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

Contents

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.

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.

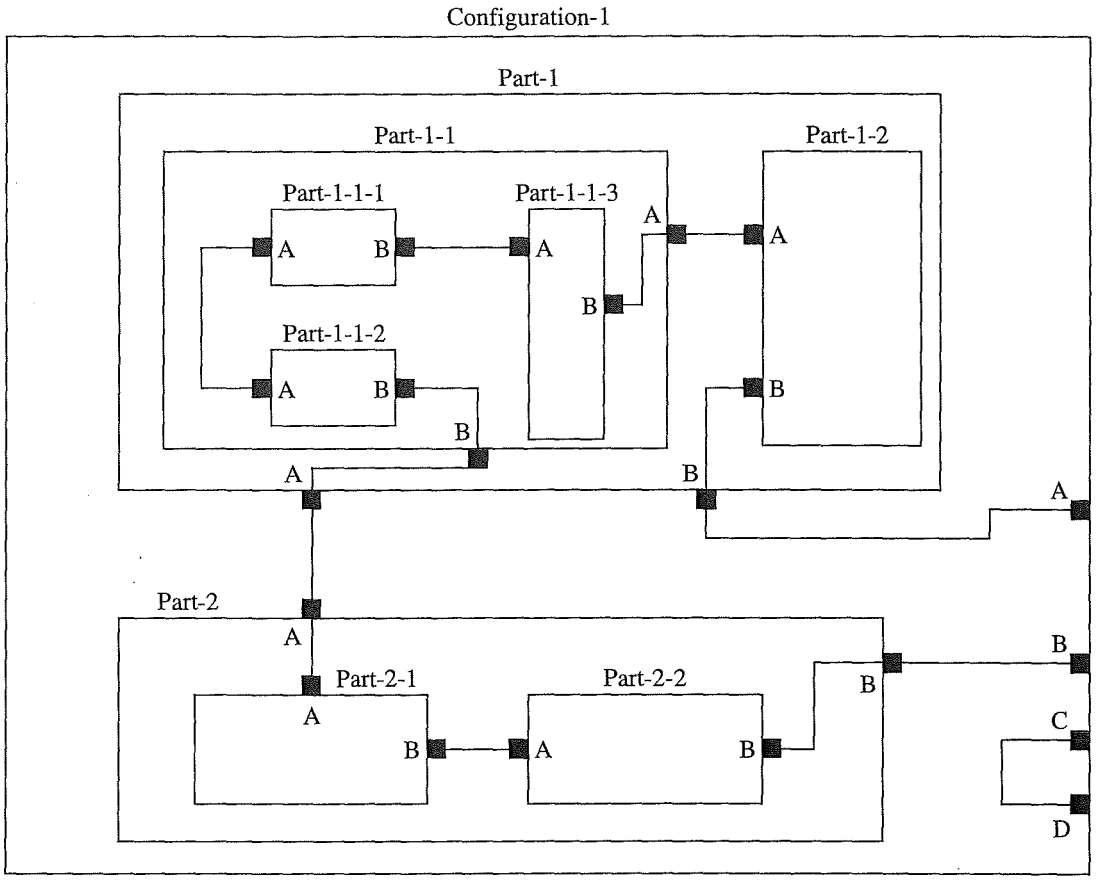

FIGURE 8.1. An example of a configuration using a port-and-connector model. In this figure, all components are shown as boxes and the top-level component is Configuration-1. It has two parts, Part-1 and Part-2. The arrangement of the configuration is indicated by the part-of hierarchy, displayed graphically 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

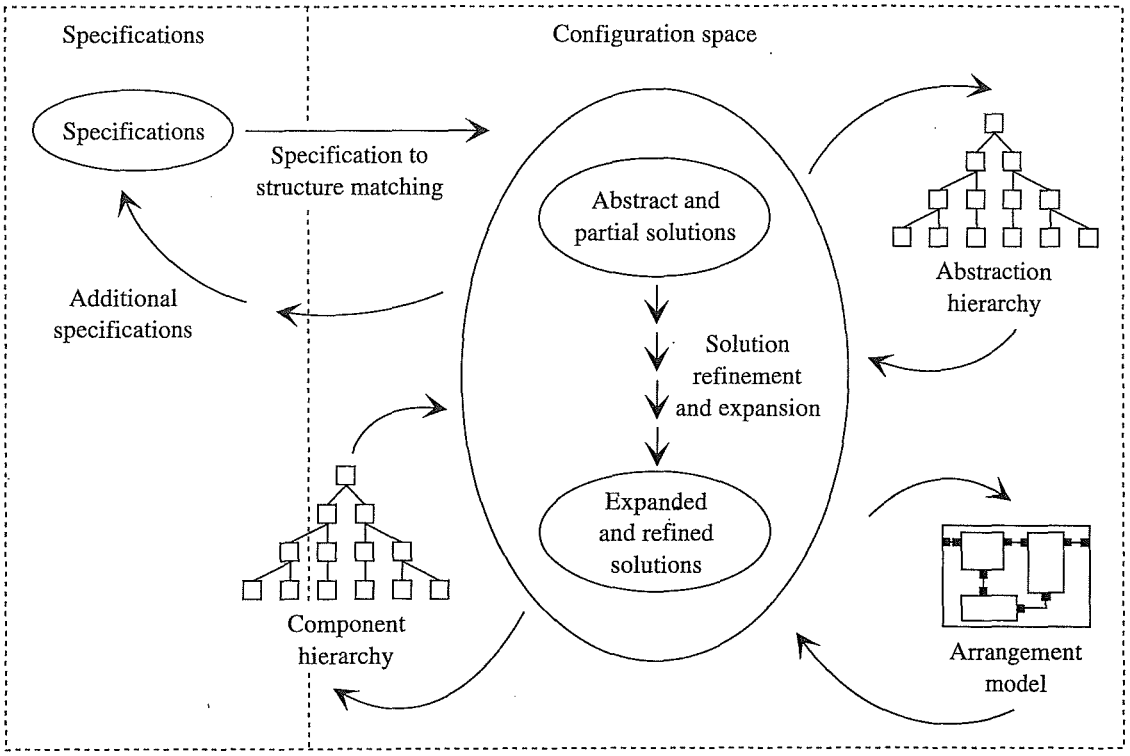

FIGURE 8.2. Spaces in configuration tasks.

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.

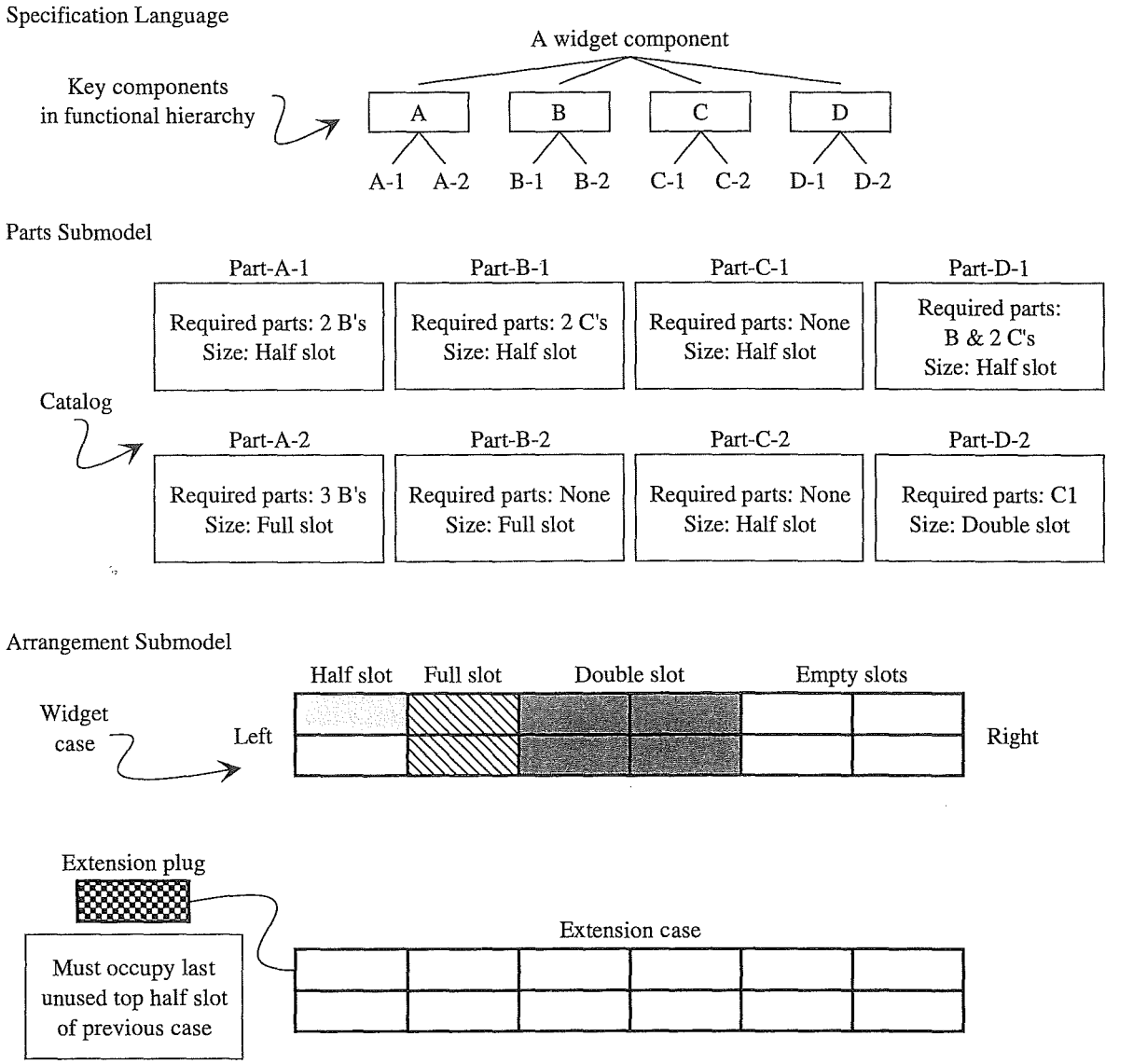

FIGURE 8.3. The widget model, W-1, illustrates phenomena in configuration problems.

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 ar-

rangement, 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.

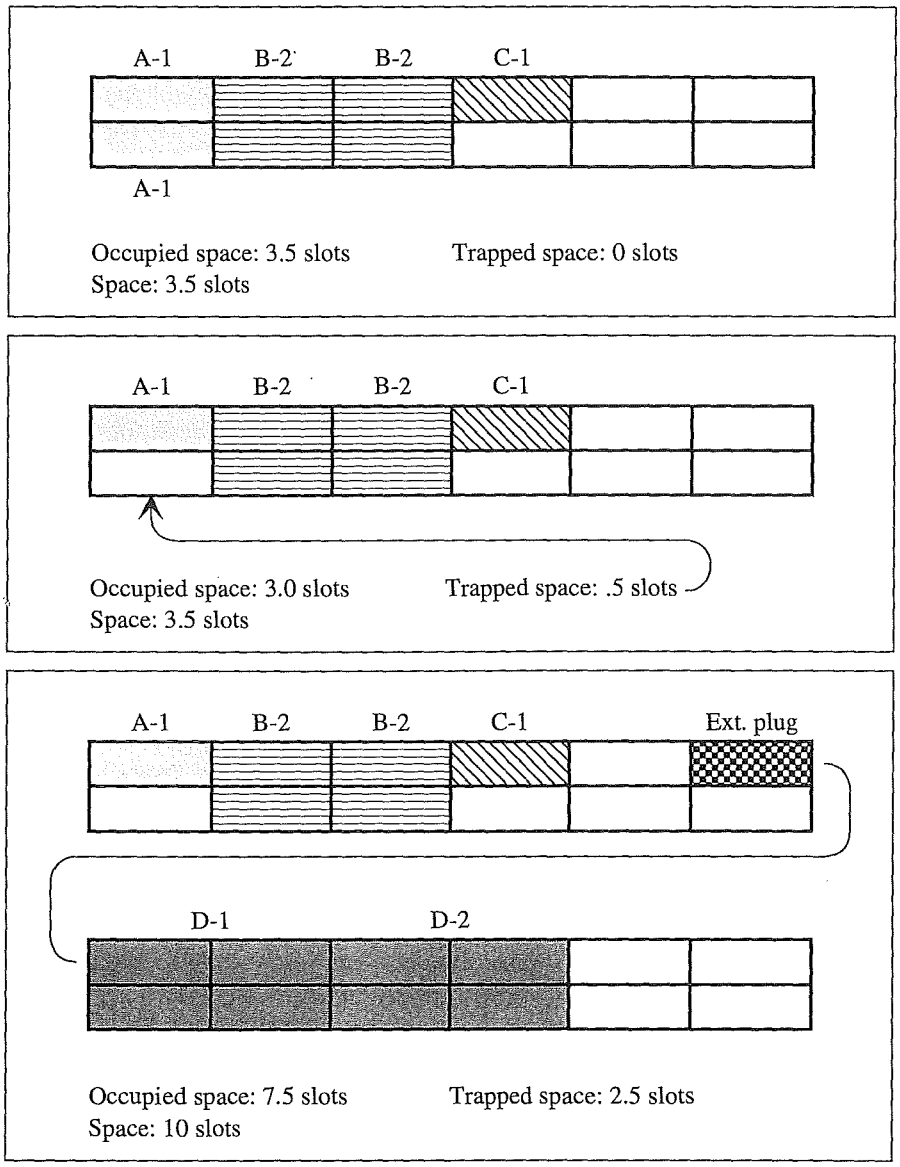

FIGURE 8.5. Examples illustrating the terms total space, occupied space, and trapped space. For most problems, the key factor is the total space, which is called simply "space."

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-

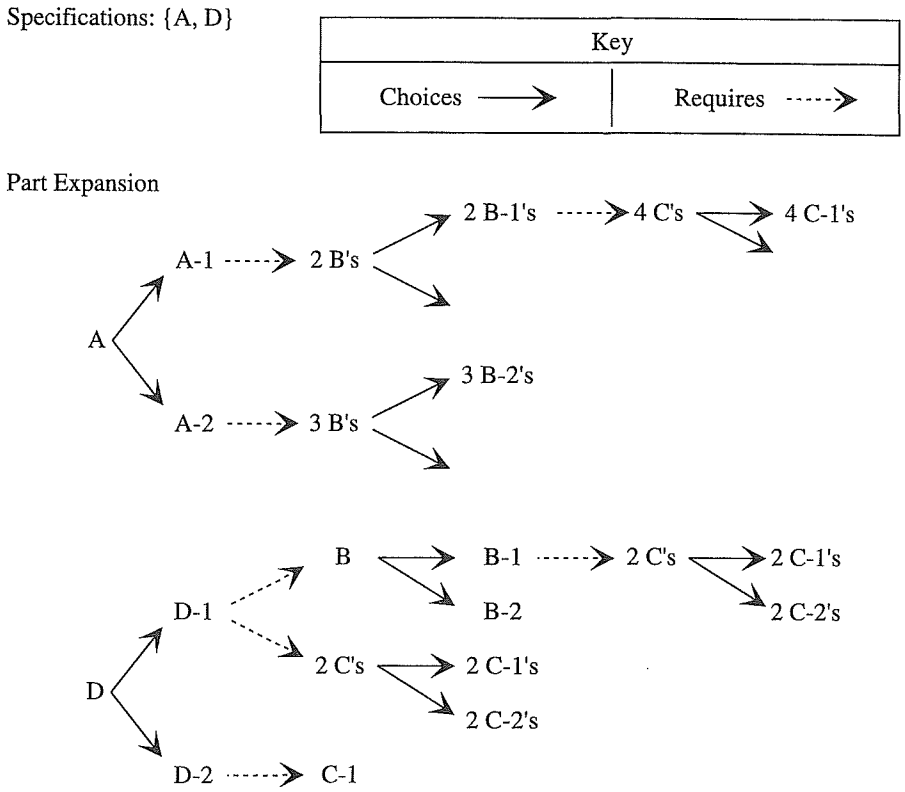

FIGURE 8.6. The dynamic nature of configuration problems. In this example, the initial specifications are {A, D}. The solid arrows indicate where there are alternatives for implementing a part. The dashed arrows indicate places where a part requires further parts. There are several possible solutions 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

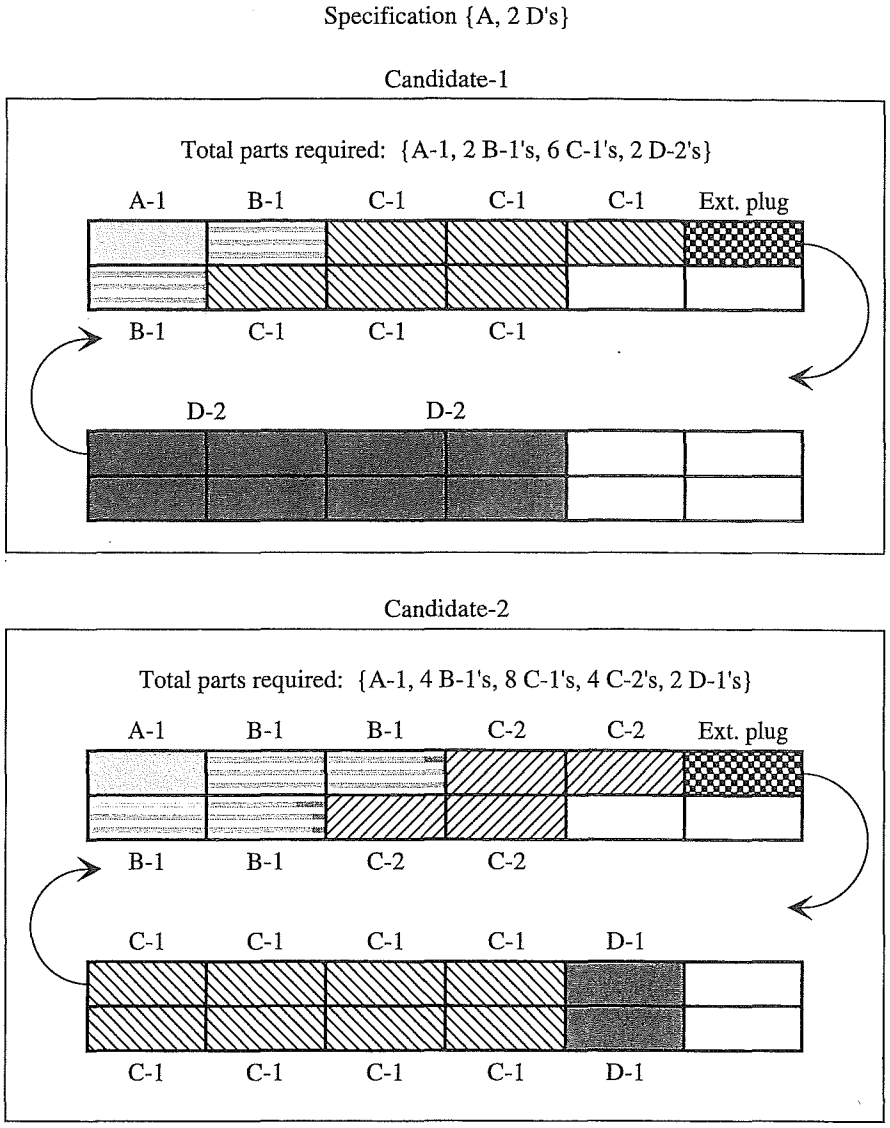

FIGURE 8.7. Threshold effects. This figure illustrates two possible solutions to a widget configuration problem with the specifications {A, 2 D's}. Both candidate solutions exceed the capacity of a single widget case. In more involved examples, exceeding one threshold can lead to a ripple effect of exceeding 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 sat-

isfy 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-

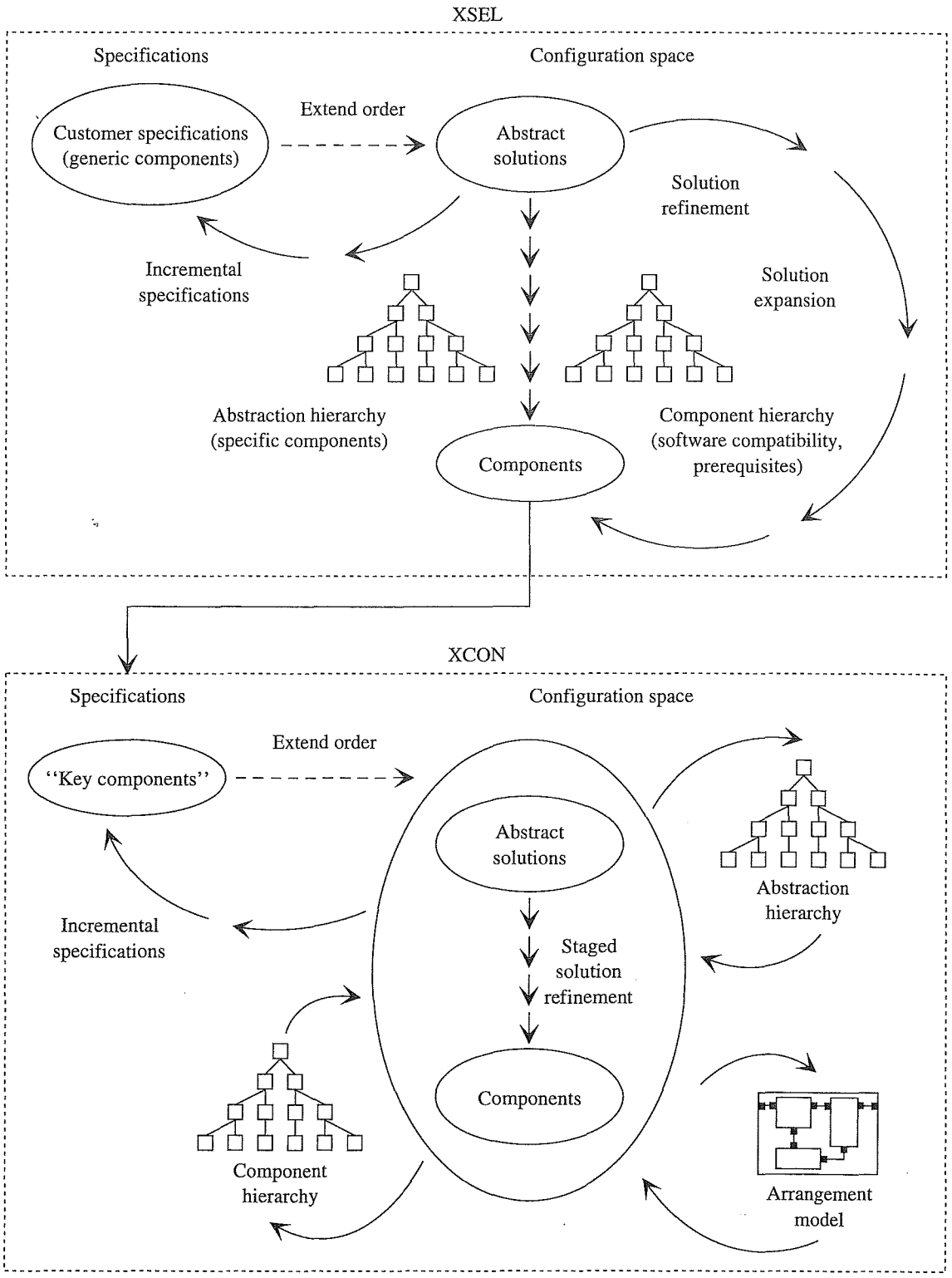

FIGURE 8.8. The pattern of knowledge use for XCON and XSEL. The systems split the responsibility, with XSEL performing the first steps and XCON performing a more detailed job. XSEL is concerned mainly with the completeness of a customer's order with regard to major system components ("key components") 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

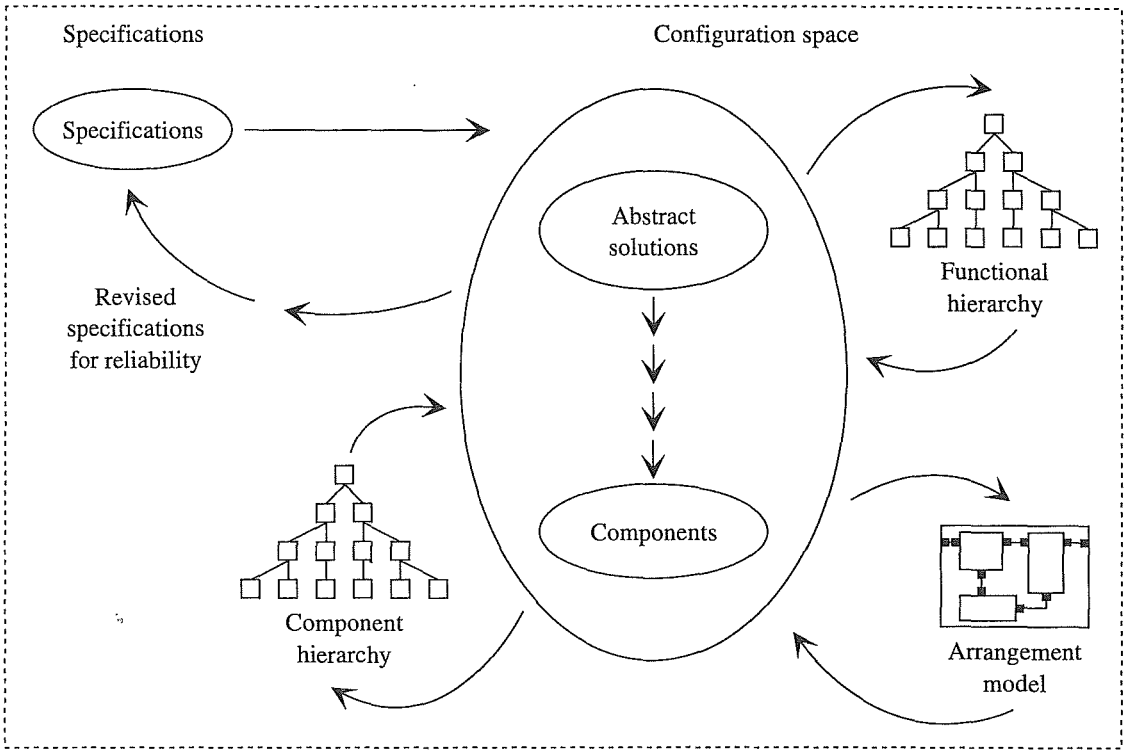

FIGURE 8.15. The specification language for M1 is a list of key components, using M1's functional hierarchy. M1 also has a component hierarchy that describes required parts, which are supporting circuitry. Its arrangement model is based on ports and connections. Not shown in this figure is M1's knowledge 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,

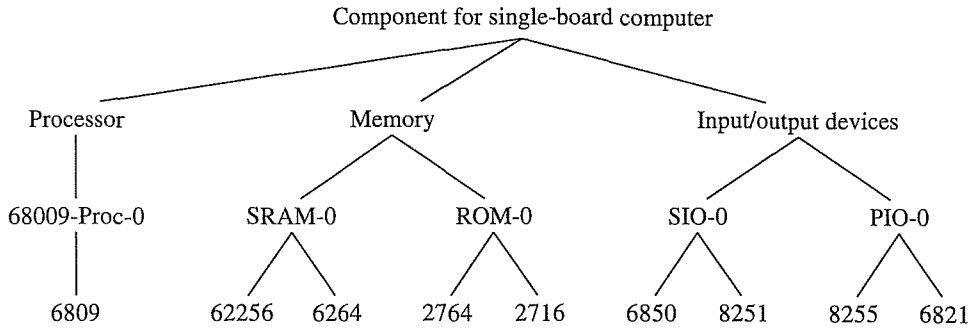

FIGURE 8.16. A functional hierarchy used in the M1 system. Some parts in this hierarchy represent abstract parts. These parts are developed by generalizing the functions of a class of physical parts. For example, the abstract part PIO-0 embodies a set of functions common to all parallel input/output 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

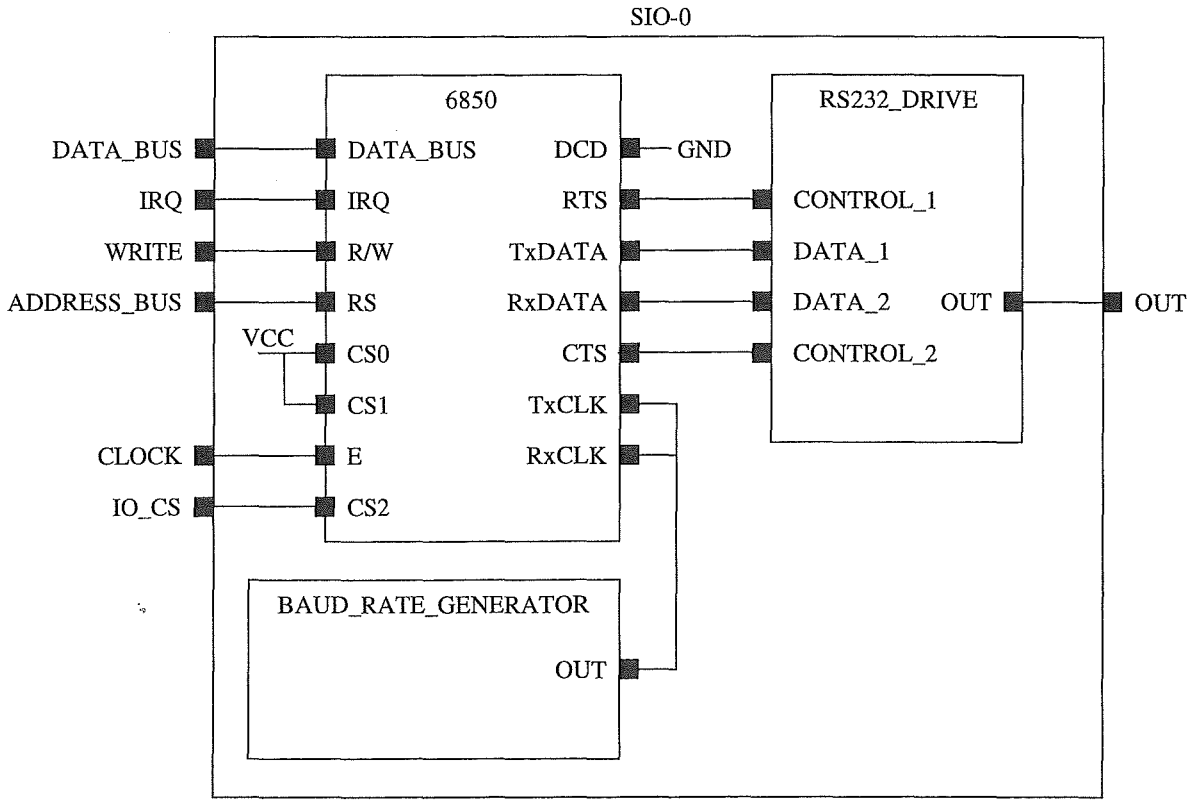

FIGURE 8.17. M1 uses templates and a port-and-connector model for describing the space of possible interconnections. The template in this figure is for the serial input/output device SIO-0. (Adapted from Birmingham, Brennan, Gupta, & Sieworek, 1988, Figure 4, page 37)

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

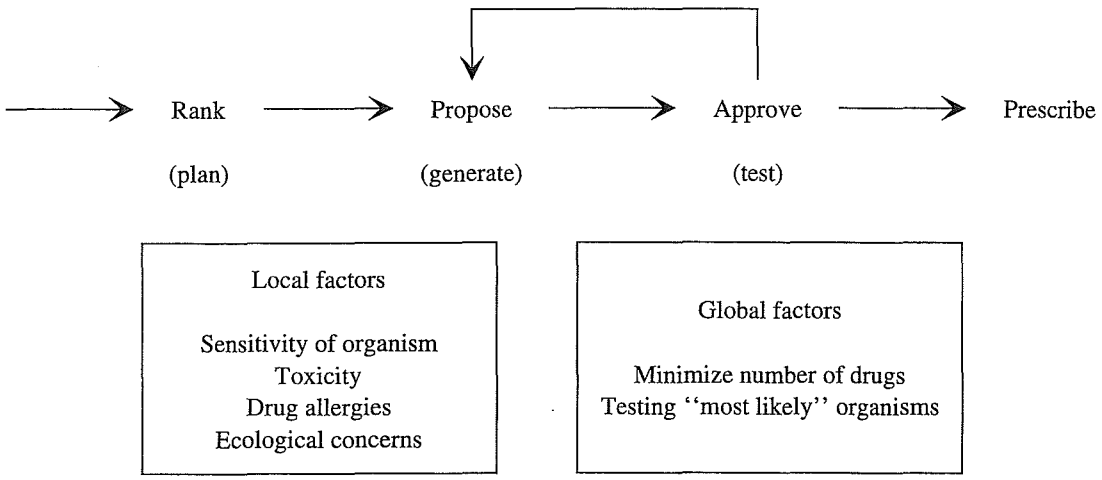

FIGURE 8.21. Therapy recommendation viewed as a plan, generate, and test process. (Adapted from Clancey, 1984, Figure 6-1, page 135.)

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.

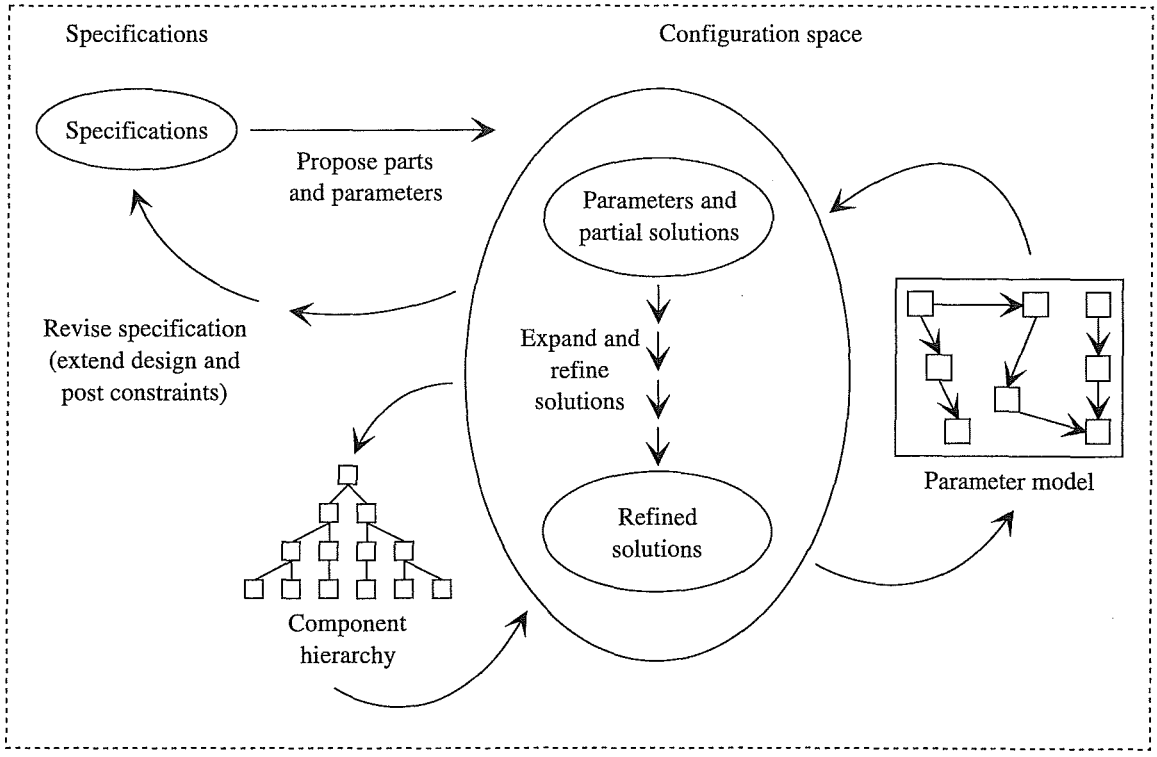

FIGURE 8.22. VT uses a propose-and-refine approach.

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.

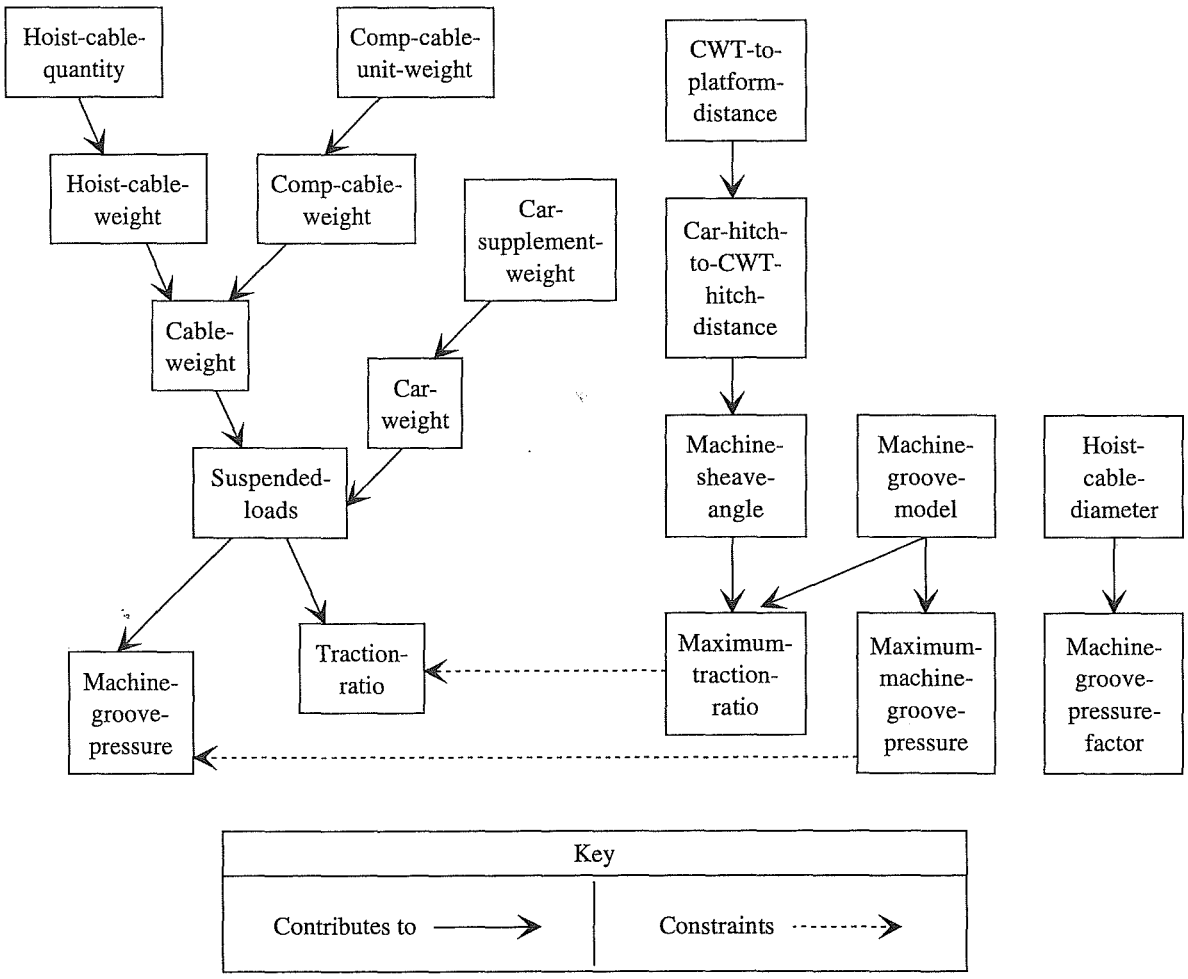

FIGURE 8.23. Segment of VT's knowledge base showing relations among parameters. Each box represents one of VT's design parameters. Arrows between the boxes indicate a partial order for determining values for parameters. (Adapted from Marcus, 1988, Figure 4-5, page 102.)

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.

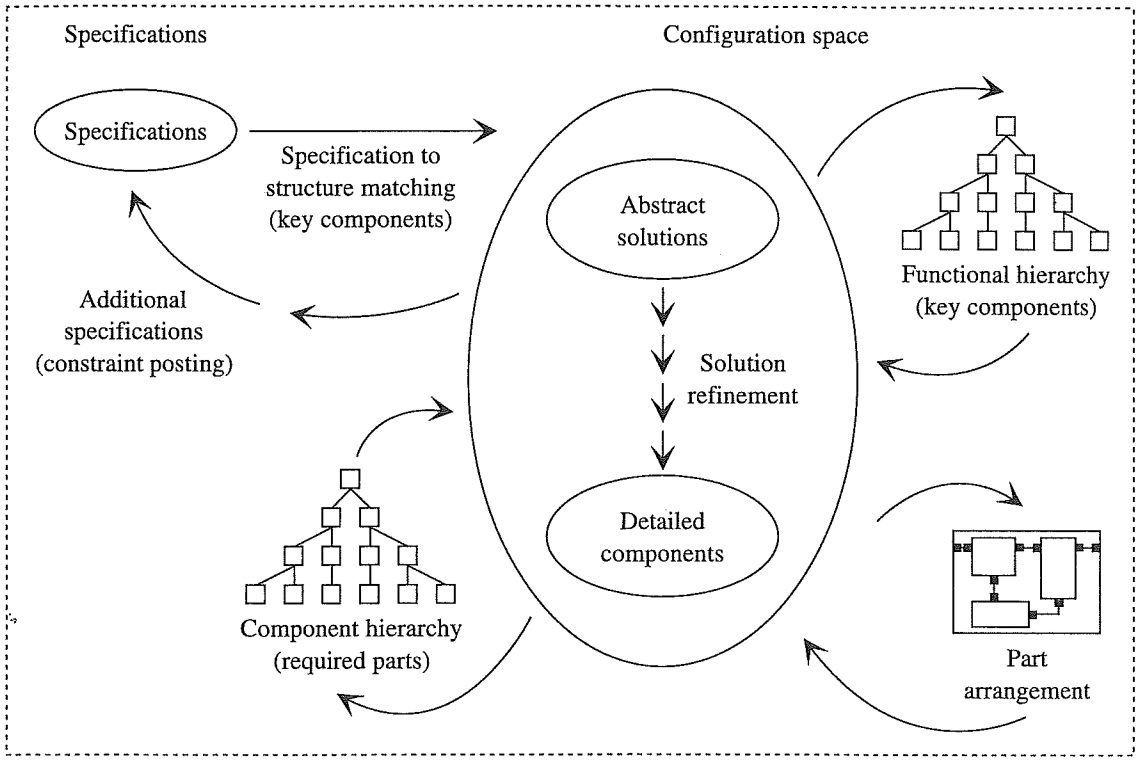

FIGURE 8.28. COSSACK begins with user specifications, in terms of evaluation criteria and functional specifications. The evaluation criteria are used to order candidates in component selection. Functionality specifications are represented using a key-component approach, where key components may be either 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-

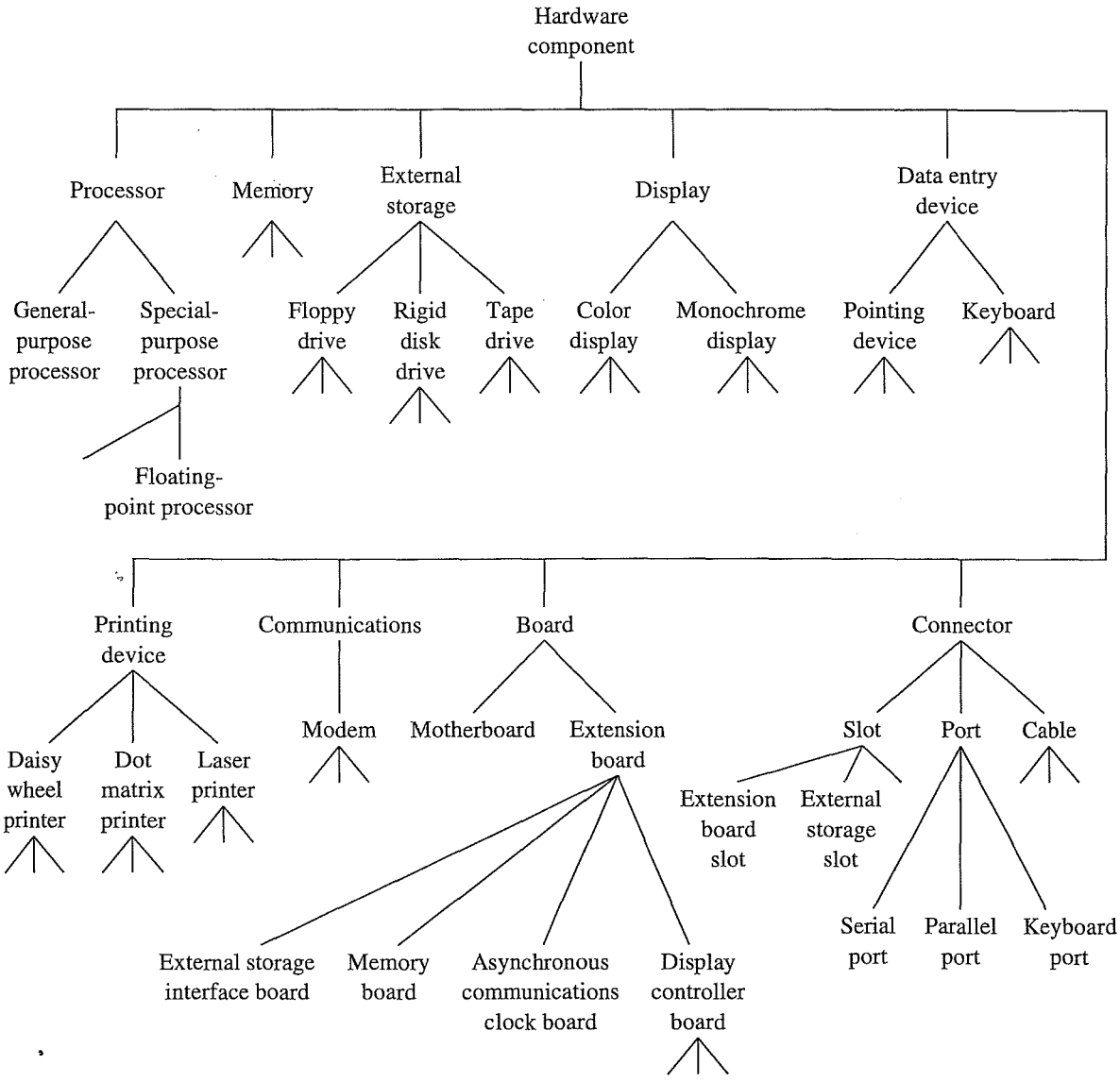

FIGURE 8.29. Portion of the functional hierarchy for components in COSSACK. Any component below the root can be a key component. COSSACK also considers software components in its configuration and has classes for word processors, accounting programs, spreadsheets, and operating systems.

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

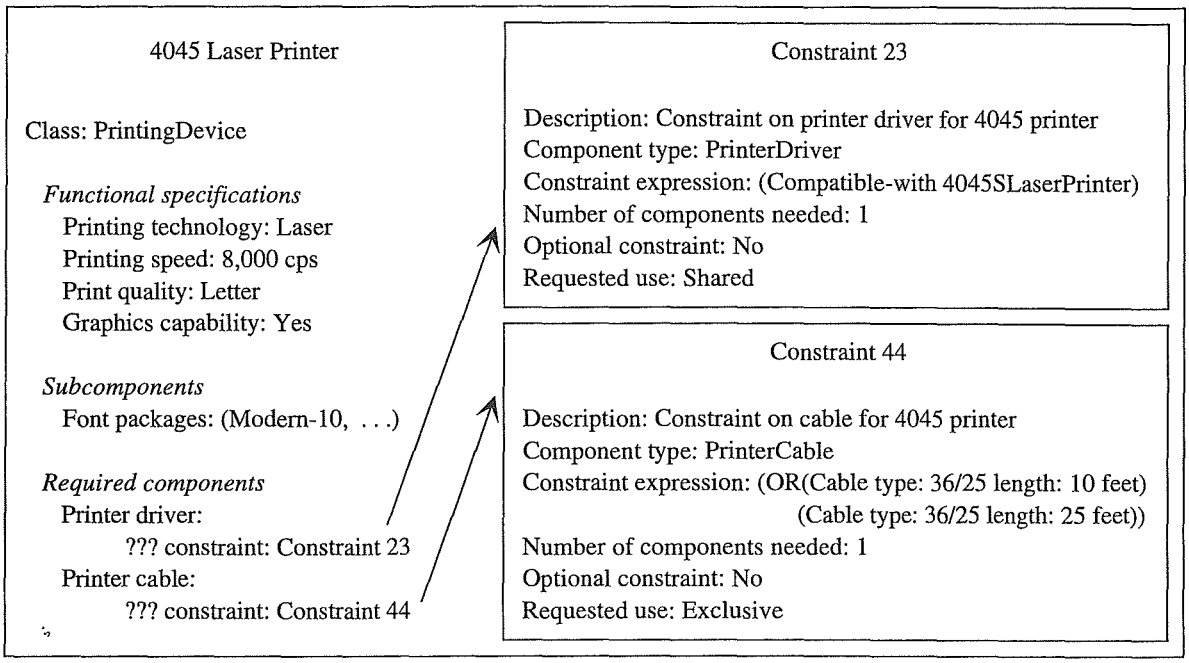

FIGURE 8.30. Frame representation of a component in COSSACK's functional hierarchy. This frame represents a 4045 laser printer. Frame slots are organized into groups: functional specifications, subcomponents, required components, and processing information. The functional specifications are used when 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

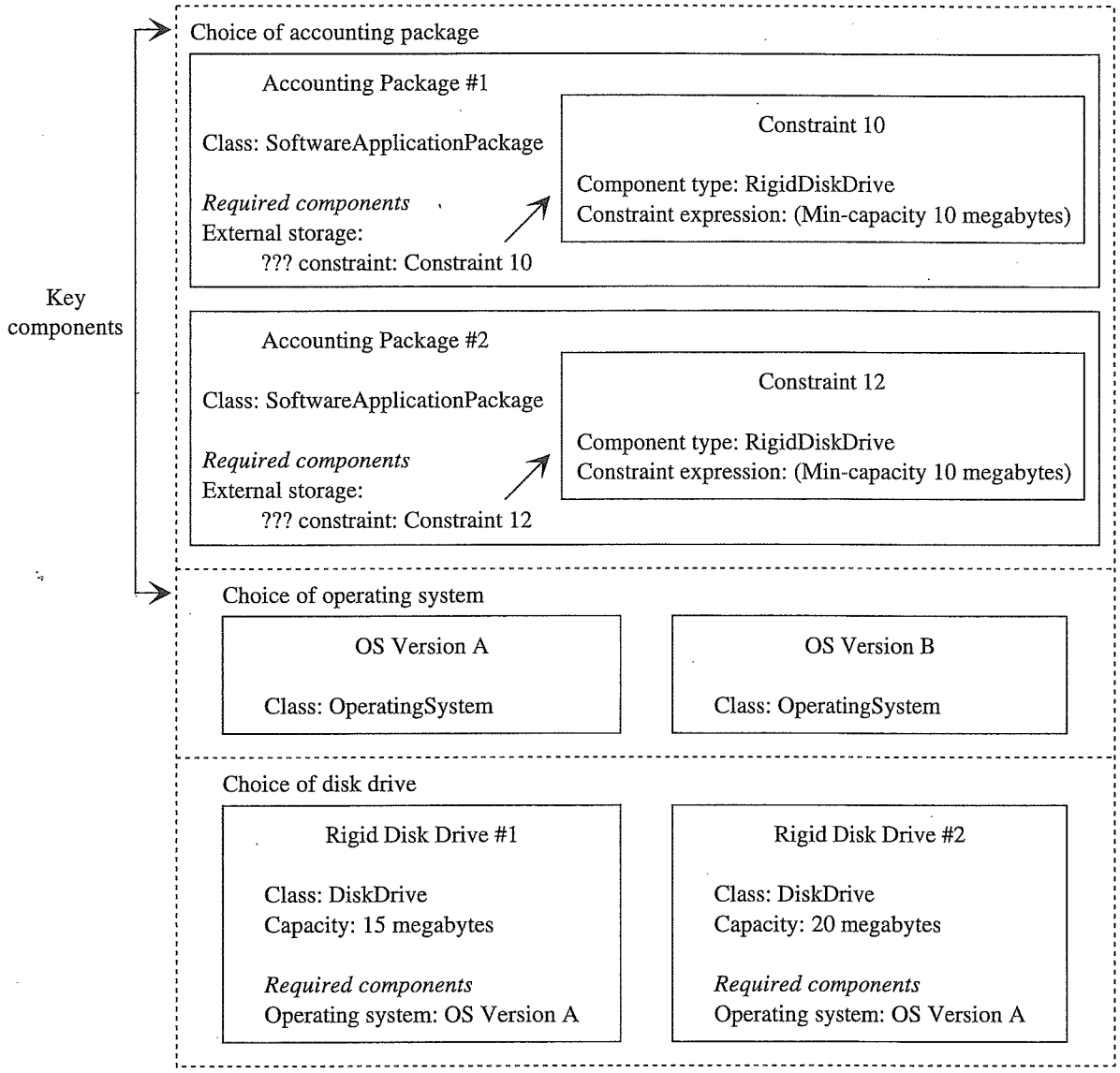

FIGURE 8.31. Simplified partial commitment example from COSSACK. In this example, we assume that an accounting package and an operating system have been specified as key components. COSSACK notices that whatever rigid disk drive it chooses, the required operating system is OS Version A.

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.

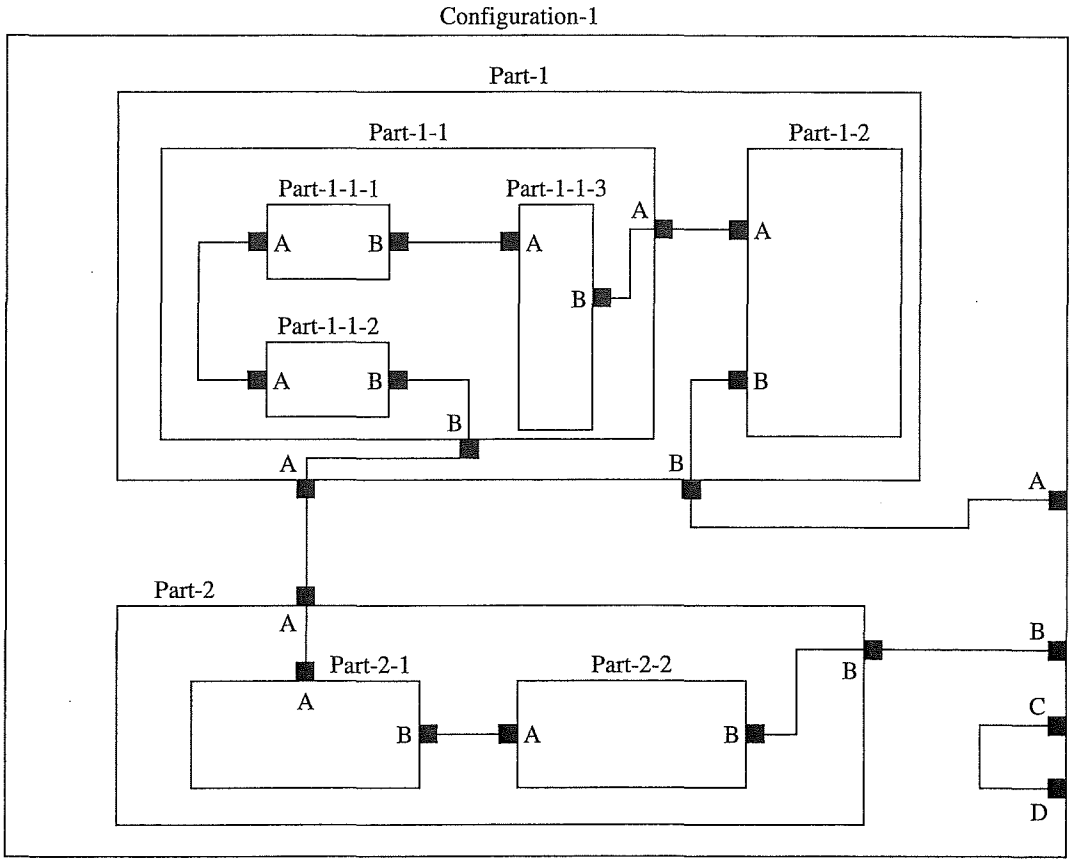

FIGURE 8.32. Ports and connectors.

(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-

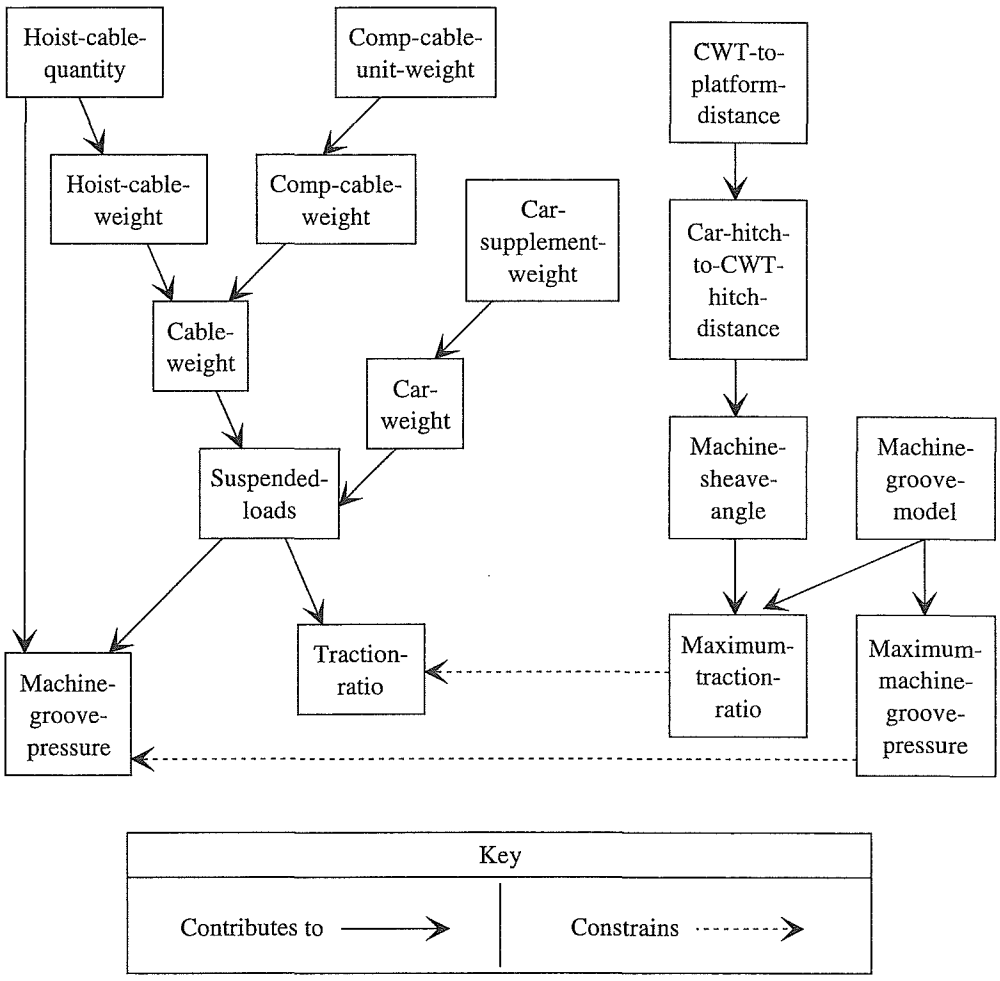

FIGURE 8.33. Interactions between antagonistic subproblems. (Adapted from Marcus & McDermott, 1989, Figure 5, page 18.)

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-

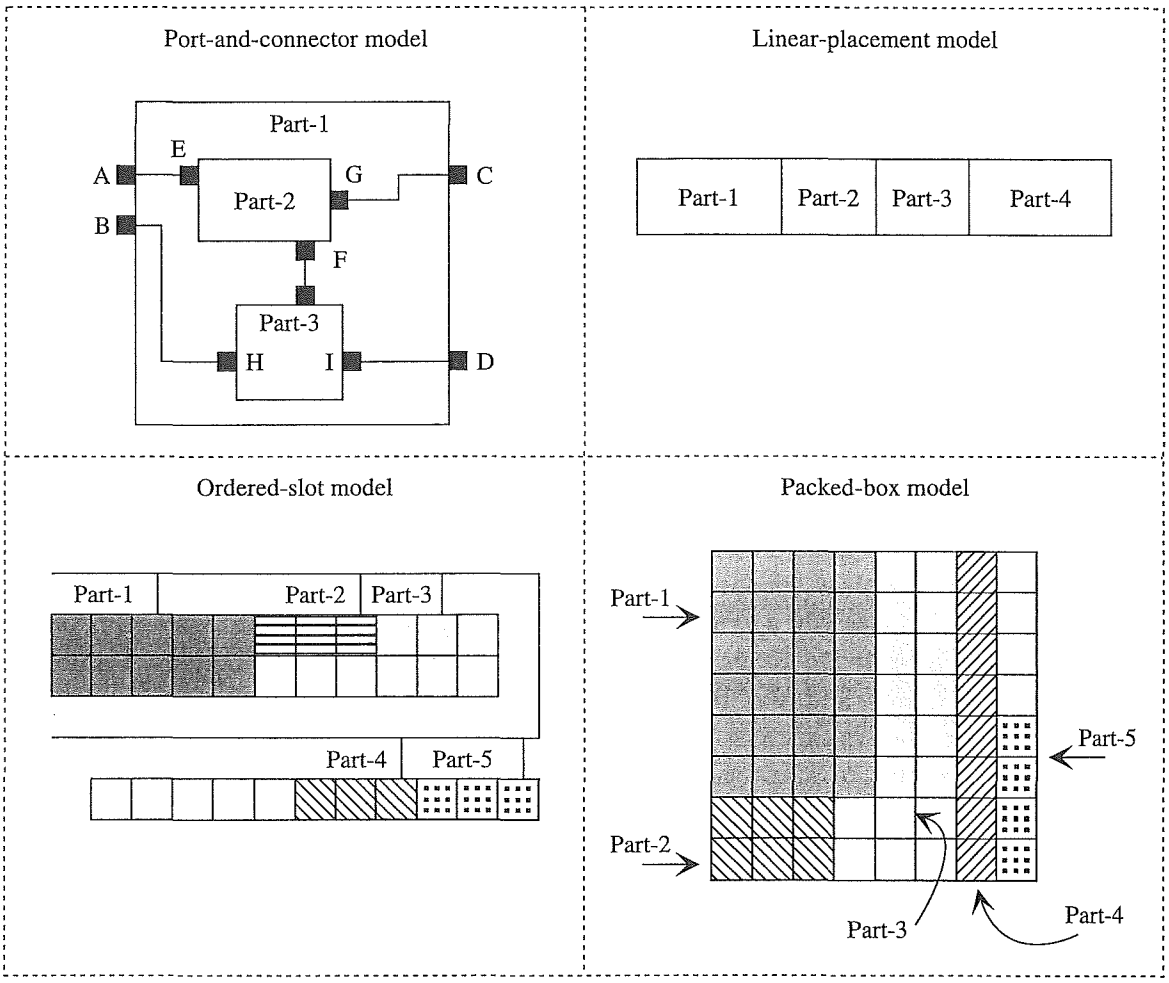

FIGURE 8.34. Alternative models of arrangement for configuration problems. Different arrangement models lend themselves to different resource allocation strategies.

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

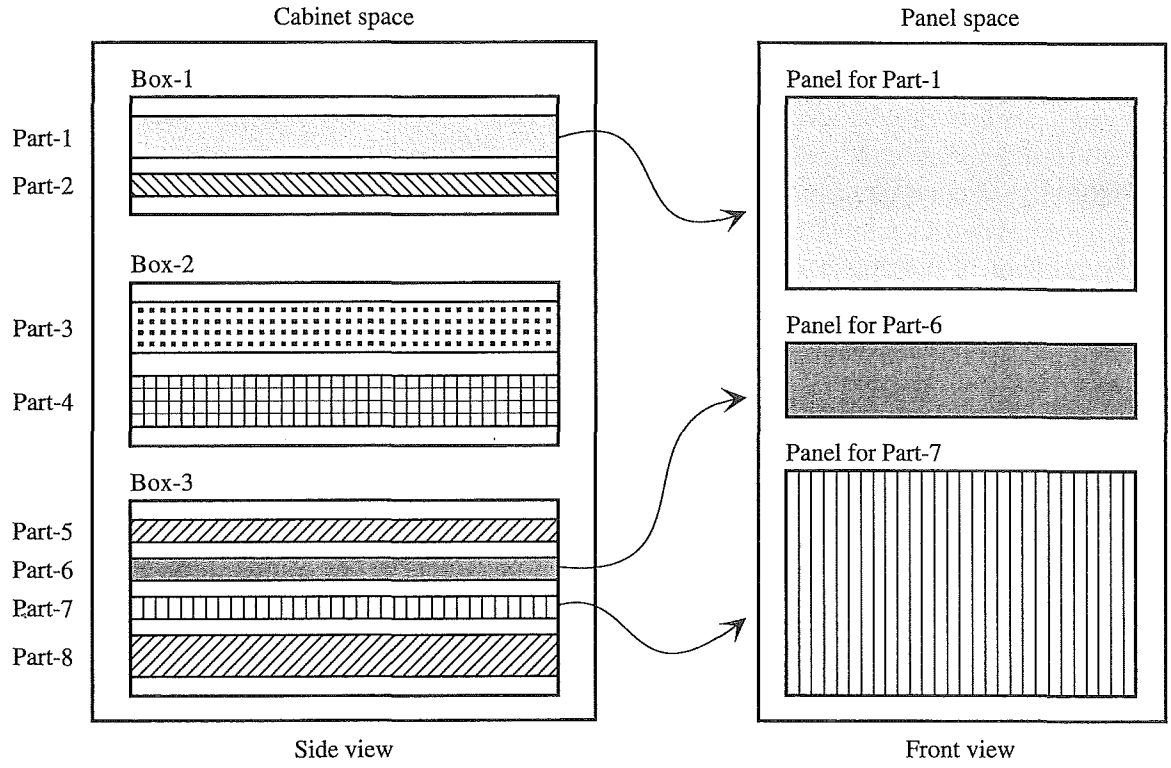

FIGURE 8.35. Another variation on models of arrangement for configuration problems. In this model, parts are arranged in boxes and boxes are arranged in cabinets. In addition, some parts require control panels and there is a limited amount of panel space. Panels for parts must be located on the cabinet 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. */
1.      Set parts-list to nil.

        /* Get-requirements returns specifications for required parts given
        a part. */
2.      Set requirements to get-requirements(initial-specifications).
3.      Set key-components to get-best-parts(requirements).

        /* Add all the required parts to the parts-list. */
4.      For each key component do
5.      begin
6.            Push the key component onto the parts-list.
7.            Add-required-parts(key-component).
8.      end

        /* Arrange parts in the configuration. */
9.      Arrange parts using arrange-parts(parts-list).
10.     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. */
1.      Add-required-parts(part)
2.      begin
3.            Set requirements to get-requirements(part).
4.            If there are some requirements, then
5.            begin
6.                  Set new-parts to get-best-parts(requirements).
7.                  For each of the new-parts do
8.                  begin
9.                        Push the new-part onto the parts-list.
10.                       Add-required-parts(new-part).
11.                 end
12.           end
13.     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. */
1.   Set parts-list to nil.
2.   Set requirements to get-requirements(initial-specifications).
3.   Set key-components to get-best-parts(requirements).

     /* Conditionally invoke Subtasks. */
4.   While there are pending subtasks do
5.   begin
6.        Test conditionals for invoking subtasks and choose the best
          one.
7.        Invoke a specialized method for the subtask.
8.   end
9.   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.
     */
10.  Method-for-Subtask-1:

     /* Subtask-1: Look ahead as needed and expand some parts. */
     /* Subtask-2: Look ahead as needed and arrange some parts. */

20.  Return.
...
30.  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. */
1.   Initialize the list of parts to empty.
2.   Set requirements to get-requirements(initial-specifications).
3.   Set key-components to get-best-parts(requirements).

     /* Apply the domain knowledge to extend and revise the configuration.
     */
4.   While there are open decisions and failure has not been signaled do
5.   begin
6.        Select the next-node in the partial configuration for which a
          decision can be made.
7.        If the decision is a design extension, then invoke method
          propose-design-extensions(next-node).
8.        If the decision is to post a constraint, then invoke method
          post-design-constraints().
9.        If there are violated-constraints, then invoke revise-design().
10.  end
11.  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 visual-

ization 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.