A Comparison of Different Techniques for Grounding Near-Propositional CNF Formulae

Stephan Schulz
Fakultät für Informatik, Technische Universität München, Germany
schulz@informatik.tu-muenchen.de

Abstract

A near-propositional CNF formula is a first-order formula (in clause normal form) with a finite Herbrand universe. For this class of problems, the validity problem can be decided by a combination of grounding and propositional reasoning. However, naive approaches to grounding can generate extremely large ground formulae. We investigate various means to reduce the number of ground instances generated and show that we can increase the number of problems that can be handled with reasonable resources.

Introduction

Automated deduction systems are computer programs that try to prove or, less often, refute, the validity of a logical hypothesis. The most mature theory and implementations of such systems exist for the cases of propositional logic and first order logic. After many years of development, deduction systems for these logics are now increasingly being applied in practice. Theorem provers for first-order logic are e.g. being used for the verification of communication protocols (Weidenbach 1999; Gürdens & Peralta 2000), hard- and software, and the retrieval of mathematical theorems (Dahn & Wernhard 1997) or software components from libraries. Propositional systems are widely applied in circuit verification.

In this paper, we are interested in the first-order fragment of *near-propositional* CNF formulae. A first order formula (in clause normal form) is called near-propositional, if it has a finite Herbrand universe, i.e. if its signature does not contain any non-constant function symbols. This fragment is decidable, the satisfiability problem is equivalent to the satisfiability problem for the Bernays-Schönfinkel class, and is NEXPTIME-complete (Dreben & Goldfarb 1979). It is interesting for a number of reasons:

- Propositional multi-modal logic can be translated into near-propositional clausal logic (Hustadt & Schmidt 1997). This logic is the base for various knowledge representation formalisms.
- Near-propositional logic is a superclass of Datalog, which is used as the basis for query languages for relational and deductive data base systems.
- Model finders like MACE (McCune 2000) transform first order formulae into near-propositional form to find models. This application is particularly interesting for verification tasks, where a proof certifies that a certain requirement is met, but a counter-model corresponds to the less desirable, but more frequent case of a bug.
- Finally, many applications naturally generate near-propositional formulae. Release 2.4.1 of the TPTP problem library for theorem provers (Sutcliffe & Suttner 1998) contains 533 near-propositional formulae, i.e. approximately 10% of the library is from this class. Of particular interest are approximately 100 problems from group theory, and about 30 problems generated by DORIS (Bos 2001), a tool for natural language understanding.

Near-propositional proof problems can be attacked using two different approaches. First, we can use standard first-order techniques. Some current first order calculi provide a decision procedure for the class of near-propositional proof problems. On the down side, the overhead of first order calculi is significant, and practical performance often is not very good. Moreover, for any given calculus, it is not necessarily obvious if it decides this class.

As an alternative, we can combine a grounding procedure and a propositional prover. Since the Herbrand universe for near-propositional formulae is finite, we can enumerate all ground instances of the formula. The resulting ground problem can be solved by a propositional prover, e.g. a DPLL-procedure (Davis, Logemann, & Loveland 1962) as implemented in SATO (Zhang & Stickel 2000).

Herbrand's theorem ensures that this approach is sufficient to show the decidability of the near-propositional fragment. However, in practice, the number of ground instances can be to large to make this feasible. In this paper we investigate different ways of generating smaller ground formulae without changing the satisfiability of the problem. The resulting grounding procedure was used in the deduction systems E-SETHEO (Stenz & Wolf 2000) and PizEAndSATO, which won the first two places in the near-propositional category of the 2001 CASC-JC theorem proving competition (Sutcliffe 2001; Sutcliffe, Suttner, & Pelletier 2002).

Preliminaries

Assume a finite set P of *predicate symbols* with associated arities. We write p/n if p has arity n. Assume further a finite, non-empty set C of *constants* and an enumerable set V of *variables*. A *(near-propositional) atom* a is a word $p(t_1, \ldots, t_n)$, where $p/n \in P$ and all $t_i \in C \cup V$. We write $a|_i$ to denote the subterm t_i in a. A *(near-propositional) literal* is either an atom a (a *positive literal*) or a negated atom $\neg a$ (a *negative literal*). We call a and $\neg a$ *complementary* literals. If l is a literal, we denote by $s(l)$ the sign of l, $+$ for positive literals, and $-$ for negative literals. We define $\neg + = -$ and vice versa.

A *(near-propositional) clause* is a finite set of literals. Finally, a *(near-propositional) formula* (in clause normal form) is a finite set of clauses. If an atom, literal, clause or formulae does not contain any variables, we call it *ground* or, in this context, *propositional*. We use standard notation in writing a clause as the disjunction $l_1 \vee l_2 \ldots \vee l_n$ of its literals.

A *substitution* is a mapping $\sigma : X \to C$ and is continued to atoms, literals, and clauses. If c is a clause, we call $\sigma(c)$ an instance of c, if $\sigma(c)$ is ground, we call it a ground instance and σ a *grounding* substitution (for c). The *Herbrand universe* for a near-propositional formula is identical to the set C of constants[1]. The *Herbrand base* is the set of all possible ground atoms, and a *Herbrand interpretation I* is a subset of the Herbrand base. A ground atom a is called *true under I*, $I(a) = \top$ if $a \in I$. We write $I(a) = \bot$ to denote that a is not true (or *false*) under I. A positive ground literal is true iff it is true as an atom, a negative literal, iff its atom is not true. A ground clause is true under I iff at least one of its literals is true, a non-ground clause is true, iff all of its ground instances are, and a formula is true, iff all of its clauses are. A formula F is called *satisfiable*, if there is an interpretation I with $I(F) = \top$ (in this case we call I a model of F), otherwise it is *unsatisfiable*.

Refined Grounding Techniques

According to Herbrand's theorem, a formula F (in clause normal form) is unsatisfiable exactly if there is a set of ground instances of clauses from F that is unsatisfiable. If there is such a set, then the set of *all* ground instances is certainly unsatisfiable. However, our aim is to find smaller sets if this is possible.

Let us consider an example: $P = \{p/1, q/2\}$, $V = \{x, y, z, \ldots\}$ and $C = \{a, b, c, d\}$. Then the following clause set is unsatisfiable:

- $c_1 = \neg p(x) \vee q(x, y)$
- $c_2 = \neg q(x, a) \vee \neg q(a, y)$
- $c_3 = p(a), c_4 = p(b), c_5 = \neg p(c), c_6 = \neg p(d)$

An example set of ground instances to show the unsatisfiability is $\neg p(a) \vee q(a, a), p(a), \neg q(a, a)$ (where the last clause results from the fact that $q(a, a) \vee q(a, a)$ collapses). This set has only three clauses. The Herbrand universe has a size of four, and the Herbrand base has a size of 20. Naive instantiation of the clause set generates 36 clauses (16 each for the two non-unit clauses c_1 and c_2, plus the 4 existing ground units), some of which occur more than once.

In general, if the size of the Herbrand universe is $|C|$, and a clause c has n different variables, there are $n^{|C|}$ different ground instances of c, since each variable can be instantiated to all constants. In order to reduce the number of generated instances, we can either try to reduce the number of variables per clause, or we can constraint the possible instantiations for each variable. We achieve the first by splitting clauses with variable disjoint parts, and the second by using structural constraints on the literals to induce constraints on the variables.

Actual generation of the ground clauses dominates CPU usage in grounding. Hence the removal or simplification of clauses after generation is less useful then reducing the number of generated clauses in the first place. However, some simplification techniques can be implemented very cheaply, and since memory consumption usually is the limiting factor, any technique that reduces the final number of clauses is worth investigating.

It is worth noting that all techniques presented here are compatible with each other, and can be combined without loss of completeness or correctness.

Clause Splitting

Some clauses contain independent parts, i.e. instantiations in one part of the clause do not influence other parts. An example of such a clause is clause c_2 from above. Since the two literals do not share any variables, we can split the clause into clauses $c_{2'} = \neg q(x, a) \vee t$ and $c_{2''} = \neg q(a, y) \vee \neg t$ (where $t/0$ is a new predicate symbol) without changing the satisfiability of the clause set. However, instead of one clause with 16 ground instances, we now have two clauses with 4 instances each. Formally, we can state the (binary) splitting rule as follows:

$$\frac{l_1 \vee \ldots \vee l_n \vee l_{n+1} \vee \ldots \vee l_{n+m}}{l_1 \vee \ldots \vee l_n \vee t \qquad l_{n+1} \vee \ldots \vee l_{n+m} \vee \neg t}$$

if $l_1, \ldots, l_n$ and $l_{n+1}, \ldots, l_{n+m}$ do not share any variables, at least one literal from each of the two subsets is non-ground, and where $t/0$ is a new predicate symbol that does not occur in the formula.

Application of the rule *replaces* the clause in the precondition with the clauses in the conclusion if the condition holds. It is easy to see that the conclusion implies the precondition (using a single resolution inference). Hence, unsatisfiability is preserved under application of the rule. To prove that satisfiability is preserved, assume a model I of $F \cup \{l_1 \vee \ldots \vee l_n \vee l_{n+1} \vee \ldots \vee l_{n+m}\}$. This model necessary satisfies $l_1 \vee \ldots \vee l_n$ or $l_{n+1} \vee \ldots \vee l_{n+m}$: Assume that there exists grounding substitutions σ and τ with $I(\sigma(l_1 \vee \ldots \vee l_n \vee l_{n+1})) = \bot$ and $I(\tau(l_1 \vee \ldots \vee l_n \vee l_{n+1})) = \bot$. Since both parts are variable disjoint, $\sigma(l_1 \vee \ldots \vee l_n \vee l_{n+1}) \vee \tau(l_1 \vee \ldots \vee l_n \vee l_{n+1}))$ is a ground instance of $l_1 \vee \ldots \vee l_n \vee l_{n+1} \vee \ldots \vee l_{n+m}$ that evaluates to false under I. This contradicts the assumption that I is a model. Now assume (without loss of generality) that

[1] Remember that we require the existence of at least one constant.

$I(l_{n+1} \vee \ldots \vee l_{n+m}) = \top$. Then $I \cup \{t\}$ is a model of $F \cup \{l_1 \vee \ldots \vee l_n \vee t, l_{n+1} \vee \ldots \vee l_{n+m} \vee \neg t\}$.

Note that we can repeatedly apply clause splitting if a clause has more than two variable disjoint parts. This is equivalent to a single application of the *hyper-splitting* rule (Riazanov & Voronkov 2001). In practice, we prefer hyper-splitting, because it can be implemented more efficiently. The restriction to clauses that split into two non-ground parts in the inference rules ensures termination of the splitting process.

Experiences with clause splitting for the full first-order case are mixed (op.cit.). As the results below show, for our application of grounding, whenever it has any significant effect, it is a positive one.

Structural constraints

Clause splitting is a local approach and considers only one clause at a time. Taking the whole formula into account, we can often determine constraints on the possible instantiations of variables. The underlying idea is to avoid the creation of clauses with *pure* literals (Davis & Putnam 1960). A literal l in a given clause is pure in a formula, if there exists no literal l' in another clause so that $\sigma(l)$ and $\sigma(l')$ are complementary for some substitution σ. In the propositional case, a literal is pure if there is no complementary literal in another clause. A clause is pure, if one of its literals is. Since we can freely chose the interpretation of a pure literal, pure clauses do not influence the satisfiability of a formula, and can be ignored or deleted. We will now find sufficient conditions for the purity of literals, and use these as constraints on the grounding substitutions.

Consider again our example formula. The predicate symbol p occurs only in positive literals in the clauses c_3 and c_4. Thus any instance of c_1 that does not instantiate the variable x to either a or b is pure. We call this restriction a *structural constraint* on the instantiation of $p|_1$.

Formally, an *elementary structural constraint* is a formula $p|_i \in D$, where $p/n \in P$, $D \subseteq C$, and $i \in [1 \ldots n]$. Semantically, an elementary structural constraint $p|_i \in D$ allows the instantiation of a variable at position i in literals with predicate symbol p with constants from D. An *elementary variable constraint* is a formula $x \in D$, where $x \in V$ and $D \subseteq C$. A *variable constraint* is a conjunction of elementary variable constraints. Note that variable constraints constrain variables independently of the positions they occur in.

Now let F be a formula, let p/n be a predicate symbol occurring in F, and $L^+(F,p)$ and $L^-(F,p)$ be the set of all positive and negative literals with predicate symbol p in F, respectively. For $\diamond \in \{+,-\}$, we define the structural constraint $S^\diamond(F,p,i) = p|_i \in \{l|_i \mid l \in L^{\neg\diamond}(F,p)\}$ if $\{l|_i \mid l \in L^{\neg\diamond}(F,p)\}$ does not contain any variable $x \in V$, otherwise $S^\diamond(F,p,i) = p|_i \in C$.

The structural constraints induced by a formula can already be used to reduce the number of instantiations considered. However, we can transform local structural constraints into variable constraints, and thus propagate them through a clause. Let $c = a_1 \vee \ldots \vee a_m$ be a clause, and let p_i/n_i be the predicate symbol of a_i. Then the structural variable constraint of c is $VC(F,c) = \wedge_{j\in[1,m]}(\wedge\{S^{s(a_j)}(F,p_j,i)|i \in [1\ldots n_i], p_i \in V\})$. Informally, $VC(F,c)$ is the conjunction of of all structural constraints induced on all variables anywhere in the clause.

It easy to see that we only need to consider instances of c that satisfy $VC(F,c)$, because any instance that violates the structural variable constraint violates at least one elementary structural constraint, and hence would create a pure literal. Note also that structural constraints efficiently approximate the *hyper-linking* condition (Lee & Plaisted 1992). While our constraints allow more instantiations than hyper-linking, they can be computed in linear time based on the size of the formula, and without any use of backtracking.

Structural constraints and clause splitting complement each other: If variables occur in multiple literals, and hence make splitting impossible, they are more likely to be constrained in at least one position.

Conventional contraction techniques

Clause splitting works on the first order-level, while structural constraints are applied during the grounding stage. We will now describe techniques that can be applied after grounding. Since most propositional provers already have very efficient implementations of simplification, we have used only techniques that can be implemented very cheaply in our framework. The three techniques currently used are *tautology deletion*, *unit subsumption* and *unit simplification*.

Tautological clauses are clauses that contain two complementary literals. These clauses always evaluate to true, and can hence be discarded without affecting the satisfiability of a formula. For the propositional case with a finite number of atoms, we can detect tautologies in $O(n)$ operations, where n is the number of literals. However, this requires either a good hashing algorithm, or a constant overhead in the order of $O(|B|)$, where B is the Herbrand base of the problem. Since clauses are typically short, while B is large, we use a conventional $O(n^2)$ algorithm which traverses the clauses in a triangular fashion, comparing each pair of literals once.

Unit subsumption and unit resolution are two forms of unit propagation already suggested in (Davis & Putnam 1960). Unit subsumption allows us to delete a clause $l \vee l_1 \vee, \ldots, \vee l_n$, if we already have a unit clause l. The reason is that any interpretation that makes l true also makes the larger clause true. Unit resolution is the complementary part to unit subsumption. It allows us to replace a clause $l \vee l_1 \vee \ldots \vee l_n$ with $l_1 \vee \ldots \vee l_n$ if we have a unit clause with a literal that is complementary to l. If we only perform forward subsumption and resolution, we can implement both techniques in $O(n)$ for each clause of length n, and still simplify or delete most generated clauses. To achieve this, we sort the original (non-ground) clause set by length. When we start grounding with the shortest clauses, most propositional unit clauses will be generated early. We assign a unique positive integer to each generated ground atom[2]. Generated unit clauses are then represented simply by an entry in an

[2]This is necessary anyway if we want to generate the DIMACS format used by most propositional provers.

array, with 1 at position n representing a positive unit clause with propositional literal n, 2 representing a negative unit, and 3 representing the presence of both positive and negative units[3]. This allows us to check in constant time for any given literal if it or its complement appears as a unit clause.

Experimental Results

We have implemented all techniques described in this paper in the program `eground` as a part of the distribution of our equational theorem prover E (Schulz 2001), and evaluated them on the set of all 533 near-propositional CNF formula from the TPTP 2.4.1 library for theorem provers (Sutcliffe & Suttner 1998). The experiments were conducted in compliance with the guidelines for use of the TPTP. Input files were unchanged except for expansion of *include* directives. The program used and the full results are available from `http://wwwjessen.informatik.tu-muenchen.de/~schulz/WORK/grounding.html`. We present only selected results here.

We conducted two sets of experiments, one with a memory limit of 128 MB, and one with a larger limit of 512 MB. The CPU time limit was 300 seconds for both sets of experiments (the first conducted on a Sun Ultra 10/300 workstation, the second on a SunBlade 1000/750). This limit was sufficient that no single experiment run out of time, all failures are caused by running out of memory.

The strategies are as follows: *Naive* uses no refinements at all, *Conventional* uses only contraction techniques, *Constraints* uses structural constraints, *Splitting* uses clause splitting, and *Combined* uses all three approaches.

Table 1 gives the number of problems successfully grounded by each strategy. We found that all generated ground problems are very simple for SATO, and can usually be solved within at most a few seconds.

Memory Limit	128 MB	512 MB
Naive	403	461
Conventional	407	461
Constraints	418	472
Splitting	479	496
Combined	491	504

Table 1: Successes on TPTP problems

All techniques increase the number of problems we handle within reasonable resource bounds. Although splitting is the most successful individual technique by far, the combination of techniques is significantly stronger than the best individual technique. We can also see that increasing the memory limit is particularly useful for the weaker strategies.

Our experiments show that the success of a technique is correlated with the type of the problem. TPTP domains are loose collections of problems with a common theme. Most groundable problems come from the domains GRP (group theory), NLP (natural language processing), PUZ (logical puzzles) and SYN (problems not directly modeling any application problem, most of these problems are translated modal logic formulae). Table 2 shows the results for the 128 MB limit by domain.

Domain	GRP	NLP	PUZ	SYN	Other
Size	101	30	27	360	15
Naive	101	10	20	260	12
Conventional	101	10	20	264	12
Constraints	101	20	21	264	12
Splitting	101	12	21	333	12
Combined	101	22	23	333	12

Table 2: Successes on TPTP domains (128 MB limit)

As we can see, the different techniques show different performance depending on problem domain. First, all GRP problems currently in TPTP are trivial for `eground`. Even naive grounding can instantiate all of these problems. NLP problems profit most from structural constraints, while SYN problems nearly exclusively benefit from clause splitting. If we use the larger memory limit, most of the differences vanish, as both splitting and constraints are sufficient to make more problems solvable.

Table 3 shows the data of some experiments in detail. From each of the four big TPTP domains we chose the hardest example (by run time) still successfully grounded by the naive strategy. We report CPU time (on the SunBlade 1000/750) and size of the resulting formula by number of clauses and literals. Times do not include writing of the result to disk.

Problem	Strategy	Time	Clauses	Literals
GRP124-3.005	Naive	2.380	114394	570256
	Conv.	1.950	38930	165077
	Constr.	1.500	72703	355737
	Split.	2.410	114394	570256
	Comb.	1.150	17507	64987
NLP116-1	Naive	17.000	200034	3600066
	Conv.	17.420	200034	3560066
	Constr.	0.020	66	642
	Split.	16.700	200036	3600070
	Comb.	0.010	68	614
PUZ018-1	Naive	2.570	110199	624655
	Conv.	2.440	93428	510846
	Constr.	0.550	24275	133871
	Split.	2.550	110199	624655
	Comb.	0.510	19769	100306
SYN439-1	Naive	24.160	677768	6581022
	Conv.	25.800	645862	6288515
	Constr.	23.890	677768	6581022
	Split.	0.100	4546	21929
	Comb.	0.130	4546	21929

Table 3: Comparison on individual problems

These results further support the conclusions from table 2. For GRP124-3.005, conventional contraction and constraints both decrease the size of the formula. However, the combination of both reduces overall size (measured in literals) by nearly a full order of magnitude. In NLP116-1,

[3] Note that in this case we can terminate immediately and write out the empty clause, since the overall formula is unsatisfiable.

structural constraints are the absolute "killer" technique, reducing the number of generated ground instances from more than 200 000 to 66! The puzzle problem is quite similar to GRP124-3.005, although the overall benefit of the refinements is somewhat less. Finally, in SYN439-1 (a translated propositional multi-modal K logic formula), splitting is the only technique that gives a significant benefit. However, this benefit is dramatic, reducing the literal count by a factor of 300.

It should be noted that our implementation of grounding is far from optimal. Since we reuse the core libraries from our equational theorem prover E, most data structures, in particularly literal and clause representations, are far more general than necessary for this task. We estimate that a specialized implementation could easily reduce the memory consumed by 75%, and also lead to some speed-ups.

Conclusion

The recent CASC-JC competition has shown that refined grounding techniques, combined with a propositional prover, are a viable method for dealing with many near-propositional problems. The program described in this paper was used by the top two contenders in the EPR (near propositional) division of the competition, E-SETHEO (where it was combined with both a propositional and various first-order provers), and PizEAndSATO (where is was coupled with SATO).

In this paper, we have explained the different techniques that contribute to this overall success. We can observe both the case that a single technique is crucial for a given problem, but more frequently, that the different techniques complement each other.

There still are significant possibilities to improve our results in the future. It is possible to split clauses which do not have two variable disjoint parts by introducing *link literals* which contain variables. Alternatively, we can interleave instantiation and splitting. This is helpful if partial instantiation of a clause results in two variable disjoint parts. The pruning effect of structural constraints can also be improved. First, we can consider only pairs of literals that unify with each other to induce constraints. Secondly, we can also interleave instantiation and constraint generation, and thus propagate constraints between clauses. Finally, many techniques from the field of propositional provers can be integrated directly into the grounding procedure.

Acknowledgments: I would like to thank Geoff Sutcliffe for giving me the motivation to implement an efficient grounding procedure and for building PizEAndSATO on top of it.

References

Bos, J. 2001. The DORIS Project Web Site. `http://www.coli.uni-sb.de/~bos/doris/`.

Dahn, B., and Wernhard, C. 1997. First Order Proof Problems Extracted from an Article in the MIZAR Mathematical Library. In *Proceedings of the 1st FTP, Linz*, 58–62. RISC Linz, Austria.

Davis, M., and Putnam, H. 1960. A Computing Procedure for Quantification Theory. *Journal of the ACM* 7(1):215–215.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A Machine Program for Theorem Proving. *Communications of the ACM* 5:394–397.

Dreben, B., and Goldfarb, W. 1979. *The Decision Problem: Solvable Classes of Quantificational Formulas*. Addison-Wesley.

Gürdens, S., and Peralta, R. 2000. Validation of Cryptographic Protocols by Efficient Automatic Testing. In Etheredge, J., and Manaris, B., eds., *Proc. of the 13th FLAIRS, Orlando*, 7–12. AAAI Press.

Hustadt, U., and Schmidt, R. 1997. On Evaluating Decision Procedures for Modal Logics. In Pollack, M., ed., *Proc. of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, volume 1, 202–207. Morgan Kaufmann.

Lee, S.-J., and Plaisted, D. 1992. Eliminating Dupliction with the Hyper-Linking Strategy. *Journal of Automated Reasoning* 9(1):25–42.

McCune, W. 2000. *MACE 2.0 Reference Manual and Guide*. Argonne National Laboratory, Argonne, USA. (available at `http://www-unix.mcs.anl.gov/AR/mace/`).

Riazanov, A., and Voronkov, A. 2001. Splitting without Backtracking. In Nebel, B., ed., *Proc. of the 17th International Joint Conference on Artificial Intelligence (IJCAI-2001), Seattle*, volume 1, 611–617. Morgan Kaufmann.

Schulz, S. 2001. System Abstract: E 0.61. In Goré, R.; Leitsch, A.; and Nipkow, T., eds., *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, 370–375. Springer.

Stenz, G., and Wolf, A. 2000. E-SETHEO: An Automated[3] Theorem Prover – System Abstract. In Dyckhoff, R., ed., *Proc. of the TABLEAUX'2000*, volume 1847 of *LNAI*, 436–440. Springer.

Sutcliffe, G., and Suttner, C. 1998. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning* 21(2):177–203.

Sutcliffe, G.; Suttner, C.; and Pelletier, J. 2002. The IJCAR ATP System Competition. *Journal of Automated Reasoning*.

Sutcliffe, G. 2001. The CASC-JC Web Site. `http://www.cs.miami.edu/~tptp/CASC/JC/`.

Weidenbach, C. 1999. Toward an Automatic Analysis of Security Protocols in First-Order Logic. In Ganzinger, H., ed., *Proc. of the 16th CADE, Trento*, volume 1632 of *LNAI*, 314–328. Springer.

Zhang, H., and Stickel, M. 2000. Implementing the Davis-Putnam Method. *Journal of Automated Reasoning* 24(1/2):277–296.