Encoding First Order Proofs in SAT

Todd Deshane, Wenjin Hu, Patty Jablonski, Hai Lin, Christopher Lynch, and Ralph Eric McGregor

Clarkson University

Abstract. We present a method for proving rigid first order theorems by encoding them as propositional satisfiability problems. We encode the existence of a first order connection tableau and the satisfiability of unification constraints. Then the first order theorem is rigidly unsatisfiable if and only if the encoding is propositionally satisfiable. We have implemented this method in our theorem prover CHEWTPTP, and present experimental results. This method can be useful for general first order problems, by continually adding more instances of each clause.

1 Introduction

There are two strands of thought in first order theorem proving today. One line of research is to design general theorem provers which address all of first order logic. The second line is to design general purpose algorithms for decidable problems and combine them together. In this paper, we attempt to design an efficient algorithm for the specialized rigid theorem proving problem, which can be used as an end in itself, or can be used incrementally to address all of first order logic.

In standard refutational theorem proving problems we attempt to prove the unsatisfiability of a set of clauses, and allow an unbounded number of renamed instances of each clause. In rigid theorem proving, only one instance of each clause is allowed. Rigid theorem proving has been studied as early as [20, 1], and is used in some tableau style theorem provers[12]. In this case, the rigid theorem proving problem is used as a means of solving the general theorem proving problem. But, as we argue in [8] the rigid theorem proving problem is useful in itself for modeling behavior that only occurs a bounded number of times, such as cryptographic protocols with a bounded number of sessions.

The most impressive success recently in theorem proving has been the efficiency of SAT solving methods based on the DPLL method[7]. The success of these methods seems to be based on the fact that the search space is defined in advance. This means that an exponential number of possibilities can be explored in polynomial space. It also means that the data structures can be predetermined in such a way that the algorithm can access everything efficiently. In this paper, we try to take advantage of such techniques in first order theorem proving. The obvious idea is to try to incorporate a SAT solver, and also to use it in such a way that it is not called often, because calling it too many times loses the advantages mentioned above.

In our method, we encode a proof of first order unsatisfiability with propositional clauses. Obviously, we cannot encode full first order unsatisfiability, since that is an undecidable problem. So we chose to encode rigid first order satisfiability. In order to do this, we need to decide what kind of proof should be encoded. One possibility would be to encode a resolution proof, as has been done in [16], although in that paper propositional proofs were encoded and not first order proofs. We chose instead to encode a connection tableau proof. The reason we chose this method is because the number of clauses that are encoded remains fixed for connection tableau whereas resolution proofs introduce clauses not contained in the original set. For that reason, we think a SAT solver will be more directed in the encoding of a connection tableau proof, as opposed to a resolution proof.

Given a propositional encoding of a rigid connection tableau proof (which we describe in this paper), the encoding is sent to a SAT solver (we used MiniSat[9]) which will return RIGIDLY SATISFIABLE if a rigid connection tableau exits, and RIGIDLY UNSATISFIABLE if a closed rigid connection tableau does not exist. If satisfiable, we can recover the tableau from the truth assignment returned by the SAT solver.

The idea of the encoding is the following. We encode the existence of a clause as the root of the connection tableau. We encode the fact that every literal assigned to a non-leaf node is extended with a clause containing a complementary literal. Those things are easy to encode, and do not take up much space. There are three things which are more costly to encode.

First, we must encode the fact that two literals are complementary, in other words that their corresponding atoms are unifiable. For that, we basically have to encode a unification algorithm. In our encoding of unification, we leave out the occurs check, because it is expensive, and because it rarely occurs. We add a check for this after the SAT solver returns the truth assignment. If there really is an occurs check, we add a propositional clause and call MiniSat again.

Second, the above encoding leaves open the possibility that the connection tableau is infinite. Therefore, we must encode the fact that the connection tableau is finite, i.e., that the connection tableau contains no cycle.

Third, we must encode the fact that every literal assigned to a leaf node is closed by a previous literal. Our encoding is simpler for the Horn case, because it is only necessary to close a literal with the previous one on the branch. For the non-Horn case, we must encode the fact that there is a complementary literal higher up in the tree. Since the same clause may occur on two different branches, and a literal on that clause may close with different literals on different branches, we may need to add more than one copy of a clause in the rigid non-Horn case, because of the fact that the literal is closed differently. But we still try to get as much structure sharing in our tree as possible. Note that rigid Horn clause satisfiability is NP-complete, but Rigid non-Horn clause satisfiability is Σ_2^p-complete[11]. So it is not surprising that a SAT solver cannot solve rigid non-Horn clause satisfiability directly.

The original contributions in this paper are to define an encoding of closed rigid connection tableau proofs as a SAT problem; provide rigid-horn, rigid/non-horn, and non-rigid algorithms and proofs of their completeness and soundness; and discuss our implementation along with initial experimental results.

2 Preliminaries

2.1 First Order Logic

We use standard notation to represent classical first order logic formula. Our alphabet consists of variables, constants, functions symbols, predicates, the quantifiers $\forall$ (universal) and $\exists$ (existential), logical connectives $\vee$ (disjunction),$\wedge$ (conjunction), $\neg$ (negation) and parentheses. Terms are defined inductively as follows. Variables and constants are terms. If f is a function with arity n and $t_1..t_n$ are terms then $f(t_1, ..., t_n)$ is a term. Atoms are of the form $P(t_1, ...t_n)$ where P is a predicate of arity n and $t_1, ..., t_n$ are zero or more terms. A literal is defined as a positive or negative atom and a clause is a disjunction of literals. We consider a formula to be constructed from elements in the alphabet according to the standard rules for constructing formula. See [4] for a detailed description of first order logic and a background discussion on the validity of a first order logic formula.

We consider a literal to be ground if it contains no variables. And we define a Horn clause as a clause which contains at most one positive literal. A clause which contains only negative literals is called a negative clause.

A formula is in conjunctive normal form (CNF) if it is a conjunction of literals such that negations are applied only to atoms and all variables are universally quantified.

A substitution σ for F is a map from the set of variables of F to a set of terms. We can view a substitution as a set of equations $x = t$ where x is a variable and t is a term. We call a single equation of a substitution σ an *assignment of* σ. Given a substitution σ, an application of σ on a formula F is denoted $F\sigma$. We say that a substitutions, σ, is a unifier of formulas G and H if $G\sigma = H\sigma$. If σ unifies G and H and for every unifier α of G and H there exists some θ such that $\sigma\theta = \alpha$ then we call σ a most general unifier (mgu) of G and H. A ground instance of G is an instance of G in which all variables are replaced by ground terms.

2.2 Propositional Logic

The alphabet for propositional logic formula consists of propositional variables and the logical connectives $\vee$ (disjunction), $\wedge$ (conjunction), $\neg$ (negation) and parentheses. As with first order logic, we will consider propositional logic formulas to be in CNF and conform to the standard rules for constructing valid propositional logic formulas[4].

2.3 Connection Tableaux

We define rigid clausal tableau as follows.

Definition 1. *Rigid clausal tableaux are trees with nodes labeled with literals, branches labeled either open or closed, and edges labeled with zero or more assignments. Rigid clausal tableaux are inductively defined as follows. Let $S = \{C_1 ... C_n\}$ be a set of clauses. If T is a tree consisting of a single unlabeled node (the root node) N then T is a rigid clausal tableau for S. The branch consisting of only the root node N is open. If N is a leaf node on an open branch B in the rigid clausal tableaux T for S and one of the following inference rules are applied to T then the resulting tree is a rigid clausal tableaux for S.*

(Expansion rule) Let $C_k = L_{k1} \vee ... \vee L_{ki}$ be a clause in S. Construct a new tableaux T' by adding i nodes to N and labeling them L_{k1} through L_{ki}. Label each of the i branches open.

(Closure rule) Suppose L_{ij} is the literal at N and for some predecessor node M with literal L_{pq} there exists some most general unifier σ such that $L_{ij}\sigma = \neg L_{pq}\sigma$ and the assignments of σ are consistent with the assignments of T. Construct T' from T by labeling the edge from L_{pq} to L_{ij} with the assignments used in the unification and by closing the branch of N.

We call the clause which is added to the root node the start clause and we say that a clause is *in* a tableau if the clause was used in an application of the expansion rule.

Definition 2. *A clausal tableaux is connected if each clause (except the start clause) in the tableaux contains some literal which is unifiable with the negation of its predecessor [13].*

Connection tableaux use an additional macro inference rule called the extension rule.

Definition 3. *(Extension Rule) Let N be a node in the tableau T and let C_k be a clause in S such that there exists a literal L_{ki} in C_k which is unifiable with the negation of N. Apply the expansion rule with C_k and immediately apply the closure rule with L_{ki}.*

The calculus for connection tableaux (or model elimination tableau [13]) therefore consists of the expansion rule (for the start clause only), the closure rule, and the extension rule. We call a tableau closed if each leaf node has been closed by an application of the closure rule.

By [13] we can require that the start clause be a negative clause since there exists a negative clause in any minimally satisfiable set.

2.4 Rigid Unsatisfiability

Unless otherwise stated, we let F be a set of first order logic formulas. The main problem in Automated Theorem Proving is to determine if the set of hypotheses in F implies the conclusion in F. For our purposes we assume that all formula in a problem are in CNF and the conclusion is negated. Therefore we seek to determine if F is (equivalently) unsatisfiable, i.e. there does not exist a model for F. The problem of rigid unsatisfiability of F seeks to determine whether there exists a ground instance of F which is unsatisfiable.

A result of Tableau Theory is the completeness and soundness of closed (rigid) connection tableaux.

Theorem 1. *There exists a closed (rigid) connection tableau for F iff F is (rigidly) unsatisfiable[12].*

3 Tableau Encoding

Our method to determine the rigid satisfiability of F generates a set of propositional logic clauses for F which encodes a closed rigid connection tableau for F. We provide two encoding, the first for problems which contain only Horn clauses and the second for those containing non-Horn clauses. Given F, we give unique symbols to each of the clauses in F and each of the literals in each clause. We represent clause i by C_i. We represent the j^{th} literal in clause i by L_{ij} (which is used to label the tableaux). Note that as multiple copies of a clause may appear in a rigid connection tableau, multiple nodes may have the same literal label. And whereas the same literal may appear in distinct clauses, they are identified with different labels. We denote A_{ij} to be the atom of L_{ij}. Therefore L_{ij} is either of the form A_{ij} or $\neg A_{ij}$.

3.1 Encoding for Horn Clauses

We define the variables c_m, l_{mn}, e_{mnq}, u_k, p_{mq} as follows: Define $c_m = T$ iff C_m appears in the tableau. Define $l_{mn} = T$ iff L_{mn} is an internal node in the tableau. Define $e_{mnq} = T$ iff C_q is an extension of L_{mn}. Define $u_\tau = T$ iff τ is an assignment implied by the substitutions used in the closure rules. Define $p_{mq} = T$ iff there exists a path from a literal in C_m to C_q.

Below we list the set of clauses that we generate and provide their meaning.

At least one clause containing only negative literals appears in the tableau:

$$\bigvee_{C_m \text{ is a negative clause}} c_m \tag{1}$$

If C_m appears in the tableau and L_{mn} is a negative literal then L_{mn} is an internal node in the tableau:

$$\neg c_m \vee l_{mn} \tag{2}$$

If L_{mn} is an internal node in the tableau then for some q_j, C_{q_j} is an extension of L_{mn}:

$$\neg l_{mn} \vee e_{mnq_1} \vee \ldots \vee e_{mnq_k} \tag{3}$$

where $\{C_{q_1}...C_{q_k}\}$ represents the set of all clauses whose positive literals are unifiable with L_{mn}

If C_q is an extension of L_{mn} then C_q exists in the tableau:

$$\neg e_{mnq} \vee c_q \tag{4}$$

If C_q is an extension of L_{mn} and τ is an assignment of the mgu used to unify A_{qr} with A_{mn} then τ is implied by the mgu:

$$\neg e_{mnq} \vee u_\tau \text{ where } \tau \in mgu(A_{mn}, A_{qr}) \tag{5}$$

If for two assignments $x = s$ and $x = t$ there does not exist a mgu θ such that $s\theta = t\theta$ then both assignments can not be true:

$$\neg u_{x=s} \vee \neg u_{x=t} \text{ where } s \text{ and } t \text{ are not unifiable} \tag{6}$$

If $x = s$, $x = t$, $\sigma = mgu(s,t)$ and $y = r \in \sigma$ then $y = r$:

$$\neg u_{x=s} \vee \neg u_{x=t} \vee u_{y=r} \text{ where } y = r \in mgu(s,t) \tag{7}$$

If C_q is an extension of L_{mn} then there is a path from C_m to C_q:

$$\neg e_{mnq} \vee p_{mq} \tag{8}$$

Transitivity of the path relation:

$$\neg p_{mq} \vee \neg p_{qs} \vee p_{ms} \tag{9}$$

There are no cycles in the tableau:

$$\neg p_{mm} \tag{10}$$

3.2 Encoding for Non-Horn Clauses

For non-Horn problems we use an alternative set of variables and generate a different set of clauses. We say that two literals are *complementary* if they have opposite signs and their atoms are unifiable.

We define the variables s_m, c_{mn}, l_{mn}, e_{mnqj}, u_k, p_{mq}, and q_{mnij} as follows. Define $s_m = T$ iff C_m is the start clause. Define $c_{mn} = T$ iff C_m appears in the tableau and L_{mn} is used to close its parent. Define $l_{mn} = T$ iff L_{mn} is a node in the tableau and is not a leaf node created by an application of the closure rule. Define $e_{mnqj} = T$ iff C_q is an extension of L_{mn} and L_{mn} is used to close L_{qi}. Define $u_\tau = T$ iff τ is an assignment implied by the unifiers used in the applications of the closure rules. Define $o_{ijkl} = T$ iff L_{kl} is used to close L_{ij}. Define $p_{mq} = T$ iff there exists a path from a literal in C_m to C_q. Define $q_{mnij} = T$ iff L_{mn} is a leaf and L_{mn} is closed by a literal between the root node and L_{ij}.

The clauses are as follows.

There exists a start clause in the tableau which only contains negative literals:

$$\bigvee_{s_m \text{ is a negative clause}} s_m \tag{11}$$

If C_m is the start clause in the tableau then each literal L_{mn} of C_m is in the tableau:

$$\neg s_m \vee l_{mn} \tag{12}$$

If C_i appears in the tableau and L_{ij} is the complement of a literal in its parent then all other literals of C_i are in the tableau:

$$\neg c_{ij} \vee l_{ik} \text{ where } j \neq k \tag{13}$$

If L_{ij} exists in the tableau and is not a leaf node created by an application of the closure rule then either L_{ij} is closed by a literal between the root and L_{ij} or there is an extension of L_{ij}:

$$\neg l_{ij} \vee q_{ijij} \bigvee_{k,l} e_{ijkl} \tag{14}$$

If L_{ij} is extended with C_k then C_k is in the tableau and some L_{kl} of C_k is closed by L_{ij}:

$$\neg e_{ijkl} \vee c_{kl} \tag{15}$$

If clause C_m is an extension of L_{ij} and τ is an assignment of the mgu used to unify A_{ml} with A_{ij} then τ is true:

$$\neg e_{ijml} \vee u_\tau \text{ where } \tau \in mgu(A_{ml}, A_{ij}) \tag{16}$$

If for two assignments $x = s$ and $x = t$ there does not exist a mgu θ such that $s\theta = t\theta$ then both assignments can not be true:

$$\neg u_{x=s} \vee \neg u_{x=t} \text{ where } s \text{ and } t \text{ are not unifiable} \tag{17}$$

If $x = s$, $x = t$, $\sigma = mgu(s,t)$ and $y = r \in \sigma$ then $y = r$:

$$\neg u_{x=s} \vee \neg u_{x=t} \vee u_{y=r} \text{ where } y = r \in mgu(s,t) \tag{18}$$

If L_{ij} is used to close L_{kl} then their atoms must be unifiable by some unifier σ, hence each assignment of σ is true:

$$\neg o_{ijkl} \vee u_\tau \text{ where } \tau \in mgu(A_{ij}, A_{kl}) \tag{19}$$

If L_{ij} has the same sign as L_{kl} or their respective atoms are not unifiable then L_{ij} is not used to close L_{kl}:

$$\neg o_{ijkl} \text{ where } L_{ij} \text{ and } L_{kl} \text{ have the same sign or } A_{ij} \text{ and } A_{kl} \text{ are not unifiable} \tag{20}$$

If leaf L_{ij} is closed by a literal between the root and L_{kl} and clause C_k is an extension of L_{mn} then L_{ij} is closed by some literal between the root and L_{mn}:

$$\neg q_{ijkl} \vee \neg e_{mnkl} \vee o_{ijmn} \vee q_{ijmn} \tag{21}$$

If C_k is an extension of L_{ij} then there is a path from clause C_i to clause C_k:

$$\neg e_{ijkl} \vee p_{ik} \tag{22}$$

Transitivity for paths:

$$\neg p_{ij} \vee \neg p_{jk} \vee p_{ik} \tag{23}$$

There are no cycles in the tableau:

$$\neg p_{ii} \tag{24}$$

If C_i is the start clause then there are no extensions into any of the literals in C_i:

$$\neg s_i \vee \neg e_{klij} \tag{25}$$

If C_i is the start clause and L_{mn} is a leaf which is closed by a literal between the root node and L_{ij}, then L_{mn} must be closed with L_{ij}:

$$\neg s_i \vee \neg q_{mnij} \vee o_{mnij} \tag{26}$$

4 Tableau Encoding Algorithm(TE)

We provide three algorithms, each with subtle differences. The first algorithm HTE attempts to find a rigid proof and takes as an argument a problem containing only Horn clauses. The second, $NHTE$, also attempts to find a rigid proof and takes as an argument a non-Horn problem. The last algorithm, $NRTE$, seeks to finds a non-rigid proof and takes either a Horn or non-Horn problem as an argument.

The rigid algorithm for non-Horn problems may require additional copies of the clauses in F in order to generate a proof for F and the non-rigid algorithm may also require additional instances of clauses. In the case of the former, copies of clauses in F are added to the set of problem clauses. The number of copies required can be bounded by k^n where n is the number of clauses in F and k is the maximum number of literals in any clause in F. In the case of the non-rigid algorithm, new instances of clauses in F which are standardized apart are added to the problem clauses.

Each algorithm initially enters a while loop. While in the loop the set of clauses S, which encode the closed rigid connection tableau, is given to an external SAT solver. The SAT solver returns satisfiable or unsatisfiable and if the set of clauses is satisfiable, the SAT solver returns a model M. If the SAT solver returns satisfiable we check if the assignments which are assigned true in M are constent. If not, we add additional clauses to S to resolve these inconsistencies and call the SAT solver again. If the algorithm determines that S is satisfiable and the assignments which are assigned true are consistent, the algorithm returns an indication that F is rigidly unsatisfiable.

The function Unify-Substitutions takes as an argument the model M generated by the SAT solver and generates additional clauses to rectify inconsistencies in the assigments used in the proof. The only inconsistency that can occur among assignments is due to cycles. For example, $\{x_1 = f(x_2), x_2 = f(x_3), x_3 = f(x_1)\}$. If a cycle is found, a clause is created which prevents the conflict. These clauses are added to the original set of clauses generated by the algorithm which are again checked by the SAT solver.

Algorithm 1: Rigid Algorithm For Horn Problems(HTE)
Input: F, a set of FO formula in conjunctive normal form.
Output: RIGIDLY UNSATISFIABLE or RIGIDLY SATISFIABLE

```
HTE($F$)
(1)     Generate $S$, the set of encodings for $F$
(2)     $S^{'} = \emptyset$
(3)     while true
(4)         SAT-Solver($S \cup S^{'}$)
(5)         if SAT-Solver returns SATISFIABLE and and the as-
            signments set true in $M$ are consistent
(6)             return (RIGIDLY UNSATISFIABLE)
(7)         else if SAT-Solver returns SATISFIABLE
(8)             $S^{'}$ = Unify-Substitutions($M$)
(9)         else
(10)            return (RIGIDLY SATISFIABLE)
```

Algorithm 2: Rigid Algorithm For Non-Horn Problems(NHTE)
Input: F, a multi-set of FO formula in conjunctive normal form.
Output: RIGIDLY UNSATISFIABLE

```
NHTE($F$)
(1)     $F^{'} = F$
(2)     $S^{'} = \emptyset$
(3)     while true
(4)         generate $S$, the set of encodings for $F^{'}$
(5)         $M$ = SAT-Solver($S \cup S^{'}$)
(6)         if SAT-Solver returns SATISFIABLE and the assignments set true in $M$
            are consistent
(7)             return (RIGIDLY UNSATISFIABLE)
(8)         else if SAT-Solver returns SATISFIABLE
(9)             $S^{'}$ = Unify-Substitutions($M$)
(10)        else
(11)            $F^{'} = F^{'} \cup F$
```

Algorithm 3: Non-Rigid Algorithm(NRTE)

Input: F, a set of FO formula in conjunctive normal form.
Output: UNSATISFIABLE

```
NRTE($F$)
(1)   $F^{'} = F$
(2)   $S^{'} = \emptyset$
(3)   while 1
(4)       generate $S$, the set of encodings for $F^{'}$
(5)       $M$ = SAT-Solver($S \cup S^{'}$)
(6)       if SAT-Solver returns SATISFIABLE and the assignments set true in $M$
          are consistent
(7)           return (UNSATISFIABLE)
(8)       else if SAT-Solver returns SATISFIABLE
(9)           $S^{'}$ = Unify-Substitutions($M$)
(10)      else
(11)          generate set of new instances, $A$, of $F$ using variables not occurring in
              $F^{'}$
(12)          $F^{'} = F^{'} \cup A$
```

5 Completeness and Soundness Theorems for HTE

In the following proofs we refer to the sets of clauses generated by HTE by the enumeration given in Section 3.1.

Theorem 2. *(Completeness) Let F be a set of first order logic Horn clauses. If F is rigidly unsatisfiable, then HTE will return RIGIDLY UNSATISFIABLE.*

Proof. Assume F is rigidly unsatisfiable and let S be the set of clauses for F generated by HTE. As F is rigidly unsatisfiable then by Theorem 1 there exists a closed rigid connection tableaux T. It also follows that the start node of T contains only negative literals. From T we will construct a map from the variables in S to {true, false} so that S is satisfiable.

If C_m appears in the tableau set $c_m = true$ otherwise set $c_m = false$. If L_{mn} is an internal node in the tableau set $l_{mn} = true$ otherwise set $l_{mn} = false$. If C_q is an extension of L_{mn} set $e_{mnq} = true$ otherwise set $e_{mnq} = false$. If τ is an assignment implied by the unifiers used applications of the closure rule set $u_\tau = true$ otherwise set $u_\tau = false$ and if there exists a path from C_m to C_q set $p_{mq} = true$ otherwise set $p_{mq} = false$.

As T has a start node containing only negative literals, there exists a variable in Set 1 which is true. Thus Set 1 of S is satisfiable.

As T is a connection tableau and each extension of T closes the branch containing the positive literal of a clause, and since each clause contains at most one positive literal, then each negative literal in T is an interal node. Hence each variable representing a clause in T is true iff its negative literal variables are also true. Thus Set 2 is satisfiable.

Since each negative literal in T must be extended it follows that each variable representing a negative literal in T is true iff the variable representing its extension is true. Therefore Set 3 is satisfiable. Furthermore since each extension of T extends a literal to all the literals in a clause, an extension variable is true iff the clause variable associated with the extension is true. Thus Set 4 is satisfiable.

Since each extension in T unifies complementary literals, it follows that an extension variable is true iff each of the variables representing the assignments in the unifier used in the unification of the complementary literals are true. Hence Set 5 is satisfiable. It also follows by the consistency of T that inconsistent assignments can not both be true, thus for each pair of variables representing inconsistent assignments we have one is true iff the other is false. Hence Set 6 is satisfiable. In addition if two assignments map the same variable to unifiable terms s and t then the assignments used in the unification of s and t must be true. Therefore Set 7 is satisfiable.

Now as there exists paths between literals and clauses via extensions in T, if a variable representing an extension is true then the variable representing the path is true. Thus Set 8 is satisfiable. And since the paths in T have a transitive relation and no cycles exist in T, Sets 9 and 10 are satisfiable respectively.

Therefore since each of the sets of clauses in S are satisfiable, then the SAT solver called in HTE returns a satisfiable model with consistent assignments, hence HTE returns RIGIDLY UNSATISFIABLE. $\square$

Theorem 3. *(Soundness) If HTE on F returns RIGIDLY UNSATISFIABLE then F is rigidly unsatisfiable.*

Our proof of soundness uses the satisfiability map produced by HTE to construct a tableau for F.

Proof. Suppose HTE on F returns RIGIDLY UNSATISFIABLE. Then there exists a set of clauses S generated by HTE and a model M for which S is satisfiable. Furthermore the set of assignment variables that are true in M correspond to a consistent set of assignments. We construct a closed rigid connection tableau T for F using M and S as follows.

Since S is satisfiable the clause $C = c_1 \vee ... \vee c_n$ in Set 1 of S, is satisfiable. Since C is satisfiable at least one of the variables in C are assigned true. Let c_m where $m \in [1..n]$ be a variable of C such that $c_m = true$. We begin constructing T by setting C_m as the start clause of T. [1]

Now as $c_m = true$ and Set 2 is satisfiable, each of the variables corresponding to the literals in C_m are true. Thus for each literal L_{mn} in C_m we create a node directly off the root and label it L_{mn}.

Let L_{mn} be a literal in C_m. Now as l_{mn} is true and Set 3 is satisfiable there exists some variable e_{mnq_i} which is true and as Set 4 is satisfiable $e_{mnq_i} = true$ implies $c_{q_i} = true$. We therefore expand the node labeled L_{mn} in T with clause C_{q_i}. We continue this process until all literal, clause, and extention variables

[1] It may be the case that more than one variable of C is assigned true. This corresponds to the fact that there may be more than one closed rigid connection tableau for F.

which are assigned true have been addressed. By the satisfiability of Sets $2-4$, T is closed.

Now let e_{mnq_i} be a variable in M which is set to true. Since Set 5 is satisfiable, e_{mnq_i} implies that a set of assignments are true. We label the edge from L_{mn} to the positive literal in C_{q_i} with these assignments. Since each extension unifies adjacent complementary literals and the assignments in M are consistent, T is connected and consistent.

The satisfiability of Sets $8-10$ ensure that there are no cycles in T, hence T is a tree. It follows then that T is a closed connection tableau. Since each clause in T is in F, T is a closed rigid connection tableau for F. Thus by the soundness theorem for closed rigid connection tableaux, F is rigidly unsatisfiable. $\square$

6 Completeness and Soundness Theorems for NHTE

Here we provide the completeness theorem of NHTE which takes as input non-Horn problems. In the proofs, we refer to the sets of clauses generated by NHTE by the enumeration given in Section 3.2.

Theorem 4. *(Completeness) Let F be a set of first order clauses. If F is rigidly unsatisfiable, then NHTE will return RIGIDLY UNSATISFIABLE.*

Proof. Assume F is rigidly unsatisfiable and let S be the set of clauses for F generated by NHTE. By Theorem 1, as F is unsatisfiable, there exists a closed rigid connection tableaux T for F. It also follows that the start node of T contains only negative literals. Let S be the set of clauses generated by NHTE. Given T we will construct a map from the variables in S to {true, false} so that S is satisfiable.

Set $s_m = T$ iff C_m is the start clause. Set $c_{mn} = T$ iff C_m appears in the tableau and L_{mn} is closed by an application of the extension rule. Set $l_{mn} = T$ iff L_{mn} is a node in the tableau but is not closed by an application of the extension rule. Set $e_{mnqj} = T$ iff C_q is an extension of L_{mn} and L_{qj} closes L_{mn}. Set $u_\tau = T$ iff τ is a assignment implied by substitutions used in the closure rules. Set $o_{ijkl} = T$ iff L_{kl} is used to close L_{ij} but not during an application of the expansion rule. Set $p_{mq} = T$ iff there exists a path from a literal in C_m to C_q. Set $q_{mnij} = T$ iff L_{mn} is a leaf and is closed by a literal between the root node and L_{ij}.

As T has a start node containing only negative literals, there exists a variable in Set 11 which is true, thus Set 11 of S is satisfiable. Since each of the literals in the start clause are in T and are not closed by an application of the expansion rule then their respective variables are true, therefore Set 12 is satisfiable.

Now as each clause in T (except for the start clause) is the result of an expansion rule, and only one literal in each clause is closed in the process of using the expansion rule, all the other literals are in the tableau but are not closed by an application of the expansion rule. Hence Set 13 of S is satisfiable.

Suppose L_{ij} is in T such that L_{ij} is not closed by an application of the expansion rule. Then either L_{ij} is extended or L_{ij} has been closed by a complementary literal on its path. It follows that Set 14 is satisfiable.

Since each extension in T adds a clause to T, Set 15 is satisfiable. Since each extension in T unifies complementary literals, it follows that an extension variable is true iff each of the variables representing the assignments in the unifier used in the unification of the complementary literals are true. Hence Set 16 is satisfiable. It also follows by the consistency of T that inconsistent assignments cannot both be true, thus for each pair of variables representing inconsistent assignments, one is true iff the other is false. Hence Set 17 is satisfiable. In addition if two assignments map the same variable to unifiable terms s and t then the assignments used in the unification of s and t must be true. Therefore Set 18 is satisfiable.

As each pair of literals which are used in a non-extension closure are complements, if a variable representing the non-extension closure between two literals is true then the variables representing the assignments implied by unification of their atoms are true. Hence Set 19 is satisfiable. Since no two literals with have the same sign or which have atoms that are not unifiable cannot be used in a non-extension closure, Set 20 is satisfiable.

Suppose L_{ij} is a leaf and is closed by a literal between the root and L_{kl}. If the clause containing L_{kl} is an extension of some node L_{mn} then either L_{mn} is a complement of L_{ij} or L_{ij} is closed by a literal between the root node and L_{mn}. It follows that Set 21 is satisfiable.

Now as there exists paths between literals and clauses via extensions in T, if a variable representing an extension is true then the variable representing the path is true. Thus Set 22 is satisfiable. And since the paths in T have a transitive relation and no cycles exist in T, Sets 23 and 24 are satisfiable respectively.

As the start clause has no expansions into it, Set 25 is satisfiable. And since if a leaf, say L_{ij} in T is closed by a non-extension closure by a literal between the root and L_{mn} of the start clause, since there are not literals between the root and the literals of the start clause, then L_{ij} must be closed by L_{mn}. Hence Set 26 is satisfiable.

Therefore as each of the sets of clauses in S are satisfiable, then the SAT solver called in NHTE returns SATISFIABLE. It follows that as T is a tableau the assignments implied by the closure rule are consistent. Hence, NHTE returns RIGIDLY UNSATISFIABLE.

Theorem 5. *(Soundness) If NHTE on F returns RIGIDLY UNSATISFIABLE then F is rigidly unsatisfiable.*

7 CHEWTPTP Implementation

We have implemented our tableau encoding method in a command line program written in C++ called CHEWTPTP. The default options assume the input file is in TPTP CNF format[18]. By default the program assumes the input problem is non-Horn and uses the non-Horn algorithm with one instance of the clauses in the input file. The user may specify alternate settings by including the following flags. The flag -h indicates the problem is Horn, -r specifies the user wishes the program to run one of the rigid algorithms, -i allows the user to input the number of instances of the problem to use, and -p instructs the program to print a proof. Other options are provided to control input and output.

The program initially parses the input file and constructs a data structure to hold the clauses in memory. The program then constructs the sets of clauses defined in section 3. While generating the clauses, a data structure is kept which maps each variable to a unique integer. We use the integers to format the clauses in a MiniSat[9] readable format. CHEWTPTP then forks a process and invokes MiniSat on the set of generated clauses and MiniSat determines the satisfiability of the set. When MiniSat returns, we inspect the file output by MiniSat. If the file contains an indication of satisfiability we check that the substitutions are unifiable and if so, we use the model provided by MiniSat to construct a proof. If MiniSat returns back an indication of unsatisfiable, the program returns SATISFIABLE in the rigid Horn case, and may add additional clauses and repeat the process in the other cases.

Preliminary results on 1365 Horn and non-Horn CNF problems without equality in the TPTP Library show that 221 of them have rigid proofs requiring a single instance. We have found that CHEWTPTP was able to solve some problems which many theorem provers could not within a 600 second time limit, e.g. the non-Horn problems ANA003-4.p and ANA004-4.p. And although we have not tested the library extensively by adding additional instances, CHEWTPTP was successful solving non-rigid problems that others were unable, e.g. ANA003-2 was proved with 2 instances in less than 5 seconds.

Below are some statistics on the problems mentioned above and a few other problems. The first column identifies the name of the problem in the TPTP library and the second column identifies whether or not the problem is Horn. The third column identifies the number of instances that were required to prove the problem. The fourth column gives the number of seconds CHEWTPTP took to generate the tableau encoding(s) and the fifth column gives the total time (in seconds) that MiniSat ran on the problem. The sixth and last columns give the number of clauses and variables respectively that were input to MiniSat when MiniSat returned SATISFIABLE.

Table 1. Statistics on Selected Problems

Name	Horn	Instances	Clause Gen (sec)	MiniSat (sec)	Clauses	Variables
ALG002-1	N	2	1.2	65.93	411020	13844
ANA003-2	Y	2	.1	4.88	183821	7238
ANA003-4	N	1	1.1	.06	34774	2616
ANA004-4	N	1	1.61	.3	44142	3160
COL121-2	Y	1	1.35	.16	47725	2322
GRP029-2	Y	1	.08	1.41	241272	7943
PUZ031-1	N	1	.24	.71	662145	14672

8 Conclusion

This is not the first paper to suggest an encoding of a proof in propositional logic. [16] has explored the idea of encoding a propositional resolution proof itself in propositional logic. Our emphasis is different from that paper. That paper is interested in using local search methods to find a proof. Since they are only considering propositional proofs, they do not encode unification. In our case, we encode tableau proofs, because we suspect that SAT solver will be able to direct its search better in that case.

We can also compare our recent work with recent papers on instantiation-based theorem proving, which either try to use a SAT solver to create a first order model[6], or else try to use DPLL-like methods directly on first order clauses [3, 5, 14, 19, 2]. Those are completely different approaches. They do not try to encode the proof. They try to find a model instead. We argue that the benefit of our approach is that the SAT solver is called rarely. Based on the reasons we understand for the success of SAT solvers, we think this is a big advantage of our method.

In our implementation, we sometimes generate large files for MiniSat to solve. Minisat usually solves them very quickly. Our implementation is still preliminary, but we think it shows promise, given that it can solve some problems quickly that many other theorem provers cannot solve. We do not have a good handle yet on which problems our method does better on. Obviously, it will perform best on problems that do not need many instances of the clauses. From our results, it appears that more than 15% of the problems without equality in the TPTP library are rigidly unsatisfiable, with only one instance of each clause.

For future work, there are several things that need to be done. We need to make our implementation more efficient. There are also several useful extensions. We need to find a good way to represent equality. We need to find a good way to decide exactly which clauses should be copied. We would like a method to decide satisfiability from rigid satisfiability. It would be useful to have an encoding of rigid clauses modulo a non-rigid theory, as discussed in [8]. This way, we could immediately identify some clauses as non-rigid, and work modulo those clauses.

Finally, the most interesting idea to improve the efficiency is to replace the SAT solver by a SAT solver modulo theories. Crude analysis of the input files to the SAT solver shows that for the Horn case less than 1% of the clauses generated are to determine the structure of the tableau whereas nearly 70% are to encode the unification. Instead of encoding the unification problem, we could work modulo a background unification theory. Besides unification, the other things that generate a lot of clauses are the encoding of no cycle in the tree (as much as 30%), and the existence of a complementary literal previously in the tree (as much as 70% in the non-Horn case). They are both somehow concerned with finding cycles. A unification algorithm could also be employed here if it had an efficient occurs check. The existence of a cycle can be succinctly encoded as a unification problem. Unification has a deterministic algorithm. Since this would remove the bulk of our propositional clauses and replace them by a deterministic algorithm, we expect it would improve the efficiency a lot.

References

1. Andrews P. B. [1981], Theorem Proving via General Matings, Journal of the Association for Computing Machinery, Vol. 28, No. 2, pp.193-214
2. Baumgartner P. [2000], FDPLL - A First-Order Davis-Putnam-Logeman-Loveland Procedure, In *CADE-17*, Springer, LNAI 1831, pp. 200-219
3. Baumgartner P. and Tinelli C. [2003], The Model Evolution Calculus, In *CADE-19*, Springer, LNAI 2741, pp. 350-364
4. Bell J.L. and Slomson A.B. [1969], Models and Ultraproducts, An Introduction, Dover
5. Billon J. [1996], The Disconnection Method: a Confluent Integration of Unifications in the Analytic Framework, In *Tableaux 1996*, Springer, LNAI 1071, pp. 110-126
6. Chandru V., Hooker J., Rago G. and Shrivastava A. [2002], Partial Instantiation Methods For Inference in First-Order Logic, Journal of Automated Reasoning, 28, pp. 371-396
7. Davis M., Logemann D. and Loveland D. [1962], A Machine Program For Theorem Proving, Communications of the ACM, Volume 5 , Issue 7, pp. 394-397
8. Delaune S., Lin H. and Lynch C. [2007], Protocol Verification Via Rigid/Flexible Resolution, submitted
9. E*é*n N. and S*ö*rensson N. [2003], An Extensible Sat-Solver, In *SAT*, pp. 502-518
10. Ganzinger H., Hagen G., Nieuwenhuis R., Oliveras A. and Tinelli C. [2004], DPLL(T): Fast Decision Procedures, 16th International Conference on Computer Aided Verification (CAV), Boston (USA)
11. Goubault J. [1994], The Complexity of Resource-Bounded First-Order Classical Logic, Lecture Notes In Computer Science, Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science, Vol. 775, Springer-Verlag, pp. 59-70
12. H*ä*hnle R. [2001], Tableaux and Related Methods, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. 1, Elsevier Science, chapter 3, pp. 101-177
13. Letz R. and Stenz G. [2001], Model Elimination and Connection Tableau Procedures, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. 2, Elsevier Science, chapter 28, pp. 2015-2113
14. Letz R. and Gernot S. [2001], Proof and Model Generation With Disconnection Tableaux, In *LPAR 2004*, LNCS 3079, Springer, pp. 289-306
15. McCune W. [1994], A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems, Technical Report ANL/MCS-TM-194, Argonne National Laboratory
16. Prestwich S. and Lynce I. [2006], Local Search for Unsatisfiability, 9th International Conference on Theory and Applications of Satisfiability Testing, Springer, Aug. 2006 , pp. 283-296
17. Riazanov A.,Voronkov A. [2002], The design and implementation of VAMPIRE, AI Communications, CASC, Vol. 15 , Issue 2-3 , pp. 91-110
18. Sutcliffe G. and Suttner C.B. [1998], The TPTP Problem Library: CNF Release v1.2.1, Journal of Automated Reasoning, Vol. 21, No. 2, pp. 177-203
19. Stenz G. [2002], The Disconnection Tableaux, PhD thesis, TU M*ü*nchen, 2002
20. Chang, C. and Lee, C.R. [1973], Symbolic Logic and Mechanical Theorem Proving. Academic Press New York and London.