R1: A Rule-Based Configurer of Computer Systems*

John McDermott

Department of Computer Science, Carnegie-Mellon University, Schenley Park, Pittsburgh, PA 15213, U.S.A.

Recommended by Edward A. Feigenbaum

ABSTRACT

R1 is a program that configures VAX-11/780 computer systems. Given a customer's order, it determines what, if any, modifications have to be made to the order for reasons of system functionality and produces a number of diagrams showing how the various components on the order are to be associated. The program is currently being used on a regular basis by Digital Equipment Corporation's manufacturing organization. R1 is implemented as a production system. It uses Match as its principal problem solving method; it has sufficient knowledge of the configuration domain and of the peculiarities of the various configuration constraints that at each step in the configuration process, it simply recognizes what to do. Consequently, little search is required in order for it to configure a computer system.

1. Introduction

R1[1] is a rule-based system that has much in common with other domain-specific systems that have been developed over the past several years [1, 13]. It differs from these systems primarily in its use of Match rather than Generate-and-test as its central problem solving method; rather than exploring several hypotheses until an acceptable one is found, it exploits its knowledge of its task domain to generate a single acceptable solution. R1's particular area of expertise is the

*The development of R1 was supported by Digital Equipment Corporation. The research that led to the development of OPS4, the language in which R1 is written, was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, and monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1151. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Digital Equipment Corporation, the Defense Advanced Research Projects Agency, or the U.S. Government. VAX, PDP-11, UNIBUS, and MASSBUS are tradmarks of Digital Equipment Corporation.

[1]Four years ago I couldn't even say "knowledge engineer", now I . . ..

Artificial Intelligence **19** (1982) 39–88
0004-3702/82/0000–0000/$02.75

configuring of Digital Equipment Corporation's VAX-11/780 systems. Its input is a customer's order and its output is a set of diagrams displaying the spatial relationships among the components on the order; these diagrams are used by the technician who physically assembles the system.[2] Two interdependent activities must be performed in configuring a VAX system.

- The customer's order must be determined to be complete; if it is not, whatever components are missing must be added to the order.
- The spatial relationships among all of the components (including those that are added) must be determined.

The criterion of success for whether a configuration is complete does not reside in any simple test, but involves instead particular knowledge about all the individual components and their relationships. The criterion of successful spatial arrangement is more compact (reflecting the uniform character of geometric structure), but it too involves particular knowledge on a component by component basis. Thus, task accomplishment is defined by a large set of constraints embodying a large amount of knowledge.

R1's approach to the configuration task is to start with the set of components on the order and construct an acceptable configuration using its knowledge of the constraints on component relationships. Since it's strategy is wholehartedly bottom-up, it may seem surprising that R1 is able to generate an acceptable configuration without doing backtracking. R1 is able to avoid search because its task domain permits the use of the Match method [9]. Match will be discussed at length in Section 2.3 below. Basically, the method is applicable if it is possible to order a set of decisions in such a way that no decision has to be made until the information sufficient for making that decision is available. This applicability condition is trivially satisfied if the set of decisions required by a task is fixed and can be ordered a priori. But in the VAX-11/780 configuration task, the set of decisions required varies widely from configuration to configuration. The Match method is a procedure for dynamically ordering a set of decisions. When Match is used, each decision brings one closer to the successful completion of the task.

Although a significant portion of this paper is devoted to a description of precisely how R1 goes about doing the configuration task, I have tried to avoid letting the details of R1's inner workings overshadow the domain independent lessons that have emerged from this research. There are two important lessons:

- An expert system can perform a task simply by recognizing what to do, provided that it is possible to determine locally (i.e., at each step) whether taking some particular action is consistent with acceptable performance on the task.
- When an expert system is implemented as a production system, the job of refining and extending the system's knowledge is quite easy.

[2] R1's output for a sample order is shown in Appendix B.

The paper is divided into three sections. The first section describes the VAX-11/780 configuration task and characterizes its difficulty. The second section describes R1 and discusses its evolution from a system with only the most limited capabilities to what might fairly be called an expert. The third section describes R1's current level of expertise and isolates the design decisions that made the building of R1 straightforward.

1. The Task

The VAX-11/780 is the first implementation of Digital Equipment Corporation's VAX-11 architecture. It is similar in many respects to the PDP-11, though its virtual address space is 2^{32} rather than 2^{16}. The VAX-11/780 uses a high speed synchronous bus, called the sbi (synchronous backplane interconnect), as its primary interconnect. The central processor, one or two memory control units, one to four massbus interfaces, and one to four unibus interfaces can be connected to the sbi. The massbuses and particularly the unibuses can support a wide variety of peripheral devices. Because the number of system variations is so large, the VAX configuration task is nontrivial.

1.1. The size of the task

A configurer must have two sorts of knowledge. First, he must have information about each of the components that a customer might order. For each component, the configurer must know the properties that are relevant to system configuration—e.g., its voltage, its frequency, how many devices it can support (if it is a controller), how many ports it has; I will call this knowledge *component information.* Second, he must have rules that enable him to associate components to form partial configurations and to associate partial configurations to form a functionally acceptable system configuration. These rules must indicate what components can (or must) be associated and what constraints must be satisfied in order for these associations to be acceptable; I will call this knowledge *constraint knowledge.*

The difficulty of the VAX configuration task is a function of the amount of component information and the amount of constraint knowledge required to perform the task. It is fairly easy to estimate the amount of component information that is needed. On the average, a configurer must know eight properties of a component in order to be able to configure it appropriately. Currently about 420 components are supported for the VAX.[3] Thus there are over 3300 pieces of component information that a VAX configurer must have access to.

Before R1 was developed, it would have been difficult to estimate accurately

[3]Of the 420 components, about 180 are actually bundles composed of various subsets of the remaining 240 components.

the amount of constraint knowledge required for the configuration task. Much of the required knowledge was not written down anywhere and thus the only source of estimates would have been individual human experts. But the experts find the task of quantifying their constraint knowledge foreign. As I extracted this knowledge from them, it became clear that their knowledge takes to forms: (1) The experts have a sparse but highly reliable picture of their task domain. When asked to describe the configuration task, they do so in terms of the subtasks involved and the various temporal relationships among these subtasks. (2) They also have a considerable amount of very detailed knowledge that indicates the features that particular partial configurations and unconfigured components must have in order for the partial configurations to be extended in particular ways.Both sorts of knowledge are easily expressable as rules. I extracted 480 rules. Of these, 96 define situations in which some subtask should be initiated. The other 384 rules define situations in which some partial configuration should be extended in some way.

1.2. The constraints

To understand a program, such as R1, that is driven almost entirely by task-specific knowledge, it is necessary to understand the nature of the constraints imposed by the task. This subsection provides an example of a specific subtask that can arise within the configuration task and indicates (1) the constraint knowledge involved, (2) the informational demands imposed by that constraint knowledge, and (3) the extent to which the subtask presupposes other subtasks. The subtask is that of placing unibus modules in backplanes. As is the case with the configuration task in general, many of the constraints in this subtask have a strongly ad hoc flavor; they are not easily derivable from more general knowledge because they are the result of design decisions whose justifications are hidden in a shadowy past. Moreover, the applicability of a particular constraint is ordinarily dependent on a variety of factors; many decisions that a configurer makes must take a large number of details into account.

Whenever more than one unibus option is ordered for a VAX, it is necessary to place the modules on the unibus in an acceptable sequence. Ignoring spatial constraints, it is straightforward to determine the theoretically optimal sequence for the modules; the modules are sorted on the basis of their interrupt priority and within that on the basis of their transfer rate. Before a module can be placed on the unibus, it is necessary to select a backplane. Several constraints come into play. Backplanes come in two sizes (4-slot and 9-slot) and can have any of several pinning types. The backplane selected must be of the pinning type required by the unibus module. To determine the size of the backplane to be selected, it is necessary, first, to determine whether the size is constrained by the box that the backplane will be placed in. 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.[4] Assuming that either a 4-slot or a 9-slot backplane would be acceptable, the next constraint to come into play is that a 9-slot backplane should not be selected unless the next N modules in the optimal sequence all require a backplane of the same type and will not all fit in a 4-slot backplane. Once a backplane is selected, the board or boards comprising the next module in the optimal sequence can be placed in the backplane. However, the first and last slots of a backplane cannot accomodate a full width board; thus the configurer must make sure that the boards will physically fit into the backplane. There are several other constraints. For example, the total amount of power that can be drawn from a regulator is limited; also, if the length of the unibus exceeds a certain limit or if the load on the unibus exceeds a certain limit, a backplane containing a unibus repeater must be placed in the box.

After a module has been placed in a backplane, there is frequently room for additional modules, but the next module in the optimal sequence may require a backplane of a different type or more space than is available in the backplane. At that point the configurer must decide whether to deviate from the optimal sequence or to leave some of the backplane slots empty; the decision is based on the total amount of box space available and the seriousness of the deviation. If there is sufficient box space to accommodate the modules when they are in the optimal sequence, the optimal sequence should be preserved (even if this entails adding additional backplanes to the order). If there is not sufficient space, the seriousness of the deviation must be determined; there are some less than optimal sequences that are acceptable. If the decision is that the deviation from the optimal would impair the functionality of the system, then the configurer must add another box (and possibly a cabinet as well) to the order; if the decision is that an acceptable suboptimal sequence can be found, then some module other than the next one in the optimal sequence is placed in the backplane. In general, it is acceptable to add a module that is not next in the optimal sequence if it meets all of the constraints mentioned above and has a transfer rate that is lower than that of any other module with the same interrupt priority that it will precede.

There are reasons, other than lack of slot space or power, why the configurer must consider deviating from the optimal sequence. If the module that is to be added is a multiplexer, for example, then in addition to the space and power constraints, there is the added constraint that sufficient panel space must be available in the cabinet containing the box that will contain the multiplexer. If there is not enough panel space in that cabinet, the configurer must decide whether to deviate from the optimal ordering or to put all of the remaining modules in the remaining cabinets. Again, the decision must be made on the

[4]The box that contains unibus modules has two +5 volt regulators. One of these regulators supplies power to the first two slot backplanes (or to the first 9-slot backplane); the second supplies power to the other backplanes. All of the modules in a backplane must draw power from the same regulator.

basis of the total space available and the seriousness of the deviation. If there are no other cabinets on the order, one must be added to the order.

This description of how unibus modules are configured brings to light most of the constraints relevant to that task for a simple, single unibus system. But it does not make very clear what the demands for component information are or the extent to which this task presupposes other tasks. For a module, the component information that the rules require is the module type, the number and size of each board in the module, transfer rate and interrupt priority, the pinning type required, the power drawn by the module, and the load it puts on the unibus. For a backplane, the information required is the pinning type, the size, the power drawn by the modules that have been placed in it, and the slots still available. For a box, the information is the box type, the amount of backplane space still available in the box, and the length of and load on its unibus. For a cabinet the information is the cabinet type and amount of box and panel space still available in the cabinet.

Some of the tasks that this task presupposes were already mentioned: determining the optimal sequence, selecting a backplane, assigning the backplane to a box, selecting some module other than the next one in the optimal sequence, verifying (when the module is a multiplexer) that there is sufficient panel space in the cabinet. There are several others: unibus adaptors have to have been configured (so unibus length can be computed); boxes have to have been assigned to cabinets and also to unibuses (so unibus length can be computed and so the amount of usable panel space will be known); the unibus cables that connect backplanes must be selected (so unibus length can be computed); before a module that is a laboratory peripheral can be put in a backplane, it must be determined that there is sufficient panel space in the cabinet and sufficient space and power in the box for the backplane that will contain the laboratory peripheral options.

2. The System

This section focuses on how to represent the knowledge required for the VAX configuration task so that the resulting system can perform the task expertly and efficiently and can easily acquire additional knowledge about the domain. The architecture in which R1 is embedded is described. Issues of search are discussed. The content and use of R1's knowledge is analyzed. Finally, some design and implementation history is provided in order to show the extent to which the development of R1 was an evolutionary process.

2.1. The production system architecture

R1 is implemented as a production system [10]. The particular production system language used is OPS4. Since detailed descriptions of this language have

been provided elsewhere [4, 5, 7], only a brief indication of the basic features of the language will be given in this paper. An OPS4 production system consists of a set of productions held in *production memory* and a set of data elements (e.g., state descriptions) held in *working memory.* A *production* is a conditional statement composed of conditions and actions; a production has the form

$$P_i(C_1C_2 \cdots C_n \rightarrow A_1A_2 \cdots A_m).$$

An action typically modifies working memory by deleting, adding, or modifying a data element; users may, however, define application specific actions. A condition is a pattern containing constants and variables; when each of the conditions in a production can be matched by an element in working memory, the production is said to be instantiated. An *instantiation* is an ordered pair of a production and a set of elements from working memory that satisfy the conditions of the production. The OPS4 interpreter operates within a control framework called the *recognize-act* cycle. During the recognition part of the cycle, it finds the instantiation to be executed; during the act part, it performs the actions. The recognize-act cycle is repeated until either no production can be instantiated or an action explicitly stops the processing. Recognition can be divided into *pattern-match* and *conflict resolution.*[5] During pattern-match, the interpreter finds the set of all instantiations of productions that are satisfied on the current cycle. During conflict resolution, it determines which instantiation to execute.

OPS4 is a general-purpose production system language; it was not developed with the configuration task in mind. Since OPS4 knows nothing of configuration, all of the domain-specific knowledge that R1 requires to do the task—including domain-specific control knowledge—is contained in productions. Each of R1's productions (rules) embodies a piece of constraint knowledge. The production's conditions typically look for situations in which a particular type of extension to a particular type of partial configuration is permissable or required; the actions then effect that extension. R1's rules are such that on almost every cycle several rules can be instantiated—frequently in a number of different ways. From R1's point of view, it often makes no difference which of these instantiations is executed; in these cases, how OPS4 determines which instantiation to execute is irrelevant.[6] R1 does, however, rely heavily on one of OPS4's conflict resolution strategies, the *special case strategy,* and so this strategy needs to be understood. Given two instantiations, one of which

[5]The term 'pattern-match' (rather than 'match') is used here to avoid confusion with the 'Match method' (Section 2.3) which is the problem solving method R1 uses in performing the configuration task.

[6]For example, a rule that bears on configuring some particular type of component will have more than one instantiation if more than one such component is available; any of these instantiations could be executed. Or if R1 is filling a cabinet, it might make no difference which part of the cabinet is filled first.

contains a proper superset of the data elements contained by the other, OPS4 will select the instantiation containing more data elements on the assumption that it is specialized for the particular situation at hand. OPS4's cycle time, though it is essentially independent of the size of both production memory and working memory [6], depends on particular features of the production system (e.g., the number and complexity of the conditions in each production). The average cycle time for OPS4 interpreting R1 is about 150 milliseconds.[7]

R1 currently has 772 rules that enable it to perform the configuration task. An English translation of a sample rule is shown in Fig. 2.1. The first condition indicates that the subtask in which this rule is relevant is the distributing of massbus devices among massbuses. The other five conditions specify one of the sets of constraints that must be satisfied within this subtask in order for a disk drive to be assigned to a massbus. When an instantiation of this rule is executed, one of the single port disk drives on the order is assigned to one of the massbuses.

To provide R1 with access to information about each component that can appear on an order, production memory and working memory have been augmented, for this application, with a third memory. This memory, the *data base*, contains descriptions of each of the 420 components currently supported for the VAX. Each entry in the data base consists of the name of a component and a set of attribute/value pairs that indicate the properties of that component that are relevant for the configuration task. Every component has a *type* attribute and a *class* attribute; the class of a component determines what other attributes are relevant. There are 15 classes: bundle, cabinet, sbi module, sbi device, box, backplane, massbus device, unibus device, unibus module, panel, power supply, software, cable, document, and accessory. Each component description consists, on the average, of eight attribute/value pairs; there are only about 50 distinct attributes, several of which are common to all (or most) of the classes. Fig. 2.2 shows two of the entries in the data base. The RK711-EA is a bundle of components; it contains a 25 foot cable (70-12292-25), a disk drive (RK07-EA*), and a bundle of components (RK611) which itself consists of three continuity boards (G727), a unibus jumper cable (M9202), a backplane (70-12412-00), and a disk drive controller (RK611*). The RK611* is a disk drive controller which, because of its interrupt priority and transfer rate, is typically located toward the front of the unibus. The module is comprised of five hex boards each of which start in lateral position 'A'; it draws 15.0 amps of +5 volt current, 0.175 amps of −15 volt current, and 0.4 amps of +15 volt current, and it generates 1 unibus load. It can support up to eight disk drives and is connected to the first of these with a 70-12292 cable of some length.

[7]OPS4 is implemented in MACLISP; R1 is run on a PDP-10 (model KL) and loads in 412 pages of core.

```
DISTRIBUTE-MB-DEVICES-3

    IF: THE MOST CURRENT ACTIVE CONTEXT IS DISTRIBUTING MASSBUS DEVICES
        AND THERE IS A SINGLE PORT DISK DRIVE
                THAT HAS NOT BEEN ASSIGNED TO A MASSBUS
        AND THERE ARE NO UNASSIGNED DUAL PORT DISK DRIVES
        AND THE NUMBER OF DEVICES THAT EACH MASSBUS SHOULD SUPPORT IS KNOWN
        AND THERE IS A MASSBUS THAT HAS BEEN ASSIGNED AT LEAST ONE DISK DRIVE
                AND THAT SHOULD SUPPORT ADDITIONAL DISK DRIVES
        AND THE TYPE OF CABLE NEEDED TO CONNECT THE DISK DRIVE
                TO THE PREVIOUS DEVICE ON THE MASSBUS IS KNOWN
  THEN: ASSIGN THE DISK DRIVE TO THE MASSBUS
```

FIG. 2.1. A sample rule.

In addition to containing descriptions of VAX components, the data base also contains a few cabinet templates. A *cabinet template* describes what space is available in a particular cabinet type. These templates serve two purposes: (1) they enable R1 to know, at any point in the configuration process, what container space is still available, and (2) they enable R1 to assign a specific location (i.e., coordinates) to each component that it places in a cabinet. Fig. 2.3 shows the template for the cpu cabinet. The components that may be ordered for the cpu cabinet are sbi modules, power supplies, and an sbi device. The template for the cpu cabinet contains descriptions of the space available for each of these classes of components and specifies what can be put where. For example, up to six sbi modules fit into a cpu cabinet; each cabinet contains a cpu module and some memory; in addition 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 cpu module must be associated with nexus 2 of the sbi; the numbers in parentheses indicate the top left and bottom right coordinates of the space that can be occupied by a cpu module.

Initially, working memory is empty. It grows, during the course of configur-

```
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
    NUMBER OF 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
    NUMBER OF UNIBUS DEVICES SUPPORTED: 8
    CABLE TYPE REQUIRED: 1 070-12292 FROM A DISK DRIVE UNIBUS DEVICE
```

Fig. 2.2. Two component descriptions.

```
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)
```

FIG. 2.3. A sample template.

ing a system, to contain descriptions of the components ordered, and as various components are associated, to contain descriptions of partial configurations as well as other component information required to do the configuration task. A component is represented in working memory as a *component-token* with associated attribute/value pairs. R1 retrieves information from the data base as the need for such information arises. There are five actions that R1 can perform that provide it access to the data base. Three of these functions, *generate-tokens*, *find-token*, and *find-substitute-token*, retrieve specified information about a component (or list of components) from the data base, create a component-token, and then add a partial description of the component to working memory. The other two, *get-attributes* and *get-template*, augment the description of an already existing component-token.

In addition to containing component descriptions, working memory contains three other types of elements:

- Elements that define partial configurations.
- Elements that indicate the results of various sorts of computations.
- Context symbols.

An element that defines a partial configuration contains a description of the relationships among two or more components. Typically, these elements indicate either that one component is to be connected to another by means of a cable or, in the case of a component that is a container, the spatial relationship between the container and each of the components it contains. An element that indicates the result of some computation contains a symbol identifying the computation and one or more values indicating the result. The component descriptions, together with the elements that define partial configurations and the elements that indicate the results of various computations, constitute what I referred to above as component information. A context symbol contains a context (subtask) name and an indication of whether or not the context is active.

2.2. Contexts

The configuration task can be viewed as a hierarchy of subtasks that have strong temporal interdependencies. The actions required within each subtask are highly variable; they depend completely on the particular combination of components that appear on an order and the way in which sets of those components have been configured up to the point when the new subtask becomes appropriate. It is possible, however, to indicate what actions (including the action of initiating a new subtask) are appropriate within a subtask in terms of a relatively small number of rules that are each sensitive to a few salient features of the current situation.

R1's approach to exploiting the temporal relationships among subtasks is straightforward. The function of several of R1's rules is to recognize, on the basis of the component information in working memory, when a new subtask should be initiated. When one of these rules fires, it adds a context symbol to working memory. Each context symbol contains a context name, an indication of whether the context is active or not-active, and an indication of when (i.e., how recently) the context was made active. Each rule contains two condition elements that are sensitive to context symbols. Together these conditions insure that only those rules that bear on the most current active context will fire. Which of these rules fires depends on what descriptions of components, partial configurations, and computational results are in working memory. Typically, a few of the rules associated with a context contain the constraint knowledge ordinarily relevant within that context. Other rules associated with the context are special case rules; these rules, when their more stringent conditions are satisfied, fire before (and often instead of) the ordinary case rules. Each context has an associated rule containing only the two condition elements sensitive to a context symbol. This rule deactivates the current context. Since each deactivation rule is a general case of all the other rules sensitive to its context, it fires only after the other satisfied rules have fired. R1, then, is a recognition-driven system that relies on its knowledge of the structure of the configuration task as well as on information about the set of components it is configuring to determine what to do. When it has several courses of action open to it, it falls back on general (non-task-specific) strategies for selecting among alternatives. But it needs only two such strategies: (1) it uses special case rules in preference to more general ones, and (2) it does all that it can within a context before leaving that context.

The contexts that R1 makes use of in order to do the configuration task were not arrived at through a rigorous analysis of the demands of the configuration task. Rather, they reflect the way in which human experts who do the task actually approach it. As might be expected, as R1 evolved it became apparent that some modifications and reordering of the contexts would make the processing easier, and so these changes were made. But basically, the approach

that R1 takes to the task is the same as that of human configurers. At the top level, the task divides up into 6 major subtasks:

Subtask 1. Determine whether there is anything grossly wrong with the order (e.g., mismatched items, major pre-requisites missing).

Subtask 2. Put the appropriate components in the cpu and cpu expansion cabinets.

Subtask 3. Put boxes in the unibus expansion cabinets and put the appropriate components in those boxes.

Subtask 4. Put panels in the unibus expansion cabinets.

Subtask 5. Lay out the system on the floor.

Subtask 6. Do the cabling.

The following paragraphs give a rough picture of what each of these subtasks involves from R1's perspective; the purpose of this excursion into the guts of the configuration task is to provide some concreteness on which to build the subsequent discussions of the effectiveness (and adequacy) of R1's problem solving method and of the use that it makes of its configuration knowledge.

Subtask 1 (196 rules)

The first subtask is to determine whether there are major problems with the order and to rectify them if possible. The work that R1 does during this stage is considerably more complex than that done by human experts during this stage. Humans tend to assume that an order will be unproblematic and wait until a problem actually arises before dealing with it. The advantage of the human approach is that it saves an unnecessary first step when configuring unproblematic orders; its disadvantage is that if problems do arise, they may impact earlier decisions and thus require redoing part of the configuration (i.e., backtracking). R1 first retrieves partial descriptions of each of the components on the order. If a component is a bundle, it retrieves a partial description of each of the components on that bundle's component list. Then R1 checks to see whether it has been instructed to treat any of the components in an exceptional way—e.g., to leave some components (presumably spares) unconfigured. It next determines whether all of the components on the order have compatible voltage and frequency requirements, and if not, substitutes components of the appropriate voltage and frequency. Finally R1 determines, to the extent that it can before actually doing the configuration, whether any of the massbus or unibus devices on the order have prerequisite components that are not on the order. If any pre-requisite components are missing, it adds them to the order.

Subtask 2 (87 rules)

The second subtask involves putting whatever components belong in the cpu and cpu expansion cabinet into those cabinets; in performing this subtask, R1 relies heavily on the templates for the cabinets. It augments the description of whatever cpu cabinet is on the order with the information in the template for

the cpu cabinet. R1 then finds a component in working memory whose class is sbi module and whose type is the type of one of the template elements for the sbi. It adds the component (and the coordinates it will occupy) to the list of components that are to be put in the cpu cabinet and deletes the template element. It repeats this step until no template elements remain that can be paired with a component on the order. If there are cpu expansion cabinets on the order, R1 fills them using the same rules that it uses for the cpu cabinet; the only difference between the two types of cabinets is that the expansion cabinet has no cpu, and may or may not contain memory. When all of the sbi modules have been placed in cabinets, R1 puts the sbi terminator in the appropriate place in the final cabinet and adds module simulators to any option slots that have not been filled.

Subtask 3 (256 rules)
The third subtask is to put boxes into the unibus expansion cabinets, and to put unibus modules into the boxes. Given a limited amount of box space, the information that is needed to determine whether a module has been configured acceptably is not available until after all of the modules have been placed on the unibuses. Thus R1 sometimes has to generate a number of candidate module-sequences before it finds one that is acceptable. There are a number of independent constraints on the placing of unibus modules:

(1) Each module must be put in a backplane slot of the appropriate pinning type.

(2) The position of each backplane in a box must be such that its modules draw power from a single set of regulators.

(3) There is a limit on the amount of power that the modules in a backplane can collectively draw from any regulator.

(4) If a module requires panel space, that panel space must be in the cabinet containing the module.

(5) If a module requires other supporting modules either in the same backplane or in the same box, space must be available for those supporting modules.

(6) The modules should be placed on the unibus in a sequence that is as close to the optimal sequence as possible.

If only the first five constraints applied, R1 would generate only acceptable unibus module configurations; but the addition of the sixth constraint, since it is elastic, makes that impossible. In order to limit the amount of search it has to do to configure the unibus modules, R1 interprets the sixth constraint somewhat liberally. It defines three equivalence classes of sequences: optimal (any ordering that is optimal), almost-optimal (any less than optimal ordering such that no module whose interrupt priority is i and whose transfer rate is j, occurs before a module whose interrupt priority is i and whose transfer rate is less than j), and suboptimal (any other ordering).

To configure a set of unibus modules, R1 first estimates the amount of space required to place the unibus modules optimally and the amount of space required to place the modules suboptimally; it then determines the optimal sequence. If the amount of box space available is greater than or equal to the space required for an optimal configuration, it tries to place the modules on the unibus in that sequence. If it fails (or if the amount of box space available is less than the space required for an optimal configuration, but greater than that required for a suboptimal configuration), it retries the subtask, modifying the sequence whenever such a modification would save space and result in an almost-optimal sequence. If this attempt fails, it retries the subtask again, but this time modifies the sequence whenever such a modification would save space. If this attempt fails or if the amount of box space available is less than that required for a suboptimal configuration, R1 adds another box to the order and retries the subtask.

With this background, the subtask of configuring unibus cabinets, boxes and modules can now be described. R1's first step is to assign each box to a unibus and each box to a cabinet. To determine whether a particular unibus module can be placed in some backplane, the distance from the beginning of the unibus (the adapter) to that backplane must be known. For R1 to have this information, the box assignments have to have been made. Once R1 has assigned each box to a cabinet, it starts to fill the first box on each unibus. Filling a box involves first selecting a module (and a box to put it in, if the system has more than one unibus). If R1 is attempting to place the modules in a way that will preserve the optimal sequence, module selection is straightforward. If R1 is willing to accept an almost-optimal or a suboptimal configuration, it selects the next module in the optimal sequence unless that module requires a backplane that will occupy more space than remains in the box; if there is insufficient space for such a backplane, R1 considers each of the remaining modules to see if there is one which will satisfy the relaxed sequencing constraint and which requires a backplane that will fit in the box. After selecting a module, R1 selects (or adds to the order) a backplane that has pinning and a size that will accommodate the module. At this point, R1 attempts to put the module in the backplane. Although it knows that there is sufficient space for the module in the backplane, it does not know whether the other constraints are satisfied. If adding this module would result in the power-drawn limit being exceeded, or if the unibus-load limit would be exceeded, or if required panel space is not available, or if there is not room in the backplane or box for supporting modules if any are required, then the attempt to add the module will fail. Here again, if R1 is willing to accept a less than optimal configuration, it will try a different module. As long as space remains in the backplane and there are modules that have not been configured, R1 will try to put additional modules in the backplane. The module selection

process is identical to the one described above, except that a backplane is already available. If R1 is able to place all of the unibus modules in the available box space, it goes on the fourth subtask. If it cannot, it either relaxes the sequencing constraint or adds another box to the order and tries again.

Subtask 4 (30 rules)
The fourth subtask is to assign panels to cabinets and to associate those panels with unibus modules and with whatever devices the modules serve. Since by this point all of the unibus modules have been configured and since panel space has been assigned to each module requiring it, all that R1 has to do is select a panel of the appropriate size and line type (or add one to the order) and assign that panel to the panel space set aside for the module. Then R1 assigns those devices that must connect to a module via a panel to a panel of the appropriate type.

Subtask 5 (61 rules)
The fifth subtask is to generate a floor layout for the system (i.e., to specify the spatial relationships among all of the cabinets and free standing devices). To do this subtask adequately, it is necessary to have information about the site at which the system will be installed (e.g., room dimensions, locations of obstructions) that is not currently available to R1. At the moment, therefore, all that R1 does is group together those components that must be spatially proximate and then lay the devices out in a line in the appropriate order. This subtask is relatively easy since all of the inter-device assignments have been made and information about how close a particular device should be to the cpu cabinet relative to other devices is in R1's data base.

Subtask 6 (36 rules)
The final subtask is to specify what cables are to be used to connect each device to the other devices to which it has been assigned. Given the inter-device assignments, all that R1 has to do is determine the distance between each pair of devices that must be connected and find (or add) a cable of the type and length required. Because R1 does not currently lay out systems in the way they will be laid out at the installation site, it does not do an adequate job of determining the precise cable lengths required. However, since much of the cabling is between devices that have a site-independent spatial relationship, the job that R1 does is more helpful than might first appear. All of the cabinets must be bolted together, and devices associated with the same controller must be physically proximate. Thus, the only cable lengths that R1 cannot determine precisely are those involving cables that connect a controller to the first device it serves. Once R1 has finished the cabling task, the system is configured; at that point, R1 generates output describing the configuration.

2.3. Searching the space of possible configurations

The configuration task performed by R1 requires finding an acceptable configuration within the space of possible configurations. The basic operations that R1 uses to explore this space are creating and extending partial configurations. Other of R1's operations, such as retrieving component information from the data base or counting the number of components of a particular type, prepare the way for the actual acts of configuration. It would appear that performing any task for which there are many constaints on an acceptable solution requires a heuristic search (i.e., a combinatorial search in which candidate partial solutions are constructed and their potential evaluated). This has been the experience of AI in all sorts of tasks [11] and in particular in real-world domains where the methods used have been characterized as forms of Generate-and-test [3]. R1, however, by and large does without search. It is useful to understand just what the structure of R1's method is and what permits search to be avoided.

R1's method is essentially a generalized form of matching. That is, R1 uses the method normally used to match a form (i.e., a symbolic expression containing variables) against an exemplar in order to instantiate the variables, thus making the form identical to the exemplar. Matching is usually not considered to be search. In typical AI heuristic search programs, it is taken to be a part of the computation performed within a state to determine which operators are applicable. However, Match is clearly also a search technique, analagous to Generate-and-test, Means-ends Analysis, etc. The Match method can be found in some of the earliest heuristic search programs [8], and later Newell included a generalized form of Match as one of his *weak methods* [9].

The search space for Match is the space of all instantiations of the variables in a form. Each state in the space is a partially instantiated form. The form is the body of domain specific knowledge that defines acceptable sets of instantiations; thus the form holds the constraints for task satisfaction. This knowledge serves two functions: (1) it enables the form to be brought into correspondence with the exemplar, and (2) it permits local tests (comparisons) to be made between the form and the exemplar at each point of correspondence. The primary condition on Match, then, is

The Correspondence Condition. *Match can be successful only if each constituent of the form can take on a locally determined value (through the comparison of the form to the exemplar at the point of correspondence).*

A consequence of the correspondence condition is that Match never requires backtracking. If there is at least one solution state, Match will find a path from the initial state to a solution state without generating any false paths. This condition abstracts from all of the details of how expressions are put in correspondence. It states only the essential condition that permits the operators to be selected and applied in a single pass.

The decision that determines the next step in the space (i.e., the next instantiation) is local. This does not imply, however, that the space is decomposable into a set of completely independent subtasks. Instantiating a variable can have global consequences for the final solution and hence for subsequent instantiations (because the substitution must take place in all occurrences). However, none of these consequences affect what has already been matched. Indeed, the result of putting the form into correspondence with the exemplar is to impose an order on the instantiations (i.e., the search through the space) so that all global consequences are pushed into the unmatched portion of the form. In typical string matching this requires only that matching start from one end of the string or the other. This can be stated as another general condition:

The Propagation Condition. *A partial ordering on decisions must exist such that the consequences of applying an operator bear only on aspects of the solution that have not yet been determined.*

Applying this analysis of Match to R1, the initial state is the set of descriptions of the components ordered. The intermediate states are sets of descriptions of partial configurations and the as yet unconfigured components. At each decision point, the constraint knowledge about what next step can be taken (i.e., what next partial configuration to produce) is provided by R1's rules. R1 has enough knowledge to satisfy the correspondence condition stated above. Thus, there is no need for backtracking, since its knowledge is sufficient to determine an acceptable next step.

R1's rules can be divided into three categories based on their role in this generalized Match method. (1) *Operator* rules take the actual next step in creating or extending a partial configuration. (2) *Sequencing* rules determine the order in which decisions need to be made to satisfy the propagation condition. These rules are primarily those involved in the sequencing of contexts, and this sequencing is thus seen to be an essential part of R1's method, not simply an additional bit of 'programming'. (3) *Information-gathering* rules access the data base or perform various computations in order to provide the information needed for operator and sequencing rule selection This information has an important subsequent use that has no analogue in the simple Match task. It provides a justification for decisions (and their consequences) that is necessary in order to preserve the propagation condition.

The power and limitations of Match are exemplified by two interesting phenomena. The first one is that humans often do not solve the configuration task by Match, but rather by a more general (weaker) heuristic search; that is, they engage in backtracking. This is true of novices and to a lesser extent also true of experienced configurers. Novices in particular frequently make decisions in the wrong order or neglect to preserve information needed for future decisions. The second phenomenon is that, for R1, Match is not in fact sufficient for the complete task. The subtask of placing modules on the unibus

is formulated essentially as a bin-packing problem—namely how to find an optimal sequence that fits within spatial and power-load constraints. No way of solving this problem without search is known (short of table lookup, if the configurations are limited enough to be enumerated). R1 does not use a fully adequate search method for this subtask, but does a simple Generate-and-test, using Match as a heuristic guide.

The desirability of using a generalized Match method is hardly in doubt; it is good to avoid search, especially when this can be done at low cost. The two phenomena above shed some light on when it is possible to use Match. As the bin-packing (i.e., the unibus module placement) subtask shows, it depends in part on the specific nature of the task environment (i.e., whether its structure is sufficiently interlocking). Bin-packing has been studied enough to attribute the necessity of search to the structure of the task. However, as the search behavior of novice and expert configurers shows, tasks almost invariably become more tractable as they become better understood. The role of experience with a task is to permit the acquisition of enough knowledge, both of decision order and of decision content, to satisfy the two conditions stated above, thus permitting Match to be used.

R1 appears to be the first of the 'knowledge-engineered' domain-specific AI systems to employ Match as its central problem solving method. As mentioned above, the more typical approach has been to develop systems whose central method is Generate-and-test. Such systems contain knowledge that enables them to generate one or more hypotheses that explain (or otherwise give structure to) the phenomena proper to their domains; these hypotheses are then tested against some particular collection of empirical data. The systems differ from one another in the degree to which they make use of the data in generating hypotheses. Some systems generate hypotheses independently of the data to be explained; others generate partial hypotheses that are then modified and extended on the basis of the data. MYCIN [2], for example, has a collection of rules that allow it to infer the likely causes of a bacterial infection in the blood. Hypothesis generation is accomplished by backward chaining of the rules. If a rule's conditions are not known to be true or false (on the basis of patient data), rules that bear on the truth or falsity of its conditions must be examined. Ultimately this process leads back to the data; the data either confirms or disconfirms the hypothesis. Though other domain-specific systems make more use of the data in selecting hypotheses to generate, they also use the data to test hypotheses.

The feature that distinguishes R1 most clearly from these other systems is that it generates only a single hypothesis—the solution. In R1, the knowledge that other systems would use to test hypotheses is part of the generator. As we have seen, except in the case of unibus module placement, this strategy is sufficient. Clearly its applicability is due in part to the structure of the configuration domain. It will be interesting to see to what extent this strategy can be successfully applied in other domains.

It is worth noting that the languages (engines) that underlie the other rule-based expert systems that have been developed to date do not provide the mechanisms necessary for implementing Match.[8] For the most part, these other languages, unlike OPS4, were developed with specific applications in mind. Each of the languages implements a specific (or small set of specific) search strategies that their designers believed would be of value for the application. Moreover, the task domains addressed by these other systems differ in a fundamental way from R1's task domain. For the most part, the other systems perform tasks that can be broadly characterized as analytic; they are given a single, complex object, and their task is to decompose that object into its constituent parts and determine the relationships among those parts. The configuration task, on the other hand, is synthetic; R1 is given a set of components, and its task is to impose relationships on those components and in so doing to form a complex object. It may turn out that the search strategies that each of these systems embody are the most effective strategies for their respective domains; it may even turn out that Match is an unsuitable method for analytic domains. But until we have more experience with a variety of approaches to the various problems raised in each of these domains, meaningful comparison of systems is difficult—especially when the systems differ as fundamentally as R1 does from the others.

2.4. The content of R1's rules

R1's rules can be distinguished from one another in terms of the functions that they perform and the extent to which they embody domain knowledge. As Fig. 2.4 shows, only 480 of R1's 772 rules contain knowledge that is directly related to the configuration task. The other 292 rules contain more general knowledge. About a third of this more general knowledge is used by R1 to generate its output; this knowledge is not used until after the configuration task has been finished. Another third consists of rules that deactivate contexts; essentially these rules contain only the knowledge that if one is in a context and there is nothing left to do, one should exit from the context.[9] The final third of the general knowledge is about evenly divided between rules whose function is to do various kinds of counting tasks, and rules that generate 'empty' data structures for the domain knowledge rules to use. Of the rules that embody domain knowledge, about a fourth generate new contexts, another fourth deal with missing pre-requisites (mostly by adding whatever component is missing to the order), and another fourth create or extend partial configurations. The final fourth is about evenly divided between rules that retrieve partial descriptions of

[8]The converse is not true. General-purpose production system languages, such as OPS4, are not limited to implementing just Match; they are equally useful for other problem solving methods [12].

[9]These rules are, embarassingly enough, completely unnecessary since their function could (and soon will) be taken over by a single general rule.

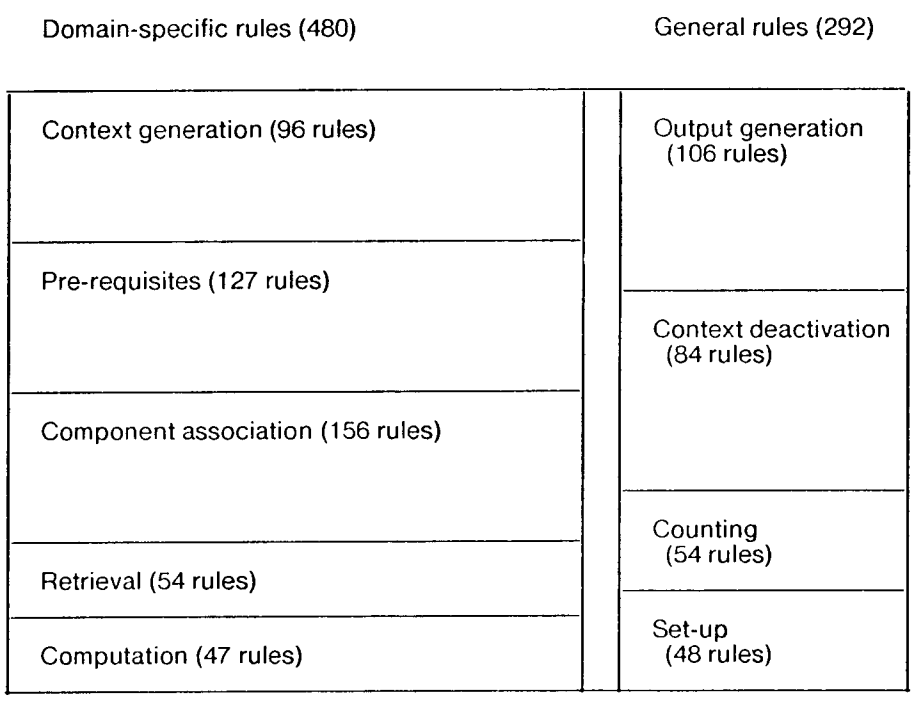

FIG. 2.4. The distribution of R1's knowledge.

components from the data base, and rules that do various sorts of computations. The classification of these domain specific rules is somewhat rough since many of them have dual functions.

The knowledge encoded in individual rules is, of course, just the knowledge needed to make use of the component information and context symbols that can appear in working memory. Thus all of R1's constraint knowledge falls into the following four classes:

- Knowledge of the significance of the various attributes of VAX components and of how to retrieve component information from the data base.
- Knowledge of how to recognize and of how to construct partial configurations.
- Knowledge of how to make and make use of various kinds of computations.
- Knowledge of what actions are appropriate within various contexts and of what contexts to enter from other contexts.

R1 has essentially no knowledge of the defining characteristics of its contexts; it simply has names for contexts, and these names are bare symbols with no associated descriptions. It recognizes 84 context names;[10] each context has, on the average, about eight rules associated with it.

Table 2.1 shows the relative frequency with which the various sorts of constraint knowledge occur as conditions and actions. The first column provides this breakdown for all of R1's rules; the second column provides the breakdown for just the domain-specific rules. In order for the numbers in Table

[10]This excludes the 18 context names recognized by the 106 rules that generate output.

TABLE 2.1. The composition of R1's knowledge

	Mean number in each rule	Mean number in each domain rule
Components		
Conditions	2.02	2.79
Actions	1.24	1.82
Assertions	0.61	0.97
Modifications	0.30	0.43
Deletions	0.33	0.42
Partial configurations		
Conditions	1.15	1.55
Actions	0.52	0.68
Assertions	0.20	0.31
Modifications	0.25	0.29
Deletions	0.07	0.08
Results of computations		
Conditions	1.06	1.20
Actions	0.72	0.82
Assertions	0.31	0.35
Modifications	0.27	0.32
Deletions	0.14	0.15
Contexts		
Conditions	2.04	2.07
Actions	0.35	0.32
Assertions	0.19	0.28
Modifications	0.16	0.04
Deletions	0.00	0.00
Total		
Conditions	6.27	7.62
Actions	2.83	3.64
Assertions	1.30	1.91
Modifications	0.98	1.09
Deletions	0.55	0.64

2.1 to have any significance, it is necessary to understand how much information each piece of knowledge contains. In general, each working memory element contains from three to six pieces of information that R1 can use. Elements of type component each contain a component-token and, on the average, two attribute/value pairs. Elements of type partial configuration almost always contain at least two component-tokens, a symbol indicating their interrelationship, and usually other symbols that further specify the relationship. When the partial configuration element is a list of the contents of a

cabinet or some other container, it may contain 20 or more pieces of information. But R1 typically does not look very far inside these lists; it merely builds them up for its output routines. Elements of type computational result contain a symbol identifying the purpose of the computation and usually two or three values. Elements of type context contain a context name, an indication of whether or not the context is active, and an indication of how recently the context was asserted. The amount of information in a rule is proportional to the amount of information in these working memory elements. Conditions are patterns that are instantiated by the working memory elements; typically a pattern will contain some constants, some variables that occur only once in the condition part of the rule, and some variables that occur more than once in the condition part of the rule and thus function to constrain the combinations of elements that can instantiate the rule. Actions are either functions that retrieve information from the data base, modify or delete specified elements in working memory, or perform various arithmetic calculations, or they are patterns that are added to working memory after being instantiated with values bound in the conditional part of the rule.

Given this background, the numbers in Table 2.1 provide some insight into the amount of constraint knowledge that R1 has. Since the primary interest is in understanding the amount of domain knowledge required for the task, only the numbers in the second column will be considered; they do not differ significantly from the numbers in the first column except they show that R1's domain rules are more discriminating that its more general rules. Each of the task-specific rules has at least two conditions that are sensitive to the context. One of these conditions specifies that a particular context must be active; the other that there not be a more recent active context. The number is actually slightly greater than 2 because a few rules use the fact that a context is no longer active to determine that an action is appropriate. Beyond the context information, there are about 5.5 conditions, each of which can draw on from three to six pieces of information. This suggests that by and large, each rule captures only a very small part of the knowledge in the domain. The ratio of conditions to actions is about 2 to 1.

Rather free English translations of four of the nine rules associated with the context of assigning power supplies to sbi modules are shown in Fig. 2.5. The first rule adds a power supply to the order if one is needed and if all of the power supplies ordered have already been configured. The second rule configures a power supply—i.e., indicates what sbi module the power supply should be connected to and where in what cabinet it should be put. The fourth rule has the same function as the second, and is fired instead of the second when both are satisfied because it is a special case. It contains the extra knowledge that when the sbi module that needs a power supply is a unibus adaptor, a particular regulator (the H7101) must be associated with the power supply. The third rule fires if all of the conditions required to satisfy the fourth

```
ASSIGN-POWER-SUPPLY-1

    IF: THE MOST CURRENT ACTIVE CONTEXT IS ASSIGNING A POWER SUPPLY
        AND AN SBI MODULE OF ANY TYPE 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 NO AVAILABLE POWER SUPPLY
        AND THE VOLTAGE AND FREQUENCY OF THE COMPONENTS ON THE ORDER IS KNOWN
  THEN: FIND A POWER SUPPLY OF THAT VOLTAGE AND FREQUENCY AND ADD IT TO THE ORDER


ASSIGN-POWER-SUPPLY-2

    IF: THE MOST CURRENT ACTIVE CONTEXT IS ASSIGNING A POWER SUPPLY
        AND AN SBI MODULE OF ANY TYPE 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
  THEN: PUT THE POWER SUPPLY IN THE CABINET IN THE AVAILABLE SPACE


ASSIGN-POWER-SUPPLY-6

    IF: THE MOST CURRENT ACTIVE CONTEXT IS ASSIGNING A POWER SUPPLY
        AND A UNIBUS ADAPTOR 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


ASSIGN-POWER-SUPPLY-7

    IF: THE MOST CURRENT ACTIVE CONTEXT IS ASSIGNING A POWER SUPPLY
        AND A UNIBUS ADAPTOR 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
```

FIG. 2.5. Sample rules from a single context.

rule are satisfied except the availability of a regulator; it adds the appropriate regulator to the order. The character of these rules is fairly representative of the rules in each of the contexts. In general, a context has a few rules associated with it whose function is to perform the basic (ordinary) actions appropriate in that context. Other rules associated with the context insure that if one or more of these basic rules are not satisfied because some required component is not on the order, the component will be added to the order so that the appropriate basic rules can fire. Still other rules associated with the context handle exceptional situations; these are typically special cases of the basic rules and fire before (and often instead of) the basic rules if their conditions are satisfied.

The rules shown in Fig. 2.6 complement the sample rules in Fig. 2.5; they give an idea of the kinds of cues that cause R1 to generate a new context and provide a somewhat broader understanding of how component information can be used to select among competing actions. The first three rules shown are all associated with contexts that become relevant within the first major subtask. The first rule checks to see if there are components on the order with

```
CHECK-VOLTAGE-AND-FREQUENCY-1

    IF: THE MOST CURRENT ACTIVE CONTEXT IS CHECKING VOLTAGE AND FREQUENCY
        AND THERE IS A COMPONENT THAT REQUIRES ONE VOLTAGE OR FREQUENCY
        AND THERE IS ANOTHER COMPONENT THAT REQUIRES A DIFFERENT VOLTAGE OR FREQUENCY
  THEN: ENTER THE CONTEXT OF FIXING VOLTAGE OR FREQUENCY MISMATCHES


VERIFY-SBI-AND-MB-DEVICE-ADEQUACY-3

    IF: THE MOST CURRENT ACTIVE CONTEXT IS VERIFYING SBI AND MASSBUS DEVICE ADEQUACY
        AND THERE ARE MORE THAN TWO MEMORY CONTROLLERS ON THE ORDER
  THEN: MARK THE EXTRA CONTROLLERS AS UNSUPPORTED (IE, NOT TO BE CONFIGURED)
        AND MAKE A NOTE TO THE SALESPERSON
                THAT ONLY TWO MEMORY CONTROLLERS ARE PERMITTED PER SYSTEM


ASSIGN-UB-MODULES-EXCEPT-THOSE-CONNECTING-TO-PANELS-4

    IF: THE MOST CURRENT ACTIVE CONTEXT IS ASSIGNING DEVICES TO UNIBUS MODULES
        AND THERE IS AN UNASSIGNED DUAL PORT DISK DRIVE
        AND THE TYPE OF CONTROLLER IT REQUIRES IS KNOWN
        AND THERE ARE TWO SUCH CONTROLLERS NEITHER OF WHICH HAS ANY DEVICES ASSIGNED TO IT
        AND THE NUMBER OF DEVICES THAT THESE CONTROLLERS CAN SUPPORT IS KNOWN
  THEN: ASSIGN THE DISK DRIVE TO EACH OF THE CONTROLLERS
        AND NOTE THAT THE TWO CONTROLLERS HAVE BEEN ASSOCIATED
                AND THAT EACH SUPPORTS ONE DEVICE


SELECT-BOX-AND-UB-MODULE-FOR-NEXT-SU-2

    IF: THE MOST CURRENT ACTIVE CONTEXT IS SELECTING A BOX AND A MODULE TO PUT IN IT
        AND THE NEXT MODULE IN THE OPTIMAL SEQUENCE IS KNOWN
        AND THE NUMBER OF SYSTEM UNITS OF SPACE THAT THE MODULE REQUIRES IS KNOWN
        AND AT LEAST THAT MUCH SPACE IS AVAILABLE IN SOME BOX
        AND THAT BOX DOES NOT CONTAIN MORE MODULES
                THAN SOME OTHER BOX ON A DIFFERENT UNIBUS
  THEN: TRY TO PUT THAT MODULE IN THAT BOX


PUT-UB-MODULE-6

    IF: THE MOST CURRENT ACTIVE CONTEXT IS PUTTING UNIBUS MODULES IN BACKPLANES IN SOME BOX
        AND IT HAS BEEN DETERMINED WHICH MODULE TO TRY TO PUT IN A BACKPLANE
        AND THAT MODULE IS A MULTIPLEXER TERMINAL INTERFACE
        AND IT HAS NOT BEEN ASSOCIATED WITH ANY PANEL SPACE
        AND THE TYPE AND NUMBER OF BACKPLANE SLOTS IT REQUIRES IS KNOWN
        AND THERE ARE AT LEAST THAT MANY SLOTS AVAILABLE
                IN A BACKPLANE OF THE APPROPRIATE TYPE
        AND THE CURRENT UNIBUS LOAD ON THAT BACKPLANE IS KNOWN
        AND THE POSITION OF THE BACKPLANE IN THE BOX IS KNOWN
  THEN: ENTER THE CONTEXT OF VERIFYING PANEL SPACE FOR A MULTIPLEXER


CHECK-FOR-UB-JUMPER-CHANGES-5

    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
        ASSIGN THE BC11A-10 TO THE BACKPLANE
        INCREMENT THE CURRENT LENGTH OF THE UNIBUS BY TEN FEET
```

FIG. 2.6. Sample rules from different contexts.

incompatible voltage or frequency requirements; if so, the rule generates the context of fixing the mismatches (which is done by isolating the set of components of the 'wrong' voltage or frequency and replacing them on the order with components of the right voltage and frequency). The second rule is

one of several rules that makes sure that various system limits are not exceeded; this particular rule is satisfied if there are more than two memory controllers on the order. The third rule assigns unibus devices to controllers; it is a special case rule since it deals with the case in which the device being assigned is a dual port device. The fourth, fifth, and sixth rules are all associated with contexts that become relevant while the unibus expansion cabinets are being filled. The fourth rule is one of several whose function is to select the next unibus module to configure and determine what box to put it in. The fifth rule tests for an exceptional case in the context of putting modules in a backplane; if the module that is to be put in the backplane is a multiplexer, then the context of verifying that there is panel space available in the cabinet is generated. The sixth rule checks to make sure that the jumper cable that connects two backplanes is of the appropriate length; this rule looks for the case in which the cable has to connect the second box in one cabinet with the first box in another.

2.5. R1's use of its rules

This subsection first provides some information from 20 sample runs to show how R1's use of its knowledge varies from order to order. Then an analysis of the extent to which each step in the task is conditional on the results of prior steps is provided. Finally, R1's use of its knowledge across the 20 runs is considered; this gives an indication of how much of R1's knowledge is 'core' knowledge and how much of it is 'exceptional' knowledge.

Each of the first 20 rows in Table 2.2 provides information about R1's performance on an order;[11] the final row shows the mean performance over the 20 orders. The column headed "Number of components ordered" shows the number of components that R1 configured. The column headed "Number of cycles" shows the number of rule firings required to configure these components. The first number includes the rule firings that generated output; the number in parentheses shows the number of cycles required excluding output generation. On 18 of the 20 orders, R1 had to generate only one candidate unibus module sequence. On each of the other two (orders 2 and 15), it generated three candidates; 281 rule firings were required, on the average, to generate each unacceptable candidate. The next two columns show the degree to which R1 used the knowledge it had available. The first of these columns shows the ratio of the number of distinct rules that fired to the total number of rules;[12] the other shows the ratio of the number of distinct domain specific rules that fired to the number of domain specific rules. The ratio of domain specific

[11]The trace of the run that configured one of these orders (order 7) is given in Appendix A.

[12]Since the 106 rules that generate output are invoked only after the configuration task has been done (and after all of the necessary domain knowledge has been used) they are excluded from consideration here and in Tables 2.3 and 2.4 as well.

TABLE 2.2. Statistics for individual runs

Order number	Number of components ordered	Number of cycles total	(domain)	Percent of rules used	Percent of domain rules used	Run time in cpu minutes
8	54	613	(476)	0.38	0.35	1.10
17	59	670	(517)	0.40	0.37	1.23
9	63	702	(541)	0.40	0.37	1.35
6	64	812	(657)	0.43	0.41	1.48
5	65	665	(502)	0.39	0.35	1.28
18	67	768	(592)	0.42	0.37	1.50
10	73	770	(643)	0.43	0.41	2.13
11	78	928	(743)	0.43	0.40	1.73
14	78	915	(706)	0.45	0.42	1.80
19	83	845	(642)	0.43	0.41	1.87
20	84	940	(734)	0.44	0.41	2.13
1	97	1113	(877)	0.46	0.44	3.27
2	99	1864	(1622)	0.47	0.39	3.90
15	99	1900	(1644)	0.48	0.41	3.80
3	102	1083	(829)	0.45	0.42	2.73
16	105	1065	(907)	0.47	0.44	2.28
13	107	1151	(905)	0.44	0.41	2.90
12	115	1167	(921)	0.43	0.41	3.07
4	129	1570	(1249)	0.53	0.51	4.75
7	142	1591	(1249)	0.53	0.51	5.20
Mean	88	1056	(847)	0.44	0.41	2.48

knowledge used to general knowledge used is about 2 to 1, which is approximately the ratio of domain specific knowledge available to general knowledge available. It should be noted that the number of distinct rules used does not grow directly with the number of cycles. On the average, for the orders that take the smallest number of cycles to configure, each rule is used twice; for the orders that take the largest number of cycles to configure, each rule is used five times. Only about 40 to 50 percent of the knowledge that R1 has available is required on any particular order (about 200 domain specific rules and about 100 general rules). The column that shows run time in cpu minutes includes the time it takes to generate output. The average amount of time required to generate output is 0.72 minutes; thus the average run time required to configure an order is 1.76 minutes.[13] The average working memory size during a run is about 500 elements. Although some elements remain in working memory after their usefulness has ended, R1 deletes elements from working memory

[13]The 20 sample runs were made on a PDP-10 (model KL).