Temporal Resolution: A Breadth-First Search Approach*

Clare Dixon
Department of Computing
Manchester Metropolitan University
Manchester M1 5GD
United Kingdom
C.Dixon@doc.mmu.ac.uk

Abstract

An approach to applying clausal resolution, a proof method for classical logics suited to mechanisation, to temporal logics has been developed by Fisher. The method involves translation to a normal form, classical style resolution within states and temporal resolution between states. The method consists of only one temporal resolution rule and is therefore particularly suitable as the basis of an automated temporal resolution theorem prover. As the application of this temporal resolution rule is the most costly part of the method, involving search amongst graphs, it is on this area we focus. A breadth-first search approach to the application of this rule is presented and shown to be correct. Analysis of its operation is carried out and test results for its comparison to a previously developed depth-first style algorithm given.

1 Introduction

Temporal logics have been used extensively for the specification and verification of properties of concurrent systems, see for example [18, 19, 15, 11, 3, 21, 2, 14]. Important computational properties such as liveness, deadlock and mutual exclusion can be expressed easily and simply in temporal logic making it useful for specification. Verifying that a temporal logic specification satisfies a temporal property usually requires some form of theorem proving. *Model checking* approaches have been the most popular, particularly based on tableau [25] or automata [23]. However, standard model checking approaches are limited as only finite state problems can be handled and, even then the number of states required soon becomes large due to the combinatorial explosion. Alternatively one can adopt a resolution based approach [22] which is is not limited to finite state problems and has a significant body of work on heuristics to control search (see standard texts such as [6] for example). Decision procedures based on resolution have been developed for temporal logics in [5, 24, 1], however in many cases they are unsuitable for implementation either because they only deal with a small number of the temporal operators or because of problems with proof direction due to the large numbers of resolution rules that may be applied.

In this paper we present a breadth-first search style algorithm which enables practical implementation of the resolution method for temporal logics developed by Fisher [9]. The resolution procedure is characterised by translation to a normal form, the application of a classical style resolution rule to derive contradictions that occur at the same points in time (termed *step resolution*), together with a new resolution rule, which derives contradictions over time (termed *temporal resolution*).

This paper is structured as follows. In §2, a description of the propositional temporal logic used and the normal form required for the temporal resolution method is given. An outline of this temporal resolution method is given in §3, while the Breadth-First Search algorithm to implement the temporal resolution step is described in §4. The output of the Breadth-First Search algorithm is examined in §5, results comparing it with a previously described algorithm and conclusions are drawn in §6.

2 A linear temporal logic

Here we summarise the syntax and semantics of the logic used and describe the normal form required for the resolution method.

2.1 Syntax and semantics

The logic used in this report is Propositional Temporal Logic (PTL), in which we use a linear, discrete model of time with finite past and infinite future. PTL may be viewed as a classical propositional logic augmented with both future-time and past-time tempo-

*This work was supported partially by an EPSRC PhD Studentship and partially by EPSRC Research Grant GR/K57282

ral operators. Future-time temporal operators include '$\Diamond$' (*sometime in the future*), '$\Box$' (*always in the future*), '$\bigcirc$' (*in the next moment in time*), '$\mathcal{U}$' (*until*), '$\mathcal{W}$' (*unless* or *weak until*), each with a corresponding past-time operator. Since our temporal models assume a finite past, for convenience, two last-time operators are used namely '$\bullet$' (*weak last*) and '$\circledcirc$' (*strong last*). When evaluated at any point other than the beginning of time, both $\circledcirc A$ and $\bullet A$ are **true** if and only if A was **true** at the previous moment. However, for any formula A, $\circledcirc A$ is **false**, when interpreted at the beginning of time, while $\bullet A$ is **true** at that point. Particularly, $\bullet$**false** is only **true** when interpreted at the beginning of time. The weak last-time operator may be defined in terms of the strong last operator as follows

$$\bullet A \equiv \neg \circledcirc \neg A.$$

Models for PTL consist of a sequence of *states*, representing moments in time, i.e.,

$$\sigma = s_0, s_1, s_2, s_3, \ldots$$

Here, each state, s_i, contains those propositions satisfied in the i^{th} moment in time. As formulae in PTL are interpreted at a particular moment, the satisfaction of a formula f is denoted by

$$(\sigma, i) \models f$$

where σ is the model and i is the state index at which the temporal statement is to be interpreted. For any well-formed formula f, model σ and state index i, then either $(\sigma, i) \models f$ or $(\sigma, i) \not\models f$. For example, a proposition symbol, 'p', is satisfied in model σ and at state index i if, and only if, p is one of the propositions in state s_i, i.e.,

$$(\sigma, i) \models p \quad \text{iff} \quad p \in s_i.$$

The semantics of the temporal connectives used in the normal form or the resolution rule are defined as follows

$$\begin{array}{l}
(\sigma,i) \models \bullet A \text{ iff } i = 0 \text{ or } (\sigma, i-1) \models A;\\
(\sigma,i) \models \circledcirc A \text{ iff } i > 0 \text{ and } (\sigma, i-1) \models A;\\
(\sigma,i) \models \Diamond A \text{ iff there exists a } j \geqslant i \text{ s.t. } (\sigma, j) \models A;\\
(\sigma,i) \models \Box A \text{ iff for all } j \geqslant i \text{ then } (\sigma, j) \models A;\\
(\sigma,i) \models A \,\mathcal{U}\, B \text{ iff there exists a } k \geqslant i \text{ s.t. } (\sigma, k) \models B\\
\qquad\qquad \text{and for all } i \leqslant j < k \text{ then } (\sigma, j) \models A;\\
(\sigma,i) \models A \,\mathcal{W}\, B \text{ iff } (\sigma, i) \models A \,\mathcal{U}\, B \text{ or } (\sigma, i) \models \Box A.
\end{array}$$

The full syntax and semantics of PTL will not be presented here, but can be found in [9].

2.2 A normal form for PTL

Formulae in PTL can be transformed to a normal form, Separated Normal Form (SNF), which is the basis of the resolution method used in this paper. SNF was introduced first in [9] and has been extended to first-order temporal logic in [10]. While the translation from an arbitrary temporal formula to SNF will not be described here, we note that such a transformation preserves satisfiability and so any contradiction generated from the formula in SNF implies a contradiction in the original formula. Formulae in SNF are of the general form

$$\Box \bigwedge_i R_i$$

where each R_i is known as a *rule* and must be one of the following forms.

$$\begin{array}{lcll}
\bullet\,\mathbf{false} & \Rightarrow & \bigvee_{b=1}^{r} l_b & \text{(an } \textit{initial } \Box\text{–rule)}\\
\circledcirc \bigwedge_{a=1}^{g} k_a & \Rightarrow & \bigvee_{b=1}^{r} l_b & \text{(a } \textit{global } \Box\text{–rule)}\\
\bullet\,\mathbf{false} & \Rightarrow & \Diamond l & \text{(an } \textit{initial } \Diamond\text{–rule)}\\
\circledcirc \bigwedge_{a=1}^{g} k_a & \Rightarrow & \Diamond l & \text{(a } \textit{global } \Diamond\text{–rule)}
\end{array}$$

Here k_a, l_b, and l are literals. The outer '$\Box$' operator, that surrounds the conjunction of rules is usually omitted. Similarly, for convenience the conjunction is dropped and we consider just the set of rules R_i.

We note a variant on SNF called merged-SNF (SNF$_m$) [9] used for combining rules by applying the following transformation.

$$\begin{array}{rcl}
\circledcirc A & \Rightarrow & F\\
\circledcirc B & \Rightarrow & G\\
\hline
\circledcirc (A \wedge B) & \Rightarrow & F \wedge G
\end{array}$$

The right hand side of the rule generated may have to be further translated into Disjunctive Normal Form (DNF), if either F or G are disjunctive, to maintain the general SNF rule structure.

3 The resolution procedure

Here we present a review of the temporal resolution method [9]. The clausal temporal resolution method consists of repeated applications of both 'step' and 'temporal' resolution on sets of formulae in SNF, together with various simplification steps.

3.1 Step resolution

'Step' resolution consists of the application of standard classical resolution rule to formulae representing constraints at a particular moment in time, together with simplification rules for transferring contradictions within states to constraints on previous states. Simplification and subsumption rules are also applied.

Pairs of initial $\square$–rules, or global $\square$–rules, may be resolved using the following (step resolution) rule where $\mathcal{L}_1$ and $\mathcal{L}_2$ are both last-time formulae.

$$\begin{array}{rcl} \mathcal{L}_1 & \Rightarrow & A \vee r \\ \mathcal{L}_2 & \Rightarrow & B \vee \neg r \\ \hline (\mathcal{L}_1 \wedge \mathcal{L}_2) & \Rightarrow & A \vee B \end{array}$$

Once a contradiction within a state is found using step resolution, the following rule can be used to generate extra global constraints.

$$\begin{array}{rcl} \circledcirc P & \Rightarrow & \mathbf{false} \\ \hline \bullet\,\mathbf{true} & \Rightarrow & \neg P \end{array}$$

This rule states that if, by satisfying P in the last moment in time a contradiction is produced, then P must never be satisfied in *any* moment in time. The new constraint therefore represents $\square\neg P$ (though it must first be translated into SNF before being added to the rule-set).

The step resolution process terminates when either no new resolvents are derived, or **false** is derived in the form of one of the following rules.

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & \mathbf{false} \\ \circledcirc\,\mathbf{true} & \Rightarrow & \mathbf{false} \end{array}$$

3.2 Temporal resolution

During temporal resolution the aim is to resolve a $\Diamond$–rule, $\mathcal{L}Q \Rightarrow \Diamond l$, where $\mathcal{L}$ may be either of the last-time operators, with a set of rules that together imply $\square\neg l$, for example a set of rules that together have the effect of $\circledcirc A \Rightarrow \square\neg l$. However the interaction between the '$\bigcirc$' and '$\square$' operators in PTL makes the definition of such a rule non-trivial and further the translation from PTL to SNF will have removed all but the outer level of $\square$–operators. So, resolution will be between a $\Diamond$–rule and a *set* of rules that together imply an $\square$–formula which will contradict the $\Diamond$–rule. Thus, given a set of rules in SNF, then for every rule of the form $\mathcal{L}Q \Rightarrow \Diamond l$ temporal resolution may be applied between this $\Diamond$–rule and a set of global $\square$–rules, which taken together force $\neg l$ always to be satisfied.

The temporal resolution rule is given by the following

$$\begin{array}{rcl} \circledcirc A_0 & \Rightarrow & F_0 \\ \cdots & & \cdots \\ \circledcirc A_n & \Rightarrow & F_n \\ \mathcal{L}Q & \Rightarrow & \Diamond l \\ \hline \bullet\,\mathbf{true} & \Rightarrow & \neg Q \vee \bigwedge_{i=0}^{n} \neg A_i \\ \mathcal{L}Q & \Rightarrow & (\bigwedge_{i=0}^{n} \neg A_i)\,\mathcal{W}\,l \end{array}$$

with side conditions

$$\left\{ \begin{array}{rcl} \text{for all } 0 \le i \le n & \vdash & F_i \Rightarrow \neg l \\ \text{and} & \vdash & F_i \Rightarrow \bigvee_{j=0}^{n} A_j \end{array} \right\}$$

where the side conditions ensure that the set of rules $\circledcirc A_i \Rightarrow F_i$ together imply $\square\neg l$. In particular the first side condition ensures that each rule, $\circledcirc A_i \Rightarrow F_i$, makes $\neg l$ true now if $\circledcirc A_i$ is satisfied. The second side condition ensures that the right hand side of each rule, $\circledcirc A_i \Rightarrow F_i$, means that the left hand side of one of the rules in the set will be satisfied. So once the left hand side of one of these rules is satisfied, i.e. if A_i is satisfied for some i in the last moment in time, then $\neg l$ will hold now and the left hand side of another rule will also be satisfied. Thus at the next moment in time again $\neg l$ holds and the left hand side of another rule is satisfied and so on. So if any of the A_i are satisfied then $\neg l$ will be *always* be satisfied, i.e.,

$$\circledcirc \bigvee_{k=0}^{n} A_k \Rightarrow \square\neg l.$$

Such a set of rules are known as a *loop* in $\neg l$.

3.3 The temporal resolution algorithm

Given any temporal formula ψ to be shown unsatisfiable the following steps are performed.

1. Translate ψ into a set of SNF rules ψ_s.
2. Perform step resolution (including simplification and subsumption) until either
   (a) false is derived - terminate noting ψ unsatisfiable; or
   (b) no new resolvents are generated - continue at step 3.
3. Select an eventuality from the right hand side of a $\Diamond$–rule within ψ_s, for example $\Diamond l$. Search for loops in $\neg l$ and generate the resolvents.

4. If any new formulae have been generated, translate the resolvents into SNF add them to the rule-set and go to step 2, otherwise continue to step 5.

5. Terminate declaring ψ satisfiable.

Completeness of the resolution procedure has been shown in [16].

3.4 Loop search

As it is the application of the temporal resolution rule, i.e. the search for a set of rules that together imply $\square\neg l$, assuming we are resolving with $\Diamond l$, that is the most difficult part of the problem it is on this we concentrate in the rest of the paper. Different approaches to detecting such loops have been described in [7] and in particular a depth-first search style algorithm was outlined in [8]. With this algorithm, rules are used as edges in a graph and nodes represent the left hand sides of rules. The Depth-First Search algorithm uses SNF$_m$ rules to try build a path of nodes, where every path leads back into the set of nodes already explored. The SNF$_m$ rules are applied one at a time in a depth-first manner, as several SNF$_m$ rules may be used to expand from a particular node, backtracking when a dead end is reached. The rules governing expansion from a node ensure that the desired looping occurs and, assuming we are resolving with $\Diamond l$, that the required literal $\neg l$, is obtained.

In the next section we give an alternative, Breadth-First Search algorithm, for detecting loops together with an example of its use.

4 Breadth-First Search

Using the Breadth-First Search Algorithm, rules are only selected for use if they will generate the required literal at the next moment in time and their right hand side implies the previous node. The algorithm operates on SNF rules, only combining them into SNF$_m$ when required. With Breadth-First Search all possible rules (but avoiding the duplication of information) are used to expand the graph, rather than just selecting one rule. The graph constructed using this approach is a sequence of nodes that are labelled with formulae in Disjunctive Normal Form. This represents the left hand sides of rules used to expand the previous node which have been disjoined and simplified. If we build a new node that is equivalent to the previous one, using this approach, then we have detected a loop. However if we cannot create a new node then we terminate without having found a loop.

4.1 Breadth-First Search Algorithm

For each rule of the form $\mathcal{L}Q \Rightarrow \Diamond l$ carry out the following.

1. Search for all the rules of the form $\bigcirc X_k \Rightarrow \neg l$, for $k = 0$ to b (called *start rules*), disjoin the left hand sides and make the *top node* H_0 equivalent to this, i.e.
$$H_0 \Leftrightarrow \bigvee_{k=0}^{b} X_k.$$
Simplify H_0. If $\vdash H_0 \Leftrightarrow \textbf{true}$ we terminate having found a loop.

2. Given node H_i, build node H_{i+1} for $i = 0, 1, \ldots$ by looking for rules or combinations of rules of the form $\bigcirc A_j \Rightarrow B_j$, for $j = 0$ to m where $\vdash B_j \Rightarrow H_i$ and $\vdash A_j \Rightarrow H_0$. Disjoin the left hand sides so that
$$H_{i+1} \Leftrightarrow \bigvee_{j=0}^{m} A_j$$
and simplify as previously.

3. Repeat (2) until
   (a) $\vdash H_i \Leftrightarrow \textbf{true}$. We terminate having found a Breadth-First loop and return **true**.
   (b) $\vdash H_i \Leftrightarrow H_{i+1}$. We terminate having found a Breadth-First loop and return the DNF formula H_i.
   (c) The new node is empty. We terminate without having found a loop.

Algorithms to limit the number of rule combinations required have also been developed and are given in [7]. Input to the Breadth-First Search will be a set of SNF rules and these algorithms show how to test whether rules can be used as they are for node expansion, require combination only with start rules, or must be combined with other rules. Similarly, when rules are combined together an algorithm is given to make the number of combinations required as few as possible. Nodes (in DNF) are kept in their simplest form by carrying out simplification and subsumption.

4.2 Example

Breadth-First Search is used to detect the loop in the set of rules given below. Assume we are trying to resolve with the rule $\mathcal{L}Q \Rightarrow \Diamond l$ where $\mathcal{L}$ is either of the last-time operators, and the set of global $\square$-rules is

$$\begin{array}{lrcl@{\qquad}lrcl}
1. & \bigcirc a & \Rightarrow & \neg l & 5. & \bigcirc e & \Rightarrow & e \\
2. & \bigcirc b & \Rightarrow & \neg l & 6. & \bigcirc (a \wedge e) & \Rightarrow & a \\
3. & \bigcirc c & \Rightarrow & \neg l & 7. & \bigcirc b & \Rightarrow & b \\
4. & \bigcirc d & \Rightarrow & \neg l & 8. & \bigcirc c & \Rightarrow & b
\end{array}$$

To create a new node H_i, we examine each rule in turn, disjoining the conjunction of literals on the left hand side of the new node if the rule satisfies the criteria given. The new node is then simplified where necessary.

1. The rules 1–4 have $\neg l$ on their right hand side. We disjoin their left hand sides and simplify (although in this case no simplification is necessary) to give the top node
$$H_0 = a \vee b \vee c \vee d.$$
2. To build the next node, H_1, we see that rules 6, 7 and 8 satisfy the expansion criteria in step (2) of the Breadth-First Search Algorithm (i.e. the right hand side and the literals on the left hand side of each rule implies H_0) but rule 5 does not. Note if we combine rule 5 with any of the other rules to produce an SNF$_m$ rule that satisfies the expansion criteria, its left hand side will be removed through simplification. So we disjoin the literals on their left hand sides of rules 6, 7 and 8 to obtain node
$$H_1 = (a \wedge e) \vee b \vee c.$$
3. Rules 7 and 8 satisfy the expansion criteria and so do rules 5 and 6 when combined together to give the rule $\circledcirc(a \wedge e) \Rightarrow a \wedge e$. Thus node H_2 becomes
$$H_2 = (a \wedge e) \vee b \vee c.$$

As $H_2 \Leftrightarrow H_1$ we terminate having detected a loop. The Breadth-First loop we have found is $(a \wedge e) \vee b \vee c$. The graph constructed using Breadth-First Search for this set of rules is shown in Figure 1.

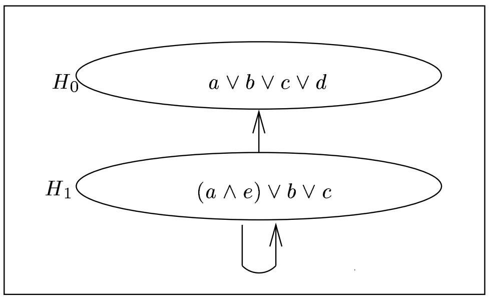

Figure 1 Breadth-First Search Example

4.3 Resolvents

The Breadth-First Search algorithm returns a DNF formula $\bigvee_i D_i$ that means (assuming we are resolving with $\mathcal{L}Q \Rightarrow \Diamond l$) if this formula is satisfied in the previous moment in time then $\neg l$ always holds, i.e.
$$\circledcirc \bigvee_i D_i \Rightarrow \Box\neg l.$$

Rather than try reconstruct a set of rules with which to apply the temporal resolution rule we use the output from Breadth-First Search directly (as if each disjunct were the left hand side of a rule).

In the previous example the loop output was $(a \wedge e) \vee b \vee c$ and the resolvents obtained are
$$\begin{array}{rcl} \bullet\,\mathbf{true} & \Rightarrow & \neg Q \vee \neg((a \wedge e) \vee b \vee c) \\ \mathcal{L}Q & \Rightarrow & \neg((a \wedge e) \vee b \vee c)\,\mathcal{W}\,l \end{array}$$
and must be further translated into SNF.

4.4 The main processing cycle

The Breadth-First search procedure finds *all* loops (see §5) for a particular eventuality. Each time round the main processing loop an eventuality is taken with which to resolve. Termination, declaring the formula satisfiable is only allowed if no more new loops may be found (and step resolution produces no new rules).

Given L_i, for $i = 1$ to n, loops detected previously we can ensure that a new loop L provides no new information by checking whether
$$L \Rightarrow \bigvee_{i=1}^{n} L_i.$$
This is because any new resolvents produced from L will be subsumed by those already generated from loops L_i.

4.5 Correctness issues

Soundness, completeness and termination of the Breadth-First Search algorithm are proved in [7].

5 Breadth-First Search loops

The two main characteristics of loops output by the Breadth-First Search are that *all* loops are detected for an eventuality and that *paths into the loop* are also detected. These are described below and compared with the output of the Depth-First Search algorithm. Finally we consider the memory implications for Breadth-First Search.

5.1 Detection of all the loops

The Breadth-First Algorithm finds *all* the loops for a particular $\diamondsuit$–rule. Considering the example given in §4.2 it is clear that the three SNF$_m$ rules,

$$\begin{array}{lrcl} M1 & \bigcirc(a \wedge e) & \Rightarrow & (a \wedge e \wedge \neg l) \\ M2 & \bigcirc b & \Rightarrow & (b \wedge \neg l) \\ M3 & \bigcirc c & \Rightarrow & (b \wedge \neg l) \end{array}$$

satisfy the criteria for a loop together (and in any combination of subsets of these three rules where if rule $M3$ occurs then rule $M2$ also occurs). We could combine these SNF$_m$ rules further to give other SNF$_m$ rules for example combining rule $M1$ and $M2$ we obtain

$$M4 \quad \bigcirc(a \wedge b \wedge e) \quad \Rightarrow \quad (a \wedge b \wedge e \wedge \neg l).$$

Performing Breadth-First Search we return the formula $(a \wedge e) \vee b \vee c$ as shown previously (because of simplification we will never return $(a \wedge e) \vee b \vee c \vee (a \wedge b \wedge e)$ for example). However performing a Depth-First Search we detect loops one at a time i.e. we would only ever return a loop relating to either rule $M1$ or rule $M2$ or rule $M4$.

5.2 The lead into the loop

Breadth-First Search finds disjuncts representing rules that are never detected in Depth-First Search. An example of this is the disjunct c representing the use of the SNF$_m$ rule $\bigcirc c \Rightarrow (b \wedge \neg l)$. This rule represents the path into the loop (there are no rules that make this rule 'fire'). Both systems are still complete but having found the Depth-First loop $\bigcirc b \Rightarrow (b \wedge \neg l)$, for example, we will have to carry out further applications of the step resolution rule to generate rules equivalent to the complete set of resolvents from Breadth-First Search.

5.3 Memory considerations

Usually a disadvantage of breadth-first search algorithms is the amount of memory required to construct the search space. The Breadth-First Search Algorithm described here does attempt to use *all* the rules or combinations of rules to construct the next node so, in the worst case, memory requirements may be a problem. However, such problems are avoided in many cases for the following reasons. Firstly, the Breadth-First Search Algorithm only requires the storage of the top node, H_0, and the last node constructed, H_i, to enable the construction of node H_{i+1} and to test for termination. The full search space constructed between these two points is not required and therefore after the construction of node H_{i+1}, and failure of the termination test, the memory required for the storage of node H_i may be released. Secondly each node constructed is a DNF formula kept in its simplest form so any redundant information is removed from the node. Finally, although the number of rules in which we search for loops may be large, containing many propositions, typically, the loops will relate to only a few of these rules containing a subset of the total number of propositions.

6 Results and conclusions

Results comparing the Breadth-First Search Algorithm with a previous loop search algorithm are given and conclusions are then drawn.

6.1 Results

A prototype implementation performing the temporal resolution method has been built. Two different loop search programs have been provided—one for Breadth-First Search, the other for the Depth-First Search algorithm. The programs are written in SICStus Prolog [4] running under UNIX and timings have been carried out on a SPARCstation 1 using compiled Prolog code. The test data is a set of valid temporal formulae taken from [13] chosen as it is a reasonably sized collection of small problems to be proved valid. An example of the type of formula being shown valid (we actually shown the negation is unsatisfiable) is

$$\diamondsuit w_1 \wedge \diamondsuit w_2 \Rightarrow \diamondsuit(w_1 \wedge \diamondsuit w_2) \vee \diamondsuit(w_2 \wedge \diamondsuit w_1).$$

We note that, although not presented in the results given here, larger examples have also been tackled, for example Peterson's Algorithm [17, 20]. The full set of results, including timings for each eventuality and example, are given in [7] however, due to space restrictions, we only present a summary of the data here.

Table 1 shows the number of times the total loop search for Depth-First Search (DFS) was less than or equal to, or greater than that for Breadth-First Search (BFS), for each eventuality and example. The figures in brackets are the values as percentages. Values are given for the full data set and for those examples where at least one of the times is greater than 60 and then 100 milliseconds. By considering these categories we hope to eliminate inaccuracies from very low timings. The figures in Table 1 indicate that Breadth-First Search performs better in significantly more examples than Depth-First Search. The increase in the percentage of examples where Breadth-First Search is quicker than Depth-First Search as we move from the examples with (at least one) time over 60 milliseconds to (at least one) time over 100 milliseconds suggests

Subset of Data	DFS $\leqslant$ BFS	BFS < DFS
(i) Full data set	20 (40 %)	30 (60 %)
(ii) Time > 60	10 (37 %)	17 (63 %)
(iii) Time > 100	6 (27 %)	16 (73 %)

Table 1 Summary of Comparative Timings

that Breadth-First Search performs better on larger examples.

Further, the raw data shows that timings for Depth-First Search have a larger range of values than Breadth-First Search. This is what we would expect from depth-first search style algorithms because if the correct search path is chosen first the solution *can* be detected more quickly than breadth-first search type algorithms. However, if a large amount of time is spent exploring fruitless paths then the overall time may be far greater than that for Breadth-First Search.

Table 2 shows the same data displayed in columns representing the number of calls made by each example to the Depth-First Search algorithm, i.e. how many times we have had to search for a loop using Depth-First Search for each example. Recall that Breadth-First Search finds *all* loops for an eventuality where as Depth-First Search find them one at a time. The first row in the table gives the total number of examples for the respective number of calls. The second and third rows give the number and percentage, respectively, of these examples where the time for detecting the loop with Breadth-First Search was less than the total time spent in loop search by Depth-First Search. The column headed 3 has been omitted as there were no examples that required 3 calls to the Depth-First Search Algorithm.

	1	2	4	5
Total	25	22	2	1
BFS < DFS (No.)	13	14	2	1
BFS < DFS (%)	52%	64%	100%	100%

Table 2 Summary by Number of Calls

The table indicates that for this set of examples the more calls to the loop finding section that Depth-First Search makes the more likely it is that it is quicker to do a Breadth-First Search than a Depth-First Search. If we take the greater number of calls to the Depth-First loop finding program as an indication of the size of the example, then this matches the observation that was made previously that Breadth-First Search performs better on larger examples.

6.2 Conclusions

A Breadth-First Search algorithm for implementing Fisher's temporal resolution method has been described and given. Output from this algorithm is analysed and compared with a previously described depth-first approach. An prototype implementation has been produced and run on a variety of valid temporal formulae. Test results suggest that breadth-first search does perform better than this alternative on larger examples.

Although we have only considered the propositional version of the logic it may be possible to extend the resolution method to first-order temporal logic (and even to other temporal logics). Indeed a first-order version of the normal form exists [10]. However, as full first-order temporal logic is undecidable [12] we must first consider subsets to which the resolution method can successfully be applied.

The suitability of Fisher's temporal resolution method to mechanisation, the algorithms developed for the temporal resolution step and prototype implementation together means that temporal resolution provides a viable option for automated temporal theorem proving.

6.3 Acknowledgements

Thanks to my supervisors Professor Howard Barringer and Dr. Michael Fisher for their guidance and encouragement during this work. Thanks also to Graham Gough for many helpful comments and advice.

References

[1] M. Abadi and Z. Manna. Nonclausal Deduction in First-Order Temporal Logic. *ACM Journal*, 37(2):279–317, April 1990.

[2] H. Barringer. Using Temporal Logic in the Compositional Specification of Concurrent Systems. In A. P. Galton, editor, *Temporal Logics and their Applications*, chapter 2, pages 53–90. Academic Press Inc. Limited, London, December 1987.

[3] H. Barringer, R. Kuiper, and A. Pnueli. Now You May Compose Temporal Logic Specifications. In *Proceedings of the Sixteenth ACM Symposium on the Theory of Computing*, 1984.

[4] M. Carlsson and J. Widen. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, Kista, Sweden, September 1991.

[5] A. Cavalli and L. Fariñas del Cerro. A Decision Method for Linear Temporal Logic. In R. E.

Shostak, editor, *Proceedings of the 7th International Conference on Automated Deduction*, volume 170 of *Lecture Notes in Computer Science*, pages 113–127. Springer-Verlag, 1984.

[6] C. L. Chang and R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[7] C. Dixon. *Strategies for Temporal Resolution.* PhD thesis, Department of Computer Science, University of Manchester, 1995.

[8] C. Dixon, M. Fisher, and H. Barringer. A Graph-Based Approach to Resolution in Temporal Logic. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic, First International Conference, ICTL '94, Proceedings*, volume 827 of *Lecture Notes in Artificial Intelligence*, Bonn, Germany, July 1994. Springer-Verlag.

[9] M. Fisher. A Resolution Method for Temporal Logic. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)*, Sydney, Australia, August 1991. Morgan Kaufman.

[10] M. Fisher. A Normal Form for First-Order Temporal Formulae. In *Proceedings of Eleventh International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, Saratoga Springs, New York, June 1992. Springer-Verlag.

[11] B. T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic*, volume 129 of *Lecture Notes in Computer Science.* Springer-Verlag, 1982.

[12] W. Hussak. Decidability in Temporal Presburger Arithmetic. Master's thesis, Department of Computer Science, University of Manchester, February 1987.

[13] Z. Manna and A. Pnueli. Verification of Concurrent Programs: The Temporal Framework. In Robert S. Boyer and J. Strother Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, London, 1981.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1992.

[15] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[16] M. Peim. Propositional Temporal Resolution Over Labelled Transition Systems. Unpublished Technical Note, Department of Computer Science, University of Manchester, 1994.

[17] G. L. Peterson. Myths about the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[18] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the Eighteenth Symposium on the Foundations of Computer Science*, Providence, November 1977.

[19] A. Pnueli. The Temporal Semantics of Concurrent Programs. *Theoretical Computer Science*, 13:45–60, 1981.

[20] A. Pnueli. In Transition From Global to Modular Temporal Reasoning about Programs. In Krysztof Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144, La Colle-sur-Loup, France, October 1984. NATO, Springer-Verlag.

[21] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In J.W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science.* Springer-Verlag, August 1986.

[22] J. A. Robinson. A Machine–Oriented Logic Based on the Resolution Principle. *ACM Journal*, 12(1):23–41, January 1965.

[23] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings IEEE Symposium on Logic in Computer Science*, pages 332–344, Cambridge, 1986.

[24] G. Venkatesh. A Decision Method for Temporal Logic based on Resolution. *Lecture Notes in Computer Science*, 206:272–289, 1986.

[25] P. Wolper. The Tableau Method for Temporal Logic: An overview. *Logique et Analyse*, 110–111:119–136, June-Sept 1985.