A Subsumption Architecture for Theorem Proving? *

Alan Bundy

Department of Artificial Intelligence
University of Edinburgh

Abstract

Brooks has criticised traditional approaches to Artificial Intelligence as too inefficient, [Brooks, 1991]. In particular, he has singled out techniques involving search as inadequate to achieve the fast reaction times required by robots and other AI products that need to work in the real world. Instead he proposes the *subsumption architecture* as an overall organising principle. This consists of layers of behavioural modules, each of which is capable of carrying out a complete (usually simple) task. He has employed this architecture to build a series of simple mobile robots, but he claims that it is appropriate for *all* AI products. Brooks' proposal is usually seen as an example of Nouvelle AI, in contrast to GOFAI (Good Old Fashioned AI — see Margaret Boden's contribution to this volume for more discussion of this contrast).

Automatic theorem-proving is the archetypal example of GOFAI. The resolution theorem proving technique once served as the engine of AI. Of all areas of AI it seems the most difficult to implement using Brooks' ideas. It would thus serve as a keen test of Brooks' proposal to explore to what extent the task of theorem-proving can be achieved by a subsumption architecture.

Tactics are programs for guiding a theorem-prover, [Gordon *et al*, 1979]. They were introduced as an efficient alternative to search-based techniques. In this paper we will compare recent work on tactic-based theorem proving with Brooks' proposals and show that, surprisingly, there is a similarity between them. It thus seems that the distinction between Nouvelle AI and GOFAI is not so great as is sometimes claimed. However, this exercise also identifies some criticisms of Brooks' proposal.

1 Intelligence Without Reason?

Rodney Brooks has championed a new approach to robotics, which he calls *behaviour-based* (see, *e.g.* [Brooks, 1991]). John Hallam and Chris Malcolm describe this approach in more detail elsewhere in this volume, so we will only summarise it here. Its key features are:

Situatedness: the robot must operate in and on the real world;

Embodiment: the robot must have sensors and actuators;

*I thank Margaret Boden, Rodney Brooks, Fausto Giunchiglia, Andrew Ireland, Sean Matthews and Geraint Wiggins for feedback on an earlier draft of this paper. Ian Green and Gordon Reid helped me fix some tricky LaTeX problems. The research reported here was supported by SERC grant GR/H/23610.

Subsumption Architecture: the robot must be organised as a network of modules. Each of these modules is behavioural, *i.e.* it is capable of performing a whole robot behaviour, *e.g.* walking, grasping. This is in contrast to a functional module, where each module performs one part of a sequence of tasks, *e.g.* perceiving, planning, acting. Behavioural modules are organised into layers, but operate independently and in parallel. Overall behaviour emerges from the combination of the behaviours of the modules. The only interaction allowed is that higher-level modules may inhibit the action of lower-level ones. There is no hierarchical organisation, no message passing between modules and no central model of the world.

Some critics have seen this as a promising approach to dealing with low-level robot control, but as inadequate in general. They advocate a hybrid approach in which behaviour-based and traditional robotics are combined (see *e.g.* [Malcolm *et al*, 1989]). Brooks repudiates this. For instance, in [Brooks, 1991, §6] he says:

> "I think that the new approach can be extended to cover the whole story, both with regards to building intelligent systems and to understanding human intelligence"

In this paper we attempt to test this strong hypothesis. We have deliberately adopted the toughest test we could devise — to see whether Brooks' approach would work on a task for which it seems least suited: mathematical theorem proving. We will see that some aspects of behavioural-based robotics are inherently unsuited to, or irrelevant for, the theorem proving task. However, perhaps surprisingly, the essential idea of the subsumption architecture does seems to echo some current approaches to theorem proving, namely the use of *tactics*.

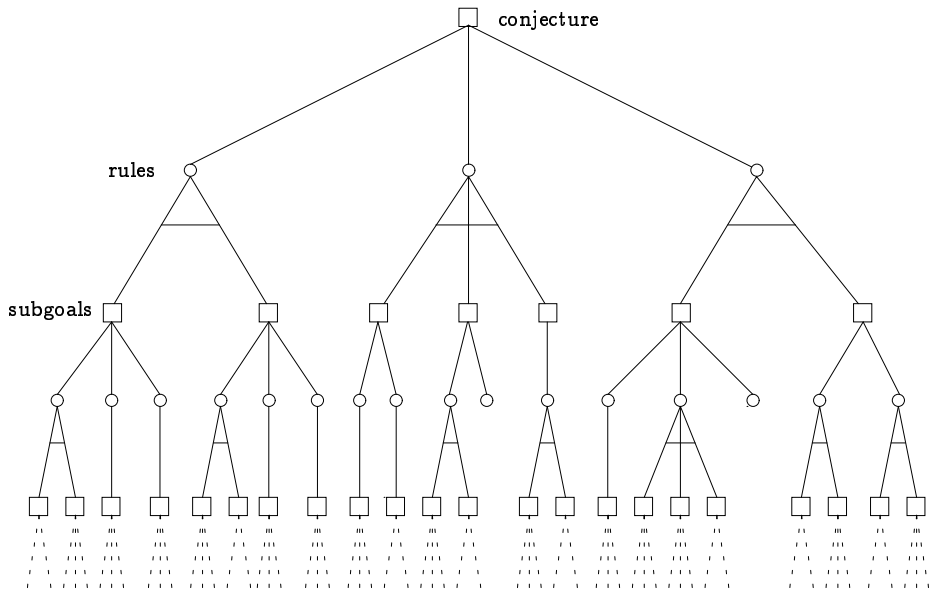
2 Automatic Theorem Proving

Work on automatic theorem proving goes back to the earliest days of artificial intelligence. A theorem prover for propositional logic was one of the first AI programs, [Newell *et al*, 1957]. The roots of the field are in mathematical logic. Most early work was based on a theorem of the logician Herbrand, which implicitly defined a procedure for exhaustively searching for a proof. A sequence of systems implementing and refining this procedure (by Gilmore, Davis and Putnam, Prawitz, *etc.*) led to the resolution method, [Robinson, 1965], which then became the basis for most future work. Many of these early papers are reprinted in [Siekmann & Wrightson, 1983].

Both resolution theorem proving and many of the alternative mechanisms can be seen as manipulating an initial conjecture by the application of rules of inference. Each rule breaks the conjecture into one or more subgoals. Rules can then be applied to each of these subgoals to break them into further subgoals. The process ends when the conjecture is reduced to a collection of trivial subgoals, *e.g.* axioms of the mathematical theory.

Several different rules may apply to each goal. To be sure of finding a proof it is necessary to try each of the alternative rules at some stage. This situation is illustrated in figure 1. Thus automatic theorem proving is a *search* problem. This is where the problems start. For non-trivial theorems the number of rule combinations which must be explored becomes astronomically large. Storing all the possibilities can easily exhaust the memory capacity of even the largest computers. Exploring

all the possibilities can take more time than anybody is willing to wait. This phenomenon is called the *combinatorial explosion*.



Search is represented as the exploration of a tree with the root at the top and the leaves at the bottom. The root is labelled with the original conjecture. Each rule applying to this conjecture is represented as a circle connected to the conjecture node. Each of these rules gives rise to a (possibly empty) set of subgoals. Each subgoal is represented as a square connected to the rule. Further rules apply to each subgoal, creating further subgoals, etc.

Figure 1: Theorem Proving as Search

Most work in automatic theorem proving has been directed to controlling the combinatorial explosion.

- Resolution-based systems have been refined so that fewer rules apply to each goal or so that each rule application makes more progress.
- Alternatives to resolution, requiring less search, have been invented, *e.g.* term rewriting (explained below).
- Heuristics have been devised to select the most promising rule applications first or to decide that some rule applications should not be tried at all.
- More efficient means of storing and applying rules have been devised so that the effect of the combinatorial explosion is reduced.
- Interactive systems have been developed in which a human user helps to direct the search.

Proving theorems in most areas of mathematics is a *semi-decidable* problem. This means that: if the conjecture is a theorem then a *complete* theorem prover will

eventually[1] find a proof; if the conjecture is not a theorem then a complete theorem prover might never terminate in its fruitless search for a proof. It is possible to prove that this is the best that can be done in general. A few simple areas of mathematics are *decidable*. This means that a *decision procedure* can be built for them. Given any conjecture this procedure will eventually terminate and say whether or not the conjecture is a theorem. Unfortunately, many decision procedures are very inefficient, *i.e.* taking a time that is much worse than exponential in the size of the conjecture. Some work in automatic theorem proving has gone into improving the efficiency of these decision procedures.

Techniques have been discovered which are incomplete[2] , but are efficient and which work in many situations of practical interest. An example is term rewriting. A mathematical theory is expressed as a set of equations, *e.g.* $x + 0 = x$. These equations are oriented to form *rewrite rules*, *i.e.* they are given a direction from left to right or right to left, *e.g.* $x+0 \Rightarrow x$, where the $=$ sign is turned into the arrow $\Rightarrow$ to indicate the direction. If the left hand side of a rule matches a subexpression of the conjecture then it is replaced by the right hand side of the rule. To prove a conjecture it is exhaustively rewritten with these rules. It is often possible to show that this process will terminate and that when it does the final result will be trivially either true or false.

Progress has been made since the 1950s. Some automatic provers can generate millions of goals in a feasible timescale and can find the proofs of open problems by exhaustive search, [Wos, 1993]. However, these open problems are typically in new and obscure areas of mathematics, for which human mathematicians have not yet developed strong intuitions. Interactive systems have been used to prove some very hard theorems, *e.g.* [Boyer & Yu, 1992]. However, these are typically in application areas, *e.g.* program verification, in which the proofs are straightforward, but long and complicated. Despite these advances, the totally automatic proof of hard mathematical theorems still seems a long way off.

3 Criticisms of Uniform Proof Procedures

In the early days of artificial intelligence, automatic theorem proving promised to be a central engine of intelligent systems. Many AI problems (in robot planning; natural language understanding; visual perception; *etc.*) could be formulated as theorem proving in a theory of commonsense knowledge. Theorem proving was a unifying factor in AI.

High expectations ended in disappointment. The combinatorial explosion proved to be an obstacle, even in commonsense reasoning. Theorem provers were far too slow. Many critics identified the uniform nature of resolution-like systems as the source of the problem. They proposed the use of inference mechanisms more closely tuned to their applications, *i.e.* containing features which would limit the amount of search to that strictly required. In expert systems, for instance, production rules were a popular choice. In natural language, semantic nets, frames and isa hierarchies were used for representing the meaning of utterances. In vision, constraint propagation was used for object recognition. In practice, these specialised inference mechanisms were often essentially similar to the uniform proof procedures they replaced, but more limited in the kinds of inference they could perform.

[1] But this could take a *very* long time.

[2] *I.e.* May not find a proof although one exists.

Brooks' proposals can be seen as a continuation of this criticism. He characterises search-based mechanisms as a dead-end in the progress of AI. For instance, in [Brooks, 1991, §3.2] he says:

> "In the early days of the formal discipline of Artificial Intelligence, search was adopted as a basic technology. It was easy to program on digital computers. It lead [*sic*] to reasoning systems which are not easy to shoe-horn into situated agents."

So, in the 1970s, AI turned away from theorem proving mechanisms[3] because of the problems of search. But proving theorems remains as a task performed by humans and requiring intelligence. Can AI emulate it, despite the problems caused by the combinatorial explosion? In particular, does Brooks' proposal of behaviour-based robots suggest an alternative mechanism to uniform proof procedures? Put another way, can we prove theorems without search?

4 Tactic-Based Theorem Proving

Divorced from AI, automatic theorem proving turned to computer programming as its main application area. Computer programming problems can be turned into theorem proving problems in the following way:

- *A computer program can be viewed as a mathematical theory.* In logic programming a computer program consists of a set of logical formulæ which are the axioms of a logical theory. In functional programming a computer program is a set of equations which are the axioms of an algebra.
- *A simple automatic theorem prover can be used to run such a computer program.* A resolution-based theorem prover can run a logic program. A rewrite-rule-based theorem prover can run a functional program.
- *Automatic theorem provers can also be used to reason about computer programs.* A program can be proved to meet a specification of its intended behaviour. A program can be synthesised which meets a specification. One program can be transformed into another, more efficient one, meeting the same specification. A program can be proved to terminate.

The specification of a program is usually represented as a logical formula. This formula defines a relationship between the input and output of a program, but without describing how the output is to be constructed from the input. For instance, a sorting program might be specified by saying that the output is an ordered permutation of the input. Automatic theorem proving can then be used to give machine assistance to the development of programs, *e.g.* by helping to synthesise a program to meet the specification.

Researchers in *automated program development* have developed interactive theorem provers to tame the combinatorial explosion. They use automatic theorem proving for the straightforward, but long and tedious phases of the proof. Human users direct the proof between these phases. An automated phase might be the application of a decision procedure or the simplification of an expression. A human directed phase might be a split into cases or the use of a key lemma.

[3] Although, in recent years there has a been a reawakening interest in automatic theorem proving within AI as a result of the increasing use of logic for knowledge representation.

A major development came with the introduction of *tactics* to direct the automated phases of the proof, [Gordon *et al*, 1979]. A tactic is a computer program whose effect is to apply the rules of a mathematical theory to a conjecture. The simplest tactics merely apply one rule. Tactics can be composed using *tacticals*. A tactical might apply two tactics in sequence, apply one tactic repeatedly or use a test to decide which of two tactics should be applied. Compound tactics can be built, for instance, to implement simplification and decision procedures.

5 Examples of Tactics

In my group at Edinburgh we have built a number of tactics for different areas of mathematics. Our methodology is to study a number of similar proofs, try to abstract from these to identify the common structure, and then implement this as a tactic. We also try to identify the conditions under which these tactics should be applied. Our theorem provers use these tactics and their preconditions to search, not at the level of low level proof rules, but at the higher level of tactics. At this higher level the proof steps are bigger and the number of choices is fewer. This helps to defeat the combinatorial explosion and make a practical theorem prover.

Some of our tactics implement the traditional simplification and decision procedures, but others go beyond this to implement larger and more difficult proof steps. For instance, we have built tactics which make case splits, apply induction, generalise conjectures and introduce key lemmas. Some of our tactics can direct a whole proof. This section gives a flavour of what is possible by describing some of these tactics.

5.1 Equation Solving: Isolation

Our PRESS system solved equations taken from 'high school' examination papers, [Sterling *et al*, 1989]. Typical equations solved are:

$$\log_2 x + \log_x 2 = 5$$
$$3.\tan 3.x - \tan x + 2 = 0$$
$$4^{2.x+1}.5^{x-2} = 6^{1-x}$$

In each case the solver is asked to find the values of unknown quantity, x, which will make the equation true.

For PRESS we developed a collection of tactics which together are highly successful in solving equations. For instance, on examples, such as those above, drawn from the GCE A Level[4] examination papers PRESS typically solves 80/90%.

One of these tactics is *isolation*. Isolation applies whenever the equation contains only a single occurrence of the unknown. It works by stripping off all functions surrounding this single occurrence until it is isolated on one side of the equation. Consider, for instance, the following simple equation:

$$2.x^2 - 3 = 5$$

Since it contains only one occurrence of x, isolation applies and solves it as follows:

$$2.x^2 - 3 = 5$$

[4] An examination taken by English students about to enter University.

$$
\begin{aligned}
2.x^2 &= 5+3 \\
x^2 &= \frac{5+3}{2} \\
x &= \pm\sqrt{\frac{5+3}{2}} \\
x &= \pm 2
\end{aligned}
$$

Each isolation step removes the outermost function surrounding x and applies its inverse to the other side of the equation. First -3 is taken from the left-hand-side and replaced by $+3$ on the right; then 2. is replaced by $/2$ and, finally, square is replaced by square root. Arithmetic is then used to simplify the result.

Isolation can be implemented by selective term rewriting. The rewrite rules all have the same essential form:

$$f(U) = V \Rightarrow U = f^{-1}(V)$$

where f^{-1} is the inverse of f. Note that U must be instantiated to an expression which contains the single occurrence of the unknown x. The isolation rules which are used in our example are:

$$
\begin{aligned}
U - W = V &\Rightarrow U = V + W \\
W.U = V &\Rightarrow U = \frac{V}{W} \qquad \text{where } W \neq 0 \\
U^2 = V &\Rightarrow U = \pm\sqrt{W}
\end{aligned}
$$

Isolation will not work on an equation containing more than one occurrence of the unknown. It isolates only one occurrence on the left hand side of the equation. The right hand side of the equation will then contain the other occurrences, which means it fails the criteria for being a solution.

To solve equations containing more than one occurrence of x requires additional tactics, *e.g. collection* and *attraction*. Collection works by reducing the number of occurrences of the unknown, x, until isolation applies. Attraction works by bringing occurrences of the unknown closer together until collection applies. Thus attraction prepares equations for collection which prepares them for isolation. In PRESS, six major tactics and a few lesser ones interact in this way to produce solutions to a wide variety of equations. All these tactics are implemented as the selective application of rewrite rules.

5.2 Inductive Proofs: Rippling

Our *Oyster-Clam* system proves inductive theorems of the kind required to reason about recursive computer programs, [Bundy *et al*, 1991]. Typical theorems proved are:

$$
\begin{aligned}
x + (y + z) &= (x + y) + z \\
len(app(x, y)) &= len(x) + len(y) \\
rev(rev(x)) &= x
\end{aligned}
$$

where $app(x, y)$ appends the lists x and y, $len(x)$ is the length of the list x and $rev(x)$ is the list x in the reverse order. We have developed a collection of tactics which are highly successful in proving such theorems.

Many of the functions which occur in these theorems are defined recursively, *i.e.* in terms of themselves. For instance, $+$ is defined on the natural numbers[5] as follows:

$$0 + y = y$$
$$s(x) + y = s(x + y)$$

where $s(x) = x+1$, *e.g.* $s(s(s(0)))$ represents the number 3. s is called the *successor function*. It is a convenient trick for representing all the natural numbers with just one constant, 0, and one function, s.

To prove theorems about recursive functions it is necessary to use mathematical induction. There are many different rules of induction[6], but the most common is:

$$\frac{P(0), \qquad P(n) \rightarrow P(s(n))}{P(n)}$$

i.e. to prove P for any natural number n, first prove it for 0, and second, assuming it for n, prove it for $s(n)$. The first subgoal is called the *base case* and the second subgoal is called the *step case*. In the step case $P(n)$ is called the *induction hypothesis* and $P(s(n))$ is called the *induction conclusion*.

Rippling is a tactic we have developed to guide the step case of inductive proofs, [Bundy *et al*, 1993]. It tries to rewrite the induction conclusion into a form where the induction hypothesis can be applied to it. In particular, it moves the innermost $s(\ldots)$ parts of the induction conclusion outwards to form a copy of the induction hypothesis within the induction conclusion, *i.e.* it rewrites $P(s(n))$ to $Q[P(n)]$. The expression $s(\ldots)$ is an example of what we call a *wave-front*. The word "rippling" comes from the following analogy.

> Imagine you are in Scotland standing beside a loch. The surrounding mountains are reflected in the loch. You throw something in the loch. The waves it makes disturb the reflection. The wave-fronts ripple outwards leaving the reflection intact again. The mountains are the induction hypothesis, the reflection is the induction conclusion and the wave-fronts are the expressions introduced into the induction conclusion by the induction rule.

We illustrate rippling in figure 2 with the example of the associativity of $+$. The wave-fronts are shaded to emphasise their movement.

Rippling is implemented as selective rewriting using rewrite rules of the form:

$$f(\boxed{w_1(\underline{U})}\ ) \Rightarrow \boxed{w_2(\underline{f(U)})}$$

where the wave-fronts in the rule must match wave-fronts in the induction conclusion. These are called *wave-rules*. Examples of wave-rules are:

$$\boxed{s(\underline{U})}\ + V \Rightarrow \boxed{s(\underline{U + V})} \tag{1}$$

$$\boxed{s(\underline{U})}\ = \boxed{s(\underline{V})}\ \Rightarrow U = V \tag{2}$$

[5] *I.e.* the non-negative integers: $0, 1, 2, 3, \ldots$.

[6] In fact, infinitely many.

$$\boxed{s(\underline{x})} + (y + z) = (\boxed{s(\underline{x})} + y) + z$$

$$\boxed{s(\underline{x + (y + z)})} = (\boxed{s(\underline{x + y})}) + z$$

$$\boxed{s(\underline{x + (y + z)})} = \boxed{s(\underline{(x + y) + z})}$$

$$x + (y + z) = (x + y) + z$$

The first equation is the induction conclusion annotated by wave-fronts. Subsequent equations show the effect of rippling these wave-fronts outwards using the rewrite rules (1) (three times) and finally the cancellation rule for s *(2). The wave-fronts ripple outwards until they finally disappear altogether. At this point the induction hypothesis can be used to complete the proof. In general, the wave-fronts do not disappear but are moved to the outside of the induction conclusion leaving an expression inside that matches the induction hypothesis. When rippling is complete the induction hypothesis is used to prove the induction conclusion. The tactic which does this is called* fertilization.

Figure 2: Rippling in the Associativity of +

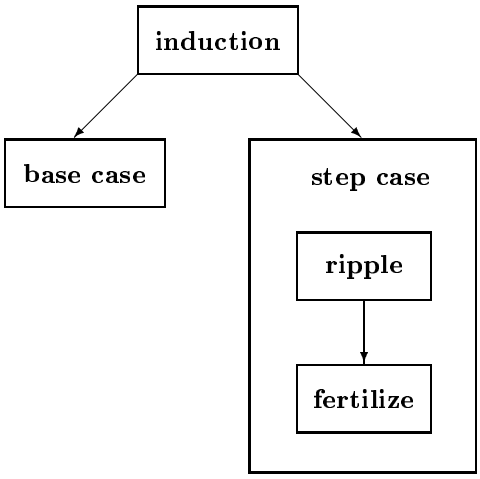
Note that in wave-rule (2) the right-hand-side wave-front, w_2, is empty.

When wave-rules are applied not only must the mathematical expressions match, but so too must the wave-fronts. This cuts down the amount of search quite drastically, [Bundy *et al*, 1993, §2.5], *i.e.* there is rarely more than one wave-rule applicable to a sub-expression. Similar remarks hold for other tactics described in this paper. As a result, our tactic-based approach avoids the combinatorial explosion which plagues other approaches to automatic theorem proving.

5.3 An Induction Strategy

By combining tactics together it is possible to make a tactic that can direct the complete application of an induction rule, *i.e.* including the proof of the base and step cases. We have implemented such a tactic in *Oyster-Clam*, which we call the *induction strategy*. It is summarised in figure 3. For some simple theorems the induction strategy can direct the whole proof, *e.g.* the associativity of +. Some slightly more complex theorems can be proved by nested applications of the induction strategy.

The induction strategy shows that the word "tactic" was, perhaps, ill-chosen. Theorem proving tactics are not restricted to directing only parts of proofs, as their name might suggest. The induction strategy alone can direct a complete proof. Nor are they restricted to directing only the simplest parts of proofs, *e.g.* decision procedures and simplification. For instance, rippling is capable of making, so called, 'eureka' steps, *i.e.* those proof steps that seem to call for human intervention.



The induction strategy consists of the application of a rule of induction followed by one or more base and step cases. Only one of each is shown here. The step case consists of rippling until the induction hypothesis appears within the induction conclusion and then fertilising to use the induction hypothesis in the proof of the induction conclusion.

Figure 3: The Induction Strategy

5.4 Proof Critics

However, in tactic-based systems there are nearly always theorems which cannot be proved solely by the use of available tactics. For instance, the ripple tactic will fail if none of the available wave-rules is applicable to the current subgoal. Then the induction strategy will also fail.

To cope with these failures we have developed a system of *proof critics*. With each tactic is associated a description of what it is trying to achieve and a collection of critics. Each critic tries to capture one of the ways in which the tactic might fail to reach its goal and to suggest a patch to the partial proof. The critic has a precondition, which is a variant of the precondition of its associated tactic, *e.g.* it might be missing one key part. This precondition is used to determine which critic to use to patch a failed proof.

The critics associated with rippling can suggest any of the following patches, according to the precise way in which rippling fails.

- Conjecturing a wave-rule which would apply to the current subgoal, proving it and then using it.
- Attempting a more complex form of induction which would produce wave-fronts at this point in the proof which would enable a wave-rule to apply that currently does not.
- Generalising the conjecture in such a way as to allow a wave-rule that currently

does not apply to do so.

- Splitting the proof into cases so that a conditional wave-rule will apply which was previously inapplicable due to the failure to prove its condition.

All of these patches represent proof steps which have previously been regarded as hard to automate. For instance, suggesting appropriate lemmas and generalising conjectures are classic examples of 'eureka' steps.

5.5 Proof Plans

Our *Oyster-Clam* system works in two phases. The *Clam* system first tries to find a plan of the proof. This is represented as a custom-built tactic constructed from the general-purpose tactics known to the system and modified by the proof critics. The *Oyster* system then runs this custom-built tactic to produce a formal proof. The preconditions of the tactics are used during the proof planning phase together with predictions of the effects of the tactics. Fairly standard AI plan formation techniques are used to construct the customised tactic from the preconditions and effects of the general-purpose tactics and proof critics.

We find this two phase process useful. For instance, it enables critics to work on a partial proof and to suggest radical rearrangements of it, *e.g.* going back to the beginning and proving a more general theorem but with a similar proof. However, it is not an essential ingredient of a tactic-based system. Tactics are most commonly used in interactive systems, in which the user decides when and which tactic to call. It is also possible to implement a totally automated one phase process in which search is conducted at the level of tactics using their preconditions to determine which tactics are applicable. Christian Horn has built a rival version of *Oyster* which uses tactics in this way, [Horn, 1992]. This one phase approach works quite well for organising tactics but it does not lend itself so well to the use of critics for patching failed proof attempts.

6 Comparison of Tactics and Subsumption Architectures

We now turn to the main question addressed in this paper; to what extent are tactic-based theorem provers similar to the subsumption architecture advocated by Brooks. To facilitate this comparison I have translated my characterisation of Brooks' subsumption architecture from §1 into tactic-based terminology.

> The theorem prover must be organised as a network of tactics. Each of these tactics is behavioural, *i.e.* it is capable of performing a whole theorem proving behaviour, *e.g.* decision making, simplification, induction. This is in contrast to a functional module, where each module performs one part of a sequence of tasks, *e.g.* formula retrieval, matching, search control. Behavioural tactics are organised into layers, but operate independently and in parallel. Overall behaviour emerges from the combination of the behaviours of the tactics. The only interaction allowed is that higher-level tactics may inhibit the action of lower-level ones. There is no hierarchical organisation, no message passing between tactics and no central model of the world.

We start by considering in what ways tactic-based theorem proving fits this description and then discuss in what ways it does not fit.

6.1 Similarities between Tactics and Subsumption Architectures

A tactic-based system is organised as a network of behavioural modules. Each tactic is a module. A tactic is "behavioural" in the sense that it performs part of the theorem proving task unaided. For instance, a decision procedure tactic will prove whole theorems if they happen to fall into its decidable class. More usually it will "finish off" the cases of a larger proof. Other tactics perform the earlier parts of a proof, *e.g.* setting up an induction. That these tactics only perform part of the overall theorem proving task does not make them less behavioural than Brooks' modules. Some of Brooks' modules only perform part of a task. For instance, a Brooks' walking module might perform the early part of a task by getting the robot to the right place, *e.g.* so that it can grasp an object.

Although we have organised them hierarchically, tactics *could* be organised in a non-hierarchical way. A theorem prover could consist of a pool of tactics together with their preconditions. The conjecture to be proved could be stored in some central pool accessible to all tactics. Any tactic whose preconditions are satisfied would fire, manipulate the conjecture and put any resulting subgoals into the central pool. The other tactics would try to solve these subgoals. Each tactic would be built on the assumption that the other tactics exist and will do the right thing with the subgoals it produced, but it would not need to communicate with them directly. A special module would need to inspect the goal central pool to determine when the theorem was proved and terminate the process. Horn's alternative version of *Oyster* already works in much this way, except for the parallelism, and this omission could be readily rectified.

It is worth noting that this kind of non-hierarchical organisation is quite common in GOFAI, *e.g.* production rules, blackboard architectures. However, it usually introduces lots of search and causes a combinatorial explosion. The preconditions of our tactics preclude too much competition and we rarely have much search.

The central store of conjecture and subgoals can be regarded as analogous to the real world in which the robot moves. Manipulations of these formulæ correspond to actions on the real world. This central store is not the forbidden (by Brooks) "model" of the real world. These are not simulations of real formulæ — they *are* real formulæ.

The fact that we did not choose this kind of organisation in our proof plans work is irrelevant to the argument in this paper. It could be done this way, Horn has shown.

6.2 Differences between Tactics and Subsumption Architectures

6.2.1 Real World Interaction

The most obvious difference between Brooks' behaviour-based robotics proposal and *any* automatic theorem prover is the lack of any real world in theorem proving. The objects of pure mathematics are symbols and expressions. These are inherently abstract. Thus there is nothing for the prover to be situated or embodied *in*.

This difference seems to be unavoidable, *i.e.* this part of Brooks' proposal seems unachievable.

However, this disconnectedness with the real world could be interpreted as an implicit criticism of automated mathematical reasoning as being too restricted to proving theorems. Real mathematicians are not concerned only with proving theorems in particular theories. They also formalise informally stated problems, design mathematical theories, decide which conjectures are interesting to prove, *etc.* Much of this mathematical activity is informed by real world problems. The classic example is the interaction between physics and mathematics, which led to the invention of the calculus, Fourier series, modern analysis and much much more. A Brooks-style behavioural-based automatic mathematician might, for instance, use a mathematics module to help solve engineering problems prior to constructing mechanical devices.

This said, it is still true that much of mathematics can and is carried out in a disembodied manner, even by human mathematicians. Such disembodied mathematics remains as a challenge to the Brooks proposal.

6.2.2 Hierarchical Organisation

In our proof planning approach to theorem proving we chose a very hierarchical organisation. This takes two different forms:

1. Large tactics are built from smaller ones, *e.g.* our induction strategy is built from the base and step case tactics; the step case is built from rippling and fertilization.

2. Our *Clam* program is a centralised planner. It constructs a customised tactic for the current conjecture by combining general-purpose tactics modified by proof critics.

Let us call these type 1 and type 2 hierarchy, respectively. I have argued in §6.1 above that type 2 hierarchy is an inessential part of tactic-based theorem proving. A flat organisation of tactics is possible and has been implemented, *e.g.* in [Horn, 1992]. We have chosen not to do this because it does not facilitate the use of proof critics to patch failed proof attempts. Critics must investigate the failure of a tactic, which requires message passing between tactic and critic. Critics have proven a powerful technique in avoiding the inherent limitations of the tactic-based approach and for suggesting key "eureka" steps in proofs.

An alternative, more Brooksian, implementation of critics is possible[7]. Each critic would stand alone, monitoring the state of the proof. However, such an implementation would be much clumsier. Currently each critic is cued by the failure of a particular method. Without message passing between a failing method and its critics it would be quite complicated for a critic to work out that its time had come. Thus type 2 hierarchy can be avoided, but at a heavy price; at best we would have to settle for an inferior implementation of critics.

Type 1 hierarchy is harder to avoid. Building compound tactics from smaller ones is an essential part of the philosophy of tactic-based theorem proving. Fortunately, *this* kind of hierarchy seems quite compatible with Brooks' subsumption architecture. He imposes no constraint on how each behavioural module is built. Hierarchical module *construction* seems to be merely good programming practice.

[7] I thank Geraint Wiggins for suggesting this.

Brooks only rules out a hierarchical *organisation* of the resulting module network, any message passing between them or any central model of the world. Indeed Brooks' proposal for 'layering' seems to come close to type 1 hierarchy[8], [Brooks, 1986, §II.B]. The main departure from layering is that we have not yet discovered a need to have 'higher-level' tactics inhibit the behaviour of 'lower-level' ones — unless you count the termination of the theorem proving process by the module that recognises that the proof is complete.

Deviation from Brooks' proposal might arise if we want a tactic both to be part of some larger tactic and also to be a member of the tactic network in its own right. From our own experience, examples of this seem to be quite rare, but they do occasionally arise, *e.g.* rippling is both part of our induction strategy and useful stand-alone.

7 Conclusion

In this paper we have tried to test Brooks' hypothesis that his behaviour-based robotics proposal can be extended to cover the "whole story" of "building intelligent systems". We have subjected it to the toughest test we could devise — to see if it could cover an aspect of intelligent behaviour for which it seemed very unpromising: automatic theorem proving. This test has been surprisingly successful. The tactic-based approach to theorem proving is remarkably similar to Brooks' subsumption architecture. Tactics are very similar to behaviour-based modules. A non-hierarchical organisation of these tactics *is* possible, although it remains to be seen whether this is the most effective organisation. There are reasons for suspecting that a hybrid tactic-based and central planning model may be more successful. For instance, this allows the efficient use of proof critics. Central reasoning is also required for analysing and modifying the existing collection of tactics, *i.e.* for learning.

Brooks' requirement that the theorem prover be situated and embodied in the real world seems inherently impossible to achieve. It is not clear what constitutes the real world in the case of mathematics. It is necessary to have some internal representation of the conjecture to be proved and the subgoals generated during a proof attempt. These must either be stored centrally or passed between tactics to enable them to interact. Unfortunately, either a central model of the world or message passing between tactics violate Brooks' proposal. One solution to this is to view the central goal pool *as* the real world, rather than as a model of it. This solution is clearly controversial.

Situatedness and embodiment might be achieved by building a robot that used mathematics to solve real world problems. However, there is still a challenge to emulate the behaviour of pure mathematicians who appear to do a lot of theorem proving without direct reference to the real world.

It is significant that it is the non-hierarchical organisation part of Brooks' proposal that has been most severely tested by our experiment. This has always seemed the most unsatisfactory aspect of his work. His proposal that modules be combined by 'layering' is vague and in practice seems to result in rather *ad hoc* programming rather than a modular and principled combination. Similar remarks apply to the restriction on message passing. For instance, in one case, Brooks avoids the letter of

[8] I thank Fausto Giunchiglia for pointing this out.

this law by having one module record a message in the external world and another module read it from there.

Thus these parts of the proposal frequently impose unwanted restrictions which the implementor is forced to hack around. It seems timely to examine them and decide whether they are serving a useful purpose or whether Brooks' proposal would be better without them.

References

[Boyer & Yu, 1992] Boyer, R. S. and Yu, Yuan. (June 1992). Automated correctness proofs of machine code programs for a commercial microprocessor. In Kapur, Deepak, (ed.), *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 416–430, Saratoga Springs, NY, USA.

[Brooks, 1986] Brooks, R. A. (1986). A robust layered control system for a mobile robot. *Journal of Robotics and Automation*, RA-2(1).

[Brooks, 1991] Brooks, R. A. (1991). Intelligence without reason. In Mylopoulos, J. and Reiter, R.., (eds.), *Proceedings of the twelfth IJCAI*, pages 569–595. International Joint Conference on Artificial Intelligence.

[Bundy *et al*, 1991] Bundy, A., van Harmelen, F., Hesketh, J. and Smaill, A. (1991). Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Bundy *et al*, 1993] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A. (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253. Also available from Edinburgh as DAI Research Paper No. 567.

[Gordon *et al*, 1979] Gordon, M. J., Milner, A. J. and Wadsworth, C. P. (1979). *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

[Horn, 1992] Horn, Ch. (1992). Oyster-2: Bringing type theory into practice. *Information Processing*, 1:49–52.

[Malcolm *et al*, 1989] Malcolm, Chris, Smithers, Tim and Hallam, John. (1989). An emerging paradigm in robot architecture. In Kanade, T., Groen, F. C.O. and Hertzberger, L. O., (eds.), *Intelligent Autonomous Systems 2*, Amsterdam. Stichting International conference of Intelligent Autonomous Systems.

[Newell *et al*, 1957] Newell, A., Shaw, J. C. and Simon, H. A. (1957). Empirical explorations with the Logic Theory Machine. In *Proc. West. Joint Comp. Conf.*, pages 218–239. Reproduced in Computers and Thought (eds Feigenbaum and Feldman), McGraw Hill, New York, pp 109-133, 1963.

[Robinson, 1965] Robinson, J. A. (1965). A machine oriented logic based on the resolution principle. *J Assoc. Comput. Mach.*, 12:23–41.

[Siekmann & Wrightson, 1983] Siekmann, J. and Wrightson, G. (1983). *Automation of Reasoning 1 & 2.* Springer-Verlag.

[Sterling *et al*, 1989] Sterling, L., Bundy, Alan, Byrd, L., O'Keefe, R. and Silver, B. (1989). Solving symbolic equations with PRESS. *J. Symbolic Computation*, 7:71–84. Also available from Edinburgh as DAI Research Paper 171.

[Wos, 1993] Wos, L. (January 1993). Automated reasoning answers open questions. *Notices of the AMS*, 5(1):15–26.