Extending the proof plan methodology to computer configuration problems *

Helen Lowe

Department of Artificial Intelligence,
University of Edinburgh.

Abstract

This paper examines whether the technique of *proof plans*, developed originally for use in theorem proving, can be successfully applied outwith mathematics. We look at a problem associated with the configuration of large computer systems, namely that of checking orders drawn up by sales people on behalf of customers. Strategies exist for this task which may be explicitly encoded. We discuss the formation of plans for order checking and the factors involved in determining whether such plans can be guaranteed to succeed.

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

1 Introduction

When large computer systems are ordered by sales representatives for customers, it is an important but often difficult task to ensure that the order contains exactly those components which are necessary for a legally configured system which meets the customer's specifications. Mistakes which often occur are:

- Failing to order vital components, *e.g.* cables.
- Ignorance of what, precisely, is given by a particular order number, *e.g.* not realising that a cable is included on the order number for a particular device, with consequent overordering.
- Transcription errors, *i.e.* wrong order codes corresponding to non-existent components or components other than those required.
- Incomplete knowledge of the constraints which exist on a given system, *e.g.* believing that more devices of one kind may be connected to a given channel than is, in fact, the case.

The difficulty with checking orders is that order forms consist solely of order codes and "options" (modifications to order codes, *e.g.* to specify port types, voltages, connection type, *etc.*) with no explicit statement of the user's goals, *i.e.* the system specifications. Given a *correct* order, the system specifications can be inferred and output for the user to check. However, we have no knowledge at the outset of the correctness or otherwise of an individual order — this is one of the pieces of information which needs to be inferred. The number of different combinations of possible errors in a given order is rather large, and the strategies employed to make sense of orders are many and various. We therefore need to ask:

1. Can we build a system with enough flexibility to account for all, or at least most, of the different orders (erroneous and otherwise) which could ever occur?
2. Can we somehow capture the "essential strategies" employed by human users to structure and direct the search for a checking strategy for all these different cases?

Proof plans have been successfully used in proving theorems by induction (Bundy *et al.*, 1991). This success has to do with the explicit encoding of strategies based on perceived commonality between proofs or algorithms which, nevertheless, display many surface differences.

Configuration synthesis involves proving that, given a specification for a computer system, there exists a set of components which can be joined together to provide a computer system meeting this specification, by constructing such a system. The open ended nature of some specifications means that there may be many different configurations possible, although there are strategies which people use to make this task more manageable than would otherwise be the case.

Order checking differs from configuration synthesis, in that there is no explicit knowledge of the original specification which led to the order. However, human experts use strategies based on configuration when checking orders. There are known methods for surmising user goals, and using the inferred specification to attempt configuration. Failure at this juncture leads to "patching" of the order.

A prototype system is outlined which uses the architecture of a proof planning system to check orders and propose amendments.

2 Related work on configuration problems

2.1 R1/XCON: a configuration checker

Perhaps the best known work in this field is XCON (alias R1), a rule-based production system, used extensively by DEC. XCON configures VAX-11/780 computer systems, in the sense that it takes a customer order as input and returns both the modifications to the order necessary to obtain a configurable system (if any) and diagrams showing the spatial relationships between the component parts. An account is given in McDermott (1982), Bachant & Soloway (1989), and Barker & O'Connor (1989). It is a production rule system, where , for example, the right hand (action) sides of rules extend partial configurations whilst the left hand sides of rules hold constraint information on whether certain partial configurations may be extended. It uses a system of contexts to constrain the applicability of the various rules, and some rules effect a change of context under conditions given by the left hand side. The rules can be summarised into three types thus:

- rules to create or extend partial configurations
- rules to determine the order in which decisions should be made

- rules which gather information for other rules, *e.g.* access the components knowledge base or perform computations

The central problem-solving method is **Match**, with **Generate-and-test** as a kind of weak backup method. **Match** "recognises" what to do next at any point in the program's execution, and the system as a whole never needs to backtrack. Even when several solutions need to be generated before an acceptable one is found, this solution will never need to be undone. This method works because (it is claimed) the domain knowledge is such that it is possible to determine *locally* whether a given action is appropriate. All the information needed to make a decision is contained in the local context, and decisions yet to be made have effects which propagate only forwards, not backwards so as to affect decisions already made. Thus very little search is involved, the main task being to dynamically order the set of decisions and generate a single acceptable solution. The knowledge base consists of a large set of constraints and component information. The expert knowledge embodied in the rule set of the system consists of two types of information:

1. The temporal relationships between the subtasks involved: *i.e.* when a subtask should be initiated.
2. Detailed information about when it is appropriate to extend partial configurations in particular ways.

XCON has a reputation for being difficult to maintain. One change has a large knock-on effect in terms of how many rules need modification. There is no separation of domain knowledge ("facts") and control knowledge ("strategy"). Most of the conditions in the condition-action rules define the context in which the rule is to be considered, and hence this kind of information is implicit rather than explicit.

Attempts have been made to improve this aspect of it (Soloway *et al.*,1987) by using a "higher level language", namely RIME, where the relationship of RIME to OPS5 (which XCON is written in) is suggested as being analogous to that between FORTRAN and assembly code. The main claim made for XCON-*in*-RIME is that control knowledge is made explicit, both in terms of the problem-solving method, said to be domain-independent; and the entering of different problem spaces (contexts), said to be domain-specific.

However, even with such improvements, it still remains the case that use of domain knowledge could not be extended, say to automating the process from user specifications to configurations, or to teaching configuration, even though the same domain knowledge would be used for all three tasks. The main intractable problem is that the central strategy of XCON is specific to this particular method of configuration checking, for these particular systems. It depends for its success on the fact that any local solution to a problem is always *the* "right" one. This approach cannot be used in more "open-ended" problems like configuration synthesis.

Another approach is that taken in COSSACK (Frayman & Mittal, 1987), a system strongly reminiscent of MOLGEN which essentially treats configuration as a constraint-driven paradigm. As such, it is eminently suitable for systems which are tailored very specifically to the specification, but not to systems which are in some sense "over-engineered": these modern architectures aim to provide scope for expansion and hence tend to be highly under-constrained. Our architecture accepts the large search space associated with such system configuration, the object-level representation ensuring the legality of the computer systems synthesized; and the control mechanisms ensuring a path through the search space directed towards solutions which are at least *locally optimal*. By this we mean that since the specification, in general, under-constrains the solution space, there will not be any one configuration which is *globally* optimal; instead we seek a (small) set of solutions which represent genuine choices: maybe one which is more efficient than other configurations of constraint-driven paradigm. As such, it is eminently suitable for systems which are tailored very specifically to the specification, but not to systems which are in some sense "over-engineered": these modern architectures aim to provide scope for expansion and hence tend to be highly under-constrained. Our architecture accepts the large search space associated with such system configuration, the object-level representation ensuring the legality of the computer systems synthesized; and the control mechanisms ensuring a path through the search space directed towards solutions which are at least *locally optimal*. By this we mean that since the specification, in general, under-constrains the solution space, there will not be any one configuration which is *globally* optimal; instead we seek a (small) set of solutions which represent genuine choices: maybe one which is more efficient than other configurations of similarly cost; another which is cheaper than other configurations of comparable utility, and so on. We feel that the search associated with a COSSACK-like system would be too great, and the number of solutions generated too large.

3 Proof plans, theorem proving, and order checking

3.1 Work on proof plans

The technique of *proof plans* has been used in Mathematical domains, for equation solving (Silver, 1985) and inductive proofs (Bundy *et al*, 1991). Proof plans provide an expression of strategies for automatic reasoning by describing *tactics*, which are algorithms for accomplishing certain tasks or subtasks, in terms of the conditions under which they are applicable (*i.e.* their *preconditions*), together with a description of their *effects* if successfully applied. These specifications of tactics are known as *methods*. These methods provide a basis for dovetailing tactics (the *effects* of one implying the *preconditions* of the next) to form a complete plan consisting of a succession of tactics which, when executed sequentially, will carry out the reasoning task (for instance, proving a theorem).

Consider a typical proof by induction. The important feature is that there *are* "typical" inductive proofs. Each proof consists of *base cases* (*e.g.* proving the theorem for $n = 1$) and *step cases* (*e.g.* proving that if the statement of the theorem is true for $n = k$, then it is also true for $n = k + 1$); each of these is typified by the use of certain kinds of rewrite rules. The *input* slot indicates the form that the input formula must take for the method to be applicable, and the *preconditions* slot expresses any additional constraints. The *output* and *effects* slots convey parallel information for the output. The *tactic* is the name of the piece of code which is applicable and results in the effects and output indicated.

Figure 1 shows the general format of a *method frame*, with slots for the name of the method, the input, output, preconditions, effects, and the tactic.

The **methods** of proof plans can be thought of as the specifications, expressed in a meta-logic, of tactics. This facilitates the task of writing proof plans to guide the search for proofs in automatic theorem proving. Planning systems based on finding appropriate sequences of methods possess properties (see Section 8.2) highly desirable in *any* knowledge based system (Bundy, 1987). Above all, they enable the general outline of a reasoning task to be captured aside from the detail of *e.g.* the exact number of times a particular step is performed. In theorem proving, a particular rewrite rule may be performed until a given condition is true; in order checking, we may need to add a particular type of component until the partial subsystem under consideration is configurable. It is this which leads us to believe that proof plans provide a useful vehicle for expressing strategies

Name	*e.g.* "induction"
input	...which must match the given input
output	form matched by the output
preconditions	...for applying this method
effects	true if method applied successfully
tactic	piece of actual code which will be run if this method is applicable

Figure 1: The format of methods

for problem-solving in non-mathematical domains as well.

3.2 Order checking

An order should consist of major components, such as the CPU, memory, disc and tape drives, and printers; together with other components necessary for connecting these into a working system, such as ports, cables, and cardcages. The disc drives may be of one of two types; to have both would be unusual but not prohibited, but to have neither would be erroneous. Similarly, we would expect at least some printers and disc drives. Mistakes involving omission of major components are easy to spot, but less likely in practice. More likely is the omission of vital cables or ports from the order; or insufficient ordering of these components. The latter may only be discovered on attempting to configure the system.

The tasks open to the checker may be divided into four categories:

1. Checking whether there are sufficient of various types of component to achieve a legally configured system.
2. Adding components to an order or deleting them if redundant.
3. Carrying out part of the configuration process.
4. Surmising user goals in order to suggest inconsistencies or fill major gaps in the system as ordered.

Orders consist of lists of codes and options, which may be decoded into lists of components, with terms such as

> **slow-channel([3,2,1])**
>
> denoting the fact that there are three *slow channels*,
>
> **slow-devices(discs([(7937H,4)]),printers([(2564B,2)]), tapes([(7980A,1),(7980A,2)]))**
>
> that there are four *7937H disc drives*, two *2564B printers*, and three *7980A tape drives*.

The inference involved in *checking* involves steps like matching totals of particular types of devices against bounds given in the knowledge base. *Adding a component* modifies one of the component terms. *Configuration* is more complex. For instance, for the *slow-devices* above, the syntactic result of allocating these to the given three channels is to add extra arguments to each of the elements of the list [1,2,3] in the *slow-channels* term which consist of devices from the *slow-devices* term.

For instance, suppose we have the two subterms

slow-channels([1,2,3]) and

slowdevices(discs([(7937H,4)]),printers([(2564B,2),(2235A,5)]))

We can allocate the first device (one of the four 7937H disc drives) to channel 1, 2, or 3 — there are no constraints. Suppose we add it to channel 1. Then the resulting rewritten terms are

slow-channels([[1,discs([7937H])],2,3]) and

slowdevices(discs([(7937H,3)]),printers([(2564B,2),(2235A,5)]))

Two other rewrites are possible at this stage (allocating the device to channel 2 or channel 3). Now we allocate the second device. It too may be allocated to any of the three channels. Sometimes there will be constraints, *e.g.* allocating a fourth disc drive to a channel would violate a constraint; allocating a particular type of printer to the same channel as a disc drive might be inadmissible. Potentially, there is a large search problem involved. By checking preconditions which ensure that (some) allocation of devices to channels is possible, not only do we avoid this complex search at the planning stage, we also cut down the search at execution time, by at least ensuring that we do not embark on a fruitless search for an allocation unless we know

1. the search for an allocation will succeed,
2. the plan as a whole will succeed.

At the end of the process, we end up with the terms rewritten to

slow-channels([[1,discs([7937H,7937H]),printers([2564B,2564B])],

[2,discs([7937H]),printers([2235A,2235A,2235A])],

[3,discs([7937H]),printers([2235A,2235A])]])

slowdevices(discs([]),printers([]))

The *slow-devices* term is removed when this process is complete, thus showing (syntactically) that there are no longer any *slow devices* to be allocated. Section 5.2 goes into this in more detail.

Surmising user goals involves obtaining parameters from certain subterms in the decoded order and computing system parameters from them, *e.g.* finding the subterm **memory(64)** and computing the range [53,96] for the number of active users supported by this system.

Configuration is computationally expensive (NP-complete), involving the allocation of N devices to M channels, where there are constraints on how many devices of different types may be allocated to each channel, and also constraints on which devices may share channels. Hence configuration should not be carried out unless certain checking procedures have been done. However, some parts of the system may be configured independently of other parts under certain circumstances.

Adding components to a system is something which we would want to control fairly tightly; if, at the other extreme, we did not require any preconditions for this, then our order could grow indefinitely, to give systems at the limits of what is configurable and beyond.

3.3 Proof plans for order checking

At first sight, there is no similarity between the tasks of theorem proving and order checking. We might not, therefore, expect the proof plan approach to buy us anything for the latter task. However, there are certain important features which both tasks have in common.

- The object-level reasoning process in configuration and order checking is analogous to object-level inference in theorem proving; in both cases, the search space involved at this level is large and subject to combinatorial explosion.

- People develop (occasionally are taught) a "feel" for what to do next at any stage, even though there is often no uniform proof procedure.

- Although we cannot formulate a set sequence of steps for proving a theorem or checking an order before seeing the particular case in question, nevertheless on being given the problem explicitly we are usually able, quite quickly, to produce such a sequence.

- We are not always right first time, and there may be alternative methods of proceeding, but we home in on the solution much more quickly than would be suggested by examination of the object level search space. Low level search strategies would have to consider all matching rewrite rules at every stage, and would thus be liable to a combinatorial explosion.

- We have a notion of "naturalness" of both proof and order correction. For instance, proofs produced by some theorem proving systems would be rejected because they contained redundant steps or were difficult to "follow". There are similar notions for order checking, best explained by an example, which is illustrated in Figures 2–5.

An erroneous order is illustrated in Figure 2. Figures 3 to 5 are pictorial representations of three alternative "corrections" to the order.

In this example, we have in the initial order four devices which may be connected via cables of type X (*e.g.* RS-232 cables) or, alternatively, via cables of type Y (*e.g.* RS-422 cables).

Suppose we have another five devices which may be similarly connected. Suppose also that we have one group of eight ports for connecting X cables and one group of eight ports for connecting Y cables. In addition, suppose we had four cables of type X.

Clearly, something is wrong with this order. The "obvious" correction would be to add five cables of type Y to the order. But there are other possibilities. We could add five X cables, another group of X type ports, and delete the group of Y ports from the order. However, this would not be seen as a sensible correction, even though it would produce a legal system — and there could be ways in which it was a "better" system.

Figure 2: An erroneous order

Figure 3: Correction to order: add Y cables

Figure 4: Correction to order: delete Y ports; add X ports; add X cables

Figure 5: Correction to order: delete Y ports; delete 5 devices

Alternatively, we could simply remove the five Y type devices and the group of Y ports from the order altogether. "Normally" this would not be a good thing to do, because there are likely to be more errors in ordering cables (fairly low-level components) than in ordering the more important devices. However, looking at other aspects of the system might reveal that it is impossible to connect the 5 devices anyway, because system limits for that type of devices might be exceeded. So in this case we might want to remove the devices, or find an alternative method of satisfying the utility which they give.

So our notion of "naturalness" cannot be predetermined, any more than our methods of checking.

4 How orders for computer systems are checked

4.1 Task definition

Given an order,

- We determine whether the order represents, as it stands, a *legally configurable* system.
- If it is not, we decide what needs to be added to the order.
- We produce a legal configuration of the components, and an output term representing the system showing how devices are connected.
- We describe the system in terms of the specification it meets, together with details of components which were added to the order and of redundant components which were found on the order.

4.2 Main stages of the task

There are three stages involved. An example of the initial input and the results of each stage is given in Figure 6.

1. *Decode* the order codes into something more structured and meaningful.
2. Try to *find a plan* which should, if executed, carry out all the stages needed for checking this particular order.

A typical order

```
order(['32490B','A1101A',('27115A',3),
       ('2345A',[('625',6)]),
       ('2345A',['803',('625',3)]),
       ('2345A',['803','805','805']),
       ('7937FL',8),('7980A',2),('7937FL',2),'7980A',
       ('2563B','393',2),('3081A',50),
       ('D1347A',8),('C1002A',15),('2235A',5),
       ('13242X',5),('24542M',8),('40242P',15)])
```

A decoded order

```
unknown code 3081A
components([spu(s950),memory(64),...,
            cardcage([3,2,1]),
            dtc([[3,[rs232,rs422,rs422]],
                 [2,[rs232,modem,modem,modem]],
                 [1,[modem,modem,modem,modem,modem,modem]]]),
            slow-channel([2,1]),fast-channel([3,2,1]),
            slow_devices(discs([]),printers([(2564B,2)]),
                         tapes([(7980A,1),(7980A,2)])),
            fast_devices(discs([(7937FL,2),(7937FL,8)])),
            serial_devices(terminals([(D1347A,8),(C1002A,15)]),
                           printers([(2235A,5)])),
            cables([(13242X,5),(24542M,8),(40242P,15)])])
```

A typical configuration

```
Possibly terminals missing...
A possible configuration is:
[...cardcage([[3,fast-channel([1,discs([7937FL])]),
                 fast-channel([2,discs([7937FL])]),
                 ...,
              [2,slow-channel([1,devices([2564B])])],
              [1,[mux(2),mux(1),
                  lanic(dtc([[3,(rs232,[(D1347A,24542M,3),(2235A,13242X,5)]),
                                 ...,
                             [2,(rs232,[(D1347A,24542M,5)]),modem,modem,modem],
                             [1,modem,modem,modem,modem,modem,modem]]))
              ]]])]
9 modem ports redundant
```

Figure 6: Stages of order checking

3. *Execute* the plan.

We are interested in such questions as:

- How should we distribute the effort between meta-level methods and object-level tactics?
- Can the plan be guaranteed to succeed?
- How much work is involved in checking preconditions which will *guarantee* success, as opposed to having preconditions which merely give a *high probability* of success?

The first of these is discussed in Section 7.3, and the other two in Section 7.2.

4.3 Variability of plans

Here are some examples of plans generated by the system.

<u>Plan 1</u>	<u>Plan 2</u>
designate-system-devices	**designate-system-devices**
check-system-limits	**check-system-limits**
	add-component(slow-channel)
	check-system-limits
check-serial-limits	**check-serial-limits**
check-major-components	**check-major-components**
config-serial-devices	**config-serial-devices**
config-fast-devices	config-fast-devices
config-slow-devices	**config-slow-devices**
config-cardcages	**config-cardcages**
check-redundancy	**check-redundancy**

<u>Plan 3</u>	<u>Plan 4</u>
designate-system-devices	
check-system-limits	
check-serial-limits	**check-serial-limits**
report-missing-component	**config-serial-devices**
check-serial-limits	assess-users
report-missing-component	guess-fast-disc-config
check-serial-limits	**designate-system-devices**
check-major-components	**check-system-limits**
config-serial-devices	**check-major-components**
config-slow-devices	**config-slow-devices**
config-cardcages	**config-cardcages**
check-redundancy	**check-redundancy**

Figure 7: Some plans for order checking

They are given in the form of sequences of steps (methods) to be carried out.

1. The order is correct, or at least it is possible to configure a legal system from the components given, which include fast discs (these are connected via fibre optic links). The plan generated by the system is as shown in Figure 7[1]. The significance of the bold face type will be explained in Section 4.4.
2. An order similar to 1, but five cables have been omitted from the order.
3. Similar to 1, but with only "slow" devices, and we have ordered insufficient numbers of serial ports: two more groups of these are needed.
4. There are no disc drives at all present on the order. However, there are fibre link ("fast") cables present. We would guess, therefore, that there should be some fast discs present, and we can use information present from other components (such as the fact that there are certain numbers of terminals on the order) to make a guess as to a likely disc configuration for the system.

4.4 Is there some commonality?

Certain tasks (*e.g.* configuring the serial devices — terminals and serial printers) must always be carried out if the task is to be successfully completed. These were in bold face in the example plans above. However, they do not always take place at exactly the same point in the checking process, and some of them are not performed the same number of times in each plan. There are other tasks which may be performed either not at all, once, or many times for a particular order, *e.g.* adding missing components. Lastly, there are tasks which are performed only occasionally, *e.g.* if there is a major component completely missing from the order as the result of (say) a transcription error, such as the disc drives, then the system must try to extract from the other parts of the order and consequent partial configuration enough knowledge of the system specifications to be able to postulate a disc configuration in order to be able to check the overall parameters of the system.

We see, therefore, that there are several tasks which are common to all order checks, and that probably a high percentage of orders will conform to a very similar pattern, identical except in the number and type of relatively minor components which have to be added to the order. However, there is a minority of cases which do not conform to this pattern yet contain most of the same subtasks, which can be thought of as major perturbations of the "basic task". We require a system with enough flexibility to cater for all these cases.

5 Architecture of the planning system

5.1 The planner

Each method has an associated algorithm. We call these **tactics**. In order to construct a plan which will, when executed, successfully carry out the checking process, we need to find the correct tactics to use, and the correct sequence in which to apply them.

At the start, we have an input which is the result of decoding the order. It consists of a list of components. We search through the available methods to find an **applicable method** by comparing the input with the slots in the method frames as follows.

[1] The arguments to the methods have been omitted, for simplicity

The **input** slot gives the form which must be matched by the input to the method. Note that the list of components given in the input slot is not complete; it indicates the syntax of component subterms which *must* be present (in any order) in the input term. The planner has access to a matcher which checks that all terms present in the input slot of the method are also present in the input term. The **preconditions** slot specifies other conditions, not covered by matching, which must be true of the input. The **output** and **effects** slots show the state which would result if the **tactic** were to be applied successfully (given that the input matches and the preconditions hold). Running the effects on the input produces the output, which then becomes the new input term and another applicable method is sought. This process continues until the plan is complete, which is signalled by an empty effects slot in the last applicable method.

5.2 The methods needed

Two typical methods are given in Figures 8 and 9.

Name	config-slow-devices(components(C),components(C1))
input	components([slow-channel(L), slow-devices(D,T,P)])
output	components(C1)
preconditions	[checked(system),simple(slow-channel(L))]
effects	[delete(slow-devices(D,T,P),C,C2), replace(slow-channel(L),slow-channel(S),C2,C4)]
tactic	config-slow-devices(components(C),components(C1))

Figure 8: Method for configuring slow system devices

Name	check-redundancy(components(C),components(C1))
input	components([cardcage(L1)])
output	components(C1)
preconditions	[¬ simple(cardcage(L1)), ¬ member(slow-channel(S),C),...]
effects	[ ]
tactic	check-redundancy(components(C),components(C1))

Figure 9: Method for checking redundant components

Figure 8 shows the method for configuring the "slow" system devices. This method is only appropriate if there are such devices, together with channels to connect them. This is checked by attempting to match the input term with the input slot. However, this is not enough to ensure applicability of the method. Before carrying out configuration, which is relatively expensive in terms of computational effort, there are certain checks (using knowledge about system limits for devices of various kinds and other constraints) which should have been done first. The first condition in the precondition slot determines whether this is the case.

Before explaining the second precondition, let us examine the effects slot. Remember that we are planning the task, not executing it. We have already said that configuration involves considerable computational effort. We do not really wish to carry out the configuration now, at the planning stage: we just want to ensure that it can be done (by checking the input and preconditions), and record the "essential effects" that performing the configuration *would* have, if executed, in such a way as to be useful when checking the input and preconditions slots of methods further along the plan.

The essential effect of this method is to take two terms of the form

$$\textit{slow-channel}([1, 2, 3, \ldots])$$
and
$$\textit{slow-devices}([(Dev_1, N_1), (Dev_2, N_2), \ldots]),$$

and attach the $N_1 + N_2 + \ldots$ Devices (of the various types $Dev_1, Dev_2, \ldots$) to the channels, which is shown by giving each channel 1,2,3,... an argument consisting of a list of the devices connected via that particular channel. We therefore end up with a modified channel term

$$\textit{slow-channel}([(1, Devicelist_1), (2, Devicelist_2), \ldots]),$$

and we should delete the *slow-devices*$([(Dev_1, N_1), (Dev_2, N_2), \ldots])$ term, since these devices have now been allocated.

It can be seen that the effects slot, does indeed, when run on the input, delete the *slow-devices* term and replace the *slow-channel*(L) term. To have satisfied the preconditions, L must be of *simple* form, which simply means that L is a list of integers. Our interest in the argument of *slow-channel* after configuration, as far as *planning* goes, is merely that it is *not simple*: we have no desire to instantiate it to show the exact way in which the devices are allocated. For future planning, we want to know only that *config-slow-devices* has been carried out, so that we neither attempt to repeat this step, nor apply methods which are appropriate only before (or instead of) *config-slow-devices*.

Running the effects slot gives us a variable argument to *slow-channel*, and variable arguments cannot be *simple*. Hence in this case, the effects slot of the method makes one of the method's preconditions *false*, ensuring that the method is never applicable twice in one plan.

The output slot ensures that the input for the next loop of the planning process is of the correct form. Note that as planning proceeds, the output and input structures change from instantiated terms of one format to partially uninstantiated terms of possibly different formats. Some methods need certain subterms to be instantiated: for instance, to check whether there are enough fast channels for the fast devices we must at least know how many of each of these components there are; however, to check whether the devices have been allocated to channels we only need to know that the channel term is not of the form *slow-channel*$([1, 2, \ldots])$. Hence the method for allocating devices simply needs to replace $[1, 2, \ldots]$ by a variable, not perform the allocation.

The method for checking redundant components on the order list is shown in Figure 9. This should only be done at the very end of a plan. The planning stage needs only to add the method to the otherwise complete plan, and no computation (*e.g.* to find "empty" ports) or rewriting (*e.g.* to replace lists containing empty ports by lists with those ports deleted) is necessary, since no further *planning* is needed. Hence the effects slot is empty. The preconditions slot ensures that there is a non-simple cardcage term, and that other terms have, by now, been deleted as devices are allocated to channels and ports, and interface cards for the channels are allocated to cardcages.

6 Proof plans and configuration

6.1 The basic case

Suppose the decoded order turns out to be correct. Then the stages needed are

1. Check whether basic system limits are exceeded, *i.e.* there are enough slots for disc drives, enough cardcages for the cards to be configured, enough ports and cables for the various devices, and so on.
2. Give the system specifications, *i.e.* number of active jobs supported, total mass storage capacity, *etc.*
3. Configure the serial devices (terminals, PCs, serial printers).
4. Configure the system devices, both fast and slow devices (disc and tape drives, system printers).
5. Configure the interface cards in the cardcages.
6. Check for redundant components, *i.e.* components which feature in the order but which are not connected to anything in the configured system.

Suppose now that there was a channel missing from the order, such that the system devices cannot be configured.

- The method to check the system devices is applicable.
- This time, however, running the effects of *check-system-limits* will cause the assertion $missing(channel)$ to the output term, instead of the assertion $checked(system)$, which would have been the result in the previous case, when no components were found to be missing.
- One of the preconditions for the method to configure the system devices is $checked(system)$, which is *false* if no term $checked(system)$ appears in the input term for the method.
- However, $missing(X)$ is a precondition for the method *add-component*, which is then applicable.

In this case, the resulting plan would be

1. Check whether basic system limits are exceeded.

   Add component (a channel).

   Check (again) whether basic system limits are exceeded.

2. Give the system specifications.

3. Configure the serial devices.

4. Configure the system devices.

5. Configure the interface cards in the cardcages.

6. Check for redundant components.

Note that in this case, we need to recheck all basic system limits, since adding a further channel could invalidate the check on the cardcages, if its addition takes the total number of channels beyond the limit for the number of cardcages on the order. If it had been the number of cardcages which was deficient, rather than the number of channels, then *only* the cardcages would need to be checked. In this case, the method *check-system-limits*, which consists of two submethods *check-interface-limits* and *check-cardcage-limits* would be replaced in the plan by the submethod *check-cardcage-limits*. (The argument to the method *add-component* would be different too, of course.)

6.2 Major variations on the theme

Suppose the order code for the disc drives was mistyped so that no disc storage at all appears on the order. Note that if only some of the disc drives are missed off, the basic plan found above will suffice. Severe curtailment will result in inconsistencies being reported at the stage where system specifications are deduced.

The complete absence of disc drives signals to any human expert that a significantly different case is to be dealt with. Our system responds to this by virtue of the fact that one of the preconditions for carrying out the system checks is invalid. Checking whether there are enough channels for all the system devices becomes a meaningless operation when we already know from the start that there are major devices missing.

Methods which in all our previous cases would have been applicable are now *not applicable*. Those methods which are independent of system devices remain applicable, notably the checking and configuration of the serial devices. Another method, *assess-users* is applicable, and has as one of its effects the provision of parameters which, if known, ensure that the preconditions for the *guess-fast-disc-config* method hold, and this latter method becomes *applicable*.

While it is not claimed that the discs configured by such a method will be identical to those really desired by the user, nevertheless the ability to carry out this operation ensures that the system constraints being checked in subsequent methods (now, finally, *applicable*) are realistic — that is to say, of the correct order of magnitude. The user is alerted to the fact that disc drives are missing, and any other errors independent of the error in disc drive ordering can be given *at the same time*. This is preferable to the system aborting when much useful partial configuration could still be done, and it may be inconvenient or impossible to contact the originator of the order after every separate mistake is discovered.

7 Implementation Issues

7.1 The emergence of basic proof plans

Although there is a great deal of variability in the plans which emerge during the testing of this system, it is nevertheless the case that there is a basic "structure" to a large number of emergent plans. That is, if there are at most a few minor components missing, and there are both slow devices and fast devices present, then the plan could be summarised thus:

1. Check everything, adding components as necessary
2. Configure everything
3. Remove redundant items

In fact, if there are no fast devices present, an equivalent plan may be formed, using a subset of methods within the first two "supermethods". This leads us to formulate such supermethods, whose preconditions may be the union of the preconditions of their constituent submethods, whose submethods dovetail in that the preconditions of the next method are guaranteed by the effects of the current method, and whose overall effects are implied by the applicability and execution of the conjunction of submethods.

The situation is illustrated in Figures 10, 11, 12 and 13. Figure 10 shows a representative subset of the methods available for consideration by the planner. At every stage, several of these may be applicable; the planner must steer a path through the task, consisting of a sequence of applicable methods dovetailing together.

In Figure 11 we see the result for the majority of cases, where there are at most only minor components missing. The arrows follow the progress of the plan from start to finish.

In Figure 12, the boxes round certain of the methods represent the "three point plan" given above. This can be thought of the proof plan for these "basic" cases.

That such a plan will not always suffice is illustrated in Figure 13. Here, there are major devices (the disc drives) missing from the order. Thus the preconditions for the supermethod *basic-check* are not satisfied and the proof plan of Figure 12 will not do. Instead, the planner must construct a plan "from scratch", using a suitable sequence of the methods available.

One supermethod is shown in Figure 14. We shall refer to *basic-check* as the *supermethod*, and to *check-system-limits* and *check-serial-limits* as its constituent *submethods*.

The input and preconditions slots check that the supermethod is applicable, in the same way as was explained for the other methods of Figures 8 and 9. The supermethod may be thought of as "*check-serial-limits*, **then** *check-system-limits*". Its effects slot runs the effects of *check-system-limits* on the initial input to the supermethod (note that the preconditions of the first submethod must be either subsumed or implied by the preconditions of the supermethod); the effects of the first submethod, together with the preconditions of the supermethod, must ensure the preconditions of the next (and last) submethod. *Applicable* then runs the effects of this submethod on the output of the first submethod to produce the final output of the supermethod.

Figure 10: Methods available to the planner

Figure 11: Plan for dealing with correct or nearly correct orders

Figure 12: Basic proof plan, for dealing with correct or nearly correct orders

Figure 13: Plan for dealing with orders in which disc drives have been omitted

Name	basic-check(components(C),components(C1))
input	components([cardcage(L1),slow-channel(L2), serial-devices(terminals(T),printers(P)),...]),
output	components(C1)
preconditions	[...,simple(cardcage(L1)),simple(slow-channel(L2)),...],
effects	[applicable(check-serial-limits(components(C),components(C2))), applicable(check-system-limits(components(C2),components(C1)))],
tactic	basic-check(components(C),components(C1))

Figure 14: Basic Check method frame

The explicit representation of such methods has considerable effect in directing the search space in cases where such methods are applicable.

7.2 Strength of preconditions

We turn now to the question of whether plans, if found, can always successfully be executed.

A method will be deemed *applicable* if its preconditions (including the matching of the input term) are satisfied. If, for all methods, satisfaction of the preconditions also implies that the method will be *executable* at that stage in the process (which is the execution of the plan), then the plan can be guaranteed to succeed.

Take, as an example, the tactic to allocate "slow" devices to channels. The initial input consists of a list of devices $[Device_1, Device_2, \ldots]$ and a list of channels $[1, 2, \ldots]$. $Device_1$ can be allocated to the first channel; $Device_2$ may be added to the second channel (if one such exists) or, alternatively, to the first channel, which it will then share with $Device_1$. The remaining devices may be allocated similarly, to an empty channel (if there is one), or to one of the channels already occupied.

There is not a free choice, however, owing to constraints of two main kinds:

1. Limits of the total number of devices which may be connected via one channel, or of a particular type of device which may be so connected. *e.g.* no more than six devices in total may be connected to one channel, of which not more than four may be disc drives. We call these *upper-bound constraints.*
2. Some devices may not share a channel with others, *e.g.* a particular model of printer may not share a channel with any disc drive; the system disc drive should not share with the system backup tape (whatever models these are). We call these *not-share constraints.*

It is, in theory, not enough for the preconditions of the corresponding method for this tactic to check the *upper-bound* constraints but not the *not-share* constraints, if the resulting plan containing this method is to be guaranteed to succeed. Unfortunately, while it is computationally cheap to check the upper-bound constraints, for the *not-share* constraints it is impossible to find a set of preconditions which will guarantee success under *all circumstances* that is not tantamount to running the tactic. Since we want the computational effort of planning to be less than that of execution (otherwise we can only lose in overall efficiency by imposing the extra overhead of planning), this will not do.

By *all circumstances*, we mean allowing for any state of the knowledge base, *i.e.* with arbitrary additional constraints of the form $not\text{-}share(X, Y)$.

The problem is equivalent to the map-colouring problem (finding the minimum number of different colours such that no adjacent countries are coloured the same), since the aim is to find the minimum number of sets containing mutually compatible devices.

As such, results from graph theory show that, in general, we can only find upper and lower bounds for the solution. In cases where the upper and lower bounds do not coincide, there may be much computational effort in calculating the minimum, tantamount to actually performing the allocation. If, rather than applying this effort, the upper bound is taken, this may result in over-ordering of channel components. On the other hand, if the lower bound is taken, the plan may fail.

So, if we have a set of constraints which throws up discrepancies between lower and upper bounds, how should we tackle this? We need to answer the following questions:

1. In practice, do additional *not-share* constraints on placings *ever* cause failure to allocate devices, in the cases where there are numerically enough slots? In some systems, there are not many of these constraints, and heuristic guidelines used by sales people and configuration experts tend to lead to "over-ordering" of channels on a scale which ensures that cases in which there is a large discrepancy between upper and lower bounds do not occur. But suppose we tried to generalise to "maxed-out"[2]systems which have many more of these constraints?

2. In many actual cases, success can be guaranteed, if one of the following holds:
   - no constraints exist
   - the total number of constraints is less than the number of slots
   - the upper and lower bounds coincide.

In other cases, we must choose either to over- or to under-specify conditions.

In extremis, we can set preconditions which do not include this kind of fine checking. It is this which has been done in this prototype implementation, given the paucity of *not-share* constraints in the test computer systems considered. This might well not be satisfactory for other computer systems.

7.3 Distribution of effort between methods and tactics

The method for allocating devices to "slow channels" as implemented leaves all the effort of actually allocating devices to the tactic. Even if the tactic can be guaranteed to succeed, there could be much search and backtracking involved in finding a legal allocation.

Much search at the object level could be avoided if this stage were planned. For instance, rather than having a method *config-slow-devices* we could apply a number of 'submethods' *allocate-slow-devices* (as many as there are devices), to give an ordering of device allocation aimed to cut down search. The supermethod *config-slow-devices* would order the devices so that the most constrained came first.

Clearly, this involves more computational effort at the planning stage. Empirical studies should reveal whether the extra overhead in plan formation is matched by increased efficiency at execution.

[2] *i.e.* everything is stretched to its maximum capacity

8 Conclusions

8.1 Tests

A prototype system based on the proof plan architecture has been implemented in PROLOG and tested on sample orders (both correct and erroneous), and seems to provide the flexibility required for the variety of problems which occur. There are methods for checking system limits for the various subsystems, methods for configuring these subsystems, and prototypical methods for dealing with degenerate cases — missing discs, for example.

The tests on which statistics are given below were provided at HP Research Labs, Bristol. The table gives runtimes in milliseconds for forming plans and for executing them.

Test	Plan formation	Plan execution
correct order	4.6	3.1
missing terminals	4.6	2.8
missing cables	4.7	3.3
wrong console	4.6	3.2

The plan formation time is around 60% of the total time to form and execute the plan. Of the execution time around two thirds is spent on the various configuration subtasks.

8.2 Benefits of the proof plan methodology

We claimed in Section 3.1 that proof plans gave rise to properties desirable in any expert system.

- Is the system *efficient*? Some results, for runtimes to form plans and execute those plans, are given in Section 8.1. Typically the time to execute a plan is dominated by the time taken to configure devices, notably slow system devices on channels. By planning, we avoid much of the object level search: in particular, we do not have to redo the expensive configuration stages.

  In addition to this, planning cuts down consideration of inappropriate methods. For instance, at the root node of the object level space there are typically ten to possible twelve branches. Although the branching rate could decline further down the tree (since it not appropriate to perform some tactics more than once in a given execution), this represents a tree of unmanageable proportions.

  In the planning space, however, there is frequently very little search, a suitable plan being found often with no backtracking; so far, this has proved to be the most suitable, or natural plan possible.

- *Generality*: each method may be used in many different plans, and a proof plan may be applicable to many cases. Many other plans can be generated by the system.

- *Maintainability*: the separation of factual from control knowledge means that either may be changed without affecting the other. Control knowledge is expressed by means of the methods. These may be added to as new partial strategies are learned. The knowledge base of component names, component specifications, and constraints is free-standing.

8.3 Summary

We have investigated whether the proof plan methodology can usefully be applied outwith mathematics, to a general knowledge based system. We have developed a prototype system for checking orders for large computer systems. This task typically requires a flexible approach if all possible combinations of errors are to be catered for, which is given by means of methods, which are specifications for the tactics actually used in the order checking task.

Developing the system has proved a useful adjunct to learning strategies for order checking; failures in the system under development usually initiate the adding of new methods to the existing system, or strengthening preconditions to existing methods, leaving the system otherwise intact. Maintenance is thus facilitated.

It is envisaged that this work can be built on in various ways

- Developing a system for configuration synthesis, *i.e.* creating orders and configurations to meet initial user system specifications. Whereas this is a distinct task (see Section 1), nevertheless much if the current work is relevant, since the object level is the same.
- Providing explanatory systems for both order checking and synthesis. Because the basic architecture of a proof plan system mirrors the strategy and the hierarchical organisation employed by human experts, it should be possible to base explanations (of options open to the user, of why a particular method was considered appropriate, *etc.*) at the right level of detail and in the correct context.

Acknowledgements

I should like to thank Alan Bundy and Frank van Harmelen (Mathematical Reasoning Group, Department of Artificial Intelligence, University of Edinburgh); and Richard Hull (Hewlett Packard Research Laboratories, Bristol) for feedback on earlier drafts of this paper.

References

Bachant, Judith and Soloway, Elliot. (March 1989). The engineering of XCON. *Communications of the ACM*, 32(3):311–317.

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 from Edinburgh as Research Paper No. 336.

Bundy, A., van Harmelen, F., Hesketh, J. and Smaill, A. (1991). Experiments with proof plans for induction. *Journal of Automated Reasoning*. In press. Earlier version available from Edinburgh as Research Paper No 413.

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.

McDermott, John. (1982). R1: a rule-based configurer of computer systems. *Artificial Intelligence*, 19:39–88.

Silver, B. (1985). *Meta-level inference: representing and learning control information in Artificial Intelligence*. North Holland, Revised version of the author's PhD thesis, DAI 1984.

Soloway, E., Bachant, J. and Jensen, K. (July 1987). Assessing the maintainability of XCON-in-RIME: coping with the problems of a very large rule-base. In *6th National Conference on Artificial Intelligence*, Philadelphia. Sponsered by the American Association for Artificial Intelligence.