Abstract

In this paper, we describe the design of a parallel theorem prover for first order logic. The parallel theorem algorithm is based on the divide-and-conquer strategy. The concept of restricted substitution is used to reduce the number of the ground clauses generated during the operation of this theorem prover. In this manner, the ground clause set generated by the theorem prover will be much smaller than that generated directly by Herbrand universe.

1 Introduction

The solution to the satisfiability (SAT) problem is the basis of most automatic reasoning mechanisms, and refutation is a major approach of theorem proving. Under this approach, the premises and negated conclusion are translated into a clause set in conjunctive normal form. Thus, testing the unsatisfiability of the clause set determines whether these premises and conclusion form a theorem or not. We formalize the concept of theorem proving in the following form:

Given a logic statement \(S\):
Premses \(P_1, P_2, \ldots, P_k\)
Conclusion \(C\)

\(S\) is a theorem if and only if \(P_1 \land P_2 \land P_k \land \ldots \land \neg C\) is unsatisfiable.

There have been many strategies to determine the unsatisfiability of a clause set in first order logic, for example, Davis-Putnam procedure [3], Robinson’s resolution principle [7], and Kowalski’s connection graph proof procedure [4], [5]. The major drawbacks of these strategies lie in creating too many clauses to be processed, and particularly having limited inherent parallelism. In present work, we make use of a divide-and-conquer proof procedure [2] for testing the unsatisfiability of a clause set. The basic principle of divide-and-conquer is to divide the problem into many independent subproblems which can be solved simultaneously. Accordingly, theorem prover will have high degree of parallelism.

The original version of this proof procedure is designed for propositional logic. Since the ground literals can be considered as propositional symbols, the proposed proof procedure is capable of processing the ground clauses in the same way as it does for the propositional clauses. Nevertheless, the proof procedure may be intractable. In previous work [3], the ground clauses are generated directly by Herbrand universe. The set of these clauses is usually too large to be handled by practical theorem proving environment. In order to overcome the intractability, we construct the restricted substitution for each variables to reduce the number of ground clauses. Consequently, the theorem prover based on this approach is more efficient.

In this paper, section 2 describes the divide-and-conquer strategy for theorem proving and gives an algorithm which is sound and complete. In section 3, we discuss the procedure to prove theorems in first order logic. The proof procedure resembles Davis-Putnam procedure with an improvement in the generating of ground clauses. A suitable architecture for such proof procedure is also discussed. Section 4 shows some examples to illustrate the behavior of the theorem prover and discuss its performance. In section 5, we give the conclusions and future works.

2 Divide-and-Conquer Strategy for Parallel Theorem Proving

The divide-and-conquer strategy can be applied to test the unsatisfiability of a clause set [2]. The basic idea of the divide-and-conquer strategy is to
split the original problem into two smaller subproblems which are independent and can be recursively solved. Hence, these independent subproblems can be solved simultaneously. Thus, we make use of the property of parallelism to speed up the processing time.

Let \( S \) be a propositional clause set, and \( P \) be an arbitrary predicate symbol in \( S \). We first define the following terminologies.

Definition 1. \( S_p \) is the set of clauses containing \( P \) in \( S \).

Definition 2. \( S_{\neg P} \) is the set of clauses containing \( \neg P \) in \( S \).

Definition 3. \( S_{xp} \) is the set of clauses containing neither \( P \) nor \( \neg P \) in \( S \).

Definition 4. \( S_{p'} \) is the set \( S_p \) with all occurrences of \( P \) deleted.

Definition 5. \( S_{\neg p'} \) is the set \( S_{\neg p} \) with all occurrences of \( \neg P \) deleted.

Definition 6. \( S(P) \) is the union of \( S_p \) and \( S_{xp} \).

Definition 7. \( S(-P) \) is the union of \( S_{\neg p} \) and \( S_{xp} \).

Example 1. Given a clause set \( S = \{ \neg P \lor R, P \lor \neg Q, \neg R, Q \} \). By previous definitions, we obtain \( S_p, S_{\neg p}, S_{xp}, S_P, S_{xp}, S(P), \) and \( S(\neg P) \) as follows:

\[
\begin{align*}
S_p &= \{ P \lor \neg Q \} \\
S_{\neg p} &= \{ \neg P \lor R \} \\
S_{xp} &= \{ \neg R, Q \} \\
S_P &= \{ \neg Q \} \\
S_{xp} &= \{ R \} \\
S(P) &= \{ \neg Q, \neg R, Q \} \\
S(\neg P) &= \{ R, \neg R, Q \}
\end{align*}
\]

Furthermore, the following theorem [2] supports our use of the divide-and-conquer strategy for theorem proving.

Theorem 1. Let \( S \) be a clause set. \( S \) is unsatisfiable if and only if both \( S(P) \) and \( S(\neg P) \) are unsatisfiable for an arbitrary proposition symbol \( P \) in \( S \).

Based on the theorem described above, a parallel algorithm is proposed to check whether a propositional clause set is unsatisfiable or not [2].

Algorithm: Check the unsatisfiability of a propositional clause set \( S \).
Input: The propositional clause set \( S \).
Output: Return YES if \( S \) is unsatisfiable, i.e. the empty clause is deduced from \( S \). Return NO, otherwise.

1. Check the clauses in \( S \). If \( S \) is an empty set, then return NO. If there is an empty clause in \( S \), then return YES.
2. Select a symbol \( P \) in \( S \) and construct the sets \( S(P) \) and \( S(\neg P) \).
3. Perform the following procedures in parallel:
   3.1 Apply this algorithm with set \( S(P) \).
   3.2 Apply this algorithm with set \( S(\neg P) \).
4. Compare the two values returned from step 3. If both of them are YES, then return YES; Otherwise, return NO.

This theorem proving procedure (algorithm) for propositional logic has been proved to be logically sound and complete [2]. The following examples demonstrate the divide-and-conquer strategy.

Example 2. Given clause set \( S = \{ \neg P \lor R, P \lor \neg Q, \neg Q \} \).

See the proof tree of \( S \) in Figure 1. \( S \) is split to two sets \( S(P) \) and \( S(\neg P) \). Next, take an arbitrary symbol, say \( Q \), from \( S(P) \) to generate \( S(P)(Q) \) and \( S(P)(\neg Q) \). Similarly, take a symbol, say \( R \), from \( S(\neg P) \) to generate \( S(\neg P)(R) \) and \( S(\neg P)(\neg R) \). There are empty clauses occurring in all leaf nodes, \( S(P)(Q), S(P)(\neg Q), S(\neg P)(R), \) and \( S(\neg P)(\neg R) \), of the proof tree. Therefore, the algorithm with \( S \) as its input should return YES, and \( S \) is unsatisfiable.

Every node in the proof tree described in the above example represents a SAT problem to be solved. The solving of them can be overlapped, and each path from the root to any leaf forms a processing pipeline. When the proof procedure starts up, only one clause is fed into the proof tree from the root node for every processing cycle until there is no clause left. Hence, a parallel theorem proving model with the pipeline property is constructed [2].

3 Parallel Proof Procedure for First Order Logic

The basic concept of our theorem proving model for first order logic is based on the Davis-Putnam procedure with some modifications. Two stages, generating of ground clauses and testing unsatisfiability of clauses, are performed in parallel. Moreover, the ground clause set generated by our generating procedure is much smaller than that generated by Davis-Putnam procedure and the
testing procedure is replaced by the algorithm proposed above. The mechanism of generating process is described in this section. The parallel theorem prover is divided into two stages ground clause generator and proof tree for testing unsatisfiability of a set of ground clause. The ground clause generator generates ground clauses which can be tested by the proof tree in parallel. The procedure continues until the unsatisfiability has been assured or all the ground clauses have been tested.

3.1 Generating of Ground Clauses

Given a clause set, we can construct the Herbrand universe $H$ of $S$ [1]. $H$ may be finite or even infinite. Many terms in $H$ are absolutely unnecessary for testing unsatisfiability. The goal is to find out the potentially useful ground terms which should be substituted to the variables of a clause, so as to generate ground clauses that are essential to the proof procedure. Our approach is to use resolution principle as an aid to judge what substitution is potentially useful. The procedure is described as follows:

Input: A formula in first order logic. The formula includes two parts: axioms and theorems.
Output: Ground clauses for proof tree.

1. Extract all literals in the theorems. For each literal, if it is unifiable with any axioms, record the substitutions and substitute the variables. The newly generated clauses and the clauses in the original formula are put into the template.

2. Extract all literals in the template except those used in step 1. Put these literals into the seed set.

3. Choose a literal from the seed set with ground literals having the highest priority. If no ground literals exist, choose a literal with least number of variables.

4. Test the unifiability of the literals with all clauses in the template. If they are unifiable, substitute the terms in the clauses with the substitutions (not to generate any resolvents).

5. If new ground clauses are generated, feed them into the proof tree. These ground clauses are further recovered to its original form with variables renamed.

6. If the proof tree returns unsatisfiable, then exit.

7. Discard the literal selected in step 3, and extract newly generated literals in step 4 into the seed set.

8. Go to step 3.

During the generating of ground clauses, two sets are maintained, namely variable clause template and seed set. The variable clause template (or template in short) is the set of clauses containing at least one variable, and the seed set is the collection of all literals which are given initially or generated during the run time.

At first, the ground clauses in the given formula is tested whether they are unsatisfiable or not. If the unsatisfiability can not be determined, we make use of the template to generate new ground clauses for subsequent testing. Initially, all literals in the formula are put into the seed set and all nonground clauses are put into the template. In each iteration of the generating procedure, one literal is chosen from the seed set. The literal is used to examine if it is unifiable with each clause in the template. The examination is to find some substitutions rather than generate resolvents. The substitution can be viewed as a reasonable constraints to define the scope of a variable. In general, these substitutions will substitute variables to ground terms and make clauses one step closer to be ground clauses. After some iterations, if there are any ground clauses generated, they can be fed into the proof tree immediately.

The seed set is initialized with the collection of all literals which occur in the given clause set. In each iteration of the ground clause generating procedure, we select a literal in the seed set. This selected literal is used to test the unifiability with all clauses in the template. The purpose here is to find some ground terms for some variables. If the selected literal itself contains only variables, it is rather difficult to generate any ground clauses. Thus, it will be most effective to choose the literal with only ground terms.

In general, more than one ground clauses will be generated in each iteration. These new ground clauses will be fed into the proof tree to test the unsatisfiability. On the other hand, many of the literals in the template become ground literals by unification, and these ground literals are added into the seed set as the new seeds which can be selected for subsequent generation. The following examples demonstrate the generating procedure.
Example 3. Given clause set $S = \{ P(X) \lor \neg Q(Y,Z), \neg P(a), \neg P(b), \neg P(c), Q(c,d) \}$. The ground clause set $G = \{ \neg P(a), \neg P(b), \neg P(c), Q(c,d) \}$. The variable clause template $V = \{ P(X) \lor \neg Q(Y,Z) \}$. The seed set $S = \{ \neg P(a), \neg P(b), \neg P(c), Q(c,d) \}$.

Because the given ground clauses $\neg P(a), \neg P(b), \neg P(c),$ and $Q(c,d)$ are not unsatisfiable, the theorem prover must generate some new ground clauses by using the template $P(X) \lor \neg Q(Y,Z)$. First, we take $\neg P(a), \neg P(b),$ and $\neg P(c)$ as the unification seeds, and there is no ground clause generated. However, variable $X$ is bound to the ground terms $a, b,$ and $c$ potentially, and 3 literals $P(a), P(b),$ and $P(c)$ are added into the seed set as the new seeds. Later, $Q(c,d)$ is selected as the unification seed, the variable $Y,$ and $Z$ are bound to $a,$ and $d$ respectively. There are 3 new ground clauses $P(a) \lor \neg Q(c,d), P(b) \lor \neg Q(c,d),$ and $P(c) \lor \neg Q(c,d)$ generated simultaneously at this time. They should be used to test the unsatisfiability. In this case, the theorem prover will response the answer YES. Figure 2 illustrate the generation of ground clauses for another example.

### 3.2 Comparisons with Other Approaches

Resolution principle and Herbrand’s theorem are the two major approaches in mechanical theorem proving. In resolution, resolvents are generated at each iteration until the empty clause is found. The main issue is to design a strategy to determine which clauses are to be unified and generate the resolvents so as to obtain the empty clause as soon as possible. For Herbrand’s theorem, we try to find a set of ground clauses of the formula and prove its unsatisfiability. The main issue is to find a set of ground clauses as small as possible. Since we have a very efficient procedure for testing the unsatisfiability of a set of ground clauses. Our approach basically follows Herbrand’s theorem and Davis-Putnam procedure. But, in the generating of ground clauses, resolution is utilized to limit the scope that a variable can be bound. A clause will not be sent into the proof tree until it becomes ground. It usually takes many resolutions to make a clause become ground. In such case, many intermediate resolvents are generated by resolution principle. But in our approach, only one ground clause is generated. Thus, our generating procedure usually generate less number of clauses in most cases.

### 3.3 Proof Tree

Conceptually, we consider each node in the proof tree as a processor. Each processor contains several registers, buffers, and flags, which keep clauses and some housekeeping data. The proof tree grows dynamically, that is, the processor is allocated when it is needed and is deallocated when the branch to it is blocked. The ground clauses are translated to the internal codes before feeding them into the proof tree. By this preprocessing, the processors can work efficiently, and the internal structures of them are regular such that the implementation is simple. Figure 3 shows the internal architecture of a processor. The processors can be connected as tree machines, but such architecture results in poor utilization of processors. Since, in most cases, the proof tree will not be a complete binary tree, thus there are always many idle processors in the tree. To utilize processors more efficiently, we have proposed a mapping scheme to map dynamically growing trees on the hypercube machines [8].

### 4 Simulations and Performance Analysis

The theorem prover has been simulated on Sun 3 in UNIX C. The prover has been used to prove theorems appeared in [9]. The theorems range from quite trivial to difficult cases including some common theorems in group theory. The simulations in [9] are mainly resolution based, and the number of clauses generated is the major measurement. We consider the number of ground clauses generated as our measurement and compare it to the results of [9] in the following table. In the following table, UR is unit resolution, HYPER is hyper resolution, and PAR is our generating procedure. G1, G2, and G3 are theorems in group theory. HP1, HP2, and HP3 are problems in Henkin models.

<table>
<thead>
<tr>
<th></th>
<th>G1</th>
<th>G2</th>
<th>G3</th>
<th>HP1</th>
<th>HP2</th>
<th>HP3</th>
</tr>
</thead>
<tbody>
<tr>
<td>UR</td>
<td>270</td>
<td>44</td>
<td>173</td>
<td>10</td>
<td>11</td>
<td>54</td>
</tr>
<tr>
<td>HYPER</td>
<td>350</td>
<td>220</td>
<td>371</td>
<td>na</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>PAR</td>
<td>292</td>
<td>198</td>
<td>302</td>
<td>6</td>
<td>6</td>
<td>73</td>
</tr>
</tbody>
</table>

Table 1. The number of ground clauses generated by different approaches.
It is shown that the generating procedure is quite effective. The number of ground clauses is usually less than that of the resolution approach. Since our generating and testing procedure work in parallel, and the testing procedure itself is a tree structure which is fully parallel and pipelined, if we compare the execution time, the parallel version will be more overwhelming.

5 Conclusions

The divide-and-conquer strategy of testing the unsatisfiability for a propositional logic formula is very efficient. If we want to apply this divide-and-conquer strategy to solve the unsatisfiability problem in first order logic, we must generate ground clauses by binding the variables to some ground terms. The generating procedure we proposed will generate fewer ground clauses than those generated by Herbrand universe. Consider the following example:

Example 4. Given clause set $S = \{ P(a), P(b), P(c), \neg P(X) \lor Q(Y), \neg Q(d), \neg Q(e) \}$. The Herbrand universe $H = \{ a, b, c, d, e \}$. The generator in Davis-Putnam procedure will substitute every term in $H$ for the variables $X$ and $Y$, and generate 25 ground clauses. The total number of ground clauses is 30. By unification, variable $X$ can only be substituted by $\{ a, b, c \}$, and variable $Y$ can only be substituted by $\{ d, e \}$. The generating procedure we proposed will generate 6 ground clauses, and the total number is 11.

In most cases, the number of the ground clauses generated by restricted substitution sets will be much less than that generated by Davis-Putnam procedure. The parallel proof tree is very efficient and has time complexity of $O(M*N)$, where $M$ is the number of clauses and $N$ is the number of literals in a clause. Since $N$ is not large in most cases, the time for theorem proving in first order logic is dominated by the number of ground clauses generated. The generating procedure still can not avoid generating redundant ground clauses, and how to generate less or even no redundant ground clauses efficiently is the most important issue for further study.

References


Fig. 1 The proof tree of the clause $S$.

```
~P \lor R
P \lor \sim Q

Q
S(P)

R

\sim R
\sim R

\sim R

\sim R

Q

Q

S(\sim P)
```

**Proof Tree**

**Variable Clause Template**

1. $\sim P(X) \lor Q(f(X))$
2. $\sim Q(Y) \lor P(f(Y))$

**Ground Clauses**

1. $P(a)$
2. $P(f(f(a)))$
3. $\sim P(a)$
4. $Q(f(a))$
5. $\sim Q(f(a))$
6. $P(f(f(a)))$

**Seeds**

Fig. 2 Mechanism of ground clauses generation.

![Diagram](image)

Fig. 3 The architecture of a processor in the proof tree.