Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1998), *12,* 345–356. Printed in the USA.
Copyright © 1998 Cambridge University Press 0890-0604/98 $12.50

Proof planning for maintainable configuration systems

HELEN LOWE,[1] MICHAL PECHOUCEK,[2] AND ALAN BUNDY[3]

[1]Department of Computer Studies, Glasgow Caledonian University, Cowcaddens Road, Glasgow, Scotland
[2]Gerstner Laboratory for Intelligent Decision Making, Czech Technical University in Prague
[3]Department of Artificial Intelligence, University of Edinburgh, England

(Received November 3, 1997; Accepted February 5, 1998)

Abstract

Configuration is a complex task generally involving varying measures of constraint satisfaction, optimization, and the management of soft constraints. Although many successful systems have been developed, these are often difficult to maintain and to generalize in rapidly changing domains. In this paper, we consider building intelligent knowledge-based systems with maintainability well to the fore in our requirements for such systems. We introduce two case studies: the initial proof of concept, which was in the domain of computer configuration, and a further field-tested study, the configuration of compressors. Central to our approach is the use of the proof planning technique, and the clean separation of different kinds of knowledge: factual, heuristic, and strategic.

Keywords: Configuration; Proof Planning; Maintainability; Knowledge-based Systems

1. INTRODUCTION

1.1. Configuration

The configuration task is generally perceived as a problem of assembling elements of a system together in such a way that no internal logical constraints are violated, or so that the extent of violation is minimal. Stefik (1995) defines four key elements:

- A **specification language**. The nature of the environment and the use of the system are reflected here. A specification language may include optimization criteria.
- A **submodel of parts** represents a catalogue of component parts. This submodel also describes the mutual interdependencies. Hence when a particular component is configured, there are links to all the other components needing to be considered.
- A **submodel for spatial arrangements** specifies the means of describing spatial arrangements of the components and thus defines what combinations of components are allowed.
- A **submodel of sharing** expresses conditions under which a component can satisfy more than one set of requirements. There can be exclusive use, limited sharing, unlimited sharing, serial reusability and measured capacity.

Formalizing a domain in logic, Najmann and Stein (1992) define configurations as mathematical structures with components:

- a set of objects
- a set of properties for each object (functionality—value pairs)
- a set of functionalities
- for each functionality, a set of values, an addition operator, and a test
- a set of demands (functionality—value pairs)

According to this model, the configuration process is a finite sequence of compositions of objects while a solution is a configuration object which the tests show satisfies the demands.

Some authors have given a logic-based description of configuration tasks, for example, Klein (1996). The development of logical formalisms guarantees soundness of resulting solutions. The particular constructive type theory of configuration developed in Lowe (1993b) goes further in that

Reprint requests to: Helen Lowe, Department of Computer Studies, Glasgow Caledonian University, Cowcaddens Road, Glasgow, Scotland. E-mail: H.Lowe@gcal.ac.uk.

https://doi.org/10.1017/S0890060498124071 Published online by Cambridge University Press

sound configuration objects are synthesized from the specification of such an object (see Section 2.2). We see that this view is very strongly related to that of constraint-based approaches (see, e.g., Faltings & Weigel, 1994). We shall see that, although it is not explicitly a constraint-based system, our approach may be viewed as "constraints as types", where many of the constraints are satisfied by allowing only objects of the correct type to be synthesized.

However, while this takes care of some classes of constraint, those remaining tend to be more difficult to manage. The difficulty, certainly in the domains we have looked at, is that the problem is essentially *underconstrained*. It is more akin to design, of which, in fact, configuration is one component (Brown, 1996). The user wishes to explore the design space. How to present (only) *essentially different* designs for perusal, rather than a whole host of similar ones, is a major control problem. We use *proof planning* (see Section 2) to represent and generate strategies for exploring the design space in an efficient manner. These strategies are expressed explicitly, and separately, from other kinds of knowledge.

Many authors (see, e.g., Steels & McDermott, 1993) have pointed out the maintenance problems faced when managing systems in which product information changes often. Mannisto et al. (1996) propose a generic structure model to support different views and classifications of the same components evolving over time. In our view, the problem he mentions, that the original engineers may not understand the way the products are described in the system, is a consequence of mixing different kinds of knowledge. Our approach, demonstrated in the two case studies described in this paper, necessitates a clean separation of object-level, heuristic, and control (strategic) knowledge, which could be separately maintained with the aid of appropriate user interfaces.

1.2. Two configuration case studies

We now introduce the problem domains of our two case studies illustrating our approach to the configuration task, followed in Section 1.4 by an overview of the different types of knowledge which must be represented and the maintenance problems arising from these.

1.2.1. Computer configuration

As an initial proof of concept, we looked at the problem of how to synthesize a configuration which meets a specification of a computer system, using data and domain expertise from personnel at Hewlett Packart, Bristol. This synthesis should result in a term representing all the components needed, together with details of the connections between them.

As we shall show in Section 1.4, the careful separation of object-level knowledge from meta-level control and heuristic knowledge is an important feature of our approach. It has the benefit of facilitating maintenance, in a domain which traditionally has been bedevilled by maintenance overheads. In our configuration systems this separation is both strict and explicit. We shall show how object-level knowledge may be extracted and formally represented in such a way as to allow the utilization of techniques analogous to program synthesis, to perform tasks such as synthesizing computer configurations which meet specifications. Such an approach makes the task of maintaining knowledge bases more tractable and reliable.

1.2.2. Compressor configuration

We followed up our initial proof of concept with a further experiment. CompAir Reavell Ltd, a member of the Siebe group, manufactures high-pressure gas compressors mainly for the naval, NGV, and breathing air markets. Our objective was to produce an automatic configurer for specifying compressors in an engineer-and-made-to-order context. The system was required to present a logically sequenced order of questions, together with all legal options, to the user. The system should then price the solution, create the construction description number, and set up the final quotation document.

Unlike the previous case study, the client wanted an interactive system, with the user presented with legal choices at each of the main stages. The system was to present an appropriate and logically sequenced series of questions, complemented with a set of all the legal options, in order to facilitate the global product specification. There are two main clusters of decision process to consider. Firstly, there are several components that constitute a basic solution; once these have been configured, it is possible to attach a ballpark figure for the final cost of the compressor. Budget-related reasoning should be carried out at this stage. The rest of the configuration, namely customer-specific settings and the addition of optional accessories, should be elaborated afterwards, and the cost refined component by component from that point.

Given the problem of rapidly changing product lines, a prime goal was for the system to be readily maintainable. Lowe (1994) claimed that the proof planning methodology should facilitate maintenance: now we had a chance to test this hypothesis in the field. We hoped that the particular formalism chosen, expressed in logic and implemented in Prolog, with its separation of knowledge and control, would facilitate the maintenance of all types of knowledge in the system. To this end, a prototype system was built and underwent field-testing at CompAir, including tests for maintainability.

1.3. Classification of knowledge

1.3.1. Object-level knowledge

Our first kind of knowledge is *factual*. We represent these facts as *object level axioms*. They include attributes of particular components, general configuration rules, and limits. For example,

1. No more than six devices may be connected to a component *example-channel* (of type *channel*);
2. No more than four objects of type *disk-drive* may be connected to it.
3. No more than four components of type *card-cage* in a configuration;
4. No more than four objects of type *channel* to be configured in them.

Each of these four limits is a *hard constraint*, in that they may never be relaxed. Any object not conforming to them is not legal.

1.3.2. Heuristics

Secondly, we have *heuristic* knowledge. Suppose that it has been discovered that configuring the maximum number of components legally possible in a configuration *may* lead to an inefficiently running computer system. Suppose in the examples of object-level knowledge above, these heuristics amount to more stringent limits as follows:

1. No more than five devices should be connected to an *ex-channel*.
2. No more than three objects of type *disk-drive* should be connected to an *ex-channel*.
3. No more than four *card-cages* in a configuration (no change from the above).
4. No more than three objects of type *channel* to be configured in an *ex-card cage*.

The changes from the previously given limits represent relaxed, *soft limits*, that is, these limits are desirable for some reason and in some sense, but objects not conforming to them may still be legal.

Soft compatibility constraints are often discovered during the lifetime of a component.

1.3.3. Control knowledge

Thirdly, we have *control* knowledge, for example, knowledge about the order in which subtasks should be carried out. An example of top-level strategy might be

1. Try to find a configuration which obeys all heuristic limits.
2. If not possible, try to find a configuration which breaks as few heuristics as possible.

Here, we are treating all heuristics as equally important. However, this leads to us “preferring” breaches of the *channel-per-card cage* heuristic, since one breach here gives us a “breathing space” while we load devices on to the “extra” channel up to heuristic limits before we have to consider any more breaches. So the strategy can be expanded as follows:

1. Try to find a configuration which obeys all heuristic limits.
2. If not possible, ignore the heuristic: No more than three objects of type *channel* to be configured in an *ex-card cage*, but try to find a configuration which meets both the other heuristics.
3. Otherwise, just try to find a legal configuration (perhaps optimal with respect to price).

Apart from managing heuristics, there are other types of control knowledge. For example, when configuring devices on channels, it pays to configure the most restricted devices first, that is, devices which may not share channels with certain other devices in the configuration. On a more global level, there are advantages to performing the task in a particular order. There may be no universally appropriate order, but, given particular specifications, there may be a way of ordering the various subtasks so as to cut out excessive search. These strategies can be expressed as proof plans.

1.4. Knowledge and the maintenance problem

Separating different kinds of knowledge into classes enables us to manage them separately. In the case of computer configuration, it is very often the case that the price list and the product list changes rapidly, whereas knowledge of how to configure solutions stays unchanged for a long time.

This separation enables each type of knowledge to be encoded *declaratively* if we so wish. This is important if we are to be able to check that the formalism given accords with our understanding of the semantics, and this is important from the maintainability aspect. We need to be able to check, separately and independently: (1) That the facts represented are “true”, or at least what we intend (e.g., the rules of configuration). (2) That procedures are captured correctly.

- It is clear that object level knowledge must be updated as new products come into being and others become obsolete.
- Heuristic knowledge is, or should be, changing with time and circumstance; for instance, the fact that particular configurations lead to inefficiency may only be learned from experience of actual running configurations. Conversely, it may be rendered obsolete as products improve.
- Explicitly and separately held control knowledge enables us to update the configuration as whole structures or new kinds of products are added or altered. This may mean that the system can be generalized if sales policy changes, or if it is required to be used for other, similar tasks.

If the three kinds of knowledge 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. We cannot make the claim that our systems will be easily maintainable in the face of all future developments as this could involve sea-changes in technology. However, it seems 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 also remain unscathed.

2. PROOF PLANNING

2.1. Introduction to proof planning

A proof plan is a means of expressing the commonality between members of the same "family" of proofs while allowing sufficient flexibility and adaptability to prove a large number of different theorems. Proof plans provide an *explicit* expression of strategies for automated reasoning by describing tactics in terms of the preconditions under which they are applicable and their effects if applied. This specification of a tactic in terms of preconditions and effects is called a method, and methods provide a basis for combining tactics to form a complete plan—in general, a tree structure—which, if executed, will carry out a reasoning task.

Bundy (1987) proposed that this, originally developed for use in theorem proving and program synthesis, be extended to intelligent knowledge-based systems (IKBS) in general. The desirable properties of the technique would be

1. *Efficiency*, because the combinatorial explosion is avoided, or at least greatly mitigated.
2. *Generality*, because a proof plan may be applicable to many cases.
3. *Maintainability*, because the separation of factual knowledge from heuristic and control knowledge means that either may be changed without affecting the other.
4. *Explanatory power*, because control decisions can be explained at the appropriate level, rather than by generating long chains of low-level choice points in the inference process.

These properties are important for *any* knowledge-based system. Thus, proof plans can provide a useful vehicle for expressing strategies for problem solving in other domains, including nonmathematical ones.

2.2. The proofs as programs paradigm

Bates and Constable (1985) give a method for synthesizing algorithms from proofs. If we express the relationship between the input and the output of a program as spec $(input, output)$, then an algorithm may be synthesized by finding a *constructive* proof of the theorem:

$$\forall input \cdot \exists output \cdot spec(input, output),$$

and from this extracting the algorithm *alg* such that

$$\forall input \cdot spec[(input, alg(input)].$$

We use an analogous technique for synthesizing configurations. Suppose we have a specification, $spec(c)$, for a computer configuration, c. For example, the specification might state (translated into informal language) that the configuration should have a certain number of terminals for running particular applications, that it should have at least a certain amount of disk storage, that it should have printers capable of certain speeds, tasks, etc. The synthesized c should

- Obey the explicit terms of the specification—have the correct number and type of terminals, printers, disk drives, etc.
- Be a *legal* configuration—function correctly, obeying the general laws for configurations, that is, possess a processor of sufficient power, enough memory, backup devices, *etc*. In addition, all devices must be correctly connected up. These laws may be formalized as a general theory of configuration.

We synthesize such a configuration from the object-level theory by proving the theorem (more properly, the conjecture: we could be given an unrealizable specification)

$$\exists c \cdot spec(c). \tag{1}$$

where c is a well-formed object of type *configuration*, and $spec(c)$ is the specification that c must satisfy; it includes the customer's inputs as to certain values and properties of the resulting system.

An alternative way of thinking about Eq. (1) is to introduce a meta-variable C and to prove

$$spec(C), \tag{2}$$

where $spec(C)$ is a conjunct of goals expressing the required properties of the configuration.

In proving conjecture, Eq. (2), C is instantiated to a well-formed term. This is a gradual process; C starts out as a simple meta-variable but acquires some structure early on in the proof; for example,

$$C \equiv proc :: L,$$

where *proc* is instantiated but L is not: read this as "C is a processor *proc* and some other terms". Later in the proof L in turn acquires some structure, and by the end of the proof it is fully instantiated.

2.3. Tactics

A tactic is a program encapsulating a significant proof step with its attendant lower level steps. The latter are typically ones of less interest to the user of the system, and correspondingly harder to keep track of. We would prefer these to be taken care of automatically so that we can concentrate on the "interesting" proof steps. For example, let us consider a Prolog tactic to configure a device, shown in Figure 1. The arguments of *configure-device* are the device to be configured, an interface channel (IC), and the configuration (C) in which these occur.

To configure a device, we need a cable for it, we need to connect the device *via* the cable to the interface, and the whole construction—the interface with the device-cable pair—must be well formed. In Figure 1, *connect-cable/3* ensures a cable, *connected-via/3* takes care of the connection *via* the interface, and *type/2* is the well-formedness check.

Reasoning at the level of the tactic facilitates the search for an acceptable solution whilst ensuring that any such solutions found will be legal; that is, they ensure the soundness of the automatic configuration system.

2.4. Methods

A method is a specification for a tactic. Figure 2 shows the general structure of a method.

Methods have slots for method name; *input*, which the input goal must match; *preconditions*, which are conditions which must be true of the input if the method is to be applicable; *output*, which will match the rewritten input goal if the method is applied; *effects*, which are conditions on this output goal if the method is applied; and the specified *tactic*: the program to be applied to the goal at this point. Figure 3 gives the method which specifies the *configure-device* tactic we saw earlier.

In this method, the goal to be proved must have the form *configure*(*Device*, *IC*, *C*), and the output of the method is *nil*: this applies to all *terminating methods* where no further rewriting will be necessary if the tactic is successfully applied. The preconditions state, in order, that there is a device to be configured in C, that IC is of type *Type*, and that the number of slots available of the correct type is greater than zero. The effect of the method is to reduce the number of such slots by one.

One nice feature of specifying tactics by methods in this way is that it models the "user" or "customer view" of configuration, as opposed to the "engineer view", which is "modelled" by actually executing the tactic. This makes developing good explanation facilities a realistic possibility. This is not true of rule- or constraint-based systems which work at a low level, where the search space is more complex and reasons for the choices made may not be readily apparent.

More importantly from the point of view of maintenance considerations, the use of methods gives an explicit place for us to write control information. For example, the fact that the tactic to configure an interface channel in a card cage should only be run if there are spare card cage slots belongs in the preconditions of the *configure-device* method. The fact that a card cage should not be configured if there are completely empty card cages already present in the configuration finds its place in a *configure-cc* method.

Name	*method name*
input	*syntactic form of input goal*
output	*syntactic form of output goal*
preconditions	*. . . for method to be applicable*
effects	*. . . of applying tactic*
tactic	*program specified by method*

Fig. 2. Method structure.

2.5. Proof plans

In the context of configuration, we can think of proof plans as the expressions of meta-level strategies. The aim in proof planning is to find a plan tailor-made for the specification which will prove the particular theorem given to us of the form of Eq. (1). A sequence of applicable methods is found. If this sequence, which we call a *plan*, is executed, then every conjunct in the specification is proved and we are guaranteed that a well-formed configuration object meeting the specification will be instantiated as a by-product.

We can go further. High-level strategies for configuration can be developed, and encapsulated as proof plans, or super-methods. For example, let us consider the most basic strategy for computer configuration, which is

```
configure-device(Device, IC, C):-
                    connect-cable(Device, Cable, C),
                    connected-via(Device, IC, C),
                    type([Device, Cable, IC], ).
```

Fig. 1. An example tactic.

Name	*configure-device*
input	*configure(Device, IC, C)*
output	*nil*
preconditions	*Device is a device of C* *and IC : Type* *and Device needs slot of type Type* *and the number of slots available of type T is s(n)*
effects	*the number of slots available of type T is n*
tactic	*configure-device(Device, IC, C)*

Fig. 3. An example method.

1. Decide an appropriate processor from information in the specification.
2. Each processor fixes a kind of "template" configuration: set this up *via* matching.
3. Attend to explicit user needs as given by the specification.
4. Add essential components not explicitly specified.

Within this strategy, we might also want to control the order of configuration of devices, as it can be shown that some sequences are (heuristically) better than others. Super-methods are distinguished from other methods in that they call other methods (or super-methods) from within their effects slots. Some methods are iterators: for example, we may want to call a method to configure a device until there are no devices so far left unconfigured.

Within the overall guidance given by this plan, there is sufficient flexibility to cater for a variety of specifications. At the same time, the existence of a proof plan, which will be applied if possible, means that there is not a random choice of methods which could lead to legal but "unnatural" configurations being generated; moreover, backtracking in order to seek alternative solutions does not lead first merely to plans which contain the same methods, but applied in a different order—in other words, to trivially different solutions—but to configurations which are significantly different.

Various strategies have been developed, such as the constraint relaxation strategies referred to earlier. Another is the strategy employed in configuring computers to comply with cost guidelines. Here, the configuration objects referenced by methods are annotated so as to keep a running check on the approximate cost of the configuration. This cuts down on much unnecessary search, as branches leading to over-expensive non-solutions are pruned early from the search.

Again, this explicit representation of strategies means that we can have the benefit of efficiency whilst retaining a declarative, transparent system. The control knowledge does not have to be "hard-wired" deep in the program.

3. THE CLEM CONFIGURATION SYSTEM

3.1. Implementation

The main task addressed in designing the architecture of the prototype computer configuration system was how to separate control information (how to go about the configuration task, using heuristics if possible) from the object-level knowledge (ensuring that the configurations synthesized are always legal).

This system was implemented in around 5500 lines of Quintus Prolog, and runs on a SUN workstation. It consists of an object-level knowledge base (components, attributes, etc.) together with heuristic knowledge, tactics, methods (specifications for tactics), and a planning mechanism to guide inference in the system.

Examples of strategic and heuristic management knowledge have been given in Section 2. We now explain the rationale behind the various design decisions that we took in designing the types and object-level rules for an automated configuration system.

3.2. Types

Adopting a hierarchical structure for storing knowledge seems initially attractive. However, in a field which is changing rapidly (as computer technology is) this gives rise to considerable problems when attempting to fit new devices into a rigid framework. It is a common problem in artificial intelligence that initial classifications within frame-based systems and the like break down when new objects are introduced which defy the original classifications, or if the information is put to a different use. This leads to rethinking either the classification or the properties attached to slots or both; or else to messy exception-handling procedures. The problem in the computer hardware domain is that we cannot predict the course that technology will take. New products might cut across existing divisions: maintenance of the system would mean not simply updating the product data but also maintaining the structure. This would add an unnecessary overhead onto an already onerous task. Our aim was to make maintenance as straightforward as possible so that the knowledge base part of the system could be updated by people who currently maintain product information—people who do not necessarily have the expertise needed to maintain a structure tree for the knowledge base. We wanted to avoid the situation whereby, unless all future products conformed to the existing structure, the addition of just one "revolutionary" component would cause problems.

Our solution, therefore, was to adopt a fairly "flat"' type system. Individual components (processors, memory modules, terminals, disk drives, tape drives, printers, channels, cables, card cages, etc.) were represented as atomic types. There are two ways in which this knowledge may be maintained:

1. New components of existing types may be added (or old ones deleted).
2. New types are added (and old ones, which have no members left, deleted).

The first is done by adding the component to the knowledge base: the name of the component, together with its type; and its attributes as appropriate (e.g., for a disk, this would include its capacity). The second arises when a new *kind* of device is added. Inevitably there will be at least one member of the type. Each new component, together with its type, is added to the knowledge base in the normal way, but other information is also needed, such as how more complex terms may be built up using this type.

Other compound types used are list types (typically lists of objects needing to be grouped together in the configuration), and pairs, which we have already seen.

An object of type configuration is a member of a complex type, of the form

$$\begin{aligned}&\textit{processor}\ \text{list} \times \textit{memory}\ \text{list} \times \textit{device}\ \text{list}\\ &\quad\times \textit{connectors}\ \text{list} \times \textit{connections}\ \text{list}\end{aligned}$$

Note that a simple interface would allow non-AI-expert people to add domain knowledge, which can then be validated, although this was never implemented for this prototype.

3.3. Object-level rules

Object-level rules can take the form of facts as shown in Section 1.3.1 and are represented by simple Prolog ground clauses. Others can be regarded as axioms of the domain. As an example, let us consider the rules concerning the fact that, in any configuration, all interface cards must be configured in a card cage. So the definition of a legal configuration (or what it means for c to be a member of the type configuration) includes the conjunct

$$\forall ch{:}channel \cdot \exists cc{:}cardcage \cdot connected\text{-}via(ch) = cc.$$

In other words, for each and every interface *ch*, there is some card cage *cc*, such that *ch* is connected via *cc*.

4. THE ICON INDUSTRIAL CONFIGURATION SYSTEM

4.1. Strategy

ICON (Pechoucek, 1996) was programmed in LPA Prolog 3.1 in the MS-Windows environment, running on a 386 PC platform or better. Ladder logic was used for formalizing object-level knowledge about components of a compressor and attributes of a solution. Proof planning methods captured the inference knowledge. An ordered set of methods was used for expressing the decision process carried out by an expert in the field. The tactics of a method were used for storing the information about how to create the particular product number.

Proof planning introduced three phases of inference:

1. Planning stage.
2. Validation stage.
3. Execution stage.

In the planning stage, a user is asked to give as much information as possible in order to give a direction to the search for possible solutions. In the validation stage, the system offers the best found solution with a complete set of attributes. The user can either return back to the planning stage and redo some of his or her decisions, or else let the solution proceed further to the execution stage. In the execution stage the system creates the product number and the final quotation document. As with CLEM, planning takes place at the meta-level (user) view, so that the planning space is small relative to the underlying object-level search space.

4.2. Planning stage

The language of methods, the domain theory, and the system of higher level predicates facilitates creating an arbitrary planner and thus various planning behaviors. There were two completely different planners implemented within ICON in order to illustrate the generality and flexibility of the system.

The *User Assisted Planner* navigates the user through the space of possible attributes and prompts him or her for a value when necessary. It simulates the behavior of a quotation expert in the field. An example dialogue is shown in Figure 4 and a quotation in Figure 5.

The *Advanced Planner* can handle partially configured solutions. In such case the system allows the user to specify the attributes and optimization constraints he or she wish, and then allows the configurer to check the legality of the solution presented and to search through the attribute space in order to create the quotation automatically. It is notable that a first version of the Advanced Planner was implemented in just a single day.

4.3. Validation stage

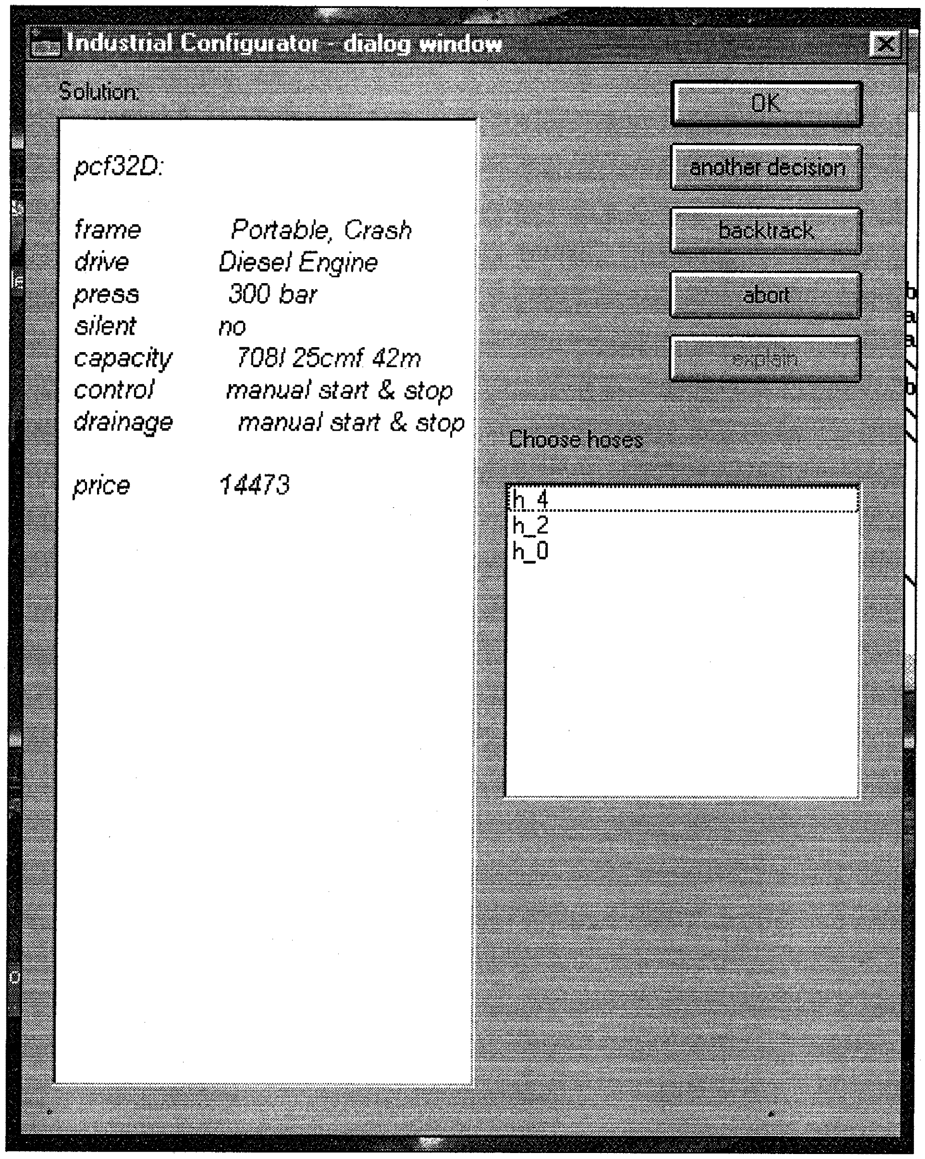
In the validation stage, the user decides whether he or she likes the solution found. Backtracking to the planning stage is an option. In the case of the advanced planning, the user may ask for another solution fulfilling the specification in question. Otherwise the program proceeds to the execution stage.

4.4. Execution stage

The system is engaged with the job of creating the product number and the final quotation document. There is a separate data base of rules about how to create a product number and the open output template. All of the text is editable and can be either printed out or saved in the conventional manner.

4.5. Object-level formalization

The object-level knowledge has been formalized by means of ladder logic, a well-known industrial representation. This formalism was chosen mainly because of its simplicity for non-experts. A parsing mechanism accepts an arbitrary ladder logic expression. Consequently, the knowledge engineer may use as complex an expression as he or she likes. This sort of freedom is a substantial virtue of the system.



pcf32D:

frame	Portable, Crash
drive	Diesel Engine
press	300 bar
silent	no
capacity	708l 25cmf 42m
control	manual start & stop
drainage	manual start & stop
price	14473

h_4
h_2
h_0

Fig. 4. Sample ICON dialogue.

4.6. Meta-level formalization

As already mentioned, the proof planning methodology introduces a slightly unusual but very efficient meta-level knowledge orientation. The ordered set of methods represents, at the meta-level, the decision process in question. When configuring the compressor, the entire quoting process can be viewed as the more or less structured ordering of the decisions to be made. Each particular decision is represented by a single method. From the maintenance point of view, the virtue of proof planning at the meta-level is the same as the virtue of the ladder logic at the factual level. It is easy to refine the method language when a new piece of control knowledge is acquired.

The preconditions of a method are intended to record all actions that need to be carried out before deciding whether a particular branch of a subtree suits the properties of a particular subsolution. Unlike the CLEM case study, where the user inputs the full specification up front, in ICON we need a user dialogue to be invoked whenever a decision between components must be made. Note that this is only activated in cases where the domain constraints have failed to restrict the choice to one.

5. TESTING

5.1. CLEM

We have tested the prototype system CLEM (Lowe, 1993*a*), and experimented with it over a wide variety of specifications, using alternative formulation of methods, and by add-

```
     pcf32D

  General model properties:

    Four Stage Air Cooled Compressor
    Frame wit Belt Dive & Guard
```

```
    Safety Valves on all stages
    2nd & 3rd Stage Oil & Moisture Separator
    Suction Filter with snorkel on engine set
    Charging Pressure Gauges
    Air Purity to BS4275/DIN3188
    Filtration Pressure Gauges
    Hours Run Meter
    High Air Temperature Switch
    Low Oil Pressure Switch
    Resilient Mount

    Drive specification:
    Power 18.5kW, Four stroke air Cooled Diesel Engine with Fuel Tank
    Starting Handle and 12 volts DC Electric Start.

  Specific model properties:

    drive           d_5409
    rounds          1800
    frame           Portable, Crash
    drive           Diesel Engine
    press           300 bar
    capacity        7081 25cmf 42m
    control         manual start & stop
    drainage        manual start & stop
    hoses           4                     XXX
    output          200/300 bar           XXX
    panel           remote
    connector       aga
    filtration      rs1
    portability     handles               XXX
    contains        DemisterVessel        XXX

  Total price is:                  XXXXX

  Product Identification Number:       pf25d8rr631hd
```

Fig. 5. An example product quotation.

ing new strategies. In addition, we successfully added new component details, to test the maintainability of the object-level knowledge of the system.

CLEM proved capable of handling all but very pathological specifications. It could successfully employ strategies for control and for taking design issues into account. We tested the maintainability of the system by adding a new processor and other components. This tested whether the object-level knowledge could be updated independently of the rest of the system. Maintainability of control information was tested throughout: control information tends to be learned gradually, and we were able to incorporate such understanding on an incremental basis, without needing to make major changes to the rest of the system.

Synthesizing a configuration in this system involves finding a plan, which is a sequence of methods, and then executing the plan, that is, running the corresponding tactics in the sequence given by the plan. One problem with using planning, or any kind of meta-level reasoning, is the overhead incurred. For very simple specifications, leading to small systems, the overhead is not cancelled out by the saving in execution time, when we compare the total time of planning execution with the time taken to find a solution using the object-level theory unaided. However, this is not the case when configuring large multi-user systems. In fact, the configurations do not have to be very large for planning to pay off (Lowe, 1993*b*). So one empirically proven advantage for this approach is that it is more efficient in finding the first solutions.

This is not the only advantage, however. One feature which was thought useful was the ability to search the planning space for alternative solutions, rather than having to execute each such solution before the next plan is found. In real life, the (human) configurer presents several solutions for the customer to choose from. The information provided at the planning level is sufficient for an informed choice to be made, since this level deals with components at the right level for the user: in terms of devices and attributes of whole configurations, rather than in terms of devices, cables, slots, etc. Only when an acceptable solution has been found at the planning level does this plan (and this only) need to be executed, to give the full details of the configuration.

Thus, not only does the proof planning approach benefit system maintenance and ensure sound solutions, it also facilitates exploration of alternative solutions. It should also be said that testing the system was also helped by this "two-stage" approach in which the most detailed information is presented only on execution, making it easier to check the top-level details of the synthesized systems at the planning level first.

5.2. ICON

ICON was warmly welcomed and well appreciated for meeting all specifications in the case of User Assisted Planner and for a worthwhile initiative in the case of the Advanced Planner. The knowledge base was tested and after a small series of refinements it seemed to behave quite like the experts. As a test case, a set of 20 customer specifications was used, representative enough to confirm the accuracy of the configurer. We evaluated

- How easy it was to capture the knowledge needed in the right form, and how long this took.
- The time taken to design and implement the prototype system.
- The performance of the system, tested by its potential users.
- How easy it would be for its users to maintain.

Two weeks were spent at CompAir Reavell carrying out the knowledge elicitation phase. Afterwards, it took around 8 weeks for one person to completely build and design the system. Two further weeks were then spent at CompAir, introducing the tool to sales people, quotation and construction departments, and management.

ICON met all requirements and typically achieved performance levels of reducing 2 d work to 1 min. One addition requested, however, was for an explanation facility, which was readily provided, to enhance the sales–customer relationship. It was thought that, with this facility, the service could be provided direct to customers *via* the Internet. Thus, with the explanation facility, the usage of the system went beyond what was originally envisaged.

A few inconsistencies were found during testing. These turned out to be easy to correct, for example a redundant attribute, some misleading vocabulary, incorporation of measurement units, and a couple of rule changes. These are typical of building such systems, and it is important that they were found and corrected easily.

For maintenance, a short Prolog tutorial was organized to explain maintenance of the knowledge base. After the tutorial session, the IT staff were asked to refine the global knowledge base in two ways:

1. Add a new set of compressors (water-cooled).
2. Add a new attribute (weight) to the set of properties.

This took one of the authors (with his expert knowledge of Prolog and his own system) 20 min. The CompAir staff averaged 30 min, and carried out the tasks correctly—this was considered a pleasing result, considering the complexity of the tasks involved and the inexperience of the staff.

The system code is well structured and self-explanatory. The openness and flexibility of the object-level formalism, the language of methods, and the lower-level predicates are illustrated by how fast and straightforward was the development of the Advanced Planner. Due to substantial field-testing, user friendliness, easy maintainability and enhancability, and overall system flexibility, ICON was successfully used at CompAir Reavel and made the quotation process considerably easier and faster.

6. RELATED AND FURTHER WORK

Other logic-based approaches (Klein, 1996; Najmann & Stein, 1992, Searls & Norton, 1990) focus on the object level. We have identified two other classes of knowledge—heuristic and strategic—which also benefit from logical formalism and separate, explicit representation, avoiding the traditional production rule systems' confusion of these classes with their consequent maintainability problems (McDermott, 1982). Our constructive type theory would seem to have much in common with various constraint-based approaches (Faltings & Weigel, 1994; Sabin & Freuder, 1996; Gelle & Weigel, 1996). The special class of constraint-based reasoning in which the configuration is being gradually refined (see ten Teije et al., 1996) is akin to our approach of gradually synthesizing a configuration from a specification by defining complex types, culminating with the *configuration* type, whose slots are eventually fully instantiated with subparts.

Further refinement of our knowledge classes could be sought by representing non-type constraints using a constraint-based system, probably by implementing systems in a constraint logic programming language rather than Prolog, as hitherto. Other possibilities would be to see the remaining constraints as "data type invariants" and to use VDM-style proof obligations to maintain the integrity of the synthesized configuration.

We have argued in this paper that proof planning can help manage complex knowledge bases by separating different kinds of knowledge, and search through large search spaces by means of its explicit search strategies. However, we would like to test this potential in more sophisticated domains, and by seeking new tasks within the existing domains requiring more sophisticated heuristics and strategies. Encapsulating expert strategies presents strong challenges for any approach.

Another useful extension would be to allow more interleaving between specification and configuration, and to allow the replaying of old plans on revised specifications. This has been more developed in ICON than in CLEM, where the complete specification had to be given at the start.

7. CONCLUSION

We have developed a formalization of the configuration domain in order to apply established theorem-proving techniques to the problem of configuring computers to meet specifications. This ensures the underlying soundness of the system, in that all solutions generated will at least be legal objects. Using this as a basis, we have been able to encapsulate the higher-level reasoning, used informally by human experts, in a more formal way in order to facilitate the generation of well-designed and natural solutions, and to generate these solutions as efficiently as possible by guiding search by means of meta-level reasoning techniques. We would not argue that we have discovered, much less implemented, all strategic knowledge brought to bear on this problem by human experts. However, the methodology of this approach enables this to be done incrementally. This is due to the fact that the meta-level knowledge is captured independently of the object-level theory. So not only can we maintain the system by the addition of new components, and even new types of components, but we can also experiment with new strategies. Given that human expert strategies are often opaque at first, there is the added psychological benefit that we can gain new insights into the process of human reasoning in a complex task. Using the proof planning methodology, we can carry out these kinds of modifications and experiments in a principled way which is easy to track.

Any student of systems analysis knows that maintainability is an important criterion for judging the success of a computer system. Usually this question is assessed with regard to systems which have been in use, in the field, for some time. This clearly has not been done with CLEM, which is merely a prototype. However, CLEM was not born, complete and perfectly formed, but evolved over a period of many months. Its expandability was, therefore, an important issue right from the start, and not simply something to worry about for the future.

ICON addresses the topic of maintainability and the ease of further enhancement, the main bottleneck of automated configuration. The entire system was designed carefully in this respect. After a short Prolog syntax tutorial and precise explanation of the system maintenance, ordinary staff from the IT department in CompAir managed successfully to enhance the system in the direction of attributes as well as in the direction of product types.

This paper advocates the proof planning programming methodology as an appropriate and convenient approach for design, development, maintenance, and enhancement of knowledge-based systems of this particular sort.

Proof planning knowledge orientation makes the knowledge-acquisition process more natural. Proof planning eases the development stage and thus considerably shortens the time needed for implementation. Proof planning facilitates easy maintenance and enhancement due to its natural knowledge orientation and natural knowledge formalization.

ACKNOWLEDGMENTS

We should like to thank everyone at Hewlett Packard Research Laboratories, CompAir Reavell Ltd, and the Mathematical Reasoning Group in the Department of Artificial Intelligence, University of Edinburgh, for their assistance and feedback. Our thanks go also to the anonymous referees who gave such helpful and constructive comments on an earlier version of this paper and helped to crystallize our ideas.

REFERENCES

Bates, J.L., & Constable, R.L. (1985). Proofs as programs. *ACM Transactions on Programming Languages and Systems 7(1)*, 113–136.

Brown, D.C. (1996). Some thoughts on configuration processes. *Proc. AAAI 1996 Fall Symposium on Configuration*, 75–83.

Bundy, A. (1987). How to improve the reliability of expert systems. In *Research and Development in Expert Systems IV*, (Moralee, S., Ed.), pp. 3–17. Cambridge University Press, New York.

Faltings, B., & Weigel, R. (1994). *Constraint-based knowledge representation for configuration systems*. Technical Report No. TR-94/59, Department d'Informatique, Laboratoire d'Intelligence Artificielle, Ecole Polytechnique Federale de Lausanne.

Gelle, E., & Weigel, R. (1996). Interactive configuration using constraint satisfaction techniques. *Proc. AAAI 1996 Fall Symposium on Configuration*, 37–44.

Klein, R. (1996). A logic-based description of configuration: The constructive problem solving approach. *Proc. AAAI 1996 Fall Symposium on Configuration*, 1–10.

Lowe, H. (1993*a*). *The CLEM configuration system, user manual and programmer manual*. Technical Paper 20, Department of Artificial Intelligence, Edinburgh, England.

Lowe, H. (1993*b*). *The application of proof plans to computer configuration problems*. Unpublished Ph.D. Thesis, University of Edinburgh.

Lowe, H. (1994). Proof planning: A methodology for developing AI systems involving design. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing 8*, 307–317 (special issue on Research Methodology).

Mannisto, T., Peltonen, H., & Sulonen, R. (1996). View to product configuration knowledge: Modelling and evolution. *Proc. AAAI 1996 Fall Symposium on Configuration*, 111–118.

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

Najmann, O., & Stein, B. (1992). A theoretical framework for configuration. In *Proc. Fifth IEAAIE*, 441–450.

Pechoucek, M. (1996). *Proof planning and industrial configuration*. Unpublished M.Sc. Dissertation, Department of Artificial Intelligence, University of Edinburgh, England.

Sabin, B., & Freuder, E.C. (1996). Configuration as composite constraint satisfaction. *Proc. AAAI 1996 Fall Symposium on Configuration*, 28–36.

Searls, D.B., & Norton, L.M (1990). Logic-based configuration with a semantic network. *Journal of Logic Programming 1*, 53–73.

Steels. L., & McDermott, J. (1993). *The Knowledge Level in Expert Systems*. Academic Press, Boston, Massachusetts.

Stefik, M. (1995). *Introduction to Knowledge Systems*. Morgan Kaufman, Los Altos, California.

ten Teije, A., van Harmelen, F., Schreiber, G., & Wielinga, B. (1996). Construction of problem-solving methods as parametric design. In *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, (Gaines, B.R. and Musen, M.A., Eds.), Vol. 1. SRDG Publications, University of Calgary, Canada.

Helen Lowe is a lecturer in Human Computer Interaction at Glasgow Caledonian University. She graduated in Mathematics from Manchester University in 1976 and obtained an MSc in Knowledge Based Systems, followed by a PhD degree at Edinburgh University under the supervision of Professor Alan Bundy in 1993.

Michal Pechoucek is a lecturer in Artificial Intelligence at Czech Technical University in Prague and a research fellow at Gerstner Laboratory for Intelligent Decision-Making. He graduated in Technical Cybernetics and obtained an MSc in Knowledge Based Systems at University of Edinburgh. He submitted his PhD thesis on Advanced Planning Techniques for Project Oriented Production at CTU in Prague.

Alan Bundy is a Professor in the Division of Informatics at the University of Edinburgh. He graduated in Mathematics in 1968 and gained his PhD in Mathematical Logic in 1971, both from Leicester University. Since 1971 he has been at the University of Edinburgh. His research is on the automation of mathematical reasoning.