Synthesizing computer configurations from specifications: structuring knowledge for design

Helen Lowe

DAI Research Paper No. 664

Synthesizing computer configurations from specifications: structuring knowledge for design *

Helen Lowe

Department of Artificial Intelligence,
University of Edinburgh

Abstract

We examine the task of synthesizing large computer configurations from specifications using a knowledge based system. We propose a new approach which breaks away from the traditional rule-based IKBS paradigm. The technique of proof planning is used to synthesize a computer configuration S from a specification $spec(S)$ by proving the theorem $\exists S.spec(S)$. Our representation gives a clean separation of the different types of knowledge: object-level, heuristics, and control, and greatly facilitates the exploration of explicit design issues in this domain.

We overview the design issues to be taken into account, concentrating on cost and efficiency. We conclude that the task cannot be viewed as either an optimization or a purely constraint driven problem, and that design choices must be preserved and not constrained in the finished configuration. Our prototype system allows this.

*The research reported in this paper was supported by a SERC CASE studentship with Hewlett Packard Research Labs, Bristol.

1 Motivation

This paper examines the issues involved in acquiring and structuring the knowledge needed for designing a computer configuration which meets a given specification.

We define a computer configuration as consisting of components such as processor, memory modules, terminals, printers, disk drives, and so on, which are connected together in some way so that the whole functions as a 'legal', working configuration.

Specifications, in general, may range from a detailed list of components to those which are expressed in terms of fairly high level goals for the configuration.

There are various motivations for seeking to automate the design process. Apart from the obvious ones, for configuring hardware at least, of saving effort or cutting costs, it is intrinsically interesting to discover whether the elusive processes of human reasoning can be understood enough to allow simulation by program.

At first sight, it seems very difficult to formalize a domain such as configuration. We need to draw on a large amount of knowledge from a variety of sources, such as the accumulated "on-the-job" experience of sales representatives; price lists, manuals; and so on. Manuals are often incomplete or ambiguous, and experienced sales representatives do not necessarily make good teachers. Generally speaking, expert systems, supposedly based on the distilled knowledge of experts, are *not reliable* (Bundy, 1987). However, by capturing knowledge, testing this knowledge in its captured form, and weeding out errors in a methodical way, we *can* hope to end up with a robust system which *cannot* produce erroneous results.

Capturing expert knowledge is difficult. The expert can take short cuts; some knowledge appears to be "hardwired" and is used automatically and non-introspectively when it is appropriate. For example, an expert may adapt last week's order for this week's specification, rather than start reasoning about the current task from scratch. Moreover, the expert is also adept at cutting through the search space to find not just *legal*, but *good* — or *well-designed* — solutions. In automating the task, we also need to find some objective measure of what a "good" solution is, and effective control mechanisms to ensure that we generate good solutions in preference to less good solutions.

2 The configuration task

2.1 Early work

An account of the original R1 system for checking orders is found in McDermott (1982), while later updates of XCON (as R1 became later known) and XSEL (a companion order generator) are in Bachant & Soloway (1989) and Barker & O'Connor (1989). Soloway *et al.* (1987) give an account of how XCON was reimplemented in RIME.

XCON was the original customer order *checker* — it takes an order for a configuration (*i.e.* a list of components) and attempts to configure them, adding any parts which are necessary but which have been omitted from the order, and making other corrections. XSEL works at a higher level, taking account of customer requirements to assemble an order for hardware components, and is interactive, in that the user is prompted for components; the system cannot generate them in any way. At the end of the interactive stage of the process XSEL passes an order to XCON.

XCON has a reputation for being difficult to maintain. Circumstantial evidence is that in the published accounts (Barker & O'Connor, 1989) the problem of training personnel to work on XCON is prominent; the solution proposed is to send all new staff on an AI course on entry.

2.2 Towards a theory of configuration

I am sceptical about claims that we can establish a general theory of configuration; the knowledge required for a powerful system seems very domain-dependent. However, it would be good to establish some generalities about configuring computer hardware systems: this is at least a desirable, as well as a realizable aim. For example:

- There are certain things which form an intrinsic part of *all* computer systems for the foreseeable future: processors, memory, storage of some kind, devices for communicating, and for producing hard copy. They may, of course, look very different in the year 2050 from what they do now.
- All devices have to be connected up — somehow. Ever faster, more compact, more versatile ways of doing so will, no doubt, be found in the future.

In order to achieve the highest level of generality, the greater the degree of abstraction possible the better. Where I can talk about "storage devices", rather than about disk drives, tape drives, *etc.*, I do so. If it is appropriate, I shall use abstractions rather than specific names — "connection" rather than "RS-232 cable", "modem", *etc.*

2.3 Definitions of configuration

Computer configuration in many ways is an ill-defined task. Folklore might have it as an art rather than a science. In rejecting this view, I need to be able to describe and define certain concepts accurately.

The configuration task has evolved into a very different problem than that addressed by the early configuration experts as implemented in XCON by McDermott (1982). Changes in the nature of the configuration task mirror the changes in technology which have, over the last two decades, obviated some parts of the task and invented new ones. A study of this period shows how technological advances cause economic changes which have profound repercussions on work activities. Our expectations of what can be achieved can be so radically altered by leaps in technological innovation that the whole basis of how tasks are performed is transformed. Cheap hardware (and relatively expensive labour) have changed the configuration task from one in which the aim is to fulfill needs, stated implicitly as lists of components, as cheaply as possible, into one where we want to be able to explore various design issues. In short, it has changed from a *highly constrained, optimization* problem into an *underconstrained, design* problem

With the advent of cheaper processing power and hardware components, modern computer systems tend to be "over-engineered", in the sense that the basic "frame" of a system has considerable scope for expansion built into it. Thus a configuration built around a particular processor will have ample sockets and extension points to allow expansion of the system to be carried out. Thus we say that the problem of configuring a computer system to meet requirements is somewhat under-constrained. A major effect of this technological advance has been to render redundant much of the detailed decision-making which originally formed a large part of configuration procedure. These decisions include those which determine the exact positioning of components in sockets (slots or ports). If not completely unnecessary, it can at least be said that this procedure now occupies a minor role as compared with, for instance, determining which of many design principles should be followed.

2.4 Modern configuration systems

2.4.1 Constraint-driven systems: COSSACK

Expert systems developed to deal with the more modern concepts of configuration are more prototypical than XCON, but the task definitions are more explicit. The authors of COSSACK (Frayman & Mittal, 1987) define the configuration task thus:

> "Given a fixed, pre-determined set of components, an architecture that defines certain ways of connecting these components, and requirements imposed by a user for specific cases, either select a set of components that satisfies all relevant requirements or detect inconsistencies in the requirements."

Configuration is here thought of as a special case of design. In fact, it falls within the category given in Chandrasekaran *et al.* (1987) as "object synthesis by plan selection and refinement", whose task specification is to "design an object satisfying specifications (object in an abstract sense: they can be plans, programs, *etc.*)" The restriction over design activity in general is that the components are pre-defined and can only be connected together in certain ways. Note that a configuration is not simply a set of components, but must also include (somehow) information about how these components are to be joined together.

Frayman & Mittal have developed a knowledge based system known as COSSACK for configuring XEROX PCs. The configuration task is restricted in that artifacts are configured according to known functional architectures: those permissible for a Xerox PC are explicitly represented and COSSACK instantiates one of these using a set of standard components, building up the configuration around *key components* — the devices of the configuration — which act as "planning islands" of *almost* independent subsystems. There is still some interdependence, however, in that devices share components, and each device may perform more than one function.

The basic strategy is to start with the functions required (*e.g.* printing, to a certain specification) and map these on to key components (*e.g.* a specific printer). This component "posts" requirements which either call for further functions or else places constraints on existing ones. The second stage is to seek a consistent connection of the components. These stages can be interleaved to prevent expensive backtracking. Experiments have been tried: using heuristics, and the use of *partial solutions* — *i.e.* sets of components which fulfill a function rather than picking one.

COSSACK is a system strongly reminiscent of MOLGEN (Stefik, 1981) which essentially treats configuration as a constraint-driven problem. As such, it is eminently suitable for hardware which which tailored very specifically to the specification, but not to hardware which is in some sense "over-engineered": these modern architectures aim to provide scope for expansion and hence tend to be highly under-constrained. The examples given in the paper cited above are of relatively small computer configurations only. Since no examples of larger hardware configurations are given, it would appear that the search space associated with a system in the style of COSSACK would be too great, and the number of solutions generated too large, for tackling problems of such an order of magnitude.

2.4.2 Logic based approaches: BEACON

BEACON (Searls & Norton, 1990), based on a semantic network, and implemented in Prolog, configures Unisys computer systems. This implementation was aimed at utilizing a more declarative approach to the configuration task, in contrast with the systems we have seen so far. The authors concern themselves with maintainability as it was apparent that this was the weakness of XCON, then still apparently seen as the torchbearer for automated configurers.

Searls & Norton recognize the need for what they call *functional* configuration — configuring to specifications couched in terms of higher level goals than merely what devices are needed, for example the requirement that there should be enough disk space to support the administration needs of a typical 200-bed hospital. They also call for the task of the sales representative and the task of the engineer to be integrated, rather than separate as is (still) common practice. The latter was the main contribution of BEACON, which is an interactive configurer where the constraints are dealt with automatically and the user presented only with feasible (legal) options. It is not clear, however, how the functional approach at the high level suggested should be implemented.

BEACON uses the semantic network KNET and an interactive ordering system. The user is presented with choices, all of which are guaranteed correct by the underlying configuration rules used in drawing up these choices. The inference strategy used is in the spirit of Prolog-style search. As the process continues, a configuration is built in a manner analogous to the instantiation of variables in a Prolog-style proof.

The approach is a useful one, in that it is logic-based, and the constraints are integrated into the logic so that, firstly, the user is never presented with illegal choices, and secondly that there is no expensive Prolog-style backtracking — which would be the case if the constraints were checked after the components had been chosen.

However, the examples given for the performance of this system are of configuring microcomputers and mini-computers which come in a limited number of sizes. It does not seem likely that this system will scale up without some kind of heuristic guidance.

2.5 Needs of a modern configuration system

Our aim is to provide a configuration system which will deal with large hardware systems, *i.e.* typically under-constrained systems in which design issue play a large part. We need to be able to cope with the large search space associated with modern hardware systems. We shall use a formal object-level representation to ensure the *legality* of the computer systems synthesized. Separate control mechanisms will maximize the chances of finding a path through the search space to solutions which are *preferable* in some way. Strategies for *design* can be absorbed on an incremental basis.

The ideal is a system which could combine logical soundness with a modern approach to the task, and still, potentially, prove as useful as XCON.

3 The process of hardware specification

3.1 Informal assessment of needs

Customers for computer hardware generally think in terms of the use to which the system is to be put, and they will go through the process of discussing customer needs with the sales representative for the computer firm. Issues include the applications to be run, the number of users involved, and possibly other considerations such as cost, and the future needs of the company as regards expanding or upgrading any hardware configured.

Sometimes there will be a formal *invitation to tender*. Here, the customer presents the requirements in written form and computer firms are invited to submit tenders, detailing the hardware to be configured, itemized costs, and indicating parts of the configuration which either fall below or exceed the specification, with reasons.

A typical (but fictitious) specification drawn up at the invitation to tender stage is shown in Figure 1.

3.2 Choice of processor

The customer may or may not specify the processor to be used. Of course, in the case of general invitations to tender the customer *cannot* specify a processor since the invitation to tender specifications are general and aimed at competing hardware firms. However, it is usual for the representative of the tendering firm to decide on the processor first. To do this requires certain domain knowledge and the ability to assimilate salient facts from the specification as a whole.

- 70 screens for data entry
- 5 screens for program development
- 20 screens for electronic mail and spreadsheet application
- 1 line printer with a speed of at least 900 lpm
- 2 laser printers, each with a speed of at least 45 ppm
- 1 laser printer with double-sided printing option
- approximately 1.7 GBytes of mass storage

Figure 1: Invitation to tender specifications

3.3 Devices

3.3.1 Disk storage devices

Disk storage requirements may be specified at various levels. The total disk storage capacity required is given, leaving the exact configuration of drives unspecified. Alternatively, disk drives could be explicitly specified, *e.g.* ten model *example_ddrive* drives. A third way is to give total capacity together with some other attributes, for example the type of disk drive, (*e.g. fibre-optic* disk drives); the access speed required (this may be a qualitative measure rather than an explicit figure); the minimum (or maximum) number of disk drives to be configured (*e.g.* that we require at least two disk drives).

It is possible that, in specifying the disk storage in any of the ways above, we might want to specify *user* disk storage separately from *system* disk drives.

3.3.2 Terminals

The terminals of a configuration could be explicitly specified: *i.e.* that we require n_1 of terminal model *term_1* and n_2 of terminal model *term_3*. However, it is more usual to specify terminals in terms of applications, or simply to state the total number of terminals required for the computer system.

3.3.3 Printers

The same remarks apply for printers as for terminals; in addition, it is possible, as in the example of Figure 1, to specify each printer required by its attributes. For printers, these attributes can be: *speed* (lines per minute; characters per second); a measure of the robustness of the printer in terms of the volume it can cope with, known as *throughput* (pages per month); availability of certain features like the ability to offer double-sided printing, high quality printing, *etc.*

3.3.4 Tape drives

Tape drives in a computer system have two main uses: backup, and transfer (of data, software, *etc.*). If backup is the only consideration it is unusual for the tape drives to appear in the specification. Instead, the tendering firm is expected to assess the backup needs of the configured system and suggest suitable tape drives. The main consideration then is the disk storage capacity of the system to be backed up. Some of the disk storage will be used for data which is never backed up: for instance, swap space. The total disk storage capacity therefore gives a convenient upper bound for the amount of backup capacity required; the suitability of various tape drives for various storage capacities is a known attribute of the drives.

3.4 Memory

It is unusual for the amount of memory required to be specified. In particular, the amount needed depends partially on the operating system used, which in turn is dependent on the processor used, amongst other factors. So a customer inviting tenders is unlikely to be able to specify memory and, as for tape drives, will expect the tendering firm to work out how much memory is needed.

For HP systems, each processor has a formula

$$memory\_capacity = F + M \times N$$

where F is a fixed amount depending on the operating system used by the processor, N is the number of concurrent active users (or the maximum of this, if it is variable), and M is a number in a range $[\alpha, \beta]$, whose position in this range is a function of the applications — some applications requiring more memory than others. Actually, "function" is probably an over-precise term for what is, essentially, rule of thumb. Since memory boards come in discrete sizes (1Mb, 4Mb, *etc.*), as long as we end up in the right band, further accuracy is unimportant.

3.5 Tendering considerations

In the tendering situation, each computer firm is in competition with others. However, this is not the straightforward situation where the lowest bid automatically wins. The specification is, to an extent, open-ended and it is up to the tendering firm to state what, exactly, is being provided for the money. The aim in tendering of this nature is to provide the best value for money, compared with the competition.

There are several points of interest here:

1. There are a very large number of possible configurations which will meet the specification given.

2. They are not all of equal merit.

3. There are a number of dimensions on which to judge solutions: for instance cost, performance criteria, expandability.

4. Because of this there is not, in general, a *global optimum* solution (see Section 5.4).

5. We would hope the system would discard any solutions which were "locally sub-optimal" (see Section 5.4); *e.g.* we could find another solution which gave better performance, cost and all other criteria being equal.

We want to be free to reason about how to arrive at locally optimal solutions safe in the knowledge that the underlying object theory of configuration is sound and complete.

The expert configurer, confronted with a specification such as the example given in Figure 1, apparently makes a leap from the user requirements to a processor (or a candidate processor). For example, for the requirements given in Figure 1, the number of users, combined with a high disk storage capacity, point, for the experienced HP sales representative, to a HP 950 processor. On the other hand, the processor may be specified from some other criterion; for example, if we know that the configuration will be upgraded in the future we might decide to specify a processor more powerful than that suggested by the other user requirements explicit at the time. Thus we see that the processor may be either explicitly specified, or else inferred from other relevant information.

4 Constraints

4.1 Hard constraints

We can draw up a set of rules which *all* configurations must follow, in order to be counted as *legal*. For a given configuration, a general rule, such as that a device must be connected, is translated into more particular rules: that it must have a cable of a compatible type, and that the other end of this cable must be connected to a port or a channel, again of a suitable type. There are constraints on which cables may be used with which devices, and on the connection method generally. For example, in some configurations a terminal will always be connected via a cable to a MUX (multiplexor: a device allowing the processor to be connected to a number of communication channels); the MUX interface resides in a card cage slot. In others, it is more complex: a terminal must be connected via its cable or a modem to a port; ports are configured in groups of six or eight on DTCs; and DTCs are configured on a LANIC (local area network interface card), whose interface occupies a card cage slot.

The choice of processor constrains what can be configured. Thus each processor is associated with a template. The choice of processor determines the template: the kinds of devices which may be supported, and the connecting components which are available; and imposes various constraints, such as the number of extra card cages which may be added, and the number of interface cards which may be configured in each card cage.

We call the various constraints which take the form of limits on the numbers of components of one type which may be associated with components of another type *limit constraints*. Here are a few examples:

1. A 950 processor must have at least two card cages.

2. A 950 processor may not have more than four card cages.

3. A card cage in a 950 system has slots for up to five interface cards.

4. An HP-IB channel may not have more than four disk drives connected.

5. An HP-IB channel may have up to six devices (disk drives, tape drives, and printers) connected in total.

There are also constraints to express that certain devices are incompatible: for example, in 950 configurations it is stated that the system disk drive (used to store the operating system) may not share a channel with the system backup tape; in older systems systems certain devices are designated as high speed devices and may not be configured alongside low speed devices. I shall refer to these henceforth as *compatibility constraints*.

However, note at this point that these rules are not sufficient in themselves to ensure that the configuration will satisfy all the needs of the customer/user. They ensure merely that the configuration can be assembled and that certain minimal requirements are met, such as there being a console to enable communication with the system, storage for the operating system, and a tape to enable software to be loaded and back up data. They do not ensure that the users will be able to run their software efficiently, or indeed at all. The related question, of achieving "better" systems, or just "adequate" systems will be addressed in Sections 4.2 and elsewhere in this paper.

4.2 Meta-level heuristics

4.2.1 Strengthening constraints

Some heuristics are, in effect, a tightening or strengthening of existing rules. They all use heuristic knowledge about what will ensure an *efficiently running configuration*, as opposed to a simply *legal* one. For example,

1. A card cage in a 950 system has slots for up to *five* interface cards.

2. An HP-IB channel should not have more than *four* disk drives connected.

3. An HP-IB channel should have no more than *six* devices (disk drives, tape drives, and printers) connected in total.

These can be thought of as "hard constraints" — the limits stated may not be exceeded in any circumstances (it is probably physically impossible to exceed them; for example, if there are only five places to insert cards in a card cage). However, we have these corresponding 'soft' constraints:

1. The *average* number of interface cards per card cage in a configuration should not exceed *three*.

2. An HP-IB channel may not have more than *three* disk drives connected.

3. An HP-IB channel may have up to *five* devices (disk drives, tape drives, and printers) connected in total.

We can think of *hard constraints* as being the rules which *all* configurations *must* obey, and of *soft constraints* as being rules which *most* configurations *should* obey, *if possible*.

Note that Number 1 above is rather strange and will be explained in Section 5.5 — it is an example of "compiled knowledge": compiled in the brain of the expert configurer, that is.

4.2.2 Acquired knowledge

Sometimes soft compatibility constraints are discovered during the lifetime of a component. For instance, there is a certain model of printer which, when connected to the same channel as a disk drive, causes the computer system to run slowly if both are operational at once. The soft constraint states that this model of printer should not be connected to any channel which has disk drives connected to it.

This type of heuristic information tends to be accumulated over the lifetime of a system product. For example, in the example above, the knowledge that this particular printer caused problems might only be known after feedback from engineers and customers. A new model of printer might come on the market whose performance in certain situations had not been predicted. We can lose these heuristic constraints too — for example, as a result of feedback from customers and engineers, a component may be altered to obviate the problem. It is almost as important that such constraints can be removed when they are no longer appropriate as it is that they are incorporated in the first place.

5 Search issues

The problem of configuring a modern computer hardware system to meet user requirements is, in general, sufficiently under-constrained to be problematical. Specifications are often expressed in terms of the functionality of the configuration, and there may be many alternative sets of devices, all of which give the desired functionality. Moreover, there may be many ways of connecting a given set of devices into the configuration, Some of these alternatives will be trivial permutations of others; other alternatives may affect the performance of the system, or its cost.

I can try to constrain the search in these ways:

1. Trying to apply all the soft constraints, if possible.
2. Suppressing trivial alternative solutions.
3. Attempting to return well-designed computer configurations.
4. Attempting not to return locally sub-optimal solutions.

I address each of these in turn in the sections which follow.

5.1 Avoiding trivial permutations

If a solution is found, we do not wish to consider, as alternatives, solutions which are in some sense permutations of the first. An example will illustrate this point.

Suppose we have a simple configuration, consisting of two card cages, two channels for configuring disk drives, tape drives, and printers, and two such devices: a disk drive and a tape drive. Since the disk drive is to be used to hold the operating system and the tape drive will provide back up facilities, we have a rule which states that we cannot configure both these devices on the same channel. A solution is found which configures the disk drive on channel 1 and the tape drive on channel 2. We are not then interested in the permutation which configures the devices the other way round.

5.2 Applying soft constraints

The principle of applying soft constraints is good in cases where it is possible to find solutions which obey all the soft constraints. If we can find such a solution, where all the user requirements have been met and, in addition, we have been able to apply heuristics which ensure that the configuration will run efficiently under every conceivable circumstance (this heuristic is the principle "apply soft constraints everywhere"), then we have no wish to return solutions which do not meet this standard. In these circumstances, we would wish to prune the search tree so that only the superior solutions are found.

However, it is sometimes the case that if we carried out such a pruning, we would prune all branches leading to solutions from the tree. In this case, we have to be able to dynamically relax one or more of the soft constraints, on one or more occasions. This is dealt with in Section 5.5.

Returning to the example in Subsection 5.1, suppose that in addition to the hardware mentioned there we have a printer of the type mentioned in Section 4.2.2: this was the one which could sometimes cause problems if connected to the same channel as a disk drive. We prefer, if possible, that this printer should not share a channel with the disk drive, but since this is a soft constraint we are prepared to relax it if necessary. In this case, it is not necessary: we can configure the disk drive on channel 1 and the tape drive with the printer on channel 2. We would wish to suppress the trivial permutation solution of configuring the disk drive on channel 2 and the other two devices on channel 1.

Note also that we would *also* wish to suppress the solution of configuring the disk drive and the printer on (say) channel 1 and the tape drive on the other channel, since it is an inferior solution and we have already found a better one. The issue here, however, is achieving a local optimum, and this is dealt with in Section 5.4 below.

5.3 Achieving good designs

Issues of what constitutes good design are subjective up to a point. I will assume only the following principle in incorporating design decisions:

- There are various "rules of thumb" — heuristics — which can be applied. We shall built these in as defaults.
- If the format of specifications is left sufficiently flexible, it will be open to the user of the system to override the defaults.

An example will illustrate these points.

1. Suppose we specify a total disk storage capacity of 3.3Mb. We could achieve this by having seven 7937H model disk drives, or by having nine 7933H disk drives. The former would be better by the rule of thumb which says "Fewer,

higher capacity disk drives are often better than more, lower capacity disk drives". Both would be acceptable, but we would hope that the first solution would be returned in preference to the second.[1]

2. In *some* circumstances we might want more disk drives (of smaller capacity). This could be because we wanted to increase the number of communication channels if our applications involved a lot of disk access. We can state this in the specification, for example by giving a minimum number of disk drives for the configuration; if this was (say) eight in the example, then the first solution will not be admissible but the second will.

3. There are many other solutions possible. However many of them would be considered poor design: for example, mixing different models of disk drives by having (say) four 7937H disk drives and three 7933H drives. But note that the user can still specify such (completely *legal*) combinations explicitly — this is the second specification option of Section 3.3.1.

5.4 Ensuring locally optimal solutions

Let us start by defining the concept of *utility*, which we shall encounter here in determining what, if anything, would constitute an optimum solution:

Definition 1 *The* utility *of a system is a multi-dimensional measure of its worth to the user. These dimensions may vary, but at present I shall enumerate them as*
(1) cheapness (inverse of cost)
(2) efficiency
(3) expandability
(4) anti-obsolescence (inverse of obsolescence)
(5) technological innovation

[1]The second would still be found, if more solutions are asked for.

These dimensions are named so that a move in the direction of the positive axis constitutes "improvement". However, I shall usually refer to *cost* rather than *cheapness*, and to *obsolescence* rather than *anti-obsolescence* in discussing configurations, since these are more natural.

Note that:

- *Obsolescence* is a property of *old* components which are likely to be discontinued (and therefore, ultimately, unsupported).
- *Technological innovation* is a property of *new* components in the process of becoming supported, which (it is hoped) represent the new "state-of-the-art" to come.

It is hard to address the question of what constitutes an *optimum* solution in a domain with several dimensions. On the one hand, we can *measure* the cost of a system: this is an objective, quantitative property. On the other hand, it is not true to say that we can *optimize* on cost considerations since cheaper solutions may be inferior in some respect to more expensive ones. We can talk about the efficiency of a system, but this is not so easily measurable. Also, two identical instances of configurations may differ in their utility, one running very efficiently and the other poorly — simply because they are running different applications, or are being used differently. In Section 4.2.1 we saw that there is a soft constraint to restrict the number of disk drives to fewer than the number given by the hard rule for the configuration. Suppose we have three identical configurations, each violating the rule. These configurations are part of *systems* (where a *system* includes the configuration, the software it is used to run, its environment, which includes the pattern of usage — interactive, batch, overnight, *etc.*) such that $\mathcal{C}_1$ runs inefficiently whereas $\mathcal{C}_2$ and $\mathcal{C}_3$ do not. How can this be? Suppose configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ both run a set of applications $\mathcal{S}_1$, and configuration $\mathcal{C}_3$ runs a different set of applications $\mathcal{S}_2$. $\mathcal{C}_1$ runs inefficiently, due to the fact that the applications it runs involve high disk access. It can be the case that $\mathcal{C}_3$ runs efficiently because

$\mathcal{S}_2$ is not involved in much disk access. $\mathcal{C}_2$ may be acceptable because its high disk access applications are run overnight, and it does not matter if they run slowly.

Thus it is not possible to address the question of optimality at a global level, as is acknowledged in Frayman & Mittal (1987), who also address this issue. However, we wish to ensure that our solutions are at least *locally* optimal. I define this concept thus:

Definition 2 *A* locally optimal solution *is one whose utility cannot be increased (improved) along one dimension without a simultaneous decrease (deterioration) along at least one other dimension.*

We can compare solutions of equal cost, expandability, obsolescence, and technological innovation — probably solutions using an identical set of components but connected differently — along the dimension of efficiency, and ensure that we reach one of the better solutions possible for that set of components. To this end, in the example of Section 5.2 we rejected the solution of placing the printer and the disk drive together because this was a less efficient solution, despite having the same cost as our favoured solution — note that it must necessarily have equal cost since it uses exactly the same components.

5.5 Soft constraint relaxation

The need to relax soft constraints arises when applying all of the soft constraints would leave the solution space completely empty. Unfortunately, constraints interact in such a way as to make this quite a difficult problem. Expert configurers tend to use rules of thumb based on experience coupled with their understanding of why the heuristic rules are there in the first place.

Let us give an example. Suppose we have four card cages and cannot configure any more, since this is the maximum allowable by the (hard) rules for this particular configuration. Suppose we have a total of twelve interface cards configured in these card cages. Of these, nine are HPIB channels. Suppose now that we have twenty-eight disk drives to configure.

Referring back to the hard and soft constraints given in Section 4.2.1, we see that these combine, given that we have four card cages, to:

- Hard constraints
  1. No more than seventeen HPIB interface cards in total
  2. No more than four disk drives to an HPIB
- Soft constraints
  1. No more than nine HPIB interface cards in total
  2. No more than three disk drives to an HPIB

While obeying both the soft constraints we can configure at most $3 \times 9 = 27$ disk drives. To configure the 28th we must relax *either* soft constraint (1) or soft constraint (2). It is not clear which would be preferable (or least unacceptable).

Relaxing constraint (1) will overload the card cages. Relaxing constraint (2) will overload one of the channels. Both of these have an effect on the efficiency of data transfer, in general. However, the actual effect of overloading in each of these ways depends on the particular extent and pattern of data transfer, and this in turn depends on three factors:

- The amount of disk access.
- The timing of disk access.
- The distribution of the data across disk drives.

So, for example, we would be prepared to relax constraint (2) if disk access were comparatively infrequent for the disk drives involved.

Soft constraint (1) is interesting. It was given to me in the course of knowledge elicitation but turned out to be "compiled knowledge". It is an example of a grossly over-simplified (in fact distorted) version of a rule given by people who *do* understand configuration to those who they believe are not capable of such understanding: the idea being that if the sales representatives blindly follow this rule it will at least keep them out of trouble. Its real *raison d'etre* is that the engineer who installs the configuration should be free to permute channels between card cages for maximum efficiency. It has not been possible, so far, to capture this knowledge, and indeed there are not many cases where it is needed. It is true that it is desirable, where possible, to allow the engineer who actually installs the configuration this flexibility, so that she can place interface cards in the optimum positions for efficient data transfer. However, there are certain patterns of system use which render it irrelevant, and therefore a constraint which can be readily relaxed. The motivation behind this heuristic is explained as: if there are not too many interface cards jockeying for position, then it is more likely that we can ensure that there are not too many of a particular type on one card cage. If the average rises above a certain level, then it is less likely that this can be done. However, by this formula the precondition for applying the heuristic has been lost. This seems to be a common problem in knowledge elicitation for AI, and for expert systems in particular.

6 Design issues

Much guidance in searching for a configuration can be provided if the user of the system has facilities for scene setting; *e.g.* stating whether an efficient configuration is important, whether there is a cost constraint, *etc.* There is a small subclass of design problems in which *the* 'optimum solution' (along one of several possible different dimensions: cost, efficiency ...) which meets the specifications is required. The underlying strategy to be used in this design scenario is quite different from, say, that to be used if the problem is to find any satisfactory solution(s) subject to keeping under a cost ceiling. As well as cost and efficiency, other design issues are *expandability*, or potential growth path, of a configuration, *i.e.* its capacity to be augmented at a future date; and technological issues, such as whether certain hardware is obsolescent and will be superseded in the near future; and anticipating new trends in hardware.

6.1 Constraint management and optimization

I believe that synthesizing computer configurations is a question of *design*, where some human intervention is necessary. At the very least, with the minimal human input, the user should be provided with a set of solutions to choose from.

Not everyone sees the problem this way. Some assumptions that are commonly made are

1. That the problem can be viewed best as one of constraint management.
2. That the problem can be seen as one of optimization.

Other systems, such as that of Frayman & Mittal (1987) use the constraints to "drive" the inference process in a MOLGEN-like system. However, when I examined the rule-base for the configurations I was considering, I found that they were not constrained to anything like the extent of the configurations described in Frayman & Mittal (1987). In all realistic examples, the solution space was very large, if I took *legality* of configuration alone into account. Applying soft constraints could prune the search tree, but could also result in no solutions being found at all. Heuristic constraint management is a complex process, especially if the motivation behind the heuristics is ill-understood. In a typical negotiation between a computer hardware firm and a customer, the firm aims to provide a solution, or a small number of alternatives, which both meets the user requirements and offers competitively priced facilities, including the hope that the configured system will run the users' software in a reliable and efficient manner.

Let us examine criteria by which a solution may be judged good or bad, and assess what optimality means

- for each criterion separately
- for *all* criteria considered together

6.2 Cost

Sometimes a ceiling is placed on what may be spent on a configuration, in which case this forms part of the specification and represents a constraint on the solution space. Even if there is no such explicit constraint, I would not usually regard cost to be completely unimportant.

It is often assumed that minimising cost is a desirable goal. This is something of an over-generalization because

1. Often cost is of little concern, provided the budget ceiling is not exceeded.
2. It may be desirable to *meet* (exactly, if possible) the budget ceiling, if there is one. Often capital which is allocated but not spent is not transferable, so that the department concerned loses out; and also risks its budget being cut for the next financial period.
3. In the common situation of tendering for a contract, it is of course important to offer a competitively-priced configuration, but not at the cost (necessarily) of cutting corners.

At best, therefore, I can *sometimes* see the need for a strategy being designed to achieve the "cheapest, all other factors being equal" configuration. I move on to a discussion of these "other factors".

6.3 Efficiency

Sometimes a configuration, while legal according to the rules in the object level knowledge base, does not run very efficiently in practice, because of the way in which the components are connected. For example, it may take a long time to transfer data to and from disk, or to print out documents, or to run compute-bound programs, or all of these. This issue was explained in Section 5.5. Additionally, a configuration may be more or less efficient, according to the choice of devices. For example, a disk drive with cache memory is faster than one without.

To appreciate the problems in trying to find 'optimum' solutions, there are a number of related points to be made.

- A configuration is *potentially inefficient* if the *heuristic* maximum (of disk drives per channel, of interface cards per card cage) is exceeded.
- If it is possible to breach the soft limits by varying amounts, then one assumes that the greater the breach, or number of breaches, the more inefficient the configuration, but...
- ...Provided the soft limits are *not* breached, a configuration is not made *more* efficient by further decreasing the number of components involved.
- Given a choice between breaching one soft limit or another, in the absence of evidence to the contrary we cannot *prefer* breaching one to breaching another; this is a genuine choice point in the search for an efficient configuration.
- Further, we cannot necessarily say that two breaches of one soft limit is worse than one breach of another.
- It *may* be the case that we would want a strategy for ensuring the least total number of breaches; however we might want it to be possible to return other solutions as well. The same applies to other strategies: ensuring the least

possible breaches of soft limit A, or ensuring the least possible breaches of soft limit B, *etc.*

What seems to be desirable is that there is some means of capturing each alternative strategy, a search procedure which gives alternative solutions based on each. Ideally we might want to allow some kind of interactive facility for choosing between them. We shall return to this in Section 11.

6.4 Reasoning with cost constraints

Suppose we have a cost constraint. Our strategy can be summarized as:

- If we can obey all soft limits, then we shall do so.
- If not, then we would want configuration with *maximum utility* (see Definition 1) subject to the cost constraints.

1. If there are two solutions which are within cost but each breaks (say) one heuristic — a different one in each case — *someone* needs to decide which of the two is acceptable/better.
2. Two solutions, where one has two breaches of soft limit A and the other has one breach of soft limit B, may not be comparable.
3. We may have to decide, say, between having faster (and more expensive) disk drives which are connected inefficiently, and slower disk drives which can be connected according to soft limits.

6.5 Reasoning without cost constraints

Just because we have no cost constraint does not mean that cost is unimportant.

1. It may be that we have to break soft limits designed to *ensure* efficient configurations anyway, because of the sheer size of the configuration. Then the same reasoning as in Section 6.4 holds.

2. If this is not the case, then a solution which does not breach soft limits may be found. However, we may be interested in returning other solutions as well, or in finding a minimum cost solution, for the purposes of negotiation. This is particularly true in tendering situations.

6.6 The problem with metrics

I do not believe that it is possible to develop metrics in order to decide a global optimum configuration for a given specification. Let us examine the reasons for this conclusion.

It might be argued that we *can* attach metrics (or measures) of how 'bad' a given breach is. Then the strategy should be to find an 'optimum solution', which mimimizes the 'total score'. There are several problems with this.

- It is unlikely that an expert configurer could give figures which would hold across all situations. How 'bad' it is to overload a channel with disk drives depends on the pattern of disk access, which depends on users, applications, and general environmental factors for the system.

- It is equally difficult to compare different 'bad' things. Sometimes we can put up with overloaded card cages but not with overloaded channels; sometimes it is the other way round: it depends on our ideas about the likely pattern of data access/transfer.

This problem is well known from cost-benefit analysis, for example, Ball (1979) writes:

> "The official figure in Britain for the value of a life in the early 1970s was around £9000. The absurdity of such a number is readily apparent. A plane carrying 100 passengers is, on this evaluation, transporting a cargo 'worth' £900,000. The jet aircraft, on the other hand, has, say, a replacement value of £2m. Imagine that this aircraft, while airborne, suddenly develops mechanical failure. The pilot is faced with the choice of irreparable damage to the aircraft in order to save the passengers, or jettisoning the passengers in mid-flight in order to save the plane. The pilot consults the cost-benefit manual to find out what society would prefer. After safely landing the now lightened aircraft, the pilot explains the absence of passengers by reference to the mechanical difficulty and the cost-benefit manual. Heroes, it seems, can come in many different guises, particularly if you believe in cost-benefit analysis."

So I would argue against hard-wiring any optimality strategy into the system: instead I need several strategies for synthesizing configurations which may possibly be less than efficient. Flexibility is the key. I will demonstrate how this flexibility can be achieved in Section 7.2.

Simply expressed, the task that I have set out to automate is that of synthesizing configurations which meet the specifications given. Search is constrained by the need to generate *legal* configurations, and by constraints which may be present in the specification, such as the need to keep within a budget.

There appear to be three facets to the task:

1. Capturing the hard knowledge that we have, *i.e.* the object-level theory of configuration. Finding solutions according to these rules ensures *legal* solutions.

2. Capturing heuristic knowledge. These are definite facts about what constitutes a *good* solution.

3. Experimenting with strategies which aim to return solutions which are *as good as possible.* This area is more fuzzy — hence the need to experiment.

Section 7.1 deals with the issue of separation of knowledge, and Section 7.2 describes how this knowledge is then put to use in proof plans for designing configurations.

7 A new framework for IKBS

7.1 Separation of knowledge

Knowledge can be:

1. Factual:
   - The total capacity of a string of disk drives is equal to the sum of the capacities of each disk drive in the string;
   - Every device in a configuration has a means of connection in the configuration.
2. Heuristic:
   - Evaluation functions for heuristic search (*e.g.* for hill-climbing)
   - Knowledge about potentially incompatible devices in configuration. If devices are known to be incompatible (under any circumstances) this "heuristic" may be elevated to a "fact"; often there is no such certainty, and we retain the knowledge as a heuristic.
3. Control:
   - A proof plan for induction: choose variable to induce on, split into base and step case(s).

- Straightforward cases of configuration: choose processor, then, ..., then configure disk drives, then assess backup needs,...

In order to be expressible in logic, a system needs to be based on such a separation. This separation enables each type of knowledge to be encoded *declaratively* if we so wish. This is of the utmost importance, if we are to be able to check that the logical formalism given accords with our understanding of the semantics. We need to be able to check, separately and independently:

- That the facts represented are "true", or at least what we intend.
- That procedures are captured correctly.

If these two are inextricably intermixed, the configuration task becomes unacceptably hard. A system based on a clean separation is easier to maintain, because knowledge is encoded declaratively.

Given such a clean separation, it is more likely that we will be able to salvage *something* in the face of technological innovation, provided it is not too extreme. For example, if there is a radical change in storage methods and components, then it will not affect the top-level architecture. Methods of generating partial configurations which are not affected by the changes to affected components will remain unscathed also.

7.2 Proof plans

A *proof plan* is a means of expressing the commonality between members of the same "family" of proofs (Bundy, 1991), whilst allowing sufficient flexibility and adaptability to prove a large number of different theorems. Proof plans provide an expression of strategies for automatic reasoning by describing *tactics*, which can be understood for now as a kind of operation (or program), in terms of the *preconditions* under which they are *applicable*, and their *effects* if successfully applied. The specification of a tactic in terms of its preconditions and effects is called the *method* for that tactic, and methods provide a basis for combining tactics (the effects of one implying the preconditions of subsequent methods) to form a complete plan which, if executed, will carry out a reasoning task, such as proving a theorem.

7.3 Benefits of proof planning

Proof planning techniques can benefit the development and maintenance of knowledge based systems based on firm logical foundations in many ways.

Firstly, using planning techniques can be advantageous in itself, provided the planning overhead does not outweigh the gains. Planning may pay off for two reasons:

1. Planning may be more efficient than execution, in that the steps involved in planning an operation may be fewer, or less complicated, than those involved in carrying it out.

2. The planning process is guided by the preconditions of methods, and if these preconditions are sufficiently strong, search should be much more directed. Instead of hundreds of choices at every point there should be only a few.

Secondly, if there is a clean separation of the object-level theory from meta-level knowledge, it is then possible to consider a number of related problems, all using the same underlying object-level theory. For example, connecting up a device is the same whether both are being synthesized (as when the top-level problem is to find a configuration to meet a specification) or whether they appear in a customer's order (as in the case where the problem is to check such an order drawn up by a sales representative).

Thirdly, there are potential benefits for interactive or co-operative problem-solving systems. If a plan fails, we want to *understand* why. Using methods as building blocks, the failure is returned at the appropriate level. In fact the analogy is that of a block which is actually a preassembled compound put together off-site to fulfill a particular need.

And finally, as mentioned earlier, the object theory can be maintained to take account of new or obsolete components and other changes, independently of the meta-level knowledge. We can, variously and separately, make changes to the object-level theory, add new heuristics known or thought to be useful, acquire recently learned strategies, or experiment with strategies.

7.4 Capturing higher level reasoning

I now turn to the question of how a technique developed for use in mathematics can be useful in another domain.

First of all, it is a feature of human attitudes to problem-solving that we tend, at least in the first instance, to use strategies which worked for us before. So when faced with a new problem, it is natural to ask "Does this remind me of anything? How is it similar? How is it different? What did I do last time? Will it work? If not, what is different?" and so on. This is ably presented in Polya's "How to solve it" list, from the book of the same name (Polya, 1945).

When learning about configuration, as practiced and explained by experts, it was encouraging to find that innumerably different problems can be fitted into one or other of a small number of "templates", and a large search space is divided and partly conquered by this strategy.

By using proof planning to guide inference, I aim to separate domain knowledge and control information. By separating the task into appropriate levels, I can then reason about aspects of the task at the right level of detail. Decisions can be left until all the knowledge necessary to make an informed choice has been gathered.

I can also explicitly represent strategies for directing the search.

8 Proof plans and configuration

The flexibility with which a proof planning system is able to use heuristics to guide the search for a solution will also be very useful in the configuration task. Let us take the analogy with the mathematician's thought processes described in Section 7.4. The expert configurer remembers examples similar to the current problem, and follows broadly the same pattern: she thinks in terms of a high level strategy (*e.g.* "now think about which disk drives are needed") to be followed at a later stage by more detailed considerations (*e.g.* "do I need a cable for this disk?"), and uses heuristics to guide the search for an acceptable configuration.

In configuration problems, as with theorem proving, we find "families" of problems, with a common structure overlaying the variability of the individual steps in different problems within the same family. In this work, the additional constraint, that of coming up with "good" solutions, according to some criteria, was very important. For instance, we might *prefer* configurations which were *efficient*, or which *cost less than £X*. It was found necessary to develop proof plans to deal with these different scenarios.

8.1 Semantics

The separation of planning and execution represents the two separate views of the problem:

1. The planning stage corresponds closely with the customer, or user, view;

2. The execution stage corresponds closely with the engineer view.

Although the planning stage does not give enough detail to configure the hardware, it contains sufficient information to allow the user to decide whether the plan, if executed, is likely to yield an acceptable configuration. For instance, the user can see which devices are being provided, even if the details of how they are connected are absent. In essence, the ability to plan weeds out most, if not all, of those solutions which would be unacceptable to the user, without expending a large amount of time executing the plans, *i.e.* actually configuring such systems. Suppose the first solution at the planning stage gave us at least one device we did not want. Then we would seek further solutions. Or we might want to see what the options were. Whatever our motivations, we would not want to wait while all the devices were connected up by the configurer before we saw what devices had been chosen. We are not at all interested in the configuration details: rather, we want to look at the "bare bones" of each solution before deciding whether this is the one we want or else passing on. With, say, a choice of ten disk drives and three tape drives this gives thirty different combinations which is quite enough.

Since it is much faster to plan than to execute, this planning stage is a kind of "rapid prototyping" leading to a likely looking plan, which can then be executed.

To recap, we can *roughly* see the two stages as

1. From high level user specifications, deciding which devices will provide these, *i.e.* how we get the required usage, and, in addition, some handle on how we provide the required utility (*e.g.* knowing whether devices must be connected according to efficiency heuristics, or not).

2. From knowledge of which devices appear in the configuration, and a few constraints on how they may be connected (*viz.* the utility considerations mentioned in 1.), deciding how they will be connected.

8.2 Strategies for using heuristics

We have seen that there may be both "hard" and "soft" limits on, for example, how many disk drives can be attached to a particular kind of channel. Four is the actual "hard" limit — any more and the configuration would not be legal — but experience has shown that configurations configured up to the legal limits run slowly. On the other hand, configurations with three disk drives connected per channel do run efficiently. We might have, therefore, a "soft" limit of three disk drives per channel.

As a general strategy, we can see that, all things being equal, a configuration obeying the soft limits is preferable and it is that kind of configuration we should go for first. However, if this is not possible, due, for example to cost constraints, or to sheer size of configuration, one or more soft limits will have to be ignored. The management of soft constraint relaxation is not trivial. There are many design issues to be considered.

Our solution to this problem is to have alternative methods: one applying only soft constraints, the other allowing relaxation of these, but obviously still applying hard constraints. This gives us methods which configure components according to efficiency heuristics as well as methods which simply ensure legal configuration. This necessitates a strong control strategy to ensure that the methods are tried in the right context and in the right order. The use of supermethods, which are methods which specify the use of other methods, deals with this issue.

8.3 Control knowledge

Even in cases where we *can* design a configuration obeying all the soft constraints, the solution space may still be large. We need to decide which additional factors to look at in deciding a final solution, or small set of alternatives. We need to incorporate the knowledge needed to support the designer of computer configurations in her decision making.

Much guidance in searching for a configuration can be provided if the user of the configuration system has facilities for scene setting; *e.g.* stating whether an efficient configuration is important, whether there is a cost constraint, *etc.* There is a small subclass of design problems in which *the* "optimum solution" (along one of several possible different dimensions: cost, efficiency,...) which meets the specific-ations is required. The underlying strategy to be used in this design scenario is quite different from, say, that to be used if the problem is to find any satisfactory solution(s) subject to keeping within a cost ceiling.

In what follows, I have considered only cost and efficiency as design issues in order to simplify the discussion. There are other factors, not mentioned here; for example:

- The *expandability*, or potential growth path, of a configuration, *i.e.* its capacity to be augmented at a future date.
- Technological issues: preferring innovative products , or avoiding potentially obsolescent components.

8.4 Constraint relaxation

I give an example which demonstrates the need for strategies to relax soft constraints when this proves necessary, whilst avoiding sub-optimal solutions.

Let us look at two soft constraints:

1. Whereas four disk drives per HPIB channel are legal, more than three can, in certain circumstances, give an inefficient configuration.
2. For a two card cage configuration, more than six interface cards can, in certain circumstances, give an inefficient configuration.

These soft limits are more stringent than the "hard" limits. They are used by heuristic strategies aimed to give "better" configurations, usually in the interests of efficiently running configurations. If possible, we configure systems so that the soft limits are adhered to. However, this is not always possible, and we have to relax one or more of these soft limits. In the next section I examine the management of this constraint relaxation, with regard to certain pitfalls which must be avoided.

I shall focus on the part of the configuration task which connects disks to the configuration. Each disk (assuming that it has a suitable cable) may be connected to any existing HPIB channel, provided there is room, *i.e.* the limits for that channel will not be exceeded. We would prefer that the soft limits are not exceeded, but must not exceed the hard limits in any case. Alternatively, we can try to generate a new HPIB channel and connect the disk to that. To do this, there must be space on an existing card cage (or we can again generate a new card cage; for the sake of brevity we shall assume that this option is not open to us — true examples exist in which the number of card cages is already at a maximum, or where cost constraints preclude the addition of another card cage).

Suppose the situation so far is as shown in Figure 2.

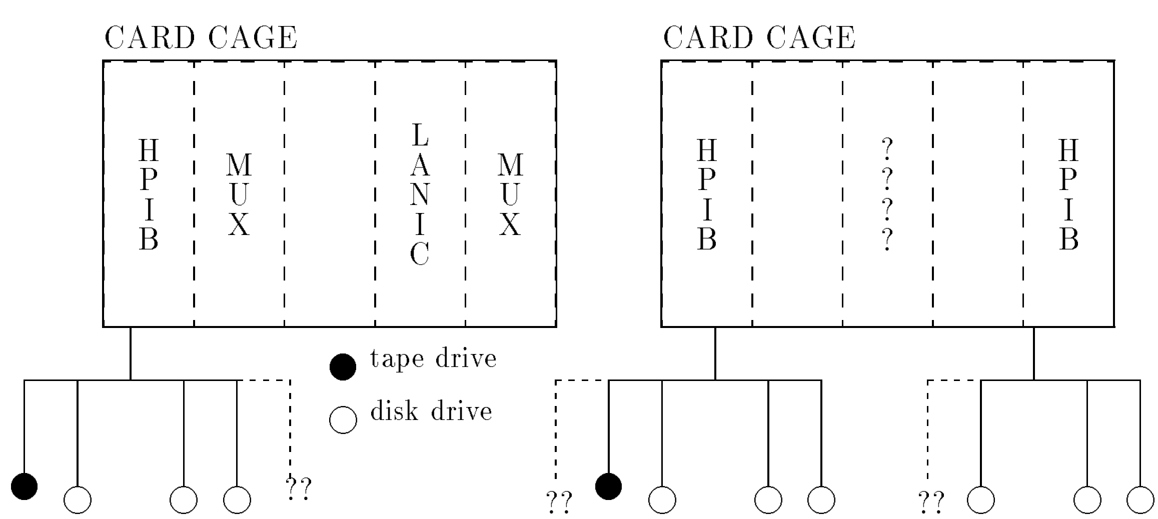

Figure 2: Situation before the next disk is added

Here, there are two card cages, with various interface cards inserted in slots: one card cage has two MUXes, one LANIC, and one HPIB; the other has two HPIBs configured. Each HPIB channel has three disk drives connected, and two of them have a tape drive connected.

We observe that the following hard constraints are satisfied:

- No more than ten ICs in total.
- No more than four HPIBs per card cage.
- No more than six devices per HPIB.
- No more than four disk drives per HPIB.

Moreover, the following soft constraints are also satisfied:

1. No more than six ICs in total.
2. No more than three HPIBs per card cage.
3. No more than five devices per HPIB.
4. No more than three disk drives per HPIB.

Suppose now that we wish to add a further disk drive to this partially configured system. Soft limit 1 will be violated if we add another HPIB channel. Soft limit 4 will be violated if we add another disk drive to any of the three existing HPIB channels. It is therefore necessary to break one of these soft limits in order to configure the next disk.

We consider two solutions.

1. Overload the third HPIB channel.
2. Overload the card cages by adding another HPIB channel.

Which of these do we prefer? The answer is contingent on circumstances. In general, both are semi-acceptable as solutions. It would take an expert on performance to decide which to prefer.

Let us carry on, and assume that there is one further disk drive to connect. Assuming that the situation is as in solution 1, we have the same choices as before. If we overload another channel, then we have violated one constraint, twice, involving two separate channels. (In this example there is not the option to violate it twice on the same channel because of hard constraints.)

If we choose to violate constraint 1 this time, however, then we arrive at the solution shown in Figure 3.

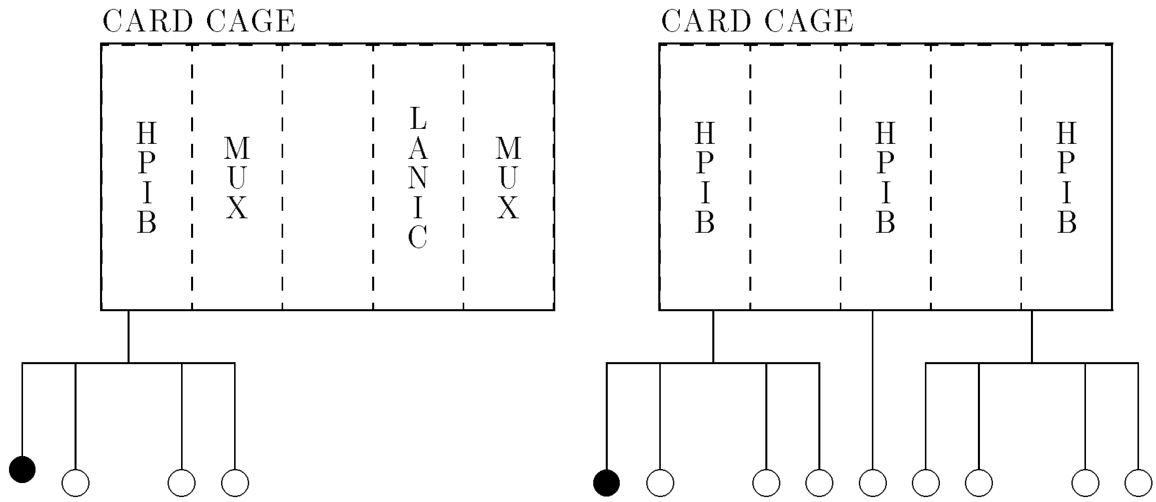

Figure 3: Locally sub-optimal solution

However, this solution is locally sub-optimal, in the sense that if we moved one of the disk drives from the overloaded channel to the newly configured one, we would have a solution with the same components (and same cost) which was better: only one constraint being violated, rather than that same constraint plus another one.

Note that if we chose the other path, namely Solution 2, no further constraints need be relaxed when configuring the last disk drive, since it can be placed on the new HPIB channel without violating any further soft constraints.

These points emerge:

- One solution, which is locally optimal, violates constraint 4 twice.
- Another, again locally optimal, solution violates constraint 1 once.
- Whereas both the above are feasible as solutions, we must avoid the situation of Figure 3.

We need to bear these points in mind when drawing up methods; in particular when considering "supermethods", or proof plans, which embody planning strategies.

8.5 Proof plans to express strategies

In tackling the problem of directing the search towards solutions which are, at the very least, locally optimal, I need *strategies*, which are expressed in terms of proof plans, which can be thought of as comprising "supermethods". A supermethod has a structure as outlined in Figure 8.5. The main feature is that the *effects* call other methods. Now, without this supermethod, the planner is capable of finding a plan which consists of the sequence

$$method_1, method_2, \ldots, method_n$$

in place of the supermethod *basic_config/1*. Hence *basic_config/1* can be thought of as *equivalent* to the proof plan so formed. I have embodied a few strategies as supermethods. For example, there is a supermethod for configuring all system devices in accordance with heuristic guidelines. There are other supermethods which will, if necessary, relax one or more soft limits whilst still keeping within the legal limits. I found that reasoning under cost constraints requires different

$Name$	$basic\_config$
$input$	$\exists c.spec(c)$
$output$	$\ldots$
$preconditions$	$\ldots$
$effects$	$\ldots$ $applicable(method_1)$ $applicable(method_2)$ $\ldots$ $applicable(method_n)$ $\ldots$
$tactic$	$basic\_config$

Figure 4: Structure of a supermethod

strategies according to whether there is a given upper ceiling or whether it is required to minimize cost. The planner will build, from the methods available, a customized plan appropriate to the particular specification given. For example, if the top-level proof plan, which turns out to have a wide area of applicability, is:

configure-processor/2

then *configure-template/1*

then *configure-terminals/1*

then *configure-disks/1*

then *configure-tapes/1*

then *configure-printers/1*

then this can be encapsulated as the supermethod which reflects the strategy of

1. Deciding what an appropriate processor would be from information given in the specification.

2. Each processor fixes a kind of "template" configuration, *e.g.* a 950 processor will support certain types and models of devices, will have at least two (and at most four) card cages, *etc.*

3. Attend to user needs explicitly given in the specification, configuring devices, adding in essential components, *e.g.* memory, back-up devices, not given explicitly.

Within the overall guidance given by this plan, there is flexibility: *e.g. configure-tapes/1* will use the methods appropriate for the specification given; either configuring tape drives explicitly given by the user or inferring a suitable tape configuration for the disk configuration synthesized if the user has not explicitly specified them.

The existence of supermethods reduces the chances of a random application of methods which could lead to locally sub-optimal solutions; in this sense the supermethods embody strategies for configuration. However, if the supermethods are not applicable, it is possible to slot together methods which will synthesize *some* configuration.

9 Ensuring good design

9.1 Managing heuristics

Let us compare two methods for configuring channels:

1. according to soft constraints and

2. according to hard constraints.

A schema for both methods is shown in Figure 5 (omitting some housekeeping preconditions); the only difference between the two methods is in whether hard or soft constraints are used when checking that there are slots available.

$Name$	$h\_configure\_device(Device, Interface, Config)$
$input$	$configure(Device, Interface, Config)$
$output$	[]
$preconditions$	$there\ are\ available\ slots\ of\ the\ right\ type$
$effects$	$reduce\ number\ of\ available\ slots\ by\ one$
$tactic$	$h\_configure\_device(Device, Interface, Config))$

Figure 5: Configuring an interface using soft constraints

We can try to "prefer" to use heuristic limits either:

1. By trusting to the ordering of the methods.
2. By incorporating supermethods which encapsulate strategies for constraint relaxation, using the methods for configuring devices as building blocks in these supermethods .

9.2 Encapsulating basic strategies as supermethods

In order to improve search control, which was limited using only the simple methods presented so far, the main problems to be overcome are

- Eliminating unproductive search and backtracking through methods.
- Incorporating strategies for constraint relaxation.

There is a discernible pattern for a high proportion of configuration tasks, and this is encapsulated in the supermethod `basic_config/1` (Figure 6).

```
supermethod(basic_config(C),
    Inputconj,
    [processors(C,P),
     [system_disks(C,P),disk_capacity(C,_DCap),total_disk_capacity(C,_TDCap)],
     printers(C,Prs),
     terminals(C,Ters)],
    [],
    [applicable_subm(...,method(configure_processor(C,P),_,...),

     ...   ...   ...   ...   ...   ...   ...   ...   ...   ...

     iterate(...
             method(match_attributes(C,_,_),_,...)),

     ...   ...   ...   ...   ...   ...   ...   ...   ...   ...


     applicable_subm(...submethod(configure_disks(C),_,...)),

     (applicable_subm(...,submethod(configure_tapes(C),_,...)),

     iterate_set(...,
             [method(h_configure_device(C,_,_),_,_,_,_,_),
              method(h_configure_ic(C,_,_),_,_,_,_,_),
              method(configure_cc(C,_),_,_,_,_,_),
              method(configure_device(C,_,_),_,_,_,_,_)],
                ...),
     applicable_subm(...,method(configure_device_list(C,Prs),
                                                   printers(C,Prs),_,...)),
     applicable_subm(...,method(configure_device_list(C,Ters),
                                                 terminals(C,Prs),_,...)),

     ...   ...   ...   ...   ...   ...   ...   ...   ...   ...

     ],
    Outputconj,
    Tactic).
```

Figure 6: The `basic_config/1` method.

In order to "recognize" that we are in such a pattern, we look at the input conjunct. Notice that, unlike the case of simple methods, we need to consider (*i.e.* pattern-match) on more than one of the input-conjunct goals. If the specification says something about terminals, disks, and printers, and if, in addition, the processor is specified, then we know we are in a straightforward example of a "normal" synthesis. The `basic_config/1` supermethod deals with such cases. It has no other preconditions — matching with the input slot is all that is required — and it works by running methods, or submethods[2], in its effects slots in order to synthesize the different parts of a configuration meeting the specification. The supermethod `basic_config/1` is not, of course, applicable to simple partial specifications — such as those used for testing the object-level. However, the preconditions for `basic_config/1` quickly fail in these cases and a custom-built proof plan is obtained by the planner, exactly as before: `basic_config/1` augments the methods base, rather than replacing existing methods. It is placed earlier than its component supermethods, to ensure that it will be tried first.

The use of supermethods obviates, to a large extent, the need to ensure that methods appear "in the right order". Following on from the point made towards the end of Section 9.1, a more directed way of attempting to configure components using heuristic limits by preference is to write a supermethod whose effects involve running submethods to configure devices according to a pattern which was discovered by hand-working many examples: the use of the predicate `applicable_subm/3` within the effects slots forces the planner to consider the methods in the desired order. It should be stressed that this is only a "heuristic" strategy for trying to make sure heuristic limits are used whenever possible.

[2]Submethods are methods which can *only* be accessed through the effects of supermethods, and not directly by the planner.

9.3 Adding strategies for cost specifications

I looked at only one strategy to do with dealing with cost specifications, namely that of finding a configuration which kept under a given cost ceiling.

To do this, I used annotation of the configuration term within methods. This was only done when the goal $cost(c) \leq x$ appeared in the specification. There is an extra overhead in dealing with costs and so this is not done where unnecessary.

An extra argument representing cost, is added to the configuration tuple (in the first position), so that

$$C \equiv [X, Proc, \ldots]$$

and when any component is added (by the predicate `add_component/2`, the cost of the component is subtracted from X, and this new value compared with zero: if it is negative, the preconditions fail.

Thus unproductive partial configurations are pruned early on. Unfortunately, there is still a lot of search involved with some examples. More strategies could be developed, but the most likely means to success is via using information about failures in the planning process to "re-route" the planner in particular directions.

10 Future development

So far, I have implemented a system CLEM which can deal with specifications as exemplified in Figure 1. Using supermethods, the search for solutions can progress in a more directed way, and the system can deal with global design goals such as those involving stipulations about cost of efficiency. This implementation is not final and complete, and various alternative formulations have been tried.

CLEM is only a prototype system; in order to be used "in the field" it would need several enhancements, notably to its interface. It also needs to be brought up to date.

In pure research terms, these are trivial issues; however, they are necessary if others are to be able to use the system effectively.

One important issue has emerged. If proof planning is to be effective in producing desirable solutions, whether this means well-designed hardware systems (in the domain of configuration) or elegant proofs (in the theorem proving domain), a true understanding of the domain is a vital ingredient in developing such a system. This does not mean one has to be a configuration expert; but there are no short cuts to be taken. It is not enough to learn a few superficial rules without also understanding what is good or bad about various alternative solutions, and striving to understand or develop strategies which return good solutions more often than bad.

Thus in order to enhance CLEM's configuration strategy it will be necessary to gain more knowledge. Various indications of this have already been given: for example, knowledge of how to relate disk storage requirements to applications; understanding of the reasons behind various heuristics.

As far as the more exciting, and domain independent, research issues are concerned, the one which seems most likely to increase the power of the system is harnessing failure to constructive ends. This is perhaps even more important for this domain than for theorem proving. In particular, failure in configuration can be more fundamental than simply searching the wrong branch of a proof tree: it can follow necessarily from impossible specifications. In this case, the system needs to be capable of a useful dialogue with the user to resolve differences and modify the specification.

The key to this is the use of preconditions to provide explanations for the user, and suggest remedial action. I hope to explore this issue in the near future.

11 Conclusions

We have developed an automatic configurer which:

- gives solutions corresponding with solutions judged to be acceptable by human experts.
- finds solutions in a reasonable time.
- finds better solutions in preference to less good ones: *i.e.* the better solutions are returned first.
- deals with search in a desirable way; *e.g.* a request for further solutions produces configurations which are acceptable, but also discernibly different from previous solutions in some significant way — *i.e.* using different components, and not merely trivial permutations of other solution configurations.

Provided a plan can be executed successfully, the resulting configuration is guaranteed to be legal, *provided* the object-level theory is correct and complete. Naturally, if there are constraints missing from the object-level theory, we cannot guarantee soundness.

Since the tactics make use of the object-level theory, which is sound, they too will necessarily result in legal configurations. Using tactics means that *completeness* will be lost. However, no expert would provide *all* solutions to a specification — for instance, many permutations exist where the same components are configured in different orders. Thus completeness is not necessarily a goal for knowledge based systems. In fact, in practice it is better to forgo completeness than allow the user to be overwhelmed by trivially different solutions. This is the strategy employed by our use of tactics.

Moreover, many solutions which are technically legal, even if they were not trivial permutations of other solutions, would be frowned upon by experts. For example, using a different model of disk drive for storing the operating system from those used for user disks would be thought odd. Where we need to use tactics to fill in gaps in the explicit specification, these are based on the same design considerations as seem to be used by experts.

The position I have taken can be summarized thus:

1. The user (customer) is always right — any specification which is legal must lead to a synthesizable configuration by the automatic configurer.
2. Any free choice left to the automatic configurer should lead, if possible, to the synthesis of well-designed configurations

The solutions found in testing conformed to this agenda. Note, however, that for large systems, even with this kind of incompleteness there will still usually be a very large number of different solutions. The user not wishing to plough through all of them would be well advised to scan through a few solutions (at the planning level only) and retry, making her specification more detailed on the second pass. This kind of negotiation is common in practice, between sales representative and customer, and cannot presently be emulated by CLEM, which is completely automatic and in no way co-operative. However, proof planning as a paradigm gives better prospects for co-operative systems than existing approaches, because methods represent a more appropriate level of reasoning on which to communicate; for example, explanations can be readily provided from method preconditions. This seems an exciting new possibility which could be explored further: harnessing together the different strengths of human designer and automaton.

References

Barker, Virginia E. and O'Connor, Dennis E. (March 1989). Expert systems for configuration at digital: XCON and beyond. *Communications of the ACM*, 32(3):298–310.

Bundy, A. (1987). How to improve the reliability of expert systems. In Moralee, S., (ed.), *Research and Development in Expert Systems IV*, pages 3–17. C.U.P. Also available as Research Paper 336, Dept. of Artificial Intelligence, Edinburgh.

Bundy, A. (1991). A science of reasoning. In Lassez, J-L. and Plotkin, G., (eds.), *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press. Also available as Research Paper 445, Dept. Of Artificial Intelligence, Edinburgh.

Frayman, F. and Mittal, S. (1987). Cossack: a constraints-based expert system for configuration tasks. In *Proceedings of the 2nd International Conference on Applications of AI to Engineering*, Boston MA.

Polya, G. (1945). *How to solve it.* Princeton University Press.

Searls, David B. and Norton, Lewis M. (1990). Logic-based configuration with a semantic network. *Journal of Logic Programming*, 1:53–73.

Stefik, Mark. (1981). Molgen part 1: Planning with constraints. *Artificial Intelligence*, 16:111–139.