2 a special case of MCF-3?

(c) In what way could MCF-3 be generalized to use other techniques from the example systems in this chapter?

8.5 *Open Issues and Quandaries*

In the 1970s, folk wisdom about expert systems said that they might be developed routinely for "analytic" tasks but not "synthetic" tasks, which were too difficult. Analytic tasks were characterized in terms of feature recognition. Medical diagnosis was cited as an example of an analytic task. In contrast, synthetic tasks reasoned about how to put things together. Design was said to be an example of a synthetic task.

In hindsight, this dichotomy is too simplistic. Some problem-solving methods combine aspects of synthesis and analysis, to wit: "synthesis by analysis" and "analysis by synthesis." Large tasks in practical domains are made up of smaller tasks, each of which may have its own methods. In the next chapter we will see that even methods for diagnosis require the combination of diagnostic hypotheses.

However, the main point for our purposes is that synthetic tasks tend to have high combinatorics and require a large and complex body of knowledge. This made them unsuitable for the first generation of knowledge systems, which performed simpler tasks mostly based on classification. Design tasks are still more often associated with research projects than with practical applications. Ambitious knowledge systems for design tasks tend to be doctoral thesis projects.

Configuration tasks specify how to assemble parts from a predefined set. It could be argued that the "real" difficulty in configuration tasks is setting up the families of components so that configuration is possible at all. For computer configuration, this involves designing the bus structures, the board configurations, the bus protocols, and so on. If these standards were not established, there would be no way to plug different options into the same slot. There would be no hope of creating a simple model of functionality around a key-component assumption. Behind the knowledge of a configurable system, there is a much larger body of knowledge about part reusability in families of systems. This brings us back to the challenges of design.

The goal in both configuration and design is to specify a manufacturable artifact that satisfies requirements involving such matters as functionality and cost. Design can be open-ended. Will the truck ride well on a bumpy road? What load can it carry? What is its gas mileage? Can the engine be serviced conveniently? There is an open-ended world of knowledge about materials, combustion, glues, assembly, manufacturing tools, markets, and other matters.

Human organizations have responded to the complexity of designing high-technology products by bringing together specialists with different training. The design of a xerographic copier involves mechanical engineers concerned with systems for moving paper, electronical engineers concerned with electronic subsystems, and computer engineers concerned with internal software. But this is just a beginning. Other people specialize in particular kinds of systems for transporting paper. Some people specialize in printing materials. Some specialize in the design of user interfaces. Some specialize in different manufacturing processes, such as plastics, sheet metal, and semiconductors. Others specialize in servicing copiers in the field.

As the number of issues and specializations increase, there are real challenges in managing and coordinating the activities and in bringing appropriate knowledge to bear. Competition drives companies to find ways to reduce costs and to speed the time to bring products to market. In this context, a large fraction of the cost of producing a product is determined by its design. Part of this cost is the design process itself, amortized over the number of products made. But the main point is about the time of decisions. Many decisions about a product influence the costs of its manufacture and service. What size engine does the truck have? Does it run on diesel or gasoline? Can the battery be replaced without unbolting the engine? How much room is in the cab? How many trucks will be sold? Many of these decisions are made at early stages of design and cannot be changed later, thus locking in major determinants of the product cost.

To reduce costs we must understand them at earlier stages of design. Thus, products are designed for manufacturing, designed for flexibility, designed for portability, or designed for servicing. All these examples of "design for *X*" attempt to bring to bear knowledge about *X* at an early stage of design. In design organizations, this has led to the creation of design teams and a practice called "simultaneous engineering." The idea is to bring together specialists representing different concerns, and to have them participate all the way through the design process.

There are no knowledge systems where you "push a button to design a truck." The challenges for acquiring the appropriate knowledge base for such ambitious automation are staggering. Instead, the response to shortening product development times involves other measures. Many of these involve ways for moving design work online. Databases and parts catalogs are online. Simulation systems sometimes replace shops for prototyping. The controls for manufacturing equipment are becoming accessible through computer networks. Reflecting the reality that much of an engineer's day is spent communicating with colleagues, another point of leverage is more advanced technology for communicating with others. This ranges from facsimile machines and electronic mail, to online computer files, to devices that enable teams of engineers to share a "digital workspace."

Systems to support design need not automate the entire process but can facilitate human design processes, simulate product performance and manufacturing, and automate routine subtasks. The elements of new design systems include knowledge systems, online catalogs, collaboration technology, simulation tools, and visualization tools. These elements reflect a move toward shared digital workspaces. As the work practice and data of design organizations become more online, many different niches appear where knowledge systems can be used to assist and automate parts of the process.

Automation advances as it becomes practical to formalize particular bodies of knowledge. Inventory and catalog systems now connect with drafting systems to facilitate the reuse of manufactured parts. Drafting systems now use constraint models and parameterized designs to relate sizing information from one part of the system with sizing information from other parts. In design tasks such as the design of paper paths using pinch roller technology (Mittal, Dym, & Morjaria, 1986), it has been practical to systematize a body of knowledge for performing large parts of the task.

Conventional tools for computer-aided design (CAD) tend to be one of two kinds: analysis tools or drafting tools. An example of an analysis tool is one for predicting the effects of vibrations on structure using finite element analysis. An example of a drafting tool is a tool for producing wire-frame models of solids. Knowledge-systems concepts are beginning to find their way into CAD systems, joining the ranks of graphics programs, simulation systems, and visualization systems. As computer technology for computer graphics and parallel processing have become available, there has been increased development of tools to help designers visualize products and manufacturing, by simulating those steps and showing the results graphically.

In summary, configuration tasks are a relatively simple subset of design tasks. They are more difficult because specifications can be more open-ended and design can involve fashioning and machining of parts rather than just selection of them. Human organizations have responded to the complexities of modern product design by engaging people who specialize in many different concerns. This specialization has also made it more difficult to coordinate design projects and to get designs to market quickly.