The Use of Theorem Proving Techniques in Expert Systems for Configuration *

Helen Lowe

Department of Artificial Intelligence,
University of Edinburgh.
80 South Bridge
Edinburgh
Scotland
EH1 1HN
tel. 44-31-650-2726

Abstract

The difficulties involved in maintaining large knowledge bases for configuration and order checking tasks are well known. An important factor in overcoming these is the clean separation of object-level facts from meta-level control and heuristic knowledge. We outline 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 systems which meet specifications. 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 utility of the knowledge base.

Keywords:

[configuration, design, theorem proving]

*The research reported in this paper was supported by a SERC CASE studentship with Hewlett Packard Research Labs, Bristol. I am grateful to Alan Bundy, David Basin, Frank van Harmelen, and Stefek Zaba, amongst others in the Mathematical Reasoning Group, Edinburgh University and at HP Labs, Bristol, for help and feedback. Note that the views expressed in this paper are my own.

1 Introduction

We are investigating whether the technique of *proof plans*, which has been successfully applied to proving theorems by induction and other related work in program synthesis (Bundy *et al.*, 1989; Bundy *et al.*, 1991) could be useful in fields outwith Mathematics.

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

We have considered the tasks associated with computer systems configuration. Although we have focussed on configuration synthesis, *i.e.* configuring computer systems to meet given specifications, we have been aware that there are related tasks which use the same object-level knowledge: namely order checking (given an order, ensuring that a viable computer system can be built from it) and system upgrading (given an existing system, and new specifications, building a new system based on the old one). We have endeavoured to separate out and formalize this object-level knowledge so that it may be used by planning systems for any of these ends. As far as we know, little or no work has been done on the last of these tasks, despite the existence of systems which successfully perform the first two (McDermott, 1982; Rosenbloom *et al*, 1985; Searls & Norton, 1990; Soloway *et al.*, 1987, Frayman & Mittal, 1987).

This paper describes the first stage, which is to separate the object-level knowledge, in the form of hard facts and rules about computer systems and their configuration, from the wealth of knowledge brought to bear by experts when carrying out configuration tasks; and then to find a suitable formalization for this knowledge. Whereas the explicit representation of strategies for configuration may be developed by questioning experts, reading configuration manuals, and other knowledge elicitation techniques, the logical representation of the object level presents additional problems associated with the formalization of an originally non-formal domain. We argue that this is the crucial stage of developing an automated process. On this depends the soundness and completeness of the resulting expert system.

We have developed an axiomization of the theory which allows us to make use of standard theorem proving techniques. In particular, we can view the synthesis of computer systems as analogous to program synthesis. We demonstrate, using an example, how computer systems may be synthesized from a specification spec(S) as a byproduct of the proof of the specification theorem $\exists$S.spec(S), where S is a term of type system.

Since this axiomization represents a correct account of configuration, we can, in theory, synthesize computer systems using this object-level knowledge alone: indeed, simple systems have been so synthesized. However, as one might expect, the search space is huge, and we need to utilize the control and heuristic knowledge which we carefully separated from the object level. We affirm that this methodology allows us to reinstate such meta-level knowledge in a principled manner. Although it is not the main subject of this paper, there is a brief discussion of how this may be done in Section 9.

2 Configuration of hardware

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. These tender specifications leave much detail unspecified: for example there may be no explicit mention of backup facilities or main memory; clearly the tendering firm must include these in the system in sufficient measure to ensure the system runs efficiently. Also, there is a sense in which these can be seen as minimal specifications, in that there is scope for enhancing the performance or expandability of the tendered system. An example of a specification which our prototype system can successfully deal with 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.

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

3 Objectives of the synthesis

A system consists of many components, which are connected together in such a way as to constitute a legal, working configuration. There are several basic types of component: for example, *printer_1* is the name of a particular kind of a printer; *terminal_3* is a terminal, *cable_11* is a rs232-serial-cable, and *system-cable_1* is a system-cable.

Our technique is analogous to program synthesis (Bates & Constable, 1985) where we are given a specification spec(input, output) of the relationship between the input and output, and synthesize the algorithm alg as follows:

Find a constructive proof of the theorem:

$$\vdash \forall \mathrm{input}.\exists \mathrm{output}.\mathrm{spec}(\mathrm{input}, \mathrm{output})$$

Synthesize from this the algorithm alg:

$$\vdash \forall \mathrm{input}.\mathrm{spec}(\mathrm{input}, \mathrm{alg}(\mathrm{input}))$$

A computer system S which meets the requirements of a specification spec(S) may be synthesized as a byproduct of proving the theorem

$$\exists S.\mathrm{spec}(S) \tag{1}$$

where S is well-formed as an object of type :system (see Section 5).

The specification spec(S) will lay down certain conditions that the desired system must fulfill; for instance, that it has disk storage capacity of at least 1.2Gb; that it has certain printers; that it can support a given number of users (terminals) running certain applications; and so on. To be well formed, a system 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(S) 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.

4 Objects of the theory

We represent individual components such as processors, memory modules, disk drives, tape drives, printers, muxes, lanics, channels, fibre-optic links, and cables as members of atomic types.

More complex objects can be represented via type constructors, these being:

- Function types.
  e.g. if nat is the type of natural numbers, and T is a type (of component), then nat $\mapsto$ T is a type whose members are functions from nat to T.
  e.g. ddrive_1 : nat $\mapsto$ disk
  (ddrive_1 is a kind of disk drive. ddrive_1(5) is an instance of this kind of disk drive in a configuration.)
- List types
  If T is a type, then list(T) is a type whose members are lists of members of T.
  e.g. the partial configuration representing the disk storage devices might be

  [ddrive_1(1), ddrive_1(2), ddrive_1(3)], of type list(disk).
- Cartesian product types
  If A and B are types, then A#B is a type whose members are pairs $< a, b >$, with a in A and b in B.
  e.g. $<$ printer_1(5), system-cable_1(16) $>$ represents a printer with a system cable attached.
- Dependent types
  x : A $\times$ B, where the B depends on x.
  e.g. ddrive_1 and ddrive_5 are both functions into type disk, but ddrive_1(3) requires a system-cable, whereas ddrive_5(2) needs a fibre-optic-link.
- Simple arithmetic and list functions, *e.g.*
  length: list(A) $\mapsto$ nat,
  plus: nat $\times$ nat $\mapsto$ nat
- Building more complex types
  We can combine these constructors to build more complex structures. This culminates in the system type, whose members are computer system configurations.
  For example, the function connected-via takes a pair of type printer $\times$ system-cable, and returns the channel to which the device, via the cable, is connected:
  connected-via : printer $\times$ system-cable $\mapsto$ channel

Our choice of atomic types formalizes the expression of an important class of constraints in this domain, namely those concerned with ensuring legal connections. Hence this is now embedded in the logical representation as well-formedness checking.

5 Systems

The syntactic structure of a system is a tuple

system =

list(processor)#list(memory)#list(cardcage)...[1]

#list(channel)#...#list(disk)...#list(cardcage-term)

where the cardcage-term is a deeply nested term, representing the connectivity of the system.

A system, to be well-formed, must also obey certain properties,

$$\forall s : \text{system}(\text{prop}_1 \wedge \text{prop}_2 \wedge \ldots) \tag{2}$$

where these properties include

- $\text{processor}(s) \neq \text{nil} \wedge \text{memory}(s) \neq \text{nil} \wedge \text{console}(s) \neq \text{nil} \ldots$ *etc.*
  expressing the fact that certain components are essential
- $\forall x \in \text{disks}(s)\ \exists y : \text{system-cable} \in \text{system-cables}(s).\ \text{connect-cable}(x) = y$
- $\forall x \in \text{disks}(s).\ \exists z : \text{channel} \in \text{channels}(s).$
  $\text{connected-via}(< x, \text{connect-cable}(x) >) = z$
  expressing the fact that all devices must be connected
- $\forall x \in \text{channels}(s)\ \exists cc : \text{cardcage} \in \text{cardcages}(s).\text{connected-via}(x) = cc$
  expressing the the fact that each interface card must be configured in a card cage
- Conjuncts to do with constraints, these falling into two classes:
  1. limits on the number of devices which can be connected to a channel or on the number of interface cards which can be configured in a card cage
  2. a small number of "incompatibility constraints", to do with the fact that certain devices, or certain devices fulfilling certain functions, may not share a channel.

A term of type system is a tuple consisting of processor, system disk, a list of memory modules, lists of all the major components appearing in the system, and a list of card cage terms. Note that there will generally be constraints on the lengths of lists, determined by the processor type, which also determines which devices and other components are configurable (they must all be "supported" by the processor).

Note that this representation of systems contains redundancy, in that a tape drive, for instance, will appear twice: once in the list of tape drives and once in the card cage term. This reflects the need to reason separately about the functionality of a system in terms of its major components whilst proving high level specification goals; and the connectivity of the system, when proving well-formedness goals.

---

[1] ... is used for terms and arguments which we are not focusing on, for clarity. The subterms involved are fairly long and complex.

The definition of well-formedness for a system object includes its internal consistency; for example, that each device is present both in the list of card cage terms and in the appropriate device list (occurring exactly once in each).

To describe a system, we need firstly to be able to indicate its major devices, *e.g.*

$$\text{terminals} : \text{system} \mapsto \text{list(terminal)}$$

We may need to identify other components:

$$\text{channels} : \text{system} \mapsto \text{list(channel)}$$

We need to identify the connections present: which card cage a certain interface card is configured in:

$$\text{connected-via} : \text{channel} \mapsto \text{cardcage}$$

Identify the cable for a device:

$$\text{connect-cable} : \text{printer} \mapsto \text{system-cable}$$

Determine which channel a device is connected to (via its cable):

$$\text{connected-via} : \text{printer} \times \text{system-cable} \mapsto \text{channel}$$

6 Axioms and rules

In the axiomization of the configuration theory, we must be careful to only include immutable rules, and exclude heuristics which are used by human experts either to control their own search problems, or because they guide the search towards systems which are desirable in some way. We cannot put these desirable properties in the object level as rules because they are heuristic only: in the event of (say) cost constraints which prevented us achieving an efficient system, we would have to settle for less. It is the task of the meta-level planning system to guide the search in this way.

We give some example axioms:

If the processor is known (or becomes instantiated in the course of the proof), then this provides a kind of skeleton or template for the system: for instance, a certain minimum configuration must exist, consisting of (say) the processor, two card cages, a certain number of channels configured in one or other of the card cages. For each kind of processor proc, there will be a rule

$$(\text{S} = [\text{proc}, \ldots]) \wedge (\text{cardcages(S)} = [\text{cc}_1, \ldots]) \wedge (\text{channels(S)} = [\text{ch}_1, \ldots])$$
$$\wedge\ (\text{connected-via}(\text{ch}_1, \text{cc}_1)), \ldots$$
$$\rightarrow \text{template(S)}$$

associating each processor with its “template”, or minimal configuration of card cages, channels, *etc.*, together with any known configuration details, such as which card cage each interface card goes in. These “minimal configuration” details are given by the system configuration manuals.

Then we have

$$\text{template}([\text{proc}, \ldots]) \rightarrow \text{processors}([\text{proc}, \ldots]) = \text{proc} \tag{3}$$

expressing the fact that a processor cannot be configured in isolation, but must exist within its template (or minimal) system configuration.

Then there are rules defining what it means for a device to be connected via a cable to a channel, *e.g.*

$$\begin{aligned}
&\text{For dd : disk, cb : system-cable, ch : channel} \\
&\quad \text{dd} \in \text{disks}(S) \\
&\quad \wedge\ \exists < \ldots, \text{chtl}, \ldots > \in \text{cardcage-term}(S). < \text{ch}, \text{dts}, \ldots > \in \text{chtl} \\
&\quad \wedge < \text{dd}, \text{cb} > \in \text{dts} \\
&\qquad\qquad\qquad\qquad \rightarrow \text{connected-via}(< \text{dd}, \text{cb} >) = \text{ch inS}
\end{aligned}$$

and for an interface card to be configured in a card cage:

$$\begin{aligned}
&c \in \text{cardcages}(S) \wedge \text{ch} \in \text{channels}(S) \\
&\quad \wedge < c, \ldots, \text{chtl}, \ldots > \in \text{cardcage-term}(S) \\
&\quad \wedge < \text{ch}, \ldots > \in \text{chtl} \\
&\qquad\qquad\qquad\qquad \rightarrow \text{connected-via}(\text{ch}) = \text{c inS}
\end{aligned}$$

7 Synthesis proofs

We give here an outline of how synthesis proofs proceed under this paradigm.

Let us take a very simple specification (and therefore synthesize a somewhat unrealistic system). Suppose we want to have a *proc_1* processor (where proc_1 is the name of a kind of processor) and disk storage capacity of at least 500Mb. We draw up the specification:

$$\text{spec}(S) \equiv (\text{processor}(S) = \text{proc\_1}(N)) \wedge (\text{capacity}(\text{disks}(S)) \geq 500) \tag{4}$$

The main highlights of the proof of equation (1) are as follows.

1. Axiom (3) matches the first conjunct of (4) with proc = proc_1

   $\text{template}([\text{proc\_1}, \ldots]) \rightarrow \text{processors}([\text{proc\_1}, \ldots]) = \text{proc}$

   The goal template(S)... is proved by synthesizing the appropriate components to stand for the existentially quantified variables:

   $S = [\text{proc\_1}, \ldots] \wedge \text{cardcages}(S) = [\ldots] \wedge \text{channels}(S) = [\ldots] \ldots$

   $\qquad\qquad \rightarrow \text{template}([\text{proc\_1}, \ldots)$

2. S is now a partially instantiated term, with values in the processor slot, and partially instantiated lists in the card cage slots, channel slots, *etc.*:

```
S = <[proc_1,_,[mem_4(1),mem_4(2),|_],
      [cc_1(1),cc_1(2),|_],
      ...,
      [ch_1(1),...],
      ...,
      [<cc_1(1),...[<ch_1,_,_,_>,...],...]>
```

   We turn our attention to the second conjunct of (4) and generate an object of type list(disk) with the required property (*i.e.* that the total capacity of the members of the list have capacity ≥ 500. At this stage, the disk subterm of S is instantiated.

3. The last stage is to ensure that S = [proc_1, . . .] is a well-formed term. This is the major part of the proof and we use the conjunct referenced by (2). This ensures that all the disk drives synthesized in step 2 are connected via suitable cables to channels.

At the end of this proof, S is a fully-instantiated and of type system.

8 Related work

8.1 Configuration systems

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

- rules to create or extend partial configurations
- rules to determine the order in which decisions should be made
- rules which gather information for other rules, *e.g.* access the components knowledge base or perform computations

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

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

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

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

However, even with such improvements, it still remains the case that use of domain knowledge could not be extended, say to automating the process from user specifications to configurations, or to teaching configuration, even though the same domain knowledge would be used for all three tasks. The main intractable problem is that the central strategy of XCON is specific to this particular method of configuration checking, for these particular systems. It depends for its success on the fact that any local solution to a problem is always *the* "right" one. This approach cannot be used in more "open-ended" problems where there are many solutions, but only a few "locally optimal" solutions: the "best-performance" solutions for a fixed cost; the cheapest solution meeting a given specification, *etc.* Our techniques include the principled inclusion of heuristics to direct search efficiently towards these local optimums.

Moreover, as described in Barker & O'Connor (1989), specialized training of personnel is essential for the constant updating of XCON and XSEL which is a feature of rapidly changing product lines. Our aim is to separate control knowledge in such a way as to make the task of maintaining the system straightforward without specialized knowledge, so that personnel responsible for product updating can perform this task in isolation; likewise configuration engineers responsible for maintaining performance heuristics and control strategies. This is made possible by virtue of the clean separation we have achieved.

Our system, as described in Section 9, has similar aims to XSEL but, we believe, lacks the maintenance overhead inherent in production rule systems. Thus a system based on the techniques that we propose would ultimately be more efficient in terms of continuing use; no intricate knowledge of expert systems would be needed to maintain the system.

COSSACK (Frayman & Mittal, 1987) is a system strongly reminiscent of MOLGEN (Stefik, 1981) which essentially treats configuration as a constraint-driven paradigm. As such, it is eminently suitable for systems which are tailored very specifically to the specification, but not to systems which are in some sense "over-engineered": these modern architectures aim to provide scope for expansion and hence tend to be highly under-constrained. Our architecture accepts the large search space associated with such system configuration, the object-level representation ensuring the legality of the computer systems synthesized; and the control mechanisms ensuring a path through the search space directed towards solutions which are at least *locally optimal*. By this we mean that since the specification, in general, under-constrains the solution space, there will not be any one configuration which is *globally* optimal; instead we seek a (small) set of solutions which represent genuine choices: maybe one which is more efficient than other configurations of similar cost; another which is cheaper than other configurations of comparable utility, and so on. We feel that the search associated with a COSSACK-like system would be too great, and the number of solutions generated too large.

8.2 Proof plan work in other domains

Proof plans, as described in Bundy (1988), Bundy *et al.* (1989), and Bundy *et al.* (1990) are a means of expressing the commonality between members of the same "family" of proofs, whilst allowing sufficient flexibility and adaptability to prove a large number of different theorems. They provide an expression of strategies for automatic reasoning by describing tactics in terms of the conditions under which they are applicable, and their effects if successfully applied. These specifications are called methods, and provide a basis for combining tactics to form a complete plan which, if executed, will carry out a reasoning task, such as proving a theorem.

Our present work is concerned with discovering whether these techniques can be applied to non-mathematical domains. The characteristics of a domain which suggest this can be summarized thus:

- There is an object-level theory, which it is possible to formalize using logic (easy for mathematical applications; much harder, as we have found, for other domains).
- There is, at the object level, a considerable search problem (otherwise the task is too trivial to justify the meta-level overhead).
- There is a body of strategies, used by experts in the field, for controlling this search, and this meta-level knowledge can be expressed in a meta-logic.

These characteristics are all found in the configuration domain. Bundy (1987) gives a motivation for using proof plans to aid the development of better, more reliable expert systems.

9 Conclusions and further work

A little thought will show that the search problem in generating systems to meet specifications *at the object level* is enormous. Also, while there may be a multitude of solutions, nothing at this level ensures that "good" systems result; instead, we are likely to end up swamped by solutions which, whilst legal in the strict terms of obeying the rules, are inefficient in practice, or overly expensive, or generally undesirable.

However, we believe that in developing a clean axiomization which contains only the "hard and fast" rules for configuration, the way is clear to develop:

1. Higher level tactics built up from the low level rules already implemented, which, for instance, will build partial configurations in response to conjuncts in the specification: *e.g.* the disk drives of the system, fully configured in the system.
2. *Methods* for each tactic which specify when the tactic is appropriate, giving preconditions for the tactic and also the effect of executing the tactic. This methodology is explained in Bundy (1988).
3. The embodiment of heuristics in the tactics: for instance, there could be a tactic for configuring devices so as to ensure an efficient computer system.
4. A planner capable of generating sequences of methods as plans which, if the corresponding tactics were executed, would synthesize systems meeting the given specification.

We have been able to incorporate the high-level strategies and heuristics in a layer above the object-level facts, which are stored separately in a knowledge base containing details of a representative subset of the product range: components,attributes, *etc.* Work is currently in progress to provide a full implementation, in PROLOG, incorporating the axiomization of the object-level theory (already implemented); high-level tactics for performing various parts of the configuration synthesis, specified by methods; and a top-level planner which seeks a sequence of applicable methods to form a plan to synthesize a configuration to meet the given specification; this plan may then be executed to give the required computer system.

So far we have a prototype system which successfully takes specifications similar to that given in Section 2; a fairly wide range of specifications is possible, with mixtures of explicit specification, *e.g.* naming models of disk drive required; and higher level specification, *e.g.* giving total disk storage capacity required. Printers may be specified explicitly or by various performance criteria, *e.g.* lines per minute, throughput per month; terminals explicitly or by application. The planning stage reveals skeleton configurations and a configuration plan which the user may choose to execute; the underlying logic is then invoked to give a fully configured, legal configuration. Alternatively, the use may request different plans; the time taken to find these is encouragingly fast, and so far seem to yield well-designed configurations.

The next stage should see both an expansion of the knowledge base to enable a greater range of systems to be configured; and the incorporation of the planning mechanisms required to tackle specifications where the appropriate processor (on which so much else in the configuration process depends) is far from clear at the start, but may become clear once other steps in the plan have been taken.

References

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

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). How to improve the reliability of expert systems. In Moralee, S., (ed.), *Research and Development in Expert Systems IV*, pages 3–17. C.U.P. Also available from Edinburgh as Research Paper No. 336.

Bundy, A. (1988). The use of explicit plans to guide inductive proofs. In *9th Conference on Automated Deduction*, pages 111–120. Longer version available from Edinburgh as Research Paper No. 349.

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 from Edinburgh as Research Paper 419.

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

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

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

Rosenbloom, Paul S., Laird, John E., McDermott, John, Newell, Allen and Orciuch, Edmund. (September 1985). R1-soar: an experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and machine intelligence*, PAMI-7(5):561–568.

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.

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