Proof Planning: a Methodology for Developing AI Systems Involving Design

Helen Lowe

DAI Research Paper No. 662

Proof Planning: a Methodology for Developing AI Systems Involving Design *

Helen Lowe

Department of Artificial Intelligence,
University of Edinburgh
80 South Bridge
Edinburgh
EH1 1HN
UK
Tel: 44-31-650-2728
Fax: 44-31-650-6516
email: hcl@uk.ac.ed.aisb

Abstract

If we can express a domain theory in a logical formalism and formulate a task in that domain in mathematical terms, then this greatly facilitates the task of building sound knowledge based systems. However, it is not immediately obvious how the *design* aspects of such tasks, where these are an important feature of problem-solving, can be incorporated in this approach. We have used a methodology based on the development of *proof plans*. The ability to conduct our research according to the Popperian framework of hypothesis, validation, testing, and modification in response to empirical evidence — the hypothetico-deductive approach — seems essential to any rigorous scientific endeavour. Proof planning, we believe, is a method which readily exploits this inherently incrementalist approach, and could prove a powerful tool in designing AI systems.

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

1 Introduction

This paper is really a description of a case study to illustrate two different kinds of "method": a method for building AI systems to perform tasks which involve design issues; and a method for conducting research and investigation in AI. We built a prototype system for configuring computer hardware, the methodology which we used being that of *proof planning*; and the framework in which we investigated was strongly Popperian in nature. This is hardly new, but tends to be under-emphasized in AI unlike more traditional areas of science and engineering.

There is a strong relationship between these two kinds of methodology in that the proof planning technique lends itself to the ideal Popperian cycle of hypothesis–experiment–test–reformulate. This *hypothetico-deductive* approach seems essential for tackling design tasks as building AI systems in this area is very much a learning experience.

Most interesting tasks involve a mixture of different kinds of knowledge: factual, heuristic, and "knowing how" control knowledge. Most knowledge that we possess can be turned to several uses. The proof planning approach is intended to grasp these two facets of human understanding and can use them to good purpose. Whether it does, in fact do so, is the subject of an ongoing study. The results of this exploration so far are reported in this paper.

2 The Case Study

Our starting point was an assertion — a hypothesis — that a technique from automated reasoning could be applied to a novel domain. The technique in question was that of *proof planning*, initially applied by Bundy *et al.* (1991b) to the domain of proving theorems by mathematical induction. We wanted to tackle a significantly different, non-mathematical domain using this basic approach, in order to investigate the hypothesis in Bundy (1988) that the proof planning technique might be expected to have

> "application to any area of automated reasoning where search control is a problem ... to guide the reasoning of expert systems." (Bundy, 1988) (Bundy, 1988)

Our approach is Popperian to the extent that we believe:

> *"The criterion of the scientific status of a theory is its falsifiability, or refutability, or testability."*
>
> *Karl Popper*

In order to investigate whether proof planning could prove useful for other, non-mathematical knowledge based systems we had to:

- Find an exemplar KBS domain.
- Express this domain in a rigorous formalism.
- Encapsulate design strategies as *proof plans.*
- Test the ensuing system on real world examples.
- Try to find examples which test the current system to destruction. These may be real world or "pathological" examples.
- Evaluate the enterprise.
- Modify our proof plans, theory, *etc.* as necessary.
- Retest again.

  etc.

The chosen domain was that of computer hardware configuration, where search control is one of the main problems (Soloway *et al*, 1987), and where strategies to control it are much sought after. We decided this was suitable because:

- Many configuration systems existed, yet the methodology for developing such systems was patchy and in all cases much different to our own.
- The task was rich in design issues which added to the interest of the search problem; and the latter was considerable. The task is not simply to find a solution, but to find a "good" solution — as with any problem where the brief tends to be under-specified.
- Human expertise in designing "good" configurations existed in abundance but little had been done hitherto in capturing this knowledge, exploiting it, or automating it.

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

The overall objective of the system is to take a specification from the "invitation to tender" stage of negotiation between a computer firm and its potential customer, and to produce a configuration which meets that specification. A typical example of a specification is:

- 70 screens for data entry
- 5 screens for program development
- 20 screens for electronic mail and spreadsheet application
- 1 laser printer (need to offer double-sided printing)
- 2 laser printers, each 45 ppm
- 1 line printer at least 900 lpm
- mass storage approximately 1.7 GBytes

There are several points of interest here:

1. There are a very large number of possible configurations which will meet this specification.
2. They are not all of equal merit.
3. There are a number of dimensions on which to judge solutions: for instance cost, performance criteria, expandability.
4. Because of this there is not, in general, a *global* optimum solution.
5. We would want to discard a great many of these which are "locally sub-optimal"; *e.g.* we could find another solution which gave better performance for equal cost and all other criteria.

Thus these can often be seen as *minimal* specifications, representing the very least that will be acceptable; and synthesizing a "more desirable" configuration can be seen as a question of *design*.

One aim of this research was the development of a robust, maintainable automatic configurer. The success of this enterprise may be judged on

- Whether the solutions given by the automatic configurer correspond with solutions judged to be acceptable by human experts.
- Whether solutions can be found in a reasonable time.
- Whether better solutions are found in preference to less good ones: *i.e.* the better solutions are returned first.
- Whether the automatic configurer dealt with search in a desirable way; *e.g.* whether a request for further solutions would produce configurations which were acceptable, but also discernibly different from previous solutions in some significant way — *i.e.* using different components, and not merely trivial permutations of other solution configurations.

- Whether any reasonable solution will, ultimately, be found.
- Whether the automatic configurer was maintainable, in a number of senses.

One aspect of the maintenance question which particularly concerns us here is whether the system proposed is adaptable to changes and additions to the theory. It is this, as we shall see, that is the major contribution of the proof planning methodology to design problems. It is particularly important in a domain such as hardware configuration where products change so frequently.

3 Proof Planning

The use of the proof planning technique is central to our methodology so without further delay we will explain the technique here.

A *proof plan* is a means of expressing the commonality between members of the same "family" of proofs (Bundy, 1991), whilst allowing sufficient flexibility and adaptability to prove a large number of different theorems. Proof plans provide an expression of strategies for automatic reasoning by describing *tactics*, which can be understood for now as a kind of operation (or program), in terms of the *preconditions* under which they are *applicable*, and their *effects* if successfully applied. The specification of a tactic in terms of its preconditions and effects is called the *method* for that tactic, and methods provide a basis for combining tactics (the effects of one implying the preconditions of subsequent methods) to form a complete plan which, if executed, will carry out a reasoning task, such as proving a theorem.[1]

[1] The use of tactics and methods means that machine proofs can be understood in higher level terms than hitherto, in a framework more akin to human reasoning. One criticism of machine proofs is that they are incomprehensible to the average user (R.A.Demillo *et al*, 1978) who then puts little faith in them. It will be interesting to see whether proof planning can deliver industrial strength proofs which are more acceptable than this.

Name	$configure\_device$
input	$configure(Device, IC, C)$
output	nil
preconditions	*Device is a device of C* *and IC* $: \tau$ *and Device needs slot of type* τ *and the number of slots available of type* τ *is* $s(n)$
effects	*the number of slots of type* τ *available is* n
tactic	$configure\_device(Device, IC, C)$

Figure 1: An example of a method frame

An example of a method frame is shown in Figure 1. It represents meta-level reasoning about a particular subtask, namely that of configuring a device. The first slot is the *name* of the method. The last slot is the *tactic* — a piece of Prolog code which carries out part of the configuration task (in this case, configuring a device on a channel). The *input* slot must match with one of the current conjuncts of the goal specification. The *preconditions* slot gives other conditions which must be true and the effects show the conditions which will hold after the tactic has been applied. The *output* slot shows any new conjuncts to be proved. (In this case, it is empty since configuring a device does not give rise to any new proof obligations.)

Using methods written in a meta-logic facilitates the task of writing proof plans to guide the search for proofs in automatic theorem proving, as it possesses the following properties:

1. Efficiency, because the combinatorial explosion is avoided, or at least greatly mitigated (Bundy *et al*, 1991b). Reasoning at the meta-level contains fewer

branches.

2. Generality, because a proof plan may be applicable to many proofs. This was found to be the case in both the theorem proving and the configuration domains.

3. Maintainability, because the separation of factual knowledge from 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. We are exploiting preconditions of methods to produce text intelligible to the average user in our current research.

These properties are important for *any* knowledge based system (Bundy, 1987). Thus proof plans can provide a useful vehicle for expressing strategies for problem-solving in other domains. Moreover, they are useful for *developing* strategies. A proof plan, via its methods, expresses, explicitly, at any given time, our "best guess" at a strategy. As we gain further knowledge and empirical evidence we can readily alter the preconditions and effects of the methods and thus change or refine the strategies. We found that the modification of the system in response to new knowledge, or empirical testing, involved a simple change — the addition of a method, or a change to the precondition slot of an existing method (Lowe, 1993).

The process of identifying common structures and patterns of reasoning, encapsulating these, and testing them on a suitable corpus of theorems, is an incremental one. Adopting this basic philosophy of building methods essentially by *understanding* mathematical problem solving techniques, rather than by a succession of *ad hoc* heuristics and firepower (hardware), means that failures — to prove a given theorem, say — do not lead to dead ends in the process of developing a powerful theorem prover, but rather to a greater understanding and further enhancement of the theorem proving system: the extension of a method, the modification of preconditions, and so on.

Mathematicians, consciously or not, use heuristics to solve problems. They see that a theorem to be proved is similar to another one with whose proof they are familiar, or they have an idea of whether applying a certain rule to an equation will make it easier or harder to solve from then on. Of course, there are many computer systems which incorporate heuristics, but they often do so in an inflexible way. The power of human mathematicians lies in their flexibility, and it is this that an AI theorem prover needs to capture.

Our technique for synthesizing configurations is analogous to program synthesis (Bates & Constable, 1985). A computer configuration c which meets the requirements of a specification $spec(c)$ may be synthesized as a byproduct of proving the theorem

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

where c is well-formed as an object of type :configuration

The specification $spec(c)$ will lay down certain conditions that the desired configuration must fulfill; for instance, that it has disk storage capacity of at least 1.2Gb; that it can support a given number of users (terminals) running certain applications; and so on. To be well formed, a configuration object must possess certain properties, which may be summarized as

1. Certain components are essential (processor, memory, ...)
2. All devices (disk drives, tape drives, printers, ...) must be connected in some (legal) way.

In proving theorem (1) above, where $spec(c)$ is, in general, a conjunct of goals expressing the required properties of the system, the existentially quantified variable is gradually instantiated to a well-formed computer system term. The logical framework employed ensures the *soundness* of the resulting term (Lowe, 1991) but will not guarantee *good design*.

Tactics for configuration are analogous to tactics in mathematics, in that they are programs which combine a significant proof step with other low level steps which are of less interest, and probably harder for the user to keep track of. The object level theory of configuration has been developed in a way which ensures legal configurations whilst facilitating an architecture for encapsulating knowledge for designing computer configurations. Although in the first instance we built a limited but nevertheless fully automated system, nevertheless this methodology was developed with an eye to future systems capable of enabling co-operation between user and machine. It is therefore eminently suited for tasks involving design, where such co-operation on the basis of shared knowledge (as imparted by preconditions and effects of methods) is essential. We would never claim that design tasks could be fully automated and still perform as well or as flexibly as humans, so the ability of the automaton to produce useful dialogue is central.

4 Formalizing the Domain

In order to use theorem proving techniques we need to formalize the domain by developing a working *theory of configuration*. However, computer configuration in many ways is an ill-defined task. Folklore might have it as an art rather than a science. In rejecting this view, we needed to be able to describe and define certain concepts accurately, in order to draw up a theory of configuration which could then be used as the basis for an automated system. The actual usefulness of these concepts and of all the axioms, rules, and heuristic knowledge assembled was tested by designing and building an automatic configurer based on them, and assessing its output — *viz* whether it gave results consistent with those expected, corresponding to the output that experts would give.

Thus we developed an axiomization of the theory of configuration which allows us to make use of standard theorem proving techniques, and, in particular, to view the synthesis of computer hardware as being analogous to program synthesis (as mentioned in Section 3).

The first issue to be addressed is that of representing the configuration domain in a logical formalism. This is desirable in order to make it more precise, to enable reasoning about the domain in a rigorous manner, and to ensure soundness. As we might have expected, this was a difficult task. The knowledge used by experts in performing their tasks is often presented in a haphazard and informal manner. Moreover, it is not easy to impose order on the plethora of objects at our disposal, or to arrive at more than an *ad hoc* description of how different objects may be combined (this may be constantly changing). However, the benefits of representing the domain as a sound logical theory far outweigh the difficulties. If we can assign a *type* to every object which is used in configuration problems, then the task of ensuring that various constraints are satisfied within a configuration is made much more elegant, in that messy *ad hoc* constraints are tidied away to the low level reasoning (*i.e.* type checking) and can be ignored at the higher level of reasoning of more interest to the user. In this respect, the division into tactics (which, amongst other things, carry out this type checking) and methods, which are specifications for tactics, represents the division between the user view of the task and the engineering view.

If we can develop an axiomization of the rules concerned with the configuration task then the tasks associated with configuration can be seen as by-products of theorem proving in the domain, with the soundness associated with such techniques. Such an axiomization is reported in Lowe (1991) and in more detail in Lowe (1993a). This describes the extrication of the object-level knowledge, in the form of hard facts and rules about computer hardware and its configuration, from the wealth of knowledge brought to bear by experts when carrying out configuration tasks; and the formalization of the object-level knowledge. Whereas the explicit representation of strategies for configuration may be developed by questioning experts, by reading configuration manuals, and by other knowledge elicitation techniques; the logical representation of the object level presents additional problems associated with the formalization of an originally non-formal domain. This is the crucial stage of developing an automated process. On it depends the soundness of the resulting knowledge based system.

5 Theory, Testing, and Experimentation

5.1 Implementation

The prototype automatic configurer, CLEM (Lowe, 1992), is the test of our theory of configuration. The issues to be addressed in tackling the task of configuring hardware to meet user specifications were: ensuring a legal configuration (*i.e.* one which works), meeting the customer needs (conforming to the specification), and incorporating at least some of the design issues (for example, finding a configuration which is reasonably priced, and runs as efficiently as possible). We hope more issues may be addressed in future. We developed an object-level theory which was a formal logical representation of the configuration domain; the proof planning technique enabled us to represent and implement the meta-level reasoning involved, including design issues.

We then assessed how far these goals were achieved, with particular regard to:

- Performance and potential in comparison with other systems, with especial consideration given to its flexibility, particularly in the face of technological change.
- Maintainability.
- Potential for adaptation to new problems in the same domain.
- Verisimilitude: performance in comparison with an expert (human) configurer.

- Perspicuity: how easy it is to understand what the system does, and how, and why.

5.2 Trial and Error

The process of developing the ideas intimated above was not a straightforward one and trial and error played a large part. Testing often revealed issues that we had not thought about, for example the importance of the order in which different types of device were configured. Far from letting us off the hook, this made adherence to a strict methodology even more important. Popper (1989, pages 312–315) analyses the process of trial and error as one in which we are actively involved in trying to solve a definite problem, rather than simply (passively) observing phenomena. Theories are merely tentative hypotheses, to be tested critically; if they are at fault then this is to be taken in a positive spirit, so that they may be revised:

> "...we make progress if, and only if, we are prepared to *learn from our mistakes.*" (Popper, 1961, page 87)

We can never "prove" that a theory is true (in the sense that it is an accurate model of the world, or subject of study), although we may sometimes refute it. Our aim should be to subject our theories to the most stringent tests possible:

> "Only if we cannot falsify them in spite of our best efforts can we say that they have stood up to severe tests.... For if we are uncritical we shall always find what we want: we shall look for, and find, confirmations, and we shall look away from, and not see, whatever might be dangerous to our pet theories.... to ensure that only the fittest theories survive, their struggle for life must be made severe for them." (Popper, 1961, pages 133–134)

5.3 Testability

A large part of our project was involved with amassing of knowledge, dividing it into categories (facts, heuristics, *etc.*), and encoding it. However, what to collect, and how to use it, were at first the objects of speculation. The relevance and accuracy of this information were tested in a systematic way. For example, configurations were divided into substructures and each of these was tested independently; the object-level theory was tested for verisimilitude before the development of the meta-level planning architecture, and so on. Results given by the program which was eventually implemented were tested against those found by human experts where possible.

However, the end result is not intended to be seen as a "proven" system. The stringent tests needed for this belong to field testing and have yet to be carried out. However, the following summarizes the steps we have taken so far:

1. Set up a problem to be solved — *i.e.* how to automate the configuration of hardware from specifications.

   > "A scientist, whether theorist or experimenter, puts forward statements, or systems of statements, and tests them step by step... by observation and experiment." (Popper, 1990, page 27)

2. Actively pursued this task in the light of a tentative theory of configuration.

   > "Experiment is planned action in which every step is guided by theory. We do not stumble over our experiences, nor do we let them flow over us like a stream. Rather, we have to be active: we have to *'make'* our experiences." (Popper, 1990, page 280)

3. Tested this system as far as possible, and made it available for others to test.

   > "...it must be possible for an empirical scientific system to be refuted by experience ... What characterizes the empirical method

> is its manner of exposing to falsification, in every conceivable way, the system to be tested." (Popper, 1990, pages 41–42)

The prototype automatic configurer implemented performs successfully on many examples (Lowe, 1993). Of course, further testing and further development can be done, by ourselves or by others. The latter is perhaps more important: it is not necessary that we attempt to refute our theories ourselves; indeed, others may be better motivated to do so, and therefore provide the more stringent testing desirable in order that only the fittest theories might survive, and indeed to motivate the development of the fittest theory. The following assist this process:

- The separation of object-level and meta-level knowledge gives clarity to the theory.
- Both were encoded declaratively in Prolog.
- The results may be reproduced: any person may access our code and verify the purported solutions to sample tests given in the text, or devise new tests and attempt to falsify the theory or its implementation.
- The theory is amenable to *disjointed incrementalism* (as outlined in the next section).

This is in marked contrast with systems where the implementation details are hidden and which are not, therefore, subject to verification by others.

5.4 Disjointed Incrementalism

One tool in developing the theory and automation of configuration deserves a brief description here.

The technique of proof planning lends itself to the disjointed incrementalist approach borrowed from planning theory (Alden & Morgan, 1974). This is often known, somewhat ironically, as "the strategy of muddling through": a first stab at a theory, program, *etc.* is tested. The results of such tests (if not wholly successful) will show up weaknesses. With suitable analysis, the nature and pattern of failure will usually suggest modifications to the theory, which it is hoped will improve it. For a theorem prover, failure might mean the inability to prove a particular theorem. If the failure is analysed it may suggest what was deficient about the theorem prover in its initial form and what could be changed to enable it to prove the theorem in question. Changes are made: obviously we must retest all the cases which were successful before, to check that they still work; and we test the failures also. Progress means that more cases are successful than previously; if the changes introduce more problems than they solve they may be abandoned; otherwise the process repeats. In theory, this process never ends, since it is always conceivable that a new test, or a new phenomenon, may introduce failure, but in practice testing may be eventually suspended. Disjointed incrementalism is a pragmatic approach, in that, if a target is set, and a given theory or system meets this target, it will not be further amended unless the target is reset. The following is a good analogy:

> "Science does not rest on solid bedrock. The bold structure of its theories rises, as it were, above a swamp. It is like a building erected on piles. The piles are driven down from above into the swamp, but not down to any natural or 'given' base; and if we stop driving the piles deeper, it is not because we have reached firm ground. We simply stop when we are satisfied that the piles are firm enough to carry the structure, at least for the time being." (Popper, 1990, page 111)

6 Stages in development

6.1 Knowledge capture

There were three main stages in developing the system:

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

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

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

The difficulties involved in maintaining large knowledge bases for configuration and order checking tasks are well known. [2] An important factor in overcoming these for us was the clean separation of object-level facts from meta-level control and heuristic knowledge. Object-level knowledge must be extracted and formally represented in such a way as to allow the utilization of techniques analogous to program synthesis. Such an approach makes the task of maintaining knowledge bases more tractable and reliable; the consequent separation of object-level knowledge from heuristics and control knowledge increases the usefulness of the knowledge base.

Many knowledge based systems suffer from the difficulty of separating object-level and meta-level knowledge. Knowledge bases are compiled with the aid of human experts who are often unable to unravel the two, or even realize the distinction. As new objects are added or new rules discovered it becomes increasingly difficult

[2]Circumstantial evidence is that in the published accounts (Barker & O'Connor, 1989) the problem of training personnel to work on XCON is prominent; the solution proposed is to send all new staff on an AI course on entry.

to maintain the system. Moreover, new applications, which ought to be based essentially on the same factual knowledge as the old one, cannot always plug into the existing knowledge base, since the information therein is irretrievably bound up with high level control knowledge used by the original application.

6.2 Testing

Our CLEM system configures a subset of HP 3000 systems. The implementation was carried out in several stages. The result to date is a prototype system, in Prolog, which can handle a variety of specifications, and which can 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.

Test runs of CLEM and runtime statistics can be found in Lowe (1993a). The results of these tests demonstrated that CLEM does indeed produce configurations in reasonable time. It is pertinent to outline the research methodology used in setting up these tests, and to describe fully what is being tested, and the significance of the results.

The first tests were very simple, and test each part of the configuration task separately: configuring storage devices; configuring terminals; configuring printers.

Later tests enhanced each of these specifications, allowing more detail to be given — for example, specifying how a terminal is to be connected in terms of which type of cable is to be used; specifying that a printer is to be configured as a serial, rather than a system, device.

Next, complete specifications were tried, starting with rather small systems (to make it easy to check the output) and going on to large ones, to assess the runtimes involved and determine whether they are acceptable. These configurations may still be synthesized in such a way that all heuristics are obeyed; the tests check that this is, in fact, done as default.

Lastly, there was a series of tests which took specifications which cannot possibly be configured whilst maintaining all heuristic constraints intact, to check how CLEM relaxes these constraints in practice. Various constraint relaxation strategies were tried as there seems little formalized information about which strategies are preferred in various situations.

To sum up, the questions to be asked are:

1. Are the solutions the ones expected, given the implementation?
2. Are these the solutions that a configuration expert would find?
3. In any of the solutions expected and obtained, are there shortcomings, and how could CLEM be modified to overcome these (if at all)?

In addition, several different potentially acceptable implementations were tried, and tests were run in order to assess the effects of changes to various factors. In particular, we ran a series of tests showing comparative runs for

1. Changing the balance of work between the planning and execution stages.
2. Stronger or weaker preconditions.
3. Using simple methods alone, or incorporating supermethods, which call methods from their effects and hence impose more ordering on the strategy used.

In addition we needed to answer the question as to whether planning is necessary at all. This question was conclusively answered, by the fact that we had to test comparative performance with and without planning by choosing very small configuration specifications: otherwise the runtime proved unacceptable. Also, the solutions obtained were not of good quality (*i.e.* ill-designed) in some cases.

7 Comparison with other approaches

We now compare the more traditional expert systems approach with the proof planning one. XCON/XSEL (Bachant & Soloway, 1989; Barker & O'Connor, 1989) is a prime, and well-established, example of the former *genre*, and CLEM is an example of what could be achieved using the proof planning approach. There are similarities in the course of development of both systems. For example, I too needed to start with a subset of components and a subset of the tasks to be performed. I needed to learn from experts and manuals much about configuration procedure and strategy. Some of my early solutions proved to be faulty in that I had overlooked certain factors.

However, XCON developed by accumulating its knowledge in a rapidly growing and unwieldy rule-base. Like the proponents of frame-based and semantic net systems, (Searls & Norton, 1990; Pierick, 1986) we too recognize the need to store knowledge "in the right place", but in my case this manifests itself, not in hierarchies of objects with procedural dæmons attached, but in a clean separation of different *kinds* of knowledge: object-level theory, heuristics, and control knowledge.

It is this separation which gives the proof planning approach the potential for growth, not merely in terms of adding more components to its repertoire and assimilating constant changes to its rules, but also in the sense that it should withstand more fundamental assaults: changing technology being the most problematical for traditional systems. It is hard to envisage a change which would result in a "dead end" for proof planning techniques. If our results are "wrong" (as given by experts) then we can ask why; relate this to our methods; and make appropriate changes. Unlike the more traditional expert systems, they are adaptable, as charting the process of the proof planning work carried out my the Mathematical Reasoning Group at Edinburgh University will show (Bundy *et al*, 1991b; Bundy *et al*, 1989; Bundy *et al*, 1991a; Ireland, 1992).

With CLEM, unlike many knowledge based systems, it is impossible to get over problems with quick "fixes": I claim that this is a benefit. The developer of the system is forced to make knowledge explicit and modular, and this can only be beneficial in the long run, in terms of saving programmer-hours. A deeper understanding of the task, the result of being obliged to formulate the domain in terms of logic, highlights the essential generalities of the configuration task. Deficiencies in the original model, as shown up by testing, leads to a recasting of the problem and the logical framework developed as a result proved much more robust for later additions to the theory.

Because the object-level theory is coded declaratively, it is easier to check:

- that components are correctly assigned to types;
- that there is a complete list of components;
- that all constraints (hard and soft) are included.

We gave various criteria on which the success or otherwise of such a venture could be assessed. Established systems, such as XSEL, already exist for the configuration task, but we believed them to be deficient in various ways. In particular, we wanted CLEM to be amenable to adaptation and change, and for its object-level knowledge, heuristics, and strategies to be explicit and transparent. Of course, the test of this lies in the future. Another aim was to demonstrate a methodology, within the proof planning paradigm, which could be transported into other knowledge based systems tasks.

Because we use tactics, specified by methods, which mirror the user view of the problem, this makes the plans readable, and the skeleton configurations provided at the planning stage provide enough information for the user to make an informed choice as to whether the plan is worth executing. This is not necessarily true of, say, constraint based systems such as Frayman & Mittal (1987), which work at a low level, as the reasons for the choices made may not be apparent on the surface.

8 Proof planning for IKBS

The motivation for turning AI in general, and the pursuit of IKBS in particular, into an engineering science is so that the resulting expert systems are *reliable*. Logic provides a firm basis on which to achieve this; in addition, meta-level reasoning, as captured by the proof planning methodology, gives us the flexibility to use this logic purposefully.

The next stage in the "proof planning for IKBS" story would be to try other IKBS problems. In fact, research in this area seems likely to switch to the Building domain for logistic reasons (mainly access to domain knowledge). It remains pertinent to ask whether this general *approach* is transportable to other domains, however. Also, will the next domain tackled prove easier by virtue of the general expertise gained from the experience of tackling the configuration task? To answer this, we need to ask whether we have learned anything fundamental from this exercise. In particular, we need to ask whether there was something peculiar to configuration which made it possible.

For the configuration task, the salient features which made proof planning both possible (1,3,4,5) and productive (2) were:

1. An object-level theory was discernible, if a little difficult to formalize.

2. A large search space: combinatorial explosion at the object-level made the use of the meta-level desirable, over and above merely paying back the overhead.

3. The ability to separate factual from heuristic and control knowledge.

4. Fairly well-developed strategies for performing the task. However, it is sometimes difficult to get hold of these in a competitive field — hence the likely switch of domain for this research to building design mentioned earlier.

5. The likelihood that new products and new procedures would fit into the overall framework.

For more static domains, of course, the last of these is not an issue. In general, these features are common to many tasks involving reasoning.

The outcome of this study has been:

- The formalization of an initially very informal domain, where it was at first difficult to ascertain which category various bits of knowledge belonged to.
- The characterization of the configuration task as synthesis in the *proofs as objects* paradigm.
- The development of a theory of configuration, and of various strategies to assist with the task of configuring a hardware system to meet a specification.
- The conceptual division of the task into two views: the user view and the engineering view, of which the former is encapsulated by the preconditions and effects of methods, which are specified by tactics, and the latter is guaranteed correct by the execution of these tactics.
- The embodiment of strategies as proof plans.
- The implementation of CLEM, an automatic configurer capable of testing these ideas and developing new ones.

We predict that many other domains could be tackled in this way. In particular:

1. Other kinds of configuration, following the same methodology, *viz.* using logic to encode an object-level theory for the domain, determining procedures and strategies and expressing these as proof plans.

2. Other design problems, for example in building, where there is a promising subset of problems which seem amenable to our approach. The advantage in this domain is in the existence of professional bodies with an interest in advancing common knowledge of how these tasks are performed (however flawed this is in practice at present!), as opposed to the field of hardware configuration, where systems are developed by competing firms.

9 A new approach?

As Kuhn (1970) states, it is not enough, when promoting a new paradigm, to demolish the old one. Part of the attraction of a paradigm, once it becomes the "established" one, is that it leaves plenty of "holes". In science, this "hole-filling" activity can be seen as virtually all of what Kuhn calls "normal scientific activity". Whereas it is an exaggeration to call our approach a new paradigm, there is a good parallel in the field of expert systems — whether seen as "scientific activity" or not — in that maintaining XSEL, for example, is seen as a satisfying activity by the maintenance programmers who perform this task. In fact, it seems to be generally expected that rule-based systems will naturally require this kind of maintenance, and that the resulting problems of control will be dealt with cleverly in an implicit way if necessary. In the world of the computer hacker, the gradual tinkering away at flawed theories in Popper's ideal world can become a grotesque sacrifice of clarity on the altar of performance.

According to Kuhn's view of change, only gradually can a new approach be expected to find adherents, until, possibly, it becomes the new paradigm. I would expect a great deal of work to be necessary, in developing logically-based knowledge based systems incorporating meta-inference in a principled and explicit manner, for different problems and different domains.

However, we believe that such work is worth doing. Basic research in IKBS is needed to place such systems on a firm logical basis, and make subsequent "hole-filling" activity more principled, more directed, and more disciplined. Bundy (1987) looks forward to a future in which AI systems are as dependable as bridges — with all that has been learned from past mistakes and the disciplined codes which effect growth and change. I share this vision.

Acknowledgements

I am most grateful to Alan Bundy, of the Department of Artificial Intelligence, University of Edinburgh, who provided the inspiration for my research. Thanks are also due to Frank van Harmelen (University of Amsterdam) and Geraint Wiggins (University of Edinburgh) for feedback on my work over the years. Many people at Hewlett Packard Research Laboratories, Bristol assisted me with knowledge elicitation and provided access to expert knowledge.

References

Alden, Jeremy and Morgan, Robert. (1974). *Regional Planning: A Comprehensive View.* Leonard Hill Books.

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.

Bates, Joseph L. and Constable, Robert L. (January 1985). Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136.

Bundy, A. (1987). AI bridges and dreams. Research Paper 300, Dept. of Artificial Intelligence, Edinburgh, Also in AI and Society, vol. 1 no. 1.

Bundy, A. (1988). The use of explicit plans to guide inductive proofs. In *9th Conference on Automated Deduction*, pages 111–120. Longer version available as Research Paper 349, Dept. of Artificial Intelligence, Edinburgh.

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

Bundy, A., van Harmelen, F., Hesketh, J., Smaill, A. and Stevens, A. (1989). A rational reconstruction and extension of recursion analysis. In Sridharan, N.S., (ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufman. Available as Research Paper 419, Dept. of Artificial Intelligence, Edinburgh.

Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A. (1991a). Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, Edinburgh, To appear in Artificial Intelligence.

Bundy, A., van Harmelen, F., Hesketh, J. and Smaill, A. (1991b). Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324. Earlier version available as Research Paper 413, Dept. Of Artificial Intelligence, Edinburgh.

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

Ireland, A. (1992). The Use of Planning Critics in Mechanizing Inductive Proofs. In Voronkov, A., (ed.), *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag.

Kuhn, Thomas. (1970). *The structure of scientific revolutions*. University of Chicago Press, second edition.

Lowe, Helen. (1991). The use of theorem proving techniques in expert systems for configuration. In Rault, J.-C., (ed.), *Proceedings of the Eleventh International Workshop on Expert Systems and their Applications, Avignon*. EC2. Also available from Edinburgh as Research Paper 536.

Lowe, Helen. (1992). The CLEM configuration system, user manual and programmer manual. Technical Paper (forthcoming), Dept. of Artificial Intelligence, Edinburgh.

Lowe, Helen. (1993). *The Application of Proof Plans to Computer Configuration Problems*. Unpublished Ph.D. thesis, University of Edinburgh, Sumbitted but not yet examined.

Pierick, Jeff. (1986). A knowledge representation system for systems dealing with hardware configuration. In Kehler, T. and Rosenschein, S., (eds.), *Proceedings of the Fifth National Conf on Artificial Intelligence*, pages 991 – 995, Philadelphia. AAAI.

Popper, Karl R. (1961). *The Poverty of Historicism*. Routledge & Kegan Paul, London, (Revised version with some corrections.).

Popper, Karl R. (1989). *Conjectures and Refutations*. Routledge & Kegan Paul, London.

Popper, Karl R. (1990). *The Logic of Scientific Discovery*. Unwin Hyman.

R.A.Demillo, R.J.Lipton and A.J.Perlis. (1978). Social processes and proofs of theorems and programs. Technical Report GIT/CS 78/04, Georgia Institute ofTechnology,Atlanta, GA.

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

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.