SCOTT: A Model-Guided Theorem Prover

John Slaney
Automated Reasoning Project
Centre for Information Science Research
Australian National University
GPO Box 4, Canberra, ACT 2601
Australia

Abstract

SCOTT (Semantically Constrained Otter) is a resolution-based automatic theorem prover for first order logic. It is based on the high performance prover OTTER by W. McCune and also incorporates a model generator. This finds finite models which SCOTT is able to use in a variety of ways to direct its proof search. Clauses generated by the prover are in turn used as axioms of theories to be modelled. Thus prover and model generator inform each other dynamically. This paper describes the algorithm and some sample results.

SCOTT (Semantically Constrained Otter) is a resolution based automatic theorem prover for first order logic. So much is hardly revolutionary. What is new in SCOTT is the way in which it blends traditional theorem proving methods, best seen as purely syntactic, with techniques for semantic investigation more usually associated with constraint satisfaction problems. Thus it bridges two aspects of the science of reasoning. It was made by marrying an existing high-performance theorem prover to an existing model generator. Neither parent program was much modified in this process. The resulting combined system out-performs its parents on many problems, for some of which it is currently the most effective prover available.

1 The Parents

1.1 Syntax: OTTER

The theorem prover OTTER, written by W. McCune and based on earlier work by E. Lusk, R. Overbeek and others, is a product of Argonne National Laboratory and is widely regarded as the most powerful program of its type for certain classes of problem ([McCune, 1990; Lusk and McCune, 1992]). Its basic method is forward chaining, applying a rule of inference *R,* seen here as a partial function on clauses, to generate new clauses as in Figure 1. The clauses are divided into two disjoint sets, the *set of support* and the *usable list* Initially the set of support is non-empty. Clearly the proof search may terminate with a successful proof, or the set of support may be emptied, showing that there is no proof in the chosen search space, or it may fail to terminate. First order logic being undecidable, this is to be expected.

```
repeat
    choose a given clause g from SOS;
    move g to UL;
    for c_1 ... c_n in UL such that
    R(c_1 ... c_i, g, c_{i+1} ... c_n) exists do
        a ← R(c_1 ... c_i, g, c_{i+1} ... c_n);
        if a is the goal then
            report the proof;
            stop
        else if a is new then
            add a to SOS
        fi
    od
until SOS is empty
```

Figure 1: Basic OTTER Algorithm

The simplest rule applied by OTTER is binary resolution, together with unification and factoring. More interesting and powerful variants include hyper-resolution, negative hyper-resolution and unit resulting resolution. For equational reasoning the available rules include the various forms of paramodulation and term rewriting (demodulation).

A crucial part of the algorithm is the decision as to whether each deduced formula is "new". In general this means that it is not subsumed by any formula already kept. Much of the high performance of OTTER is due to its sophisticated techniques for reducing the time spent computing subsumption and related properties. These techniques are not the focus of the present paper. Also not shown in the simple version of the algorithm in Figure 1 is the rewriting due to demodulation and the like which may take place before the subsumption test. Nor have back subsumption and back demodulation (whereby the generated clause is used to simplify the existing clause database) been made explicit, although in many applications they are important.

1.2 Semantics: FINDER

Deduction is only one form of reasoning. Another is the generation of models of a given theory. A model of

a theory shows that theory to be consistent, but it also shows much more. It is an account of *what it might be like* for the theory to be true. It divides the whole language into truths and falsehoods in a way consonant with the theory. Thus, given an effective means of detecting the truth values it assigns to formulas, it can be used to reach out beyond the given basis to determine semantic-properties of all formulae.

If a theory has finite models, one way of generating them is to fix the domain as consisting of a few objects (say three objects, or thirty) and then to perform an exhaustive search in the space of functions definable by enumeration over that domain. A model is an interpretation of the non-logical symbols in the theory's language making all of its axioms true. Since all of the structures in question are finite, it is easy to recognise models when they are found. There are, of course, more efficient and less efficient methods of searching. FINDER(see [Slaney, 1992]) performs a backtracking search, building a database of facts about the search space to enable it to avoid ever having to backtrack twice for the same reason. The details are interesting but are not germane to the present application since any reasonable constraint satisfaction algorithm could be used to much the same effect provided it is not heavily dominated by the overheads of starting up a search.

FINDER accepts input in a fairly friendly format. It works with first order theories in clausal form couched in a many-sorted language. Function symbols must be given first order types in terms of the sorts. There are a few pre-defined function symbols: for example, each sort comes with an identity relation, symbolised by ' = ' Each sort is also equipped with a total order, symbolised with '<' and '>'. Evidently, this convention does no harm and it is often useful. A clause is a set of literals and is true iff for every assignment of elements from the domains to variables occurring in it one of those literals is true.

It is worth noting that what FINDER does is rather different from logic programming. In the first place, it imposes no order on the evaluation of clauses and has no 'flow of control'. The clauses meet each candidate model in a body; if they are all true then the model is good, while if any clause is false, the model is adjusted to deal with that badness, resulting in a new candidate. In the second place, FINDER is not able to show the nonexistence of models and hence cannot prove sets of clauses inconsistent. It returns a null result if there is no model within a specified finite search space, but always leaves open the possibility that there may be models outside that space. In the third place, it finds and specifies the models without any reference to the Herbrand universe. The domain consists simply of the first object, the second object and so forth, any relationship with terms of the language being accidental.

2 Putting it Together

There are several distinct ways in which semantic information such as the truth value of a formula in a model can be used to direct proof searches. The oldest and simplest is *goal deletion.* In attempting to show by backward chaining that a goal formula is a theorem of some theory, we decompose the goal into simpler subgoals and work recursively on those. In typical cases, most of the subgoals are unprovable. Having discovered that a subgoal cannot be reached, the proof search must backtrack and try another. Hence techniques for rapidly detecting and deleting unprovable subgoals are valuable. One good technique is to test goals for truth in some simple model of the theory. Any that are false are unprovable and may be deleted. See [Ballantyne and Bledsoe, 1982] and [Thistlewaite *et* al, 1988] for some discussion. OTTER is a forward chaining prover, so goal deletion is not the main present concern, though it is fairly obvious how a method similar to that of SCOTT could be used for goal deletion in backward chaining systems.

```
repeat
    choose a given clause g from SOS;
    label g safe or unsafe;
    move g to UL;
    for c_1 ... c_n in UL such that
    R(c_1 ... c_i, g, c_{i+1} ... c_n) exists
    and such that at least one of
    g, c_1 ... c_n is unsafe do
        a ← R(c_1 ... c_i, g, c_{i+1} ... c_n);
        if a is the goal then
            report the proof;
            stop
        else if a is new then
            add a to SOS
        fi
    od
until SOS is empty
```

Figure 2: Basic SCOTT Algorithm

Another use for semantic information is in the *false preference* strategy. This is appropriate to forward chaining proof search and uses some model or models in which the goal is false. The strategy is to prefer parent clauses of inferences to be ones false in the guiding model or models, on the thought that the goal is more likely to be deduced from clauses which imply it in the models than from those which do not. Since this is only a heuristic, it can usually be shown not to affect the prover's completeness. SCOTT (optionally) implements the false preference strategy in a straightforward way, testing each kept clause against a guiding model and assigning greater weight to clauses which are true in the model than to those which are false.

Most interesting for present purposes is the idea of *semantic resolution.* Given any model M, a simple theorem assures us that if there is a derivation of the empty clause from a set of clauses using unification, resolution and factoring then there is one in which no inference has parents both of which are true in *M.* This is a well worn result, for a fuller account of which see [Chang and Lee, 1973] or [Slagle, 1967]. SCOTT implements semantic resolution by means of two very simple amendments to OTTER's algorithm. These are underlined in Figure 2. The safe clauses are those true in the guiding model. They are "safe" because they are known to form a con-

```
if generating then
    if g is false in M then
        N ← Model(T ∪ {g});
        if N is not null then
            M ← N
        fi
    fi
    if g is true in M then
        T ← T ∪ {g}
    fi
fi

if g is true in M then
    label g 'safe'
else
    label g 'unsafe'
fi
```

Figure 3: Basic TESTER Algorithm

sistent set whose immediate consequences may therefore safely be omitted. If the given clause *g* is safe then it may not react with other clauses unless they are unsafe. This cuts down the number of generated clauses sufficiently to have a marked effect on OTTER.

Before the guiding model can be used it must be found. The clause testing module which assigns labels to given clauses calls FINDER from time to time, to generate a model of a set of clauses or to return the information that no model was found within the delimited search space. The logic of the clause tester is given in Figure 3. Note that at any given time it is either in generating mode or in simple testing mode. At the start of the proof search it is in generating mode; after a while it stops trying to generate any new models and becomes just a tester. This is because model generation is expensive in comparison with testing and because there comes a time when no better model can be found without enlarging the search space to an unacceptable degree. The cutoff point at which the mode is switched may be set as a parameter, the default being after 100 clauses have been evaluated. Also at any given time the tester has associated with it a theory *T* (the set of safe clauses so far) and a model *M* of that theory. The further procedure 'Model' is the model generator (in effect FINDER, in SCOTT's case) and its parameter is the theory to be modelled. It must be given a finite search space, so that it always completes even when it fails to find a model. In order to set up the semantic apparatus consistently, FINDER is called initially with the theory consisting of the clauses (if any) in the initial usable list. If it fails to model this, the whole proof attempt fails, since otherwise dynamic semantic resolution would not fit with the set of support algorithm.

Two input scripts are needed for SCOTT. One is the normal OTTER input consisting of the problem to be solved. It specifies the initial contents of the usable list and the set of support, together with any settings for OTTERS optional parameters such as which rules to use, what weight limit to set and what results should be printed. The other is a FINDER input file, consisting of the sort and function specifications for the problem, together with any clauses in the initial usable list and optionally FINDER settings such as verbosity modes. Expert knowledge about the problem domain may be added in the form of extra clauses to help direct FINDER to the models. In practice this facility is best used sparingly, as the program may make more intelligent choices of model than the "expert". Any extra clauses given to FINDER are not communicated to the OTTER part of the prover, so they do not form part of the proof.

3 Case Studies

A systematic evaluation of SCOTT has yet to be made, since it has been designed and developed only in 1992. The following sample results show its effect in three selected cases where OTTER is already one of the most effective theorem provers available. These involve condensed detachment axiomatisation of propositional logics and are among the hardest solved problems in the field. It would have been easy to show dramatic speedup effects in cases where OTTER does poorly—non-Horn ground problems for example[1] but it is of more interest and value to improve what OTTER does best.

3.1 Classical Pure Implication

One problem set for which OTTER is particularly suitable is determined by the single axiom for the classical propositional calculus of pure implication

$$((p \to q) \to r) \to ((r \to p) \to (s \to p))$$

given by Lukasiewicz. The problem is to derive from this some axioms known to be sufficient for the theory, using the rule of condensed detachment

$$\frac{A \to B \qquad C}{B\sigma}$$

where σ is the most general unifier of *A* and *C*. A convenient sufficient axiom set consists of these five:[2]

(1) $p \to p$

(2) $p \to (q \to p)$

(3) $((p \to q) \to p) \to p$

(4) $p \to ((p \to q) \to q)$

(5) $(p \to q) \to ((q \to r) \to (p \to r))$

A very simple OTTER input file for the first axiom, for example, reads as follows.

[1] An amusing example is the pigeonhole problem with naive input, where the time taken in one case is reduced from hours to about a minute. However, on closer examination it emerges that the FINDER half of SCOTT actually solves the problem completely in a few milliseconds, after which the OTTER half labours through a proof search, treating the solution just as a hint!

[2] Axiom 4 is in fact redundant, but it has been included in the problem set because it is nicely intermediate in difficulty between axioms 3 and 5.

```
set(hyper_res).     % Hyper-resolution

list(usable).
  -p(x) | -p(i(x,y)) | p(y).
  -p(i(a,a)).       % This is the goal
end_of_list.

list(sos).
  p(i(i(i(x,y),z),i(i(z,x),i(u,x)))).
end_of_list.
```

The predicate *p* is for provability, and the function symbol *i* for implication. The goal is a Skolemized denial of the axiom to be derived. OTTER input for the other four problems is similar, with the different goals of course. In practice there may be some further settings such as a weight limit and a limit on the number of distinct variables per clause. The fifth problem is difficult. To the best of my knowledge, OTTER is the only prover to have solved it automatically.3

The basic FINDER input used by SCOTT in tandem with the OTTER input reads as follows. This is for the first problem; for the second, it is necessary only to declare another constant *b* and to change the goal.

```
sort value  { cardinality < 4 }

function {
   p: value -> bool {}
   i: value,value -> value {}
   a: value. {}
}

clause {
   p(i(x,y)), p(x) -> p(y).
   p(i(a,a)) -> false.
}

end
```

In fact, it pays to use some expert knowledge amounting to the fact that the implication relation may be expected to be acyclic. By embedding that relation in the default total order on the values we may cause the FINDER half of SCOTT to avoid searching too many isomorphic subspaces. This alternative clause set is the one used for problem (3) and, with a change of goal, for problem (4).

```
clause {
   x < y, p(i(y,x)) -> false.
   x < y, p(x) -> p(y).
   p(i(i(i(a,b),a),a)) -> false.
}
```

For hard problems like (5) we shall probably also want to help FINDER along with some more expert knowledge about what models of implication logics are like. In the experiment reported here it was directed to a good model by being informed ahead of time of some of the clauses which it would eventually have modelled anyway. The OTTER input was also changed a little, by imposing a length limit in order to cut down the number of clauses kept and by making it less verbose.

3Since this paragraph was written, the theorem proving group in the 'Fifth Generation' project ICOT in Tokyo have reported a proof using their Model Generation Theorem Prover MGTP-N which however is closely modelled on OTTER.

	clauses generated	clauses kept	clauses given	time (sec)
1: OTTER	2828	376	70	3.42
1: SCOTT	245	68	30	1.13
factor	11.5	5.53	2.33	3.03
2: OTTER	3070	386	74	3.72 J
2: SCOTT	450	99	35	2.04
factor	6.82	3.90	2.11	1.82
1 3: OTTER	8774	595	134	8.56
3: SCOTT	1782	130	60	2.59
factor	4.95	4.58	2.23	3.31
4: OTTER	1204953	1762	1196	1010 '
4: SCOTT	220734	1069	733	352
factor	5.46	1.65	1.63	2.87
5: OTTER	4282987	17508	2285	3865
5: SCOTT	1544423	13967	1823	2070
factor	2.77	1.25	1.25	1.87

Figure 4: IMP Problem: Results

Note that FINDER needs some termination condition such as a cardinality limit or a time limit. The more generous this limit, the longer SCOTT will spend trying (and failing) to model inconsistent sets of clauses, but the more restrictive it is made the more likely SCOTT is to miss some useful model. At this stage, the course between these two undesirable outcomes is steered manually, though clearly such heuristics are programmable.

As will be observed from Figure 4, SCOTT improves on OTTER on every measure. The 'factor' in each case is the figure for OTTER divided by that for SCOTT. The counts of clauses give measures of the amount of work done. In each case about two thirds of the given clauses were labelled 'safe', so approximately four ninths (two thirds squared) of clauses which would otherwise have been generated are avoided because their parents are safe. The amount of the time spent generating models and testing clauses for safety was 23% for problem 4 but only 8% for problem 5. In order to keep these experiments tidy for easy reporting, very simple FINDER input was used. Where SCOTT is used to tackle really serious problems, the option of giving it extra semantic information is clearly valuable. The input of expert knowledge and heuristics may be restricted to the semantic side, leaving the theorem prover itself clean.

3.2 Extending Intuitionist Implication

In [Karpenko, 1992] Karpenko raised the problem of finding an axiom which would strengthen intuitionist pure implication to classical pure implication but whose addition to various substructural systems would produce

	clauses generated	clauses kept	clauses given	time (sec)
Natural				
OTTER	22725376	89639	3606	25591
SCOTT	6395	149	87	7.90
factor	3553	602	643	3239
Tuned				
OTTER	209242	1862	377	289
SCOTT	3428	82	52	5.16
factor	61.0	22.7	7.25	56.1

Figure 5: Intuitionist Logic: Results

new substructural logics. There are several solutions to this problem, one of which is the formula

$$((p * q) \rightarrow r) \rightarrow (((q * p) \rightarrow r) \rightarrow r)$$

where *A * B* is defined as

$$((A \rightarrow B) \rightarrow B) \rightarrow A$$

The theorem-proving problem is to derive from this, together with some standard axioms for intuitionist implication, goal 3 of the set given in the last example:

$$((p \rightarrow q) \rightarrow p) \rightarrow p$$

A suitable set of 'standard axioms' is the following.

(1) $p \rightarrow (q \rightarrow p)$

(2) $(p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r))$

(3) $(p \rightarrow (p \rightarrow q)) \rightarrow (p \rightarrow q)$

(4) $(p \rightarrow (q \rightarrow r)) \rightarrow (q \rightarrow (p \rightarrow r))$

and again the problem is easily set up for OTTER to solve by enumerating consequences by condensed detachment.

Observe that the second axiom assures us that the implication relation is transitive. Hence we are entitled to add a new clause giving us the rule of condensed transitivity.

$$\frac{A \rightarrow B \qquad C \rightarrow D}{(A \rightarrow D)\sigma}$$

where *a* is the most general unifier of *B* and *C*. The OTTER input clause for this reads:

```
-p(i(x,y)) | -p(i(y,z)) | p(i(x,z)).
```

This is a useful proof-shortening device. Figure 5 shows the results of running OTTER and SCOTT on this problem, first without any special settings, except for a limit of three distinct variables per formula, then (after many experiments) with settings hand-tuned to help OTTER. It is worth noting that this problem was open until closed by SCOTT.

	clauses generated	clauses kept	clauses given	time (sec)
OTTER	4016344	11304	1991	11708
SCOTT	3047711	17621	6558	29593
factor	1.32	0.64	0.30	0.40

Figure 6: Many Valued Logic: Results

3.3 Many Valued Logic

A similar problem to that presented by the Lukasiewicz single axiom arises in the axiomatisation of his many valued logic. This time there is a negation connective as well as implication. The rule of inference is again condensed detachment, and the axioms are:

(1) $p \rightarrow (q \rightarrow p)$

(2) $(p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r))$

(3) $((p \rightarrow q) \rightarrow q) \rightarrow ((q \rightarrow p) \rightarrow p)$

(4) $(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$

The thought behind axiom (3) is that $(A \rightarrow B) \rightarrow B$ defines *AVB,* so (3) amounts to the commutativity of disjunction. Now the hard problem is to show that implication is a total order:

$$((p \rightarrow q) \rightarrow (q \rightarrow p)) \rightarrow (q \rightarrow p)$$

In view of axiom 2, we may add the clause for condensed transitivity as before. Next, it is not hard to see that the logic being axiomatised satisfies a rule of replacement of proved equivalents. We can secure the *effect* of replacement by adding a clause to make derived two way implications into equalities thus:

```
-p(i(x,y)) | -p(i(y,x)) | EQUAL(x,y).
```

together with settings to make OTTER add demodulators (rewrite rules) whenever it derives a directable equality. This, for instance will cause any subformula of the form $\neg\neg A$ to be rewritten as A, thus eliminating huge numbers of redundant equivalents.

With careful setting of weight limits and the like, OTTER can now solve the problem. It takes 3 hours 15 minutes on a Sparc-2, generating 4016344 clauses of which 11304 are kept and 1991 are given (added to UL). FINDER can easily be instructed to find a model of all the axioms except for the commutativity one, in which model negation is well-behaved in that $x \rightarrow y$ is everywhere the same as $\neg y \rightarrow \neg x$ and in which the goal formula is false. The results of running SCOTT with that FINDER file and with the same problem input as OTTER are interesting (Figure 6). Overall performance actually degrades, despite the fact that over 95% of all given clauses are labelled 'safe'.

More useful for this problem is the false preference strategy. OTTER normally selects given clauses by weight, lightest first, taking the weight to be the number of symbols in the clause. SCOTT applying the false preference strategy with a safe-weight of *S* tests each kept

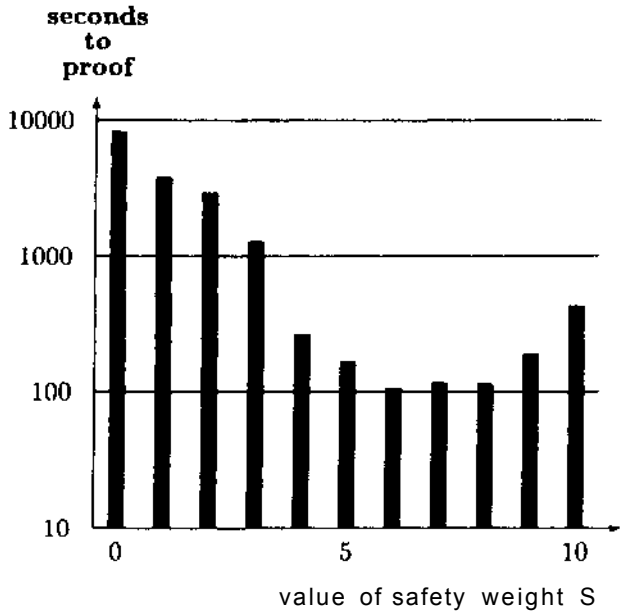

Figure 7: False Preference Strategy

clause before its insertion in the set of support and adds *S* to its weight if it is true in the current model. This causes selection of 'safe' clauses to be delayed. Figure 7 shows the effect on OTTER's proof search. Note that the scale is logarithmic. The case 5 = 0 is just OTTER, since dynamic semantic resolution was not used in this experiment. An improvement of almost two orders of magnitude is possible, but only if the choice of *S* is right. At present, no automatic method of finding an appropriate value for *S* is known, though it seems likely that the optimal value will be similar for cognate problems.

4 Remarks

SCOTT brings semantic information into the service of forward chaining resolution proof search. Looked at abstractly, this is *obviously* a move in the direction of intelligence. OTTER is powerful but blind. When asked to prove Axiom (1) in the Lukasiewicz implicational calculus problem, it starts performing *exactly* the same search as it does when asked for a proof of Axiom (4). In other words, it never refers to the goal except to check whether the proof is finished, so the goal has no effect on the search. The really remarkable fact is that such a strategy works at all! The effect of injecting models is to enable SCOTT to look ahead. Even a crude model carries information. Whether through the false preference strategy, semantic resolution of a combination of the two, this information allows the goal to affect the direction of the search. In cases where this leads to incompleteness[4] or inefficiency, it can be disabled since SCOTT has OTTER as a sub-program, so nothing is lost in the move from OTTER to SCOTT. Moreover, the addition a complex set of tools to OTTER, has opened new possibilities for heuristics on which research may usefully be focussed.

In theorem proving, there are no magic bullets. No one technique gives easy solutions to all problems. Dynamic semantic resolution and the false preference strategy are like most other worthwhile ideas in the field in that they work spectacularly in a few cases and solidly across a fair range, have little or no effect in other cases and sometimes make matters worse. Hence the present paper issues no claim to have solved all the problems. Indeed, in that it opens questions about the effectiveness of semantic resolution relative to problem-specific models it may be taken to have posed some new ones. The significance of so doing depends in part on whether these are *interesting* problems. My feeling, for what it is worth, is that if we could explain both the successes and the failures of SCOTT in its various configurations then we should understand the heuristics of first order theorem proving better than we do. Meanwhile, whether it be seen as a theorem prover of a new type or simply as OTTER with yet another optional add-on, SCOTT is both an intriguing departure and a power tool.

[4]SCOTT's dynamic semantic resolution is obviously complete where the rule of inference is binary resolution, but incomplete in general for hyper-resolution and the like. Its completeness for condensed detachment is an open question.

References

[Ballantyne and Bledsoe, 1982] M. Ballantyne and W. Bledsoe, *On Generating and Using Examples in Proof Discovery,* Machine Intelligence, 10, pp. 3-39.

[Chang and Lee, 1973] C. Chang and R. Lee, Symbolic Logic and Mechanical Theorem Proving, New York, Academic Press.

[Karpenko, 1992] A. Karpenko, *Lattices of implicational logics,* Bulletin of the Section of Logic, 21, pp.82-91.

[Lusk and McCune, 1992] E. Lusk, and W. McCune, *An FJntry in Overbeek's 1992 Theorem Proving Contest,* Journal of Automated Reasoning, forthcoming.

[McCune, 1990] W. McCune, *OTTER 2.0 Users Guide,* Argonne National Laboratory, Argonne, Illinois.

[Slagle, 1967] J. Slagle, *Automatic Theorem Proving with Renamable and Semantic Resolution,* Journal of the ACM, 14, pp. 687 697.

[Slaney, 1992] J. Slaney, *FINDER (Finite Domain Enumerator): Notes and Guide,* Technical Report TR-ARP-1/92, Australian National University Automated Reasoning Project, Canberra.

[Thistlewaite *et al,* 1988] P. Thistlewaite, M. McRobbie and R. Meyer, Automated Theorem Proving in Non-Classical Logics, Pitman, London.