Parallel Temporal Resolution

Clare Dixon
Department of Computer Science
University of Manchester
Manchester M13 9PL, UK.
dixonc@cs.man.ac.uk

Michael Fisher and Rob Johnson
Department of Computing
Manchester Metropolitan University
Manchester M1 5GD, UK.
{M.Fisher,R.Johnson}@doc.mmu.ac.uk

Abstract

Temporal reasoning is complex. Typically, the proof methods used for temporal logics are both slow and, due to the quantity of information required, consume a large amount of space. The introduction of parallelism provides the potential for significant speedups together with the increased memory size required to handle larger proofs. Thus, we see the effective utilisation of parallelism as being crucial in making temporal theorem-proving practical.

In this paper we investigate and analyse opportunities for parallelism within a clausal resolution method for temporal logics. We show how parallelism might be introduced into the method in a variety of ways, providing a range of options for the parallel implementation of proof procedures for this important class of logics.

1 Introduction

Temporal logics are non-classical logics that have been specifically developed for reasoning about properties that vary over time [10]. Varieties of temporal logic have been used in both computer science and AI to represent and reason about dynamic systems and systems dealing directly with temporal information. In many of these areas, some form of proof or validation is required. However, as complex systems are often represented by correspondingly large temporal formulae, proofs about such systems are usually long and computationally intensive. A further problem is that, in general, deciding whether a temporal formula is valid or not is difficult. Although the worst-case complexity for temporal proof methods may not arise frequently, a large amount of temporal information needs to be handled even in the average-case.

We see the practical viability of large-scale temporal theorem proving as being dependent upon the effective utilisation of parallel architectures. The successful implementation of parallel proof methods appears to be the only means whereby these large amounts of temporal information can be handled tractably. Although parallel techniques cannot always be applied productively, certain proof methods that have been developed for temporal logics, particularly clausal resolution and semantic tableau, show potential for parallelisation. The exploitation of parallelism is, unfortunately, far less straightforward for temporal logics than in the case of classical logics since the additional complication of partitioning graphs, together with other graph manipulation operations, must be introduced. It is a variety of approaches to tackling this problem that we seek to address in this paper.

1.1 Parallel Theorem-Proving

The parallel implementation of a theorem-prover, whilst not affecting the worst-case complexity of the underlying proof procedure, often enables us to handle much larger formulae, and to process them much faster. In classical logics, parallel implementations of various proof procedures have been developed, including resolution [7], semantic tableau [13], and model elimination [21]. The parallel implementation of these theorem-provers is closely related to the development of parallel implementations of logic programming systems, where potential for parallelism occurs at several different levels of granularity within logical formulae (e.g., formula, clause, or literal) and where several alternative evaluation strategies (e.g., AND/OR parallelism) can be utilised [15].

1.2 Temporal Theorem-Proving

What makes *temporal* theorem-proving more difficult than the classical case, and why can't we directly use the systems already developed? The answer to both questions lies in the fact that temporal logics, particularly the logic we study in this paper, provide more expressive power than their classical counterparts. As discrete temporal logics essentially encode a simple form of induction, then decision procedures for such logics usually involve two forms of search — *tree search* within non-temporal formulae and *graph search* within temporal formulae [11]. Thus, the parallelisation of such decision procedures will involve the integration and parallel implementation of two, distinct, types of search. While the first of these is equivalent to the problems encountered in mechanising classical logic, the necessity of implementing additional graph search mechanisms means that temporal theorem-proving is *fundamentally* different to classical theorem-proving.

1.3 Structure of the Paper

We begin, in §2, by outlining the temporal logic that we wish to mechanise. In §3, the clausal resolution method that we use for proofs in this logic is described [11]. This description includes not only an outline of the proof method, but also an overview of the areas within this algorithm where parallel implementation is either possible or particularly beneficial. The subsequent sections describe the potential for parallelism within these areas in more detail, examining translation to the clausal form (§4), non-temporal resolution (§5), simplification (§6), temporal resolution (§7), and the possibility of executing several of these elements in parallel (§8). Finally, in §9, we present a summary and outline our future work in this area.

2 A Propositional Temporal Logic

In recent years, temporal logics have been used for representing time-dependent information [3], for the specification and verification of reactive systems [17], and for direct execution [5]. Within the broad category of temporal logics, a range of different models of time have been utilised, including discrete, dense, linear, branching and interval-based models [10]. In this paper, we will consider a discrete, linear model of time, and will outline a propositional temporal logic based on this model. This logic is simply called PTL.

Rather than give the syntax and semantics of the full logic, we will present a brief description of the elements of the language, together with their semantics, required for representing the clausal resolution proof method [11].

2.1 Syntax of PTL

The set of *well-formed formulae* of PTL (WFF_p) is defined as follows.

- Any proposition symbol is in WFF_p.
- If A and B are in WFF_p, then so are

$$\begin{array}{lllll} \neg A & A \vee B & A \wedge B & A \Rightarrow B & \\ \Diamond A & \Box A & \bullet A & \circledcirc A & A \mathcal{W} B \end{array}$$

Here, the temporal operators used consist of the *future-time* temporal operators, '$\Diamond$' and '$\Box$', and the *past-time* temporal operators, '$\bullet$' and '$\circledcirc$'.

2.2 Semantics of PTL

Intuitively, the models for PTL formulae are based on discrete, linear structures having a finite past and infinite future, i.e. sequences such as $s_0, s_1, s_2, s_3, \ldots$, where each s_i, called a *state*, provides a propositional valuation. We can visualise this as representing a sequence of 'moments' in time.

Thus, the semantics of a proposition is defined by the valuation given to it at a particular state. While the semantics of the standard propositional connectives are as in classical logic, the semantics of the future time temporal operators are as follows: $\Diamond A$ is satisfied at a particular state if A is satisfied at *some* state in the future; $\Box A$ is satisfied at a particular state if A is satisfied at *all* states in the future; $A \mathcal{W} B$ is satisfied at a particular state if A is satisfied unless a state where B is satisfied occurs.

We are also able to refer to properties in the *past*. As temporal formulae are interpreted at a particular state-index, i, then indices less than i represent states that are 'in the past' with respect to state s_i. The two past-time operators have similar semantics but, as there is a unique start state, termed the *beginning of time*, they have slightly different behaviour when interpreted at this state. Thus, both $\bullet A$ and $\circledcirc A$ are satisfied at a particular state if A is satisfied at the previous state, but $\bullet A$ is satisfied, while $\circledcirc A$ is not, when interpreted at the beginning of time. In particular, $\bullet$ **false** is *only* satisfied when interpreted at the beginning of time.

Note that the following equivalence relates these two *last-time* operators.

$$\circledcirc \neg A \Leftrightarrow \neg \bullet A$$

Although this implies that only one of these operators need be defined, we will see later that it is useful, particularly when defining the normal form, to utilise both operators.

Note that a variety of other temporal operators are available but, as these will be represented within the normal form discussed in the next section simply in terms of the above operators, we will omit the definitions of these derived operators.

2.3 Proof Methods for PTL

Discrete temporal logic essentially characterises the combination of a classical logic system with a minimal form of induction. Decision procedures for such temporal logics are correspondingly more complex, being PSPACE complete in the worst-case. Several proof methods for propositional discrete temporal logics, such as PTL, have been developed, the most widely used being semantic tableau or automata-based methods [23, 12]. More recently, proof methods have been developed based on a translation to classical logics [18] and on resolution, both clausal [11] and non-clausal [1]. Several of these proof techniques, particularly clausal resolution and semantic tableau, show potential for parallelisation. The parallelisation of the temporal tableau method is discussed in [14], while the parallelisation of the resolution method is the subject of this paper.

3 Clausal Temporal Resolution

The temporal resolution system that we investigate here is presented in [11]. The basic idea behind it is to translate an arbitrary PTL formula into a set of formulae in a normal form. This normal form, which corresponds to clausal form in classical resolution, is called Separated Normal Form (SNF). Formulae in SNF are of the form

$$\Box \bigwedge_{i=1}^{n} (P_i \Rightarrow F_i).$$

Here, each P_i is a *strict* past-time temporal formula and each F_i is a *non-strict*[1] future-time formula. Each of the '$P_i \Rightarrow F_i$' (called *rules*) is further restricted to the following:

$$
\begin{array}{rclr}
\bullet\,\mathbf{false} & \Rightarrow & \bigvee_{b=1}^{m} q_b & \text{(an } \textit{initial } \square\text{-rule)} \\
\bigcirc \bigwedge_{a=1}^{l} p_a & \Rightarrow & \bigvee_{b=1}^{m} q_b & \text{(a } \textit{global } \square\text{-rule)} \\
\bullet\,\mathbf{false} & \Rightarrow & \Diamond s & \text{(an } \textit{initial } \Diamond\text{-rule)} \\
\bigcirc \bigwedge_{a=1}^{l} p_a & \Rightarrow & \Diamond s & \text{(a } \textit{global } \Diamond\text{-rule)}
\end{array}
$$

where each p_a, q_b or s is a literal.

Once the translation to SNF has been carried out, then all temporal statements within PTL are represented as sets of such rules. Thus, if we are to derive a contradiction, there are effectively only two ways to achieve this, given a set of SNF rules. The first is to apply resolution *within* a state (i.e., between $\square$-rules), the second is to apply resolution to $\Diamond$-rules. The former is equivalent to classical resolution within a state, the latter involves a search for sets of rules which, together, represent a $\square$-formula that complements the selected $\Diamond$-formula.

3.1 Sequential Resolution Method

Given an arbitrary temporal formula, φ, that we wish to show is unsatisfiable, the clausal temporal resolution method [11] proceeds through the following steps:

1. rewrite φ into Separated Normal Form (SNF), giving a set of *rules* φ_s (note that this transformation preserves satisfiability);
2. perform non-temporal (effectively classical) resolution on pairs of rules within φ_s — if false is derived, terminate, noting that φ is unsatisfiable, otherwise continue;
3. perform simplification and subsumption;
4. choose an eventuality that appears in the set of SNF rules, e.g., $\Diamond\neg p$, and look for sets of rules forming "loops in p" (i.e., sets of rules representing '$\square p$'), add resolvents (rewritten into SNF) for all the loops found;
5. if any new formulae are generated, rewrite them into SNF, add them to φ_s and go to 2;
6. terminate, noting that φ is satisfiable.

Before examining the potential for parallelism in each step of the proof process, we will consider which steps involve the greatest complexity.

[1] Here, 'non-strict' means "including the present".

3.2 Complexity of the Procedure

The complexity of the current (sequential) resolution procedure [9] is:

- translation to SNF — although used many times throughout the proof process, this transformation is not expensive, usually involving only a linear increase in the length of formula and a linear increase in the number of symbols;
- non-temporal resolution — as in classical case, i.e. NP-complete;
- temporal resolution — exponential for each $\Diamond$-formula.

In practice, the translation to SNF is relatively quick, the non-temporal resolution step *can* be slow, while the temporal resolution step is *usually* slow. As this suggests, the main effort has involved investigating the parallelisation of the temporal resolution step.

3.3 Opportunities for Parallelism

There are several places in the sequential algorithm where parallel techniques can (possibly) be exploited:

1. in the translation from an arbitrary formula to SNF;
2. in the non-temporal resolution step;
3. during simplification and subsumption;
4. in the application of temporal resolution to distinct eventualities;
5. when searching for loops.

We will consider the potential for parallelising these parts of the process in the subsequent sections[2]. In §8, we consider the coarser-level parallelism available by executing some of these components in parallel.

4 Conversion to SNF

Although parallelisation is possible in the conversion of an arbitrary formula into SNF, we note that

1. this translation is relatively efficient anyway, and,
2. even if parallel techniques are applied, the bottleneck of ensuring that instances of the same formula are identically renamed, is a (potentially severe) restriction upon parallelism.

Thus, we will not consider the parallelisation of this step (although, in the journal paper, we will outline approaches to this problem).

5 Non-Temporal Resolution

Non-temporal resolution is a variation on classical resolution, being given as the rule:

$$
\begin{array}{rcl}
\bigcirc A & \Rightarrow & F \vee l \\
\bigcirc B & \Rightarrow & G \vee \neg l \\
\hline
\bigcirc (A \wedge B) & \Rightarrow & F \vee G
\end{array}
$$

[2] Apart from (4), which will be considered in more detail in the journal version of this paper.

The exploitation of parallelism is possible here and, indeed, many systems have been developed with this in mind. This enables us to utilise the parallel resolution schemas that have been examined by others for classical logics. For example, the following elements have already been developed within parallel resolution systems.

- The generation of new literals (ROO [16]).
- OR-parallelism with spanning sets (DelPhi [8]).
- OR+AND parallelism (the Andorra model [22]).
- OR-independent AND parallelism (PEPSys [4]).
- Pipelined parallelism (POPE [6]).

Rather than describe any of these options in more detail, we simply note that our non-temporal resolution step is able to utilise such advances in parallel resolution for classical logic.

We will give an agent-oriented (see §8) description of such non-temporal resolution in the journal paper.

6 Simplification and Subsumption

6.1 Simplification

The simplification rewrite rules used in the resolution method for PTL are:

$$Simp_1 : \{ \bigcirc \mathbf{false} \Rightarrow A\} \rightarrow \{\};$$

$$Simp_2 : \{P \Rightarrow \mathbf{true}\} \rightarrow \{\};$$

$$Simp_3 : \{ \bigcirc P \Rightarrow \mathbf{false}\} \rightarrow \left\{ \begin{array}{c} \bullet\,\mathbf{false} \Rightarrow \neg P \\ \bigcirc\,\mathbf{true} \Rightarrow \neg P \end{array} \right\}$$

The first two remove valid formulae, while the third is the basic mechanism for transferring constraints between states.

Since the pattern of the left-hand sides of each of these three simplification rewrite rules is different we may concurrently apply each to the ruleset (R). If we attribute a heterogeneous simple process to the application of each rule $Simp_i$ they may all operate in parallel on R. Furthermore homogeneous copies of each simplification process may execute concurrently. Therefore we may (theoretically) utilise a potential $3 \times |R|$ processors in the simplification process at this level of granularity.

6.2 Subsumption

As in the classical case, a particular bottleneck for parallel resolution is in the subsumption checking for newly generated resolvents.

The formulation of processes for this portion of the algorithm can be tailored according to the required granularity. We may combine all simplification and subsumption elements together. Alternatively we may consider two processes, one dealing with simplification, and the other with subsumption.

In the journal paper, we will expand this section, outlining the finer structure of processes for simplification and subsumption (and actually defining them in terms of 'agents').

7 Temporal Resolution

The general temporal resolution rule, written as an inference rule, can be described as

$$\begin{array}{rcl} \bigcirc A & \Rightarrow & \square p \\ \mathcal{L}Q & \Rightarrow & \Diamond \neg p \\ \hline \bullet\,\mathbf{true} & \Rightarrow & \neg A \vee \neg Q \\ \mathcal{L}Q & \Rightarrow & (\neg A)\,\mathcal{W}\,(\neg p) \end{array}$$

where '$\mathcal{L}$ is a last-time operator. Here, the first resolvent shows that both A and Q cannot be satisfied together, while the second that once Q has occurred then $\neg p$ must occur (i.e. the eventuality must be satisfied) before A can be satisfied.

The full temporal resolution rule is given by expanding the '$\bigcirc A \;\Rightarrow\; \square p$' rule into its constituent parts composed of *global* $\square$-rules, as follows.

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

with side conditions that for all i such that $0 \leq i \leq n$,

1. $\vdash\ F_i \;\Rightarrow\; p$, and,
2. $\vdash\ F_i \;\Rightarrow\; \bigvee_{j=0}^{n} A_j$.

Thus, the side conditions ensure that each $\square$–rule makes p true and the right hand side of each $\square$–rule ensures that the left hand side of one of the $\square$–rules will be satisfied. So if any of the A_i are satisfied then p will be *always* be satisfied, i.e.,

$$\bigcirc \bigvee_{k=0}^{n} A_k \Rightarrow \square p.$$

Such a set of rules are known as a *loop* in p.

Thus, the temporal resolution step essentially consists of a search for a set of rules which together represent a $\square$-formula, complementary to the $\Diamond$-formula to which the resolution is applied. It is this search that is usually the most costly element of the whole resolution process. Fortunately, it is also an element with a significant potential for parallelisation.

There are (currently) two general ways to implement the temporal resolution search, given a set of rules and a $\Diamond$-formula.

1. Merge all possible SNF rules to give SNF_m rules[3], then search for strongly connected components (SCCs) within the graph defined by the SNF_m rules [11].

[3] SNF_m is effectively a form of SNF where the right-hand side of $\square$-rules are in Disjunctive Normal Form (DNF), rather than just being a disjunction of literals.

2. Attempt to *lazily* construct portions of the above SNF_m structure directly from the SNF rules, searching for SCCs during this construction [9].

In the worst case, the second approach has the complexity of the first. The complexity of the first approach is an exponential step ($\mathrm{SNF} \rightarrow \mathrm{SNF}_m$) followed by a linear one (SCC detection using Tarjan's algorithm [2]).

We will term the first approach the *naive* search, and the second approach the *lazy* search.

7.1 Parallelising Naive Search

The basic bottleneck in the naive search is the SNF to SNF_m translation. This involves conjoining all possible subsets of the set of SNF rules and rewriting the new rules' right-hand sides in to DNF. Potentially many parallel processes can be employed in order to achieve this. The main problem that occurs is that of merging all the generated sets of rules back in to one set (and removing subsumed rules). One approach that may alleviate this is by using an intelligent merge, where the sets of rules generated are first sorted before merging.

We will not consider the parallelisation of the naive search in any more detail here, as it is the lazy search that has been the focus of development for the sequential proof method. Its advantage in the sequential case, namely the minimal space consumed, ensures that this approach is also (initially) more desirable in the parallel case. Thus, it is this lazy search that we concentrate upon.

7.2 Parallelising Lazy Search

Within the lazy search approach, there are two basic algorithms:

1. depth-first search through the rule-set looking for a 'loop';
2. breadth-first search through the rule-set looking for a 'loop'.

As in the sequential case, as the breadth-first algorithm detects *all* loops for a particular eventuality whereas the depth-first search described in [9] finds one at a time, we will predominantly consider the parallelisation of the former. However, we should mention that, as the depth-first search is an example of independent search with back-tracking, we can utilise many of the techniques developed for standard OR-parallel search. In particular multiple depth-first searches could be invoked in parallel, with the first successful process being the loop considered. An important element of this would be the sharing of results between independent branches, which would avoid several searches 'discovering' the same loops.

Returning to the breadth-first algorithm, we first give a brief outline of its operation, assuming we wish to detect a loop in the literal p. The breadth-first algorithm builds a graph structure consisting of nodes labelled by formulae in disjunctive normal form (DNF). When building each node we attempt to use *all* possible combinations of SNF rules that satisfy the expansion conditions rather than picking one and backtracking as in the depth-first algorithm. The top node, N_0, is the disjunction (and simplification), of the conjunction of literals appearing on the left-hand side of rules that ensure p in the next moment in time. A new node N_{i+1} is constructed from node N_i by using all possible SNF_m rules where, for each rule, the conjunction of literals on the left-hand side implies the top node and whose right-hand side implies the previous node N_i. The former makes sure that the rule is guaranteed to give p in the next moment in time, and the latter is for constructing the *looping* we require. Termination occurs either when the node we have just constructed is equivalent to the previous node (or equivalent to true), or when we have been unable to construct a new node. If the former, then we have detected a loop, if the latter then we have not found a loop to resolve with the eventuality that we are considering.

More formally, we can describe the breadth-first loop-search algorithm as follows. For each rule of the form $\mathcal{L}Q \Rightarrow \Diamond\neg p$ (where $\mathcal{L}$ is either of the last-time operators) do the following.

1. Search for all the rules of the form $\bigcirc T_i \Rightarrow p$, (called *start rules*), disjoin the left hand sides, simplify and make the *top node* N_0 equivalent to this i.e.
$$N_0 \Leftrightarrow \bigvee_i T_i.$$
If $\vdash N_0 \Leftrightarrow \mathbf{true}$ we terminate having found a loop.
2. Given node
$$N_i \Leftrightarrow \bigvee_k D_k$$
where D_k is a conjunction of literals, to create node N_{i+1} for $i = 0, 1, \ldots$ look for rules or combinations of rules of the form $\bigcirc A_j \Rightarrow B_j$ where $\vdash B_j \Rightarrow N_i$ and $\vdash A_j \Rightarrow N_0$. Disjoin the left hand sides so that
$$N_{i+1} \Leftrightarrow \bigvee_j A_j$$
and simplify as previously.
3. Repeat 2. until
   (a) $\vdash N_i \Leftrightarrow \mathbf{true}$. We terminate having found a breadth-first loop and return **true**.
   (b) $\vdash N_i \Leftrightarrow N_{i+1}$. We terminate having found a breadth-first loop and return the DNF formula N_i.
   (c) The new node is empty. We terminate without having found a loop.

We next consider a range of strategies for parallelising this algorithm. Although this list is not meant to be exhaustive, it does provide a basis for comparison between approaches. Indeed the prototype implementations of some of these strategies have shown them to be relatively successful.

7.2.1 Strategy 1: Top-level partition of disjuncts

This strategy essentially consists of examining the first node, partitioning the disjuncts within this node, and performing breadth-first search separately on each of these new nodes as normal.

Advantages

If we pick a *good* subset of disjuncts we can produce a breadth-first graph for this subset that not only finds the loop, but has fewer nodes and contains fewer literals (just by totaling the number of occurrences of each literal in the graph) than the full graph. Thus, if we can apply some heuristic to detect a *good* subset then we can detect a solution more quickly than with full breadth-first search.

Disadvantages

If we pick a *bad* subset of disjuncts we may produce a breadth-first graph that finds the loop, contains the same number of nodes, but contains more literals (since we have had to carry out more combinations of SNF rules) than the full breadth-first graph.

If a loop is detected each solution (i.e., a DNF formula) will imply the solution from the full breadth-first graph. However, even though this holds, we may obtain a less general solution than with the full breadth first search. This means that if the less general solution does not give specific enough resolvents to produce a contradiction we may have to do another round of temporal resolution to obtain a more general solution.

Perhaps most importantly, we do not yet have a suitable heuristic for identifying *good* partitions, although work on developing heuristics to discard rules that will never form part of the loop (in the sequential algorithm) may be applicable here. Also, since a smaller set of disjuncts is not guaranteed to lead to a solution, then, to retain completeness, we must always ensure that the full breadth-first search runs in parallel.

7.2.2 Strategy 2: Parallel node construction

Here, we carry out a normal breadth-first search but, when constructing the next node, apply the expansion of each disjunct in parallel.

Advantages

Recall that we are searching for combinations of rules where the right hand side implies the previous node, and the conjunction of literals on the left hand side implies the top node. Thus, for each disjunct $d_{i,j}$ in the previous node we look for sets of SNF rules to combine where each right hand side contains a literal in $d_{i,j}$. We combine them so that every literal in $d_{i,j}$ is covered, making sure that the right hand side of the combined rule we have built does actually imply the previous node. By expanding each disjunct separately we will obtain a DNF formula representing the disjunction of the left-hand side of the combined rules used to expand each $d_{i,j}$ as long as the right-hand side of the combined rule has no disjunctions in it.

By carrying this out in parallel we will construct the (unsimplified) new node more quickly than in the sequential version. Further, as the same disjunct may appear in several nodes, and therefore need expansion several times, by storing the DNF formula obtained by expanding a particular disjunct we can avoid repeating calculations to expand the same disjunct several times.

Disadvantages

By expanding each disjunct in parallel we may merge the same combinations of rules more than once to cover two or more disjuncts when the combined rule we are applying has disjuncts on the right-hand side (i.e. we may repeat work by looking at each disjunct separately).

Having expanded each disjunct in node N_i we may end up with several (large) DNF formulae which we must now disjoin and simplify to build node N_{i+1}. In the sequential version simplification is carried out as we use a new rule to expand a node so that the DNF formula does not become excessively large.

7.2.3 Strategy 3: Search re-use

Here, the basic approach is to take some subset of the set of disjuncts from the top node (as strategy 1), try to build a breadth-first graph and, if we fail, add new disjuncts to the subset from the original top node and extend the graph, saving what we had before.

Advantages

Appears to allow the *incremental* construction of the breadth-first graph by reusing parts of the graph that have previously been computed.

Disadvantages

Unfortunately, in most practical cases we have to amend what has been saved so extensively it is just as costly as building the full breadth-first structure in the first place. Specifically, when trying to reuse parts of the graph saved from a previously failed search we must check that

1. we have not overcombined the required rules, and,
2. we have combined the rules enough times.

This occurs simply due to the fact that we have added disjuncts to the top node and now have extra conjunctions of literals that will generate the required literal. However, there seems to be no way to easily test whether, for the disjuncts already constructed, it was necessary to add in more disjuncts, "uncombine" them, or just leave them as they were.

A further disadvantage is that, while the basic breadth-first search algorithm only requires that the top and previous nodes be kept, this strategy requires that *all* the constructed nodes be kept for re-use. Thus, the use of this approach in practice would have major storage implications.

In summary, strategy 2 seems to have the most potential, strategy 1 also seems worth pursuing, while strategy 3 appears to have too many drawbacks. In fact, the most productive approach may be to use a combination of the first two strategies.

In the journal paper we will include a more detailed comparison of these approaches, as well as relative timings for specific examples.

8 Combining Agents

The temporal resolution algorithm contains a number of opportunities for parallelisation. These opportunities vary in their granularity and impact. We can examine the potential parallelism in the algorithm by deconstructing it in order to extract the fundamental operations involved, and to identify any necessary inter-dependencies between components. Following deconstruction the fundamental operations may be composed in order to carry out the same function as the original algorithm. However, the aim of the deconstruction is to liberate parallelism wherever possible by identifying the parts of the algorithm that may execute in parallel without affecting the overall soundness or completeness of the algorithm.

In the journal paper we derive the collection of fine-grained subtasks (fundamental operations) that compose the steps of the algorithm. Each subtask has a processing object associated with it. We shall refer to these objects as *simple agents*. We subsequently reformulate the steps of the algorithm by providing cohesive arrangements of simple agents that reflect the coarser-grained processes evident in the algorithm. We refer to these coarser-grained objects as *process agents*. Given differing arrangements of simple agents and process agents we may reflect the potential parallelisations available to us with varying task grain sizes.

We will describe the constraints and necessary synchronisation points (dependencies) in the algorithm that inhibit parallelism. These dependencies will generally affect the soundness or the completeness of the proof method and hence must not be disregarded during the reformulation process. We will deconstruct the steps of the algorithm with a view to identifying subtasks that may be carried out by simple agents executing in parallel.

When considering forms of parallelism it is useful to consider the activities of homogeneous and heterogeneous agents. Heterogeneous agents carry out different tasks in a domain in parallel, whereas homogeneous agents carry out the same task but on different data. These agent schemas may be viewed (broadly speaking) as mechanisms for describing control and data parallel activities respectively.

Finally, we note that pipelined parallelism may be employed between several of the steps, for instance new constraints produced by temporal resolution may be rewritten as soon as they are derived. Similarly there is no reason to wait for the completion of temporal resolution before commencing non-temporal resolution operations with the rewritten rules produced previously.

9 Conclusions and Future Work

Parallelism seems to provide the potential for efficiently implementing temporal reasoning methods. While we have outlined some of the opportunities for parallelism within one particular proof method, it is important to note that, as there has been very little work in the specific area of *parallel* temporal theorem proving, it is not yet clear what type of parallel architecture is best suited for this application. It appears that some implementation of the system on different machines will be necessary to determine the "best fit".

In the longer paper we present a decomposition of the approach we describe here such that component parts of the proof process are identified to enable the reconstruction of the algorithm as an agent-based design.

9.1 Future Work

Obvious future work includes the further investigation of alternative strategies for parallelising both the temporal resolution method as a whole, and the temporal resolution step within this. The approaches outlined within this paper are currently being developed and implemented — work will continue on these. In the journal paper, a deeper comparison of the approaches outlined here will be presented. This will incorporate both correctness and efficiency issues. As several of the graph search algorithms described in §7 have already been implemented, comparative timings will be provided.

We also hope to utilise advances in state-space search [20], in order to improve the tree-based search procedures, and parallel graph algorithms, such as [19], in order to improve the graph-based search procedures.

Finally, we note that, developing *efficient* parallel algorithms is a difficult task. For example, it is sometimes the case that algorithms considered naive in the sequential case turn out to be optimal given the correct parallelisation. Thus, it is possible that the algorithms considered unsuitable for temporal theorem-proving may be extended and, possibly, generalised, in order to develop appropriate parallel versions.

Acknowledgements

This work was partially supported by a SERC Research Studentship and under SERC Research Grant GR/J48979.

References

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

[2] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] J. Allen and P. Hayes. A Common Sense Theory of Time. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 528–531, Los Angeles, California, August 1985.

[4] U. Baron, J. de Kergommeaux, M. Hailpern, M. Ratcliffe, M. Roberts, J-Cl. Syre, H. Westphal. The parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *Future Generation Computing Systems*, 1988.

[5] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A Framework for Programming in Temporal Logic. In *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Mook, Netherlands, June 1989. (Published in *Lecture Notes in Computer Science*, volume 430, Springer Verlag).

[6] J. Beer and W. Giloi. POPE - A Parallel Operating Prolog Engine. In *Future Comp. Systems*, 3:83-92, 1987.

[7] R. Butler, I. Foster, A. Jindal, and R. Overbeek. A High-Performance Parallel Theorem Prover. *Lecture Notes in Computer Science*, 449:649–650, 1990.

[8] W. Clocksin. Principles of the DelPhi Parallel Inference Machine. In *Computer Journal* vol.31, no.3, 1987.

[9] C. Dixon, M. Fisher, and H. Barringer. A graph-based approach to temporal resolution. In *First International Conference on Temporal Logic (ICTL)*, July 1994.

[10] E. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier, 1990.

[11] 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.

[12] G. Gough. Decision Procedures for Temporal Logic. M.Sc. Thesis, Department of Computer Science, University of Manchester, U.K., October 1984.

[13] R. Johnson. Concurrent Theorem Proving with Tableaux and the Connection Method using Strand over a Distributed Network. technical report 538, Department of Computer Science, Queen Mary and Westfield College, University of London, July 1991.

[14] R. Johnson. A Blackboard Approach To Parallel Temporal Tableaux. In *Proceedings Artificial Intelligence, Methodologies, Systems, and Applications (AIMSA)*, World Scientific, 1994.

[15] F. Kurfeß. *Parallelism in Logic*. Vieweg, 1991.

[16] E. Lusk and W. McCune. Experiments with ROO: A Parallel Automated Deduction System In *Proceedings of Parallelization in Inference Systems, International Workshop*. Springer-Verlag. 1990.

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

[18] D. Plaisted and S-J. Lee. Inference by clause matching. In Z. Ras and M. Zemankova, editors, *Intelligent Systems*, chapter 8, pages 200–235. Ellis Horwood, Chichester, England, 1990.

[19] V. Ramachandran and J. Reif. An Optimal Parallel Algorithm for Graph Planarity. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 282–287, 1989.

[20] V. Saletore and L. Kalé. Consistent Linear Speedups to a First Solution in Parallel State-Space Search. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI)*, pages 227–233, Boston, Massachusetts, 1990. MIT Press.

[21] J. Schumann and R. Letz. PARTHEO: A High-Performance Parallel Theorem Prover. *Lecture Notes in Computer Science*, 449:40–56, 1990.

[22] D. Warren. The SRI model for OR-parallel execution of Prolog-abstract design and implementation issues. In *International Symposium on Logic Programming*, pp92-102, 1987.

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