ISSN 1361 - 6161

THE UNIVERSITY *of* MANCHESTER

Computer Science
University of Manchester

Strategies for Temporal Resolution

Clare Dixon

Department of Computer Science
University of Manchester
Technical Report Series
UMCS-95-12-1

Strategies for Temporal Resolution[1]

Clare Dixon

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
C.Dixon@doc.mmu.ac.uk

December 1995

[1]Refereed by: Graham Gough

STRATEGIES FOR TEMPORAL RESOLUTION

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

September 1995

By
Clare L. Dixon
Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

Verifying that a temporal logic specification satisfies a temporal property requires some form of theorem proving. However, although proof procedures exist for such logics, many are either unsuitable for automatic implementation or only deal with small fragments of the logic. In this thesis the algorithms for, and strategies to guide, a fully automated temporal resolution theorem prover are given, proved correct and evaluated.

An approach to applying 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 over states. It has only one temporal resolution rule and is therefore particularly suitable as the basis of an automated temporal resolution theorem prover.

As the application of the temporal resolution rule is the most costly part of the method, involving search amongst graphs, different algorithms on which to base an implementation are developed. The correctness of each algorithm is given and a comparison of their performance is made.

Strategies have been useful in classical resolution to guide the application of the resolution rule, making proofs more efficient and generating less redundant information. Similarly, a range of strategies related to each step of the temporal resolution theorem prover are suggested to improve the efficiency of the basic system. Heuristics to reduce the amount of information input to the temporal resolution step are described. Suggestions are given on how a classical resolution theorem prover might be incorporated to perform resolution within states. For each of these heuristics and strategies, correctness is considered and results given for a set of examples.

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

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning. However, although a depth-first search algorithm was published originally my MSc. Thesis, "A Graph-Based Approach to Resolution in Temporal Logics", the version presented in Chapter 3 is a significant improvement.

Copyright

Acknowledgements

I would like to thank my supervisors Prof. Howard Barringer and Dr. Michael Fisher for their support, guidance, and encouragement during my years at Manchester. Thanks to everyone in the Temple Group in the Department of Computer Science at the University of Manchester particularly to Graham Gough for his helpful comments and assistance, David Carlisle for his patience and help with many LaTeX and other questions, Rajeev Goré for his encouragement and advice and Martin Peim for the many invaluable discussions, suggestions and for writing the Emacs Lisp interface between Prolog and OTTER. My gratitude is extended to Sean Bechhofer for reading an earlier draft of this thesis.

The financial support from the Engineering and Physical Sciences Research Council is gratefully acknowledged.

I would also like to thank my family and friends for their understanding and patience and for being there when I needed them. Thanks particularly to Sean for his support in so many ways and for keeping me going at the end. I do appreciate it.

The Author

The author graduated from the University of Hull in 1986 with a B.Sc. Honours Degree in Mathematics and Management Sciences. After teaching mathematics for three years, and working as a software engineer in a commercial environment for two years, she joined the Computer Science Department at the University of Manchester in October 1991. Graduating from the University of Manchester in 1992 with an M.Sc. in Computer Science she continued to study for a Ph.D.

Dedication

This thesis is dedicated to Sam Blundell.

Chapter 1

Introduction

1.1 Temporal logic

Temporal logics are non-classical logics that are used to describe and reason about situations that change over time. They are an extension of classical logic having specific operators that deal with time and can be viewed as a special form of modal logics [HC68, Pop94]. Operators used in modal logics, $\Box$ and $\Diamond$, meaning *necessity* and *possibility*, are interpreted as *always* and *eventually* respectively in temporal logics. Different temporal logics have been developed for differing underlying orderings of events and so for example, may be linear [Pnu77, Pnu81, MP92] or branching [BAMP81, CE81, ES88], discrete or dense [BG85, BKP86] or may be based on intervals [HMM83, Mos83, SMSV83]. However, here we will consider linear, discrete temporal logics, unless we state otherwise. Models of propositional temporal logic can be represented as a sequence of *states* indexed by the natural numbers. Each state, s_i, is the set of propositions that is satisfied at the i^{th} moment in time.

Temporal logics have been used extensively for the specification and verification of properties of concurrent systems [Pnu77, Pnu81, OL82, Hai82, Lam83, BKP84, Pnu86, Bar87, MP92]. 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 such properties hold for a program specified in temporal logic involves proofs within the logic itself. There is an extensive body of work on automated verification mostly in the area of *model checking* style systems, see for example [CE81, CES86, GB88, BFG89b, Fis92a], particularly based on tableau [Gou84, Wol85] or automata [WVS83, VW86, Var87]. Model checking involves the generation of a structure that captures all possible behaviours of a system followed by a phase to check that these behaviours match the desired behaviour. 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. Symbolic model checking, Binary Decision Diagrams (BDDs) [Bry86, BCM$^+$90, BCL91], have been introduced as a mechanism to overcome the state space explosion problem. Although this works well for restricted applications the combinatorial explosion can still occur.

Alternatively one can adopt a theorem proving approach which is symbolic but the approach is not limited to finite state problems and has a significant body of work on heuristics to control search (see standard texts such as [CL73, WOLB84, Lov78] for example). Due to this, the approach to temporal theorem proving we develop in this thesis is resolution-based.

1.2 Resolution in Classical Logic

Resolution was proposed as a proof procedure by Robinson in 1965 [Rob65b] for propositional and first-order logics. Resolution was claimed to be "machine-oriented" as it was particularly suitable for proofs to be performed by computer having only one rule of inference that may have to be applied many times. To check the validity of a logical formula Φ, it is negated and $\neg\Phi$ is translated into a normal form, $\tau(\neg\Phi)$.

The resolution inference rule (see below) is applied repeatedly to the set of conjuncts of $\tau(\neg\Phi)$ and new inferences added to the set. If a contradiction (**false**) is derived then $\neg\Phi$ is unsatisfiable and the original formula Φ must therefore be valid. The process of determining the unsatisfiability of the negation of a formula is known as *refutation*. The resolution proof procedure is *refutation complete* for classical logic as, when applied to an unsatisfiable formula, the procedure is guaranteed to produce false.

Classical (clausal) resolution as applied to propositional logic requires formulae to be in a particular form, Conjunctive Normal Form (CNF), before resolution rules may be applied. A formula in CNF may be represented as

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

where each C_i, for $i = 1$ to m, is a disjunction of literals and is known as a *clause* and a literal is an atomic formula or its negation. A formula in Disjunctive Normal Form (DNF) may be represented as

$$D_1 \vee D_2 \vee \ldots \vee D_m$$

where each D_i, for $i = 1$ to m, is a conjunction of literals. Pairs of clauses are resolved using the classical (propositional) resolution rule

$$\frac{\begin{array}{c} A \vee p \\ B \vee \neg p \end{array}}{A \vee B}$$

where A and B are disjunctions of literals. Here $A \vee p$ and $B \vee \neg p$ are called the *parents of the resolvent*, p and $\neg p$ are called *complementary literals* and $A \vee B$ is known as the *resolvent*. Resolvents are added to the set of clauses, $\mathcal{C}$, and the resolution rule is applied to pairs of clauses in $\mathcal{C}$ until an empty resolvent (denoting **false**) is derived or no further resolvents can be generated.

Although the basic resolution method is complete, it is not that efficient as many new resolvents may be generated which are not needed directly in the refutation. Strategies have been developed to guide the search for a refutation in order to reduce the search space, which usually restrict the allowable parents of resolution applications. Consider, for example, unit resolution, where at least one of the parent clauses must be a single literal (known as a *unit clause*); the unit preference strategy, where resolution between clauses is constrained so that at least one is a unit clause, is performed before they would normally be done in the proof; input resolution that requires that at least one of the two parent clauses is a member of the original clause-set; and linear resolution requires that at least one of the parents is either in the initial clause-set or is an ancestor of the other parent. There are other strategies, see [CL73, WOLB84, Lov78] for more details. Unfortunately although these strategies may reduce the search space they may not be refutation complete. For example, unit and input resolution are not complete for propositional logic, while linear resolution is.

Theorem provers for propositional and first-order classical logics based on resolution have been developed. OTTER [McC90] is a resolution theorem prover that allows the use of some of the strategies for propositional and first-order resolution described here and later in the thesis. Resolution is also used as the basis of the logic programming language Prolog. Prolog systems are based on a resolution theorem prover, for a form of logic restricted to Horn clauses (clauses that have at most one positive literal). The particular strategy that Prolog uses is based on linear input resolution and is complete for Horn clauses. Details of logic programming and Prolog may be found in [Kow74, Kow79].

1.3 Resolution in Temporal Logics.

In some cases a form of the classical resolution rule can be applied to temporal logic formulae directly, for example when attempting to resolve pairs of formulae within the context of the $\Box$–operator, such as

$$\frac{\begin{array}{c} A \vee \Box p \\ B \vee \Box \neg p \end{array}}{A \vee B}$$

Generally, though, this is not the case and the problem of how to resolve two complementary literals occurring in *different* temporal contexts arises. Even so, in some cases such formulae may still be resolved. For example, pairs of formulae including the $\square$ and $\Diamond$–operators may be resolved using the following temporal resolution rule.

$$\frac{\begin{array}{c} A \vee \square p \\ B \vee \Diamond \neg p \end{array}}{A \vee B}$$

On the whole, though, this is not possible. The formulae $A \vee \Diamond p$ and $B \vee \Diamond \neg p$, for example, have no obvious resolvent.

Although it would appear sensible to be able to resolve clauses which have complementary literals enclosed in the $\square$ and $\Diamond$–operators as above, further complications occur as the $\square$–formulae may be *hidden*, that is sets of formulae may together imply a $\square$–formula. This happens due to the interaction between $\bigcirc$ (meaning informally in the next moment in time) and $\square$–operators. For example, the formula

$$\square(a \Rightarrow \bigcirc a) \wedge a \wedge l \wedge \square(a \Rightarrow \bigcirc l)$$

implies $\square l$ although this is not immediately obvious, and so this formula should resolve with $\Diamond \neg l$.

Such difficulties in how to apply resolution to temporal logics have led to only a few such methods being suggested. The two main ways are clausal systems, i.e. those that require translation to a normal form [CdC84, Fis91, Ven86] and non-clausal [Aba87]. Problems exist with both approaches however as only a small fragment of the logic is dealt with in [CdC84] (it is not clear whether the completeness proof extends for more general operators) and proof direction causes difficulties in [Aba87] due to the number of resolution rules required. Hence, we chose Fisher's clausal resolution method for temporal logics [Fis91] as the basis of our work as it allows the full range of propositional temporal operators, requires only one temporal resolution rule so is suitable for implementation and has been shown complete [Pei94]. We note that the detection of $\square$–formulae described above is crucial to Fisher's resolution method.

1.4 Overview of the Thesis

Propositional Temporal Logic is the logic used in this thesis. Its syntax and semantics are given in Chapter 2. Then, Fisher's clausal resolution method for this logic is explained. The method has three main parts: the translation to a normal form, the application of resolution within states, and the application of resolution over states. Fisher's method is clausal, so temporal formulae are translated to a normal form constructed using $\Diamond$ and $\bullet$[1] (meaning informally in the previous moment in time). Resolution within states, called *step resolution*, is similar to classical resolution as the step resolution rule is applied to constraints *within* a state. Classical style simplification and subsumption is carried out and a rule to transfer constraints between states is added. Resolution over sequences of states, called *temporal resolution*, attempts to match a formula, φ, that must eventually be satisfied with sets of formulae that together imply that φ will never be satisfied. As it is the latter that is the most expensive part of the search, involving search through graphs, as well as the most novel, it is on the application of the temporal resolution rule that we concentrate.

The three different algorithms for applying the temporal resolution rule are given in Chapter 3. The first algorithm presented forms the foundation of proofs of correctness, given in the following chapter, of the other two. For the other two algorithms we have developed prototype implementations. The structures these algorithms construct and their output are compared. Experimental timings are given and sets of formulae that together imply a $\square$-formula that are more easily detected with one algorithm than another are discussed. The correctness of all three algorithms is established in Chapter 4.

As strategies have been used to guide the application of the resolution rule in classical logics in Chapters 5 to 8 we develop strategies to improve the efficiency of the temporal resolution theorem prover. For non-trivial problems, a large amount of information is input to the algorithms applying the temporal resolution rule. A way of reducing this is suggested in Chapter 5. It is shown that correctness

[1]Actually two last-time operators are used, $\bullet$ and $\odot$ to denote weak and strong last respectively.

is maintained and that, on most examples tried, efficiency is improved. A variety of heuristics used to improve the basic temporal resolution algorithms are described and analysed.

Strategies developed for classical resolution are adapted for use during step resolution. The use of a classical resolution theorem prover to implement step resolution and the limitations of this approach are also described in Chapter 6. Efficient translations to the normal form are investigated in Chapter 7.

In Chapter 8, we relate and compare our approach to resolution in temporal logics to other temporal proof methods using resolution, tableaux and automata. Our conclusions are presented in Chapter 9. We note areas for future work including how our method might be extended to other forms of logic for example to μTL, branching time and first-order temporal logic.

Chapter 2

A Clausal Temporal Resolution

In this chapter we begin by giving the syntax and semantics of a propositional temporal logic, the basic logic used in this thesis. Then, the approach to temporal resolution suggested by Fisher is explained. This was first described in [Fis91] and has been further discussed in [Fis92c].

2.1 Propositional Temporal Logic

Propositional Temporal Logic (PTL) is based on a linear, discrete model of time with finite past and infinite future [GPSS80]. This logic may be viewed as a classical propositional logic augmented with both future-time and past-time temporal connectives.[1] Future-time temporal connectives include:-

connective	informal description
$\diamondsuit$	sometime in the future
$\square$	always in the future
$\bigcirc$	in the next moment in time
$\mathcal{U}$	until
$\mathcal{W}$	unless (weak until)

while past-time connectives include :-

connective	informal description
$\blacklozenge$	sometime in the past
$\blacksquare$	always in the past
$\bullet$	in the previous moment in time (weak last)
$\odot$	in the previous moment in time (strong last)
$\mathcal{S}$	since
$\mathcal{Z}$	zince (weak since)

2.1.1 Syntax

PTL formulae are constructed using the following connectives and proposition symbols.

- A set $\mathcal{P}$ of propositional symbols
- Propositional constants **true** and **false**
- Propositional connectives, $\neg$, $\vee$, $\wedge$, $\Rightarrow$, and $\Leftrightarrow$
- Future-time temporal connectives

[1] We also refer to temporal connectives as temporal operators.

unary connectives : $\bigcirc$, $\Diamond$, $\Box$
binary connectives : $\mathcal{U}$, $\mathcal{W}$

- Past-time temporal connectives

unary connectives : $\bullet$, $\circledcirc$, $\blacklozenge$, $\blacksquare$
binary connectives : $\mathcal{S}$, $\mathcal{Z}$

The set of well-formed formulae of PTL[2], WFF, is inductively defined as the smallest set satisfying :

- Any element of $\mathcal{P}$ is in WFF
- **true** and **false** are in WFF
- If A and B are in WFF then so are

$$\begin{array}{llllll} \neg A & A \vee B & A \wedge B & A \Rightarrow B & A \Leftrightarrow B & \\ \Diamond A & \Box A & A \,\mathcal{U}\, B & A \,\mathcal{W}\, B & \bigcirc A & \\ \blacklozenge A & \blacksquare A & A \,\mathcal{S}\, B & A \,\mathcal{Z}\, B & \circledcirc A & \bullet A \end{array}$$

Temporal formulae may be classified further as follows. A *literal* is defined as either a proposition or the negation of a proposition. A *state-formula* is either a literal or a propositional combination of other state-formulae.
Future-time formulae may be defined as follows

- if A is a state-formula, then A is a future-time formula;
- if A and B are future-time formulae then $\neg A$, $A \wedge B$, $A \vee B$, $A \Rightarrow B$, $A \Leftrightarrow B$, $A \,\mathcal{U}\, B$, $A \,\mathcal{W}\, B$, $\bigcirc A$, $\Diamond A$, and $\Box A$ are all future-time formulae.

As state-formulae that are not in the context of any temporal connective are included in the definition of future-time formulae the notion of a present-time formula will be included for our purposes with future-time formulae.
Pure *past-time formulae* may be defined as follows

- if A and B are either state-formulae or pure past-time formulae then, $\circledcirc A$, $\bullet A$, $\blacklozenge A$, $\blacksquare A$, $A \,\mathcal{S}\, B$, and $A \,\mathcal{Z}\, B$ are also pure past-time formulae;
- if A and B are pure past-time formulae then $\neg A$, $A \vee B$, $A \wedge B$, $A \Rightarrow B$ and $A \Leftrightarrow B$ are also pure past-time formulae.

2.1.2 Semantics

PTL is interpreted over a discrete, linear model of time, for example the natural numbers. Models of PTL can be represented as a sequence of *states*

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

where each state, s_i, is a set of propositions, representing those propositions which are satisfied in the i^{th} moment in time. As formulae in PTL are interpreted at a particular state in the sequence (i.e. at a particular moment in time), the notation

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

denotes the truth (or otherwise) of formula f in the model σ at state index $i \in \mathbf{N}$. For any formula f, model σ and state index $i \in \mathbf{N}$, then either $(\sigma, i) \models f$ holds or $(\sigma, i) \models f$ does not hold, denoted by $(\sigma, i) \not\models f$. If there is some σ such that $(\sigma, 0) \models f$, then f is said to be *satisfiable*. If $(\sigma, 0) \models f$ for all models, σ, then f is said to be *valid* and is written $\models f$.[3]
The semantics of a proposition is defined by the valuation given to it in the model

[2]As usual, parentheses are also allowed to avoid ambiguity.
[3]Note that formulae here are interpreted at s_0. This is an alternative definition to the one usually used in temporal logics, but is equivalent.

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

The semantics of the standard propositional connectives are as in classical logic, for example

$$(\sigma, i) \models A \wedge B \quad \text{iff} \quad (\sigma, i) \models A \text{ and } (\sigma, i) \models B.$$

The semantics of a negated formula is defined as follows

$$(\sigma, i) \models \neg A \quad \text{iff} \quad (\sigma, i) \not\models A.$$

The semantics of the unary future-time temporal connectives are defined as follows

$$\begin{array}{lll}
(\sigma, i) \models \bigcirc A & \text{iff} & (\sigma, i+1) \models A; \\
(\sigma, i) \models \Diamond A & \text{iff} & \text{there exists a } j \geqslant i \text{ s.t. } (\sigma, j) \models A; \\
(\sigma, i) \models \Box A & \text{iff} & \text{for all } j \geqslant i \text{ then } (\sigma, j) \models A.
\end{array}$$

The semantics of the binary future-time temporal connectives are defined as follows

$$\begin{array}{lll}
(\sigma, i) \models A \,\mathcal{U}\, B & \text{iff} & \text{there exists a } k \geqslant i \text{ s.t. } (\sigma, k) \models B \\
 & & \text{and for all } i \leqslant j < k \text{ then } (\sigma, j) \models A; \\
(\sigma, i) \models A \,\mathcal{W}\, B & \text{iff} & (\sigma, i) \models A \,\mathcal{U}\, B \text{ or } (\sigma, i) \models \Box A.
\end{array}$$

The semantics of the unary past-time temporal connectives are defined as follows

$$\begin{array}{lll}
(\sigma, i) \models \bullet A & \text{iff} & i = 0 \text{ or } (\sigma, i-1) \models A; \\
(\sigma, i) \models \odot A & \text{iff} & i > 0 \text{ and } (\sigma, i-1) \models A; \\
(\sigma, i) \models \blacklozenge A & \text{iff} & \text{there exists } j \text{ s.t. } 0 \leqslant j < i \text{ and } (\sigma, j) \models A; \\
(\sigma, i) \models \blacksquare A & \text{iff} & \text{for all } j \text{ s.t. } 0 \leqslant j < i \text{ then } (\sigma, j) \models A.
\end{array}$$

The semantics of the binary past-time temporal connectives are defined as follows

$$\begin{array}{lll}
(\sigma, i) \models A \,\mathcal{S}\, B & \text{iff} & \text{there exists } k \text{ s.t. } 0 \leqslant k < i \text{ and } (\sigma, k) \models B \\
 & & \text{and for all } j \text{ s.t. } k < j < i \text{ then } (\sigma, j) \models A; \\
(\sigma, i) \models A \,\mathcal{Z}\, B & \text{iff} & (\sigma, j) \models A \,\mathcal{S}\, B \text{ or } (\sigma, i) \models \blacksquare A.
\end{array}$$

For convenience there are two last-time connectives "$\bullet$" and "$\odot$", but only one next connective "$\bigcirc$" as the model of time being used has finite past but infinite future. The use of two last-time connectives enable the distinction of the unique start state, known as *the beginning of time*, i.e. s_0. Thus for any formula A, $\odot A$ is **false** at the beginning of time but $\bullet A$ is **true** at the beginning of time. Particularly, $\bullet$ **false** is only **true** when interpreted at the beginning of time.

The past-time connectives are interpreted in a strict sense, that is the current index is not included whereas the future-time connectives are non-strict and include the current index. Being antisymmetric with the past and future in this way does seem natural for the *past* $\Rightarrow$ *future* form required for the normal form we use (see Section 2.2.1) then the strict past is fixed while the present and future remain to be decided. However the rules for temporal resolution could be reformulated with alternative temporal connectives if so desired.

2.2 The Temporal Resolution Method

The temporal resolution method may be divided into three main procedures:

- translation to a normal form;
- step resolution; and
- temporal resolution.

As Fisher's method is clausal, to ascertain the validity of a PTL formula f, we negate and translate $\neg f$ into a normal form, to which resolution, both step and temporal, may be applied. Step resolution consists of the application of a standard classical style resolution rule to formulae that represent constraints holding at a particular moment in time. Temporal resolution is applied to a $\Diamond$–formula and sets of formulae which together imply a $\Box$–formula. The procedure terminates when either **false** has been derived, showing that $\neg f$ is unsatisfiable and therefore f is valid or when no further resolution rules can be applied, showing $\neg f$ is satisfiable (and f is invalid).

2.2.1 SNF

Formulae in PTL can be transformed to a normal form, called Separated Normal Form (SNF), which is the basis of the resolution method described here. SNF was introduced first in [Fis91] and has been extended both in [FN92] and [Fis92b]. After defining SNF, some of the transformation rules required for the translation from an arbitrary temporal formula to SNF are given (for the full set see [Fis95]). An algorithm for translating PTL formulae into SNF is presented and the theoretical aspects of this translation considered. 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 where each particular k_a, k_b, l_c, l_d and l stand for literals.

$$\begin{array}{rcll}
\bullet\,\mathbf{false} & \Rightarrow & \bigvee_c l_c & \text{(an \textit{initial} } \Box\text{–rule)} \\
\circledcirc \bigwedge_a k_a & \Rightarrow & \bigvee_d l_d & \text{(a \textit{global} } \Box\text{–rule)} \\
\bullet\,\mathbf{false} & \Rightarrow & \Diamond l & \text{(an \textit{initial} } \Diamond\text{–rule)} \\
\circledcirc \bigwedge_b k_b & \Rightarrow & \Diamond l & \text{(a \textit{global} } \Diamond\text{–rule)}
\end{array}$$

The outer '$\Box$' connective, 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 also define a variant on SNF called merged-SNF (SNF_m) [Fis91], for combining rules. Given a set of rules in SNF, the relevant set of SNF_m rules may be generated by repeatedly applying the following rule.

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

The right hand side of this SNF_m rule generated may have to be further translated into Disjunctive Normal Form (DNF), if either F or G are disjunctive, to maintain the basic rule structure. Thus, SNF_m represents all possible conjunctive combinations of SNF rules.

Writing Formulae into SNF

The process of transforming formulae into SNF is outlined below and is fully explained in [Fis91, Fis92c]. Usually, in performing these transformations, a set of rules is replaced by a new (usually larger) set of rules.

To translate a PTL formula A into SNF we first anchor it to the beginning of time using the following rewrite rule.

$$A \longrightarrow \Box(\bullet\,\mathbf{false} \Rightarrow A)$$

This does not effect satisfiability as for A to be satisfiable we must find some model such that A is satisfied by that model at the beginning of time.

We then rewrite formulae so that negations are applied only to propositions using rewrite rules such as the following.

$$\begin{array}{rcl} \neg(A\,\mathcal{U}\,B) & \longrightarrow & \neg B\,\mathcal{W}\,(\neg A \wedge \neg B)) \\ \neg(A\,\mathcal{W}\,B) & \longrightarrow & \neg B\,\mathcal{U}\,(\neg A \wedge \neg B) \\ \neg \Box A & \longrightarrow & \Diamond \neg A \\ \neg \Diamond A & \longrightarrow & \Box \neg A \end{array}$$

We note that other such rewrite rules exist, for example for past-time temporal connectives, and that as the left and right sides of the rules are well known logically equivalent temporal formulae such rewrite rules trivially preserve satisfiability.

In an attempt to write a formula into one of the four forms required by SNF we first write it into the form

$$\Box \bigwedge_i (P_i \Rightarrow F_i)$$

where P_i is a strictly past-time formula and F_i is a (non-strict) future-time formula where past-time and future-time formulae are defined in Section 2.1.1. It is always possible to write into this *past* $\Rightarrow$ *future* form as it has been shown [Gab79, Bur84, Gab87], that for a temporal logic such as the one that we have described, it is possible to *separate* an arbitrary formula into an equivalent boolean combination of pure past, present and pure future formulae.

This *past* $\Rightarrow$ *future* form can also be achieved by *renaming*. Renaming allows the replacement of subformulae by a new proposition symbol provided the truth value of the new proposition symbol is then linked with the subformulae it replaced. The technique of renaming was used in [PG86] to produce a normal form for classical logics that maintained the structure of the original formula and has been further investigated in [BdlTC90]. Consider the temporal formula $\Diamond(a \wedge (b\,\mathcal{U}\,c))$, for example. The subformula $(b\,\mathcal{U}\,c)$ can be renamed by the new proposition symbol t giving $\Diamond(a \wedge t) \wedge \Box(t \Leftrightarrow (b\,\mathcal{U}\,c))$. If the subformula has positive polarity (i.e. lies under an even number of negations) then '$\Leftrightarrow$' can be replaced with '$\Rightarrow$'. In the translation to SNF we use '$\Rightarrow$' rather than '$\Leftrightarrow$' and note that it preserves satisfiability which is what we require for resolution. However, a model for the transformed formula is not guaranteed to be a model of the original formula. An example of a renaming transformation to remove nested last-time connectives, where $\mathcal{L}_1$ and $\mathcal{L}_2$ are last-time connectives, F is a future-time formula and t is a new proposition symbol, is

$$\{\mathcal{L}_1\mathcal{L}_2 A \Rightarrow F\} \longrightarrow \left\{ \begin{array}{l} \mathcal{L}_2 A \Rightarrow t \\ \mathcal{L}_1 t \Rightarrow F \end{array} \right\}$$

To obtain the required form we must remove all the past-time temporal connectives except for "last" (either $\bullet$ or $\circledcirc$) and all future-time connectives except $\Diamond$. In the rewrite rules given below P and F represent arbitrary past-time and future-time formulae respectively and t is a fresh proposition symbol. We will just give examples of the future-time temporal temporal connectives but there are analagous rules for their past-time counterparts, a few of which are given at the end of the section.

Using the temporal equivalences

$$\begin{array}{rcl} \Box A & \equiv & A \wedge \bigcirc \Box A \\ A\,\mathcal{W}\,B & \equiv & B \vee (A \wedge \bigcirc(A\,\mathcal{W}\,B)) \end{array}$$

we can derive rewrite rules for the removal of $\Box$ and $\mathcal{W}$ as follows where t is a fresh proposition symbol.

$$\text{Removal of } \Box: \quad \{P \Rightarrow \Box A\} \quad \longrightarrow \quad \left\{ \begin{array}{lcl} P & \Rightarrow & A \wedge t \\ \circledcirc t & \Rightarrow & A \wedge t \end{array} \right\}$$

$$\text{Removal of } \mathcal{W}: \quad \{P \Rightarrow A\,\mathcal{W}\,B\} \quad \longrightarrow \quad \left\{ \begin{array}{lcl} P & \Rightarrow & B \vee (A \wedge t) \\ \circledcirc t & \Rightarrow & B \vee (A \wedge t) \end{array} \right\}$$

We may remove $\mathcal{U}$ by rewriting it in terms of the $\Diamond$–connective.

$$\text{Removal of } \mathcal{U} \ : \quad \{P \Rightarrow A \, \mathcal{U} \, B\} \quad \longrightarrow \quad \left\{ \begin{array}{lcl} P & \Rightarrow & B \vee (A \wedge t) \\ P & \Rightarrow & \Diamond B \\ \circledcirc t & \Rightarrow & B \vee (A \wedge t) \end{array} \right\}$$

Formulae with $\bigcirc$ as their main connective on the right hand side may be rewritten as follows.

$$\text{Removal of } \bigcirc : \{P \Rightarrow \bigcirc A\} \longrightarrow \{ \circledcirc P \Rightarrow A\}$$

The rewrite rules are assumed to take place within the context of a $\Box$–connective and as mentioned previously should only be applied after any negations have been 'pushed' down to propositions. Usually in performing these transformations a set of rules is replaced by a new (usually larger) set of rules.

The transformation rules for the removal of some of the past-time connectives are as follows.

$$\text{Removal of } \mathcal{S} \ : \quad \{A \, \mathcal{S} \, B \Rightarrow F\} \quad \longrightarrow \quad \left\{ \begin{array}{l} \circledcirc (B \vee (A \wedge t)) \Rightarrow F \\ \circledcirc (B \vee (A \wedge t)) \Rightarrow t \end{array} \right\}$$

$$\text{Removal of } \mathcal{Z} \ : \quad \{A \, \mathcal{Z} \, B \Rightarrow F\} \quad \longrightarrow \quad \left\{ \begin{array}{l} \bullet B \vee (A \wedge t)) \Rightarrow F \\ \bullet B \vee (A \wedge t)) \Rightarrow t \end{array} \right\}$$

Other transformations are necessary to ensure that the rules are in the exact form we require so that for example

1. $\bullet$ is only applied to **false**;
2. $\circledcirc$ is only applied to a conjunction of literals;
3. $\Diamond$ is only applied to a literal;
4. the right hand side of rules can only be disjunctions of literals or a $\Diamond$–formula etc.

Here we note a few such transformations for dealing with the two last-time connectives.

$$\begin{array}{rcl} \{ \bullet A \Rightarrow F\} & \longrightarrow & \left\{ \begin{array}{lcl} \circledcirc A & \Rightarrow & F \\ \bullet \mathbf{false} & \Rightarrow & F \end{array} \right\} \\ \{(\circledcirc A \wedge \bullet B) \Rightarrow F\} & \longrightarrow & \{ \circledcirc (A \wedge B) \Rightarrow F\} \\ \{(\circledcirc A \vee \bullet B) \Rightarrow F\} & \longrightarrow & \{ \bullet (A \vee B) \Rightarrow F\} \end{array}$$

For more details of these transformations see [FN92].

Theoretical Considerations

Properties of SNF The following theorem that the translation to SNF preserves the satisfiability of formulae is given and proved in [Fis95]. Let the translation of PTL formulae into SNF be characterised by τ where $\tau(A)$ is the formula A translated into SNF.

Theorem 1 *Given a PTL formula A if A is satisfiable then so is $\tau(A)$.*

Complexity We do not give a detailed analysis of complexity of the translation to SNF but note that the complexity results from [PG86], on which our renaming transformations are based, give a worst case complexity that is a linear increase in the length of the original formula (expressed as a character string) for the propositional calculus. For the renaming transformations described here if we use '$\Rightarrow$' rather than '$\Leftrightarrow$' then by renaming subformulae at the deepest level of nesting first, subformulae are not duplicated and there is a linear increase in both the number of rules and the number of propositions [Fis95]. Note that if '$\Leftrightarrow$' is used however, as renamed subformula are duplicated, there is potentially an exponential increase in the size of the formula. Further, to get the left and right hand sides of rules into the correct form we may have to translate them into DNF and CNF respectively. The cost of translating an arbitrary (classical propositional) formula into CNF or DNF may increase the length of a formula by an exponential amount in the worst case [PG86]. However, see [Fis95], for a fuller description of SNF, its transformations correctness and complexity.

Summary: An Algorithm to Translate PTL Formulae into SNF

The translation from an arbitrary formula to SNF involves a series of transformations as described above. An algorithm for transforming arbitrary PTL formula to SNF is given below.

1. Anchor formula to the beginning of time.
2. Push negations through to proposition symbols.
3. Separate past-time formulae from future-time formulae using renaming.
4. Remove all temporal connectives except $\Diamond$, $\bullet$ and $\circledcirc$ using renaming and temporal transformations.
5. Rewrite the past to the correct form by
   (a) merging $\mathcal{L}_1 A$ and $\mathcal{L}_2 B$ for $\mathcal{L}_i$ $i = 1, 2$ one of the last-time connectives;
   (b) rewriting into DNF and splitting disjuncts;
   (c) expanding $\bullet A$ where A is not **false**.
6. Rewrite the future to the correct form by
   (a) removing embedded $\Diamond$ (renaming)
   (b) ensuring the arguments of $\Diamond$ are literals
   (c) rewriting into CNF and splitting conjuncts.

2.2.2 Step Resolution

Once a formula has been transformed into SNF, both step resolution and temporal resolution can be applied. Step resolution effectively consists of the application of the standard classical resolution rule to formulae representing constraints at a particular moment in time, together with simplification rules, subsumption rules, and rules for transferring contradictions within states to constraints on previous states.

Step Resolution Rules

The step resolution rule is a form of classical resolution applied between $\Box$–rules, representing constraints applying to the same moment in time. Pairs of initial $\Box$–rules, or global $\Box$–rules, may be resolved using the following (step resolution) rules.

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & A \vee r \\ \bullet\,\mathbf{false} & \Rightarrow & B \vee \neg r \\ \hline \bullet\,\mathbf{false} & \Rightarrow & A \vee B \end{array} \qquad \begin{array}{rcl} \circledcirc P & \Rightarrow & A \vee r \\ \circledcirc Q & \Rightarrow & B \vee \neg r \\ \hline \circledcirc (P \wedge Q) & \Rightarrow & A \vee B \end{array}$$

These are both instances of the following more general rule

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

where $\mathcal{L}$ may be either of the last-time connectives but must be the same one throughout an application of the rule.

Simplification

The following SNF rules can be removed during simplification as they represent valid subformulae and therefore cannot contribute to the generation of a contradiction.

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

The first rule is valid as $\circledcirc\,\mathbf{false}$ can never be satisfied, and the second is valid as **true** is always satisfied.

The following inference rule may be used for rules which imply **false**

$$\frac{\circledcirc P \;\Rightarrow\; \mathbf{false}}{\bullet\,\mathbf{true} \;\Rightarrow\; \neg P}$$

and states that, if by satisfying P in the previous moment in time a contradiction is produced, then P must never be satisfied. The new constraint therefore represents $\Box\neg P$. It must be further translated into SNF giving the pair of rules

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

Subsumption

Subsumption also forms a part of the step resolution process. Here, as in classical resolution, a rule may be removed from the rule-set if it is subsumed by another rule. Subsumption may be expressed as the following inference rule

$$\frac{\begin{array}{rcl} P & \Rightarrow & A \\ Q & \Rightarrow & B \end{array}}{Q \;\Rightarrow\; B} \qquad \text{where } \vdash P \Rightarrow Q \text{ and } \vdash B \Rightarrow A,$$

and where the symbol '$\vdash$' means 'is provable' or 'is true'. The side conditions $\vdash P \Rightarrow Q$ and $\vdash B \Rightarrow A$ must hold before this subsumption step can be applied.

Termination

The step resolution process terminates when either no new resolvents are generated or **false** is derived by generating one of the following unsatisfiable formulae

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

Theoretical Considerations

Complexity Again we do not give a detailed analysis of complexity of performing step resolution. However, as step resolution is similar to classical propositional resolution we note that satisfiability for classical propositional logic is an NP-complete problem [Har87, Hak85] and therefore detecting propositional tautologies (a problem equivalent to recognising the unsatisfiablity of the negation of formulae) is co-NP-complete [Hak85].

2.2.3 Temporal Resolution

The temporal resolution process consists of (possibly multiple) applications of the temporal resolution rule resolving together formulae with the '$\Box$' and '$\Diamond$' connectives. However, the interaction between the '$\bigcirc$' and '$\Box$' connectives in PTL ensures that the definition of such a rule is non-trivial. Further, as the translation to SNF restricts the rules to be of a certain form, resolution will be between a $\Diamond$–rule and a *set* of rules that together imply an invariance, i.e. a literal will *always* hold, which will contradict the $\Diamond$–rule. Thus, given a set of rules in SNF, then for every rule of the form

$$\mathcal{L}Q \Rightarrow \Diamond l$$

where $\mathcal{L}$ may be either of the last-time connectives, temporal resolution may be applied between this $\Diamond$–rule and a set of global $\Box$–rules, which taken together force $\neg l$ always to be satisfied. Here, if Q is satisfied then the rule is satisfiable unless the set of global $\Box$–rules force $\neg l$ to always be satisfied. In this case, we must ensure that there is no situation where both Q and any of these rules are satisfied at the same moment. Similarly, once Q has been satisfied we must also ensure that none of the global $\Box$–rules that force $\neg l$ to hold are satisfied until l is satisfied. To resolve with the above rule then, a set of rules must be identified so that, the set of rules, when combined, have the effect of

$$S \Rightarrow \Box\neg l.$$

Here, the S may only be of the form $\circledcirc A$ where A is in DNF. So, the general temporal resolution rule, written as an inference rule, becomes

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

where the first resolvent states that both A and Q cannot be satisfied together, and the second that once Q has occurred then l must occur (i.e. the eventuality must be satisfied) before A can be satisfied. The '$\mathcal{W}$' connective is used as we already have a rule guaranteeing that $\Diamond l$ *will* occur.

The full temporal resolution rule is given by the following

$$\begin{array}{rcl} \circledcirc A_0 & \Rightarrow & B_0 \\ \dots & & \dots \\ \circledcirc A_n & \Rightarrow & B_n \\ \mathcal{L}Q & \Rightarrow & \Diamond l \\ \hline \bullet\,\mathbf{true} & \Rightarrow & \neg Q \vee \bigwedge_{i=0}^{n} \neg A_i \\ \mathcal{L}Q & \Rightarrow & (\bigwedge_{i=0}^{n} \neg A_i)\,\mathcal{W}\,l \end{array}$$

with side conditions

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

where the side conditions ensure that each $\Box$–rule makes $\neg l$ true and the right hand side of each $\Box$–rule ensures that the left hand side of one of the $\Box$–rules will be satisfied.[4] So, if any of the A_i are satisfied then $\neg l$ will be *always* be satisfied, i.e.,

$$\circledcirc \bigvee_{i=0}^{n} A_i \Rightarrow \Box\neg l.$$

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

Thus, the first of the new resolvents ensures that none of the rules which form a loop in $\neg l$ can occur at the same time as a Q occurs, so none of the left hand sides of the rules which ensure a loop in $\neg l$, (i.e., A_i for $i = 0, 1, \dots, n$) can be satisfied at the same time as Q. The second new resolvent ensures that

[4] In its original presentation [Fis91] there was a third side condition but this was found to be unnecessary [Pei94].

once Q has been satisfied, meaning that the eventuality l must be satisfied at some point in the future, none of the conditions for entering a loop is allowed to occur, i.e., none of the A_i (for $i = 0, 1, \ldots, n$) can be satisfied until the eventuality (l) has been satisfied. For more details of the resolution method see [Fis91].

We note that although we have followed Fisher's presentation and described a loop in $\neg l$ as a set of rules that satisfy the side conditions given, the detection of the DNF formula

$$\bigvee_{i=0}^{n} A_i$$

such that

$$\circledcirc \bigvee_{i=0}^{n} A_i \Rightarrow \Box \neg l$$

is sufficient to be able to apply the temporal resolution rule. This is because the right hand side of rules that make up the set forming a loop in $\neg l$ do not appear in the resolvents.

Further, in general in the rest of this thesis most of the formulae we will consider will be in SNF so we give the above resolvents translated into SNF for $i = 0, \ldots n$ where t is a fresh proposition symbol.

$$\begin{array}{rcl}
\mathcal{L}Q & \Rightarrow & \neg A_i \vee l \\
\circledcirc t & \Rightarrow & \neg A_i \vee l \\
\circledcirc \mathbf{true} & \Rightarrow & \neg A_i \vee \neg Q \\
\bullet \mathbf{false} & \Rightarrow & \neg A_i \vee \neg Q \\
\mathcal{L}Q & \Rightarrow & t \vee l \\
\circledcirc t & \Rightarrow & t \vee l
\end{array}$$

We note that the first four resolvents depend on the particular loop detected but the latter two do not. When resolving with an initial eventuality, that is

$$\bullet \mathbf{false} \Rightarrow \Diamond l$$

the third and fourth of these rules will be removed during simplification as they represent valid formulae.

Complexity We note that the satisfiability problem for PTL is PSPACE-complete [SC85].

2.2.4 The Temporal Resolution Algorithm

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

1. Translate ψ into SNF 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 and go to step 2.
5. Terminate declaring ψ satisfiable.

Note, variations on the scheduling of the parts of the algorithm are possible, as long as we ensure that no new loops are found for any eventuality and no more step resolution is possible, before we terminate finding ψ satisfiable. For example, temporal resolution may be started before all the step resolution is completed. Further, in step 3, all the loops for a $\Diamond$–rule may be detected or just one loop and each time round the main processing cycle loops may be detected for each eventuality in turn or processing may be continued for one eventuality until no new loops are found.

The detection of a set of rules that form a loop, for resolution with a $\Diamond$-rule, is a difficult part of the algorithm. Due to this we devote the following chapter to considering different ways of detecting loops in sets of rules.

Chapter 3

Loop Search Algorithms

Recall that to apply the temporal resolution rule given in Section 2.2.3 an appropriate set of $\Box$–rules must be detected for resolution with one of the $\Diamond$–rules. Specifically, given that we want to resolve with a rule of the form

$$\mathcal{L}Q \Rightarrow \Diamond l$$

where $\mathcal{L}$ is either of the last-time operators, our aim is to detect a set of rules $\odot A_i \Rightarrow B_i$, for $i = 0$ to m, that together form a loop in $\neg l$, i.e.

$$\odot \bigvee_{i=0}^{m} A_i \Rightarrow \Box\neg l.$$

As this is the area that we feel most efficiency gains are to be made, we devote the following chapter to a detailed consideration of three approaches to detecting such a set of rules.

1. Building a graph of the set of global $\Box$-rules and detecting subgraphs where $\neg l$ always holds and extracting the corresponding SNF_m rules.
2. Applying a depth-first search to the global $\Box$-rules and detecting suitable cycles where $\neg l$ is guaranteed—known as *Depth-First Search*.
3. Applying a breadth-first search to the global $\Box$-rules and detecting the repetition of the set of (left hand sides of) rules used where the right hand side of each rule implies $\neg l$—known as *Breadth-First Search*.

The first approach is only dealt with theoretically and is mentioned here both as a comparison to the other two approaches and as it is used in the following chapter for proving completeness. For the Depth and Breadth-First Searches we describe the idea behind the approach, give the necessary algorithms and illustrate with an example. A comparison of Depth-First and Breadth-First Searches is made at the end of the chapter. Finally, the type of formulae each search performs best on is discussed. Correctness arguments for each of the algorithms are detailed in the following chapter.

3.1 A Graph of the Global $\Box$-rules

Here we will show how to build a directed graph, known as a *behaviour graph* for a set of global $\Box$–rules and identify the parts of the graph constructed that represent a loop. We assume that all the step resolution possible has been carried out. The approach is based on a suggestion in [Fis91] where a directed graph is built to represent the global $\Box$–rules by combining them into SNF_m. We build the reduced behaviour graph for the set of global $\Box$–rules following the construction presented in [Pei94]. However Peim builds the graph for the full set of SNF rules not just the global $\Box$-rules. We choose to construct the behaviour graph for the global $\Box$-rules alone, as it in in this set that we search for loops. Nodes in the behaviour graph are valuations of the propositions in the set of SNF rules.

Directed edges are constructed between nodes n and n' if n' satisfies the right hand sides of the set of global $\Box$–rules where the literals on the left hand side are satisfied by n. Deletions are performed on the graph by removing any nodes (and the edges that lead into them) that have no edges leading from them. In the resolution system this represents the application of the step resolution rule and the transferral of contradictions within states to constraints on previous states, given in Section 2.2.2.

Loops in a particular literal, for example l, may be detected by searching for terminal subgraphs where the literal, l, occurs in each node.

3.1.1 Building the Reduced Behaviour Graph

Given a set T of SNF rules let

- $I \subseteq T$ be the set of initial $\Box$–rules;
- $R \subseteq T$ be the set of global $\Box$–rules; and
- $S \subseteq T$ be the set of $\Diamond$–rules.

Given T, we construct a finite directed graph $G = (N, E)$, for R the set of global $\Box$-rules, where N is the set of nodes and E is the set of edges, as follows. Nodes, n, of G are valuations of all the propositions occurring in T. Let $R_n \subseteq R$ be the set of rules such that n satisfies the conjunction of literals on the left hand side of each rule. Let C be the set of clauses on the right hand side of R_n. For each valuation n' that satisfies C there is an edge in G from n to n', and these are the only edges originating from n. Note, for node n, if there are no (explicit) rules whose left hand sides are satisfied by n, then there is an edge from n to every other node, including itself, representing the use of the valid rule $\circledcirc\,\mathbf{true} \Rightarrow \mathbf{true}$ to construct edges. However, if the rules in R_n are combined and produce an SNF rule of the form $\circledcirc\, P \Rightarrow \mathbf{false}$ then no edges lead out of the node n. G is known as the *behaviour graph* of R

To obtain the *reduced behaviour graph*, given a behaviour graph for R delete any node with no successors (and all edges into that node), until no more deletions are possible. We call the graph we obtain having performed all possible deletions the *reduced behaviour graph* for R.

3.1.2 Loop Search in the Reduced Behaviour Graph

Having built the reduced behaviour graph $G_R = (N_R, E_R)$ for R as described above, for each rule in S of the form $\mathcal{L}Q \Rightarrow \Diamond l$, where $\mathcal{L}$ is either of the last-time operators, we must look for a terminal subgraph where $\neg l$ holds at every node. Two methods to detect such a subgraph are given below.

Finding A Strongly Connected Component

Given R the set of global $\Box$–rules of T build the reduced behaviour graph, $G_R = (N_R, E_R)$, for R. Select a $\Diamond$-rule, $\mathcal{L}Q \Rightarrow \Diamond l$ in T for resolution. Use an algorithm to find the strongly connected components (s.c.c.'s) of G_R, for example, Tarjan's Algorithm (see [AHU74]). Remove any s.c.c.'s that are not terminal or do not have $\neg l$ holding at each node and ensure that any single nodes have an edge leading back to themselves. For each node n in each remaining s.c.c. find the set of rules, R_n, whose left hand sides are satisfied by n. Combine them together to make an SNF$_m$ rule. Return this set of SNF$_m$ rules.

Maximal Terminal Subgraph Algorithm (MTSA)

Given a set of SNF rules T, where $R \subseteq T$ is the set of global $\Box$–rules, carry out the following steps.

1. Build the reduced behaviour graph, $G_R = (N_R, E_R)$, for R.
2. Select a $\Diamond$–rule in T, of the form $\mathcal{L}Q \Rightarrow \Diamond l$ and do the following.
   (a) Find the subgraph $G_0 = (N_0, E_0)$ such that for any node $n \in N_R$, $n \in N_0$ iff $n \models \neg l$.

(b) Build $G_{i+1} = (N_{i+1}, E_{i+1})$ from $G_i = (N_i, E_i)$ as follows. Set $N_{i+1} = N_i$ and $E_{i+1} = E_i$ and for all $n \in N_{i+1}$ if n has an edge $e \in E_R$ from n to n' and $n' \notin N_i$ then delete e, and any other edges leading out of n, from E_{i+1}, and delete n from N_{i+1}. Repeat until N_{i+1} contains no nodes that lead out of N_i.

(c) If $G_{i+1} = G_i$ or G_{i+1} is empty we terminate and go to step 3, otherwise increment i and repeat step 2(b).

3. If G_{i+1} is empty then there are no loops in $\neg l$. Otherwise, for each node n in G_{i+1} find the set of rules, R_n, whose left hand sides are satisfied by n. Combine the rules in R_n to make an SNF$_m$ rule. Return this set of SNF$_m$ rules.

Note if G_{i+1} is empty then there are no terminal subgraphs where $\neg l$ holds at each node to be found in G_R, otherwise G_{i+1} represents (all) terminal subgraphs where $\neg l$ holds. The set of rules that represents this subgraph constructed in step 3 will form a loop in $\neg l$ and so we can use this set in the application of the temporal resolution rule.

It makes no difference to the completeness of temporal resolution whether we construct rules from the nodes that make up all (terminal) s.c.c.'s where $\neg l$ holds or from the terminal subgraph where $\neg l$ holds for resolution with the $\diamondsuit$–rule. This is because the set of nodes in G_R representing *all* the terminal s.c.c.'s where $\neg l$ holds, N'_R, will be a subset of those representing the terminal subgraph where $\neg l$ holds, N''_R, i.e. $N'_R \subseteq N''_R$. Adding the resolvents from all the terminal s.c.c's to the set of rules will remove any edges into N'_R where l is an outstanding eventuality. As there are no cycles in $N''_R - N'_R$ (if there were they would have been detected as part of N'_R) so all nodes in this set *lead into* N'_R, performing step resolution will remove the remaining nodes $N''_R - N'_R$. See Section 4.3, Lemma 11, for a proof of a similar situation.

3.1.3 Example

As this type of graph is complex and impractical to draw we will give only a small example for the purpose of illustration. Assume we are resolving with $\mathcal{L}Q \Rightarrow \diamondsuit l$ and the set of $\square$–rules are

$$
\begin{array}{rcl}
\bigcirc a & \Rightarrow & a \\
\bigcirc a & \Rightarrow & \neg l.
\end{array}
$$

The nodes in the reduced behaviour graph are all valuations of the propositions a and l, namely,

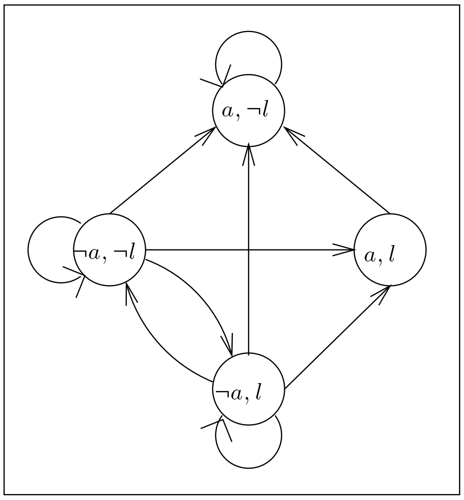

Figure 3.1: An Example Using the Reduced Behaviour Graph Approach

$\{\{a, l\}, \{a, \neg l\}, \{\neg a, l\}, \{\neg a, \neg l\}\}$. To construct the edges between nodes we take each node and extract those rules where the literals on the left hand side are satisfied by the valuation. For example taking the node $\{a, \neg l\}$, the literals on the left hand side of both rules are satisfied by this valuation. Edges from $\{a, \neg l\}$ are drawn to nodes that satisfy the combined right hand side of both these rules, i.e. $a \wedge \neg l$. So we draw a single edge from $\{a, \neg l\}$ back to itself. Note, as no explicit rules satisfy the valuations $\{\neg a, l\}$ and $\{\neg a, \neg l\}$ we apply the rule $\bigcirc\, \mathbf{true} \Rightarrow \mathbf{true}$ and draw edges from each of these nodes to every other node. No deletions to the graph can be made. The reduced behaviour graph for these two rules is shown in Figure 3.1.

Next we apply step (2) of the MTSA to detect terminal subgraphs where $\neg l$ holds at each node. We set N_0 to be the nodes where $\neg l$ is satisfied, that is $N_0 = \{\{a, \neg l\}, \{\neg a, \neg l\}\}$. Then N_1 is set to N_0 and as $\{\neg a, \neg l\}$ has an edge to $\{\neg a, l\}$ which is not a member of N_0 this node is deleted from N_1. $N_1 = \{\{a, \neg l\}\}$ and no further deletions can be made. We continue and set $N_2 = N_1$ (step 2(b) of the algorithm) but as no edges lead out of N_1, no nodes can be deleted from N_2, so $N_2 = \{\{a, \neg l\}\}$. Now $N_2 = N_1$ so we terminate with the subgraph $N_2 = \{\{a, \neg l\}\}$. In step (3) the set of rules that represents this loop is constructed by combining both rules (as the left hand sides are satisfied by $\{a, \neg l\}$) giving the rule

$$\bigcirc\, a \Rightarrow a \wedge \neg l$$

for resolution with the $\Diamond$–rule.

3.2 Depth-First Search

In the reduced behaviour graph approach the construction of the graph is the most complex part of the problem. The number of nodes required in the graph (before deletions) grows exponentially with the number of propositions in the rule-set, an example containing n propositions requires of the order of 2^n nodes. Similarly, calculating which nodes to draw edges between is non-trivial. However, once the graph is built, searching for loops is comparatively simple. Tarjan's algorithm, for example, has a worst case complexity for finding the strongly connected components of a directed graph G of $\mathrm{O}(\mathrm{MAX}(n, e))$ for an n-vertex, e-edge directed graph [AHU74].

Thus, rather than constructing the complete reduced behaviour graph which is expensive, and then searching for loops in the graph constructed, the following two algorithms, Depth-First Search and Breadth-First Search, build graph structures piecemeal. Rules are only used to expand the graph if they generate the literal required and provide the looping necessary. Loops are detected, in each case, as the graph is built.

3.2.1 Overview

The work on Depth-First Search was first presented in [Dix92] and later in [DFB94] however the presentation given in this chapter has been refined and improved. We note however that the completeness proof for Depth-First Search given in Chapter 4 has not been published before and relates a Depth-First Search to a path through the Breadth-First Search graph. Strategies applicable to Depth-First Search are given in Section 5.2.

The Depth-First Search Algorithm starts by searching for rules that generate the required literal in the next moment in time. Nodes from which to start the search are built from the sets of literals on the left hand side of these rules. The algorithm operates from each node in turn attempting to extend the path being built and detect a loop.

Nodes and edges that are built while expanding the graph represent a global $\Box$–rule (or set of global $\Box$–rules combined into SNF_m). For example a rule $\bigcirc\, a \Rightarrow b$ could be represented as shown in Fig. 3.2a. Any rule containing a disjunct on its right hand side, for example $\bigcirc\, a \Rightarrow b \vee c$, can be represented as in Fig. 3.2b using special 'disjunctive' arcs. In fact, for our purposes we need not distinguish between the two types of arcs as the edges leading out of a node will only relate to one rule (in SNF_m), however, we do so for clarity. The set of rules $\bigcirc\, a \Rightarrow b \vee c$, $\bigcirc\, b \Rightarrow a$, $\bigcirc\, c \Rightarrow a$ can be represented as shown in Fig. 3.2c.

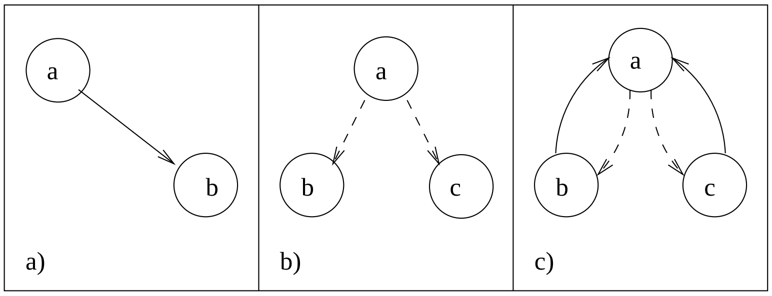

Figure 3.2: Some rules represented as graphs

Assuming we wish to apply temporal resolution to $\mathcal{L}Q \Rightarrow \Diamond l$, then we would like to construct a portion of the graph representing $S \Rightarrow \Box\neg l$, i.e., a subgraph where $\neg l$ is generated at each node as the graph is built. If this subgraph forms a strongly-connected component, we have a loop in $\neg l$, that represents $S \Rightarrow \Box\neg l$, and we can apply the temporal resolution rule. In fact other subgraphs may be sufficient to ensure we have a loop, as long as each edge from a node leads back into the subgraph. So, the basic idea underlying the algorithm is to construct paths through the graph piecemeal until either a suitable subgraph representing a loop is found, or until we have tried all possible rules and combinations of rules to construct the "next" node from each node in the path and no loop has been found (in which case no temporal resolution was possible). Note, however, that the only edges that we use to construct the path are ones which will give $\neg l$ at each node; all other possible edges are ignored.

For example, if we are trying to resolve with $\mathcal{L}Q \Rightarrow \Diamond l$, given a set of global $\Box$–rules R of which $\{\bigcirc a \Rightarrow b \lor c,\ \bigcirc b \Rightarrow a,\ \bigcirc c \Rightarrow a,\ \bigcirc a \Rightarrow \neg l,\ \bigcirc b \Rightarrow \neg l,\ \bigcirc c \Rightarrow \neg l\}$ is a subset, the portion of the graph representing these rules is given in Fig. 3.2c. This subgraph represents a loop in $\neg l$ as once we generate one of these nodes then we will always be led back to a node in this subgraph and $\neg l$ is always generated.

3.2.2 Definition of Terms

In the description of the algorithm and its usage the following terms will be used. Given a set of $\Box$–rules R, a graph of R, $G_R = (N, E)$ is the set of nodes, N, and the set of edges E. A *node*, n, is a set of literals. As we frequently convert from conjunctions of literals (and later in the chapter DNF formulae) to sets of literals, we define a function lits(F), to extract the set of literals from (simplified) DNF formulae F with negations pushed through to propositions,

$$\begin{array}{rcl}
\mathrm{lits}(\mathbf{true}) & = & \{\} \\
\mathrm{lits}(\mathbf{false}) & = & \{\} \\
\mathrm{lits}(a) & = & \{a\} \text{ if } a \in \mathcal{P} \\
\mathrm{lits}(\neg a) & = & \{\neg a\} \text{ if } a \in \mathcal{P} \\
\mathrm{lits}(A \land B) & = & \mathrm{lits}(A) \cup \mathrm{lits}(B) \\
\mathrm{lits}(A \lor B) & = & \mathrm{lits}(A) \cup \mathrm{lits}(B)
\end{array}$$

where $\mathcal{P}$ is the set of proposition symbols. Similarly, to convert from nodes (non-empty sets of literals) to formulae we define the relation con(S) as follows.

$$\begin{array}{rcl}
\mathrm{con}(\{a\}) & = & a \text{ if } a \in \mathcal{P} \\
\mathrm{con}(\{\neg a\}) & = & \neg a \text{ if } a \in \mathcal{P} \\
\mathrm{con}(\{a_0, a_1, \ldots, a_n\}) & = & a_0 \land \mathrm{con}(\{a_1, \ldots, a_n\})
\end{array}$$

The set of edges E is the distributed union of *sets of disjunctive edges for nodes*, i.e.

$$E = \bigcup_{n \in N} D_n.$$

If $n = \text{lits}(A)$, the *set of disjunctive edges for node* n, D_n are directed edges in the graph formed from the rule

$$\bigcirc A \Rightarrow \bigvee_{j=0}^{m} B_j.$$

so that $D_n = \{(n, n_0), \ldots, (n, n_m)\}$ where $\text{lits}(B_j) \supseteq n_j$ for $j = 0$ to m. The *disjunctive list* for this rule is $[\text{lits}(B_0), \ldots, \text{lits}(B_m)]$. A *path* is a structured list of nodes

$$[n_1, n_2, \ldots, n_m, [n_{m_{1_1}}, n_{m_{1_2}} \ldots], [n_{m_{2_1}}, n_{m_{2_2}} \ldots], \ldots, [n_{m_{r_1}}, n_{m_{r_2}} \ldots]]$$

such that $(n_i, n_{i+1}) \in E$ for $i = 1$ to $m - 1$, $(n_{m_{x_i}}, n_{m_{x_{i+1}}}) \in E$ for $x = 1$ to r, $i = 1, \ldots$, and $(n_m, n_{m_{x_1}}) \in E$ for $x = 1$ to r where each sublist may be structured in the same way. Note later in the chapter we index the nodes in paths relating to the order they are constructed in rather than from left to right as we have done here to illustrate the path structure. A *Depth-First loop* is a subgraph $G_l = (N_l, E_l)$ of G_R such that

1. every node in N_l has an edge leading from it back into G_l, i.e. for every n in N_l there exists an edge $e = (n, n')$ and $e \in E_l$ and $n' \in N_l$;

2. every node in N_l has an edge leading into it from a node in G_l, i.e. for every n in N_l there exists an edge $e = (n', n)$ and $e \in E_l$ and $n' \in N_l$; and

3. the graph is *complete* for sets of disjunctive edges, i.e. for all edges e if $e \in D_n$ where D_n is a set of disjunctive edges for a node $n \in N_l$ then $D_n \subseteq E_l$ and for all $e = (n, n') \in D_n$, $n' \in N_l$.

A *self-contained loop* is one where there is only one element in each set of disjunctive edges for each node. A *partial loop* is a subgraph of the loop where $D_n \not\subseteq E_l$ for some node n. Typically, we will have followed one of the disjuncts, but will have others remaining to be explored, and will have found a looping path back to one of the nodes in the partial loop.

Assuming we are resolving with $\mathcal{L}Q \Rightarrow \Diamond l$ and there is a loop in the set of global $\Box$-rules, the Depth-First Loop Search Algorithm returns either $\{\textbf{true}\}$ (representing the detection of the rule $\bigcirc\textbf{true} \Rightarrow \neg l$ in the set of global $\Box$-rules) or the set of nodes that form the Depth-First Loop.

3.2.3 Loop Search Algorithm

Paths are constructed from particular nodes, called start nodes and tested for looping as the paths are built. The paths are constructed in a depth-first manner using the global $\Box$–rules (either SNF or SNF$_m$), i.e. when there is a choice of rules to use to expand the graph, only one is followed. If we find a situation where no rules may be applied to create a new node, then backtracking is invoked and a different choice is made at an earlier choice point.

The algorithm used (assuming that we want to resolve with $\mathcal{L}Q \Rightarrow \Diamond l$) is given below. The forwards and backwards search algorithms mentioned in the steps (4) and (6) are given in Sections 3.2.4 and 3.2.5 repectively. The path constructed during the search is described in Section 3.2.6. In this section as well as Sections 3.2.4, 3.2.5 and 3.2.6 we index nodes in the order they are visited. Any node built from the left hand side of a start rule we search from is labelled n_0 and for subsequent nodes visited the index is increased. The node representing a new disjunct from which forwards search takes place is also labelled by n_0 and for subsequent nodes found, the index is increased. As we describe Backwards and Forwards Search separately this should not be a problem.

1. Search for all the rules of the form $\bigcirc X \Rightarrow \neg l$ (known as *start rules*). If $\bigcirc\textbf{true} \Rightarrow \neg l$ is found terminate returning $\{\textbf{true}\}$.

2. Use the sets of literals from the left hand side of each $\bigcirc X \Rightarrow \neg l$ as the start nodes in the graph.

3. Set the current node, n_0, equal to the next start node if one is available, and path equal to $[n_0]$, otherwise the algorithm terminates having found no new loops.

4. Perform a backwards search from n_0 until either
   (a) no loop has been detected from any of the successors to n_0 - repeat step 3;
   (b) a self-contained loop has been found - continue processing at step 7.
   (c) a partial loop has been found, i.e. we have used a rule r that has two or more disjuncts on its right hand side - remove any nodes that do not form part of the partial loop (the prefix to the loop), store the disjunct list obtained from r after removing the disjunct already used in the partial loop and continue processing with step 5.
5. Set n_0, the current node, equal to a new disjunct from the disjunct list if one is available, otherwise we have detected a loop and continue processing at step 7.
6. Perform a forward search from n_0 until either
   (a) no loop has been detected from any of the successors to n_0, backtrack to where the disjunctive rule was used (i.e. if previously performing a backwards search to within step 4, or if previously performing a forwards search to within step 6) and continue processing;
   (b) a self-contained loop has been found - continue processing at step 7.
   (c) a partial loop has been found - continue processing with step 5.
7. Remove any nodes from the path that do not form part of the loop, called *the prefix to the loop* (see Section 3.2.6 for an example of this). Extract the set of nodes from the path constructed and the algorithm terminates returning this set of nodes.

For a rule-set with more than one eventuality the algorithm is carried out using each $\diamondsuit$–rule in turn until a new loop is detected or there are no more $\diamondsuit$–rules to process.

3.2.4 Backwards Search Algorithm

Here we describe the Backwards Search Algorithm used to start Depth-First Search from one of the start nodes.

If $\circledcirc X_j \Rightarrow \neg l$, for $j = 0$ to m, is the set of start rules the Backwards Search Algorithm proceeds as follows.

1. For n_i, the current node, search for an SNF$_m$ rule

$$\circledcirc \bigwedge_{a=0}^{g} k_a \quad \Rightarrow \quad \bigvee_{b=0}^{r} C_b$$

   where k_a are literals, C_b are conjunctions of literals and the following conditions are satisfied
   (a) there exists a b, $0 \leqslant b \leqslant r$, such that $C_b \Rightarrow \mathrm{con}(n_i)$;
   (b) for all $b, 0 \leqslant b \leqslant r, C_b \Rightarrow \bigvee_{j=0}^{m} X_j$; and
   (c) $\bigwedge_{a=0}^{g} k_a \Rightarrow \bigvee_{j=0}^{m} X_j$.
2. If a new such rule exists
   (a) set the current node n_{i+1} to be $\{k_0, k_0, \ldots, k_g\}$;
   (b) if $r > 1$ (i.e. there is more than one disjunct on the right hand side of the rule) structure the search path to represent this and store the disjuncts that have not been matched to the current node in a list for future processing; and

(c) go to step 3;

otherwise if no such rule exists

(a) if $i > 0$ (i.e. this is not one of the start nodes) backtrack setting the current node to n_{i-1} and repeat step 1; or

(b) if $i = 0$ (i.e. this is one of the start nodes) terminate backwards search and return to the main algorithm.

3. (a) if n_{i+1} is already in the search path return to the main algorithm - a loop or partial loop has been detected; otherwise

   (b) increment i and continue processing at step 1.

We look for rules where one of the disjuncts from the right hand side implies the current node, and the other disjuncts and the literals on the left hand side imply one of the start nodes. This ensures both $\neg l$ is generated and we obtain the looping we require. Thus the new node will be the set of literals from the left hand side of the rule and any disjuncts remaining are saved for future processing. If such a rule does not exist we either backtrack to a previous node and select a different rule or return to the main processing algorithm as there are no different rules to try for any nodes.

3.2.5 Forwards Search Algorithm

The Forwards Search Algorithm is used after a partial loop has been detected using Backwards Search but disjuncts remain to be processed. If $\circledcirc X_j \Rightarrow \neg l$, for $j = 0$ to m, is the set of start rules the Forwards Search Algorithm proceeds as follows.

1. For n_i, the current node, search for an SNF_m rule

$$\circledcirc \bigwedge_{a=0}^{g} k_a \quad \Rightarrow \quad \bigvee_{b=0}^{r} C_b$$

   where k_a are literals, C_b are conjunctions of literals and the following conditions are satisfied

   (a) $\text{con}(n_i) \Rightarrow \bigwedge_{a=0}^{g} k_a$; and

   (b) for all $b, 0 \leqslant b \leqslant r, C_b \Rightarrow \bigvee_{j=0}^{m} X_j$.

2. If a new such rule exists

   (a) amend n_i to be $\{k_0, k_1, \ldots, k_g\}$;

   (b) set the current node n_{i+1} to be $\text{lits}(C_b)$ for some b such that $0 \leqslant b \leqslant r$;

   (c) if $r > 1$ (i.e. there is more than one disjunct on the right hand side of the rule) structure the search path to represent this and store the disjuncts that have not been matched to the current node in a list for future processing; and

   (d) go to step 3;

   otherwise if no such rule exists

   (a) if $i > 0$ backtrack setting the current node to n_{i-1} and repeat step 1; or

   (b) if $i = 0$ terminate forwards search and return to the main algorithm.

3. (a) if there is a node n_j such that $n_j \subseteq n_{i+1}$ then replace n_{i+1} by n_j in the search path and return to the main algorithm - a loop or partial loop has been detected; otherwise

(b) increment i and continue processing at step 1.

We look for rules where the current node implies the conjunction of literals on the left hand side and the disjuncts on the right hand side imply one of the start nodes. We replace node n_i with the set of literals from the conjunction on the left hand side as nodes in the graph represent the left hand sides of rules. The new node will be the set of literals from one of the disjuncts on the right hand side of the rule. If there is more than one disjunct on the right hand side of the rule the search path is structured to represent this and the remaining disjuncts are saved for future processing. If such a rule does not exist we either backtrack to a previous node and select a different rule or return to the main processing algorithm as there are no different rules to try for any nodes.

3.2.6 The Path Constructed

As the graph is expanded, the path through the nodes visited are recorded as a structured list of nodes. Initially we begin our search from a start node n_0 and record this as

$$[n_0].$$

Applying the backwards search algorithm we may construct node n_1. As the graph will now have a directed edge between the nodes n_1 and n_0 we insert n_1 in front of n_0 to obtain the path

$$[n_1, n_0].$$

We continue to apply the backwards search algorithm, adding nodes to the front of the path as described until either

1. we detect a loop; or
2. we can't find a rule with which to expand the graph; or
3. we use a rule with two or more disjuncts on the right hand side.

Let us assume we have constructed the path

$$[n_4, n_3, n_2, n_1, n_0] \tag{3.1}$$

by repeatedly applying the backwards search algorithm and none of cases 1–3 above have occurred. If we visit a node already in the path (and have not used any rules with two or more disjuncts on the right hand side) we have found a loop. So by revisiting node n_2 we add n_2 to the head of the path giving

$$[n_2, n_4, n_3, n_2, n_1, n_0]$$

and then delete the portion of the path that does not form part of the loop, known as the *prefix to the loop*. Here this is $[n_1, n_0]$ as we have a cycle $n_2 \longrightarrow n_4 \longrightarrow n_3 \longrightarrow n_2 \longrightarrow n_4$ etc. leaving the path

$$[n_2, n_4, n_3, n_2].$$

If we had the path labelled 3.1 and then find there are no successors to node n_4 by applying the backwards search algorithm we backtrack until we can make an alternative choice. For example we may backtrack to where the path is

$$[n_1, n_0]$$

and then find n_2' as an alternative successor to n_1 giving

$$[n_2', n_1, n_0]$$

and then continue with backwards search as above.

Finally by expanding the graph using a rule with two or more disjuncts on the right hand side the path is structured to represent the branching. For example given the rule $\bigcirc A \Rightarrow B_1 \vee B_2$ where A, B_1 and B_2 are conjunctions of literals, $n_5 = \text{lits}(A)$, $n_4 \subseteq \text{lits}(B_1)$, the other conditions for backwards search apply and $n_{0_{1_1}} = \text{lits}(B_2)$ then we structure the path as follows

$$[n_5, [n_4, n_3, n_2, n_1, n_0], [n_{0_{1_1}}]]$$

and continue performing backwards search and adding nodes to the front of the path until we reach a node we have visited before. Note in Sections 3.2.3 and 3.2.5 the node $n_{0_{1_1}}$ would have been labelled n_0, as soon as we begin to carry out a forwards search from it, as we have considered backwards and forwards search separately.

When performing forwards search we extract the current portion of the path being expanded (for example $[n_{0_{1_1}}]$) and apply the forwards search algorithm this time adding new nodes to the end of the subpath (as we are moving in the opposite direction to previously). If we use a rule with two or more disjuncts we structure the path as described above to represent this. When we visit a node visited before, for example if we the visit n_2 giving the subpath for forwards search from $n_{0_{1_1}}$ as

$$[n_{0_{1_1}}, \ldots, n_2],$$

we replace the subpath $[n_{0_{1_1}}]$ by this in the main path structure. If we cannot find a successor we backtrack to a node in the path from which we can apply either the forward or backwards search algorithms, whichever is currently applicable, using another rule.

3.2.7 Combinations of Rules

If no successor to the current node can be found, rules are combined using the SNF_m transformation in an attempt to generate a successor. To avoid the expense of combining all the rules, only those which may generate an appropriate successor are extracted from the rule-set for combination purposes. Start rules can also be used for combination purposes.

3.2.8 Addition of Rules

Before each cycle of temporal resolution we need to add the (valid) rule $\bigcirc \mathbf{true} \Rightarrow a \vee \neg a$ for any pair of complementary literals, a and $\neg a$, occurring on the left hand sides of any rules. This is to enable us to be able to detect loops in sets of rules such as

$$\begin{array}{rcl} \bullet\, \mathbf{false} & \Rightarrow & \Diamond l \\ \bigcirc \neg a & \Rightarrow & \neg l \\ \bigcirc a & \Rightarrow & \neg l. \end{array}$$

Here the last two rules satisfy the conditions for being a loop in $\neg l$ given in Section 2.2.3 so we have $\bigcirc(a \vee \neg a) \Rightarrow \Box\neg l$ or $\bigcirc \mathbf{true} \Rightarrow \Box\neg l$. The reason we need to add such rules is because during step resolution we only permit the resolution of complementary literals on the right hand side of rules. Adding an inference rule

$$\begin{array}{rcl} \bigcirc P \wedge r & \Rightarrow & A \\ \bigcirc Q \wedge \neg r & \Rightarrow & B \\ \hline \bigcirc (P \wedge Q) & \Rightarrow & A \vee B \end{array}$$

is another possibility. In this example we would generate $\bigcirc \mathbf{true} \Rightarrow \neg l$ from its application. The rule is certainly sound so will not hinder the soundness of the temporal resolution method. However, in practice the application of such a rule may generate many extra rules. Due to this we prefer to use rules such as $\bigcirc \mathbf{true} \Rightarrow a \vee \neg a$ where necessary.

3.2.9 Example

To aid comparison between the methods we will use the same example for both Depth and Breadth-First searches. Let us assume we are trying to resolve with the rule

$$\mathcal{L}Q \Rightarrow \Diamond l$$

and we are given the following rule-set (already translated into SNF).

$$\begin{array}{rlcl}
1. & \bigcirc x & \Rightarrow & \neg l \\
2. & \bigcirc a & \Rightarrow & \neg l \\
3. & \bigcirc b & \Rightarrow & c \vee d \\
4. & \bigcirc c & \Rightarrow & a
\end{array}
\qquad
\begin{array}{rlcl}
5. & \bigcirc d & \Rightarrow & a \\
6. & \bigcirc a & \Rightarrow & x \\
7. & \bigcirc x & \Rightarrow & b
\end{array}$$

We observe that the loop is made up from combining the following rules.

$$\begin{array}{llcl}
(2+3+6) & \bigcirc(a \wedge b) & \Rightarrow & (c \wedge x \wedge \neg l) \vee (d \wedge x \wedge \neg l) \\
(1+4+7) & \bigcirc(c \wedge x) & \Rightarrow & (a \wedge b \wedge \neg l) \\
(1+5+7) & \bigcirc(d \wedge x) & \Rightarrow & (a \wedge b \wedge \neg l)
\end{array}$$

We illustrate how to detect a loop from this set of rules using Depth-First search. Diagrams of steps 2–4 and steps 5–7 are given in Figures 3.3 and 3.4 respectively. Edges shown using broken lines represent the use of a rule with two or more disjuncts on the right hand side.

1. The $\Diamond$–rule is $\mathcal{L}Q \Rightarrow \Diamond l$ so we search for loops in $\neg l$.

2. Start nodes are $\{x\}$ and $\{a\}$ from rules 1 and 2. Without loss of generality the current node is set to $\{x\}$ and path to $[\{x\}]$.

3. Backwards search matches $\{x\}$ with the right hand side of rule 6. The new current node is $\{a\}$, and the path is $[\{a\}, \{x\}]$.

4. We try to use backwards search to match $\{a\}$ with the right hand side of one of the rules. Although $\{a\}$ matches with the right hand side of rule 4 and rule 5, the left hand side of these rules do not imply the literals on the left hand side of one of the start rules as required. We therefore combine rules 1 and 4 giving

   $$\bigcirc(x \wedge c) \Rightarrow a \wedge \neg l.$$

   Note we could have combined rules 2 and 4, 1 and 5, 2 and 5 etc. The new current node becomes $\{c, x\}$, and the path becomes $[\{c, x\}, \{a\}, \{x\}]$.

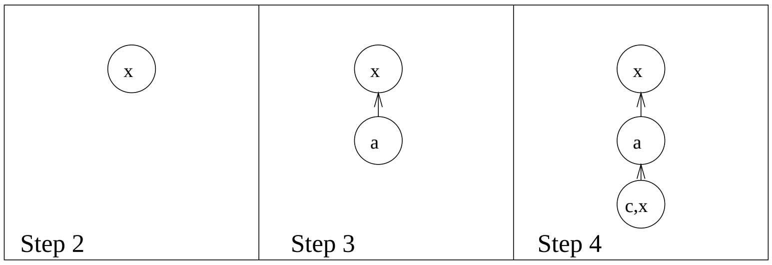

Figure 3.3: Steps 2–4 Using Depth-First Search

5. Backwards search is carried out by attempting to match the right hand side of a rule with node $\{c, x\}$. As no rule matches, rules 3 and 6 are extracted as they will give us a c, or an x, on the right hand side, and are combined to give a new SNF_m rule

   $$\bigcirc(a \wedge b) \Rightarrow (c \wedge x) \vee (d \wedge x).$$

The first disjunct on the right hand side of this new rule now matches the current node, so the new current node is set to $\{a, b\}$, the disjunct $(d \wedge x\}$ is stored remaining to be solved and the path is structured to record the branching becoming

$$[\{a, b\}, [\{c, x\}, \{a\}, \{x\}], [\{d, x\}]].$$

6. Again no rule in SNF matches the current node $\{a, b\}$. Rules 4, 5, and 7 are extracted as they will generate the required nodes on the right hand side to match with the current node. Combining rules 4 and 7 into SNF$_m$ gives a new rule

$$\circledcirc (c \wedge x) \Rightarrow (a \wedge b).$$

The current node matches with the right hand side of this rule, the new current node is set to be the left hand side of the rule $\{c, x\}$, and the path becomes

$$[\{c, x\}, \{a, b\}, [\{c, x\}, \{a\}, \{x\}], [\{d, x\}]].$$

As we have already searched through node $\{c, x\}$, we have found a *partial loop* but still have the disjunct $\{d, x\}$ to solve. We remove the part of the path that does not form part of the loop giving

$$[\{c, x\}, \{a, b\}, [\{c, x\}], [\{d, x\}]].$$

7. The remaining disjunct $\{d, x\}$ is solved using forward search. We need to combine rules 5 and 7 together to give

$$\circledcirc (d \wedge x) \Rightarrow (a \wedge b).$$

Forward search matches $\{d, x\}$ with the left hand side of this rule. The current node becomes $\{a, b\}$ and the sub-path for the disjunct becomes $[\{d, x\}, \{a, b\}]$. As $\{a, b\}$ has already been searched through before, we have found another partial loop, so the sub-path $[\{d, x\}]$ is replaced by this sub-path to give

$$[\{c, x\}, \{a, b\}, [\{c, x\}], [\{d, x\}, \{a, b\}]].$$

8. There are no remaining disjuncts left to solve so the algorithm terminates having detected the loop

$$[\{c, x\}, \{a, b\}, [\{c, x\}], [\{d, x\}, \{a, b\}]]$$

and returns the set of nodes $\{\{a, b\}, \{c, x\}, \{d, x\}\}$.

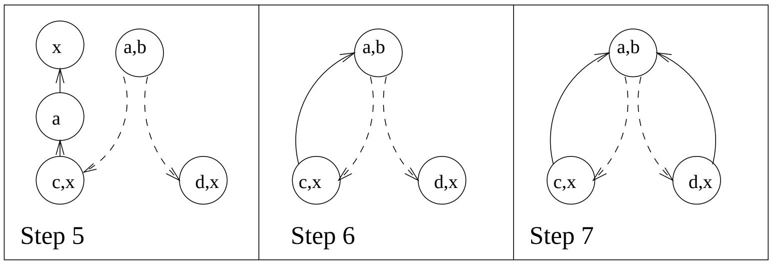

Figure 3.4: Steps 5–7 Using Depth-First Search

Note that although in this example we have performed forwards search only from the node $\{d, x\}$, in general there will be a set of nodes from which forwards search must be carried out. Each node in this set corresponds to each disjunct remaining from the right hand side of a rule used in the construction of the loop so far.

3.2.10 Resolvents

In the previous sections we considered the paths built during Depth-First Search and illustrated them by diagrams, rather than the rules that they represent. We could obtain the rules we have used to build a Depth-First loop by storing the SNF_m rules used during processing. The nodes in the diagrams we have drawn, and in the paths we have constructed, correspond to the left hand sides of these rules. However, during the construction of the path, when we have used a rule with two or more disjuncts on the right hand side, we construct nodes from each disjunct and insert them into the path (see step 5 in the preceeding example and the disjunct $\{d, x\}$). When performing a forwards search from such nodes, n, we must replace n by the set of literals from the left hand side of the rule used in forwards search. Recall from Section 2.2.3 that the resolvents from temporal resolution only require the left hand sides of the set of rules that together imply a $\square$–formula and the right hand sides of these rules are only used to make sure that this set of rules really is a loop. We assume that the latter is ensured by applying the Depth-First algorithm (and prove this is so in the following chapter) so we have no need to keep track of which rules have been used as we can just extract the relevant information from the path that the algorithm generates.

In the example given in Section 3.2.9 the algorithm constructs the path

$$[\{c, x\}, \{a, b\}, [\{c, x\}], [\{d, x\}, \{a, b\}]].$$

From this the three distinct nodes

$$\{\{a, b\}, \{c, x\}, \{d, x\}\}$$

are extracted and returned. The resolvents obtained

$$\begin{array}{rcl} \bullet\,\mathbf{true} & \Rightarrow & \neg Q \lor (\neg(a \land b) \land \neg(c \land x) \land \neg(d \land x)) \\ \mathcal{L}Q & \Rightarrow & ((\neg(a \land b) \land \neg(c \land x) \land \neg(d \land x))\, \mathcal{W}\, l \end{array}$$

must be further translated into SNF.

The set of nodes given above is stored so we can detect when we have discovered no new loops. Mechanisms for checking whether a loop provides no new information are given in Section 5.2.3.

3.2.11 Inclusion in the Main Processing Cycle

The Depth-First search algorithm returns the set of nodes from the loop found (assuming this has not previously been detected). There are several ways that the Depth-First search algorithm could be incorporated into the main processing cycle.

1. All the loops could be detected for a particular eventuality before they are translated to SNF and further step resolution is applied.

2. A single loop could be detected for an eventuality, translated to SNF, and step resolution carried out, then another loop for the *same* eventuality etc. When no further loops can be detected for an eventuality the next eventuality is used.

3. A single loop could be detected for an eventuality, translated to SNF, and step resolution carried out, and then a different eventuality picked in which to detect a loop.

It is the third method we concentrate on, for if we incorporate a strategy to detect the *easy* loops first (see Section 5.2.1) we can quickly find the loops involving the fewest rules and the smallest number of literals for each eventuality. This may lead to the detection of a contradiction avoiding having to detect larger more complex loops. Further, we choose to process each eventuality in turn as the loops required for resolution with one eventuality may depend on the resolvents generated by another. However, one of the other two approaches could be adopted if desired.

3.3 Breadth-First Search

The Depth-First Search Algorithm expands the current node using each suitable rule (i.e. those that satisfy the conditions for forwards or backwards search depending on which search mode we are using) in turn, attempting to construct a loop and backtracking to make an alternative choice if no loop can be built. As is common with depth-first search algorithms, this algorithm manages to detect loops relatively quickly if the *correct* rules are used to expand the graph first, but takes relatively longer if the *wrong* rules are used first, i.e. time is wasted by searching down fruitless paths. So, a Breadth-First Search Algorithm has been developed that uses all possible rules to expand the graph at each point rather than selecting just one and backtracking to select a different rule if a loop was not found on the path.

In this section we give the algorithms to detect loops using Breadth-First Search. The high level algorithm for creating the graph and its termination conditions is given in Section 3.3.2. The process of stepping through the rule-set rule by rule and attempting to use or discard each rule to generate the next node is given in Section 3.3.3. In Section 3.3.4 we explain how to combine the rules that cannot be used directly. The simplication performed on the nodes built is explained in Section 3.3.5 and in Section 3.3.6 we illustrate the algorithms with an example. How to obtain resolvents from the output of a Breadth-First Search is shown in Section 3.3.7 and an explanation of the inclusion of Breadth-First Search in the main algorithm is given in Section 3.3.8. Strategies that have been incorporated into the Breadth-First Search Algorithm are given in Section 5.3.

3.3.1 Overview

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 imply the previous node, as with Depth-First Search. However, 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.

3.3.2 Graph Creation

For each rule of the form $\mathcal{L}Q \Rightarrow \Diamond l$ do 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 (see Section 3.3.5). If $\vdash H_0 \Leftrightarrow \mathbf{true}$ 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) $\vdash H_i \Leftrightarrow \mathbf{true}$. We terminate having found a Breadth-First loop and return $\mathbf{true}$.

(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.

3.3.3 Node Creation

As part of step (2) in Section 3.3.2 we provide an algorithm to test whether to use an SNF rule directly, whether it needs to be combined into SNF_m or can be discarded, so as to satisfy the conditions in step (2) above. First we give the algorithm and then explain the justification behind each of the steps. Recall the function $\text{lits}(F)$, defined in Section 3.2.2, which extracts the set of literals from the simplified DNF formula F.

In order to create the new node, H_{i+1}, (part (2) of the algorithm in Section 3.3.2) each rule, excluding the start rules, in the set of SNF global $\Box$–rules, $\bigcirc X \Rightarrow Y$ is examined and

1. if $\vdash Y \Rightarrow H_i$ and $\vdash X \Rightarrow H_0$ then X is added as a disjunct of H_{i+1};
2. if $\vdash Y \Rightarrow H_i$ but $\nvdash X \Rightarrow H_0$ then $\bigcirc X \Rightarrow Y$ is combined with every start rule in turn and the left hand sides are added as new disjuncts to H_{i+1};
3. if neither of the above hold, but $\text{lits}(Y) \subseteq \text{lits}(H_i)$ then $\bigcirc X \Rightarrow Y$ is added to a list for combinations;
4. if none of the above hold the rule is discarded for the purposes of this node.

The idea of these tests is that we want to use as many rules as possible to ensure every possible way of generating the previous node. If a rule can be used on its own that will ensure $\neg l$ holds, and a disjunct from the previous node will hold in the next moment, then the left hand side of this rule is added directly as a disjunct of the new node (step 1). Although we want to try and use as many of the rules as we can, there is no need to combine them unnecessarily as these combinations will be removed during the simplification of the node (see Section 3.3.5). In step (2) we try to use a rule but we have not ensured the generation of the required literal, $\neg l$, in the next moment in time. The way to do this which makes the smallest number of combinations is by combining the rule with each start rule. Step (3) above catches rules that may need be combined with others (that is other than start rules) to give a rule in SNF_m that may then satisfy steps (1) or (2). For example if we are resolving with $\mathcal{L}Q \Rightarrow \Diamond l$ and have the rules

$$\begin{array}{llcl}
1 & \bigcirc(r \wedge q) & \Rightarrow & \neg l \\
2 & \bigcirc(s \wedge q) & \Rightarrow & \neg l \\
3 & \bigcirc(r \wedge q) & \Rightarrow & r \vee s \\
4 & \bigcirc r & \Rightarrow & q
\end{array}$$

by applying the Breadth-First Search Algorithm we obtain the top node,

$$H_0 = (r \wedge q) \vee (s \wedge q)$$

and note that $\text{lits}(H_0) = \{q, r, s\}$. To build node H_1 we need SNF_m rules that satisfy step (2) of the Graph Creation Algorithm (Section 3.3.2). Now using the algorithm for Node Creation neither of rules 3 or 4 satisfy steps (1) or (2) but both satisfy the condition for step (3) (for rule 3 $\text{lits}(r \vee s) = \{r, s\} \subseteq \{q, r, s\}$ and for rule 4 $\text{lits}(q) = \{q\} \subseteq \{q, r, s\}$). Thus rules 3 and 4 are set aside to combine together. Combining rules 3 and 4 together we get

$$\bigcirc(r \wedge q) \Rightarrow (q \wedge r) \vee (q \wedge s)$$

and now this combined rule satisfies step (1) of the Node Creation algorithm and $(r \wedge q)$ is added as a new disjunct to node H_1. How these rules may be combined together in an efficient way is described in Section 3.3.4.

Step 4 discards any rules that do not satisfy conditions 1–3 and are not required for the creation of the new node. As we have completed all the step resolution possible (and here we assume we have), if we have a set of rules R' that may be combined to satisfy the criteria for building a new node where one of the rules satisfies step 4 of the Node Creation Algorithm, then we can delete this rule from R' and the rule obtained from combining the remaining rules should still satisfy the criteria for building a new node. In the above example we may have an additional rule

$$5 \quad \bigcirc t \quad \Rightarrow \quad t$$

where $\mathrm{lits}(t) = \{t\} \not\subseteq \{q, r, s\}$ and by combining rules 3, 4 and 5 we obtain the rule

$$\bigcirc(r \wedge q \wedge t) \Rightarrow (q \wedge r \wedge t) \vee (q \wedge s \wedge t)$$

with which we can expand the graph so that $H_1 = \{(r \wedge q \wedge t)\}$. However combining rules 3 and 4 alone and adding the left hand side to H_1 gives $H_1 = \{(r \wedge q \wedge t) \vee (r \wedge q)\}$ which can be simplified to give $H_1 = \{(r \wedge q)\}$. So rule 5 can be ignored for combination purposes.

3.3.4 Combining Algorithm

We now give an algorithm to combine the rules remaining from step 3 in Section 3.3.3. What we are trying to do here is reduce the number of rule combinations we need to perform to a minimum. In step (1) of the algorithm below we work on each literal from the previous node separately, extracting any rules from the set whose right hand side contains this literal. In step (2), for each disjunct in turn we extract a set of rules one for each literal in the disjunct. These rules are combined taking one rule from each set in step (3). The combined rule is tested to see whether it can be used to build the new node. If not it is added to the relevant sets for each literal on its right hand side.

Let the previous node be H_i, and the new node we are trying to create be H_{i+1}. The previous node is made up from several disjuncts, i.e.

$$H_i \Leftrightarrow \bigvee_j D_j,$$

and each of these disjuncts is a conjunction

$$D_j = \bigwedge_{k=0}^{m} c_k$$

where each c_k, for $k = 0$ to m is a literal. The set of rules to be combined (part 3 of the algorithm in Section 3.3.3) are the rules $\bigcirc X \Rightarrow Y$ where $\mathrm{lits}(Y) \subseteq \mathrm{lits}(H_i)$ but that do not satisfy conditions 1 and 2 of the algorithm.

1. For each literal, l, that is a member of $\mathrm{lits}(H_i)$ build R_l the set of rules, $\bigcirc X \Rightarrow Y$, such that $l \in \mathrm{lits}(Y)$.

2. For each disjunct, D_j, in H_i extract the sets R_l such that $l \in \mathrm{lits}(D_j)$.

3. For each D_j

   (a) Take a new combination of rules one from each set R_l such that $l \in D_j$.

   (b) Test the combined rule, $\bigcirc A \Rightarrow B_0 \vee B_1 \vee \ldots \vee B_n$, as steps 1 and 2 in the algorithm in Section 3.3.3. If this combined rule, $\bigcirc A \Rightarrow B_0 \vee B_1 \vee \ldots \vee B_n$, fails these tests again add the rule to the relevant sublist for each literal, constructed in step 1, such that

   $$l \in \mathrm{lits}(\bigvee_{j=0}^{n} B_j)$$

   for $j \neq k$ where $B_k \Rightarrow H_i$. Remove any rules that are equivalent from each set.

   (c) Repeat steps (a) and (b) until we have taken all possible combinations of rules extracting one from each R_l.

Continuing from the example given in Section 3.3.3 where $H_0 = (r \wedge q) \vee (s \wedge q)$ we build sets R_r, R_q and R_s by collecting the rules whose right hand sides contain the literals r, q and s respectively. Sets R_r and R_s will both contain rule 3 and R_q contains rule 4. Now for the first disjunct in H_0, $(r \wedge q)$, we take all possible combinations of rules, taking a rule from R_r and a rule from R_q, so in this case we combine rules 3 and 4. We then repeat for the other disjunct and combine rules 3 and 4. Thus we obtain the combined rule

$$\bigcirc (r \wedge q) \Rightarrow (q \wedge r) \vee (q \wedge s)$$

which can now be used to expand the graph.

3.3.5 Simplification of Nodes

Here we note the simplification rules used to manipulate the nodes in DNF that the Breadth-First Algorithm builds. Assume that the node is in DNF and that each disjunct is in its simplest form. This will have been carried out during the merging of rules into SNF_m. Let X and Y be formulae in DNF (or **false**), A and B be conjunctions of literals (or **true**) and l a literal. Note that ordering of disjuncts makes no difference, i.e. we can apply the rules $X \vee Y \longrightarrow Y \vee X$ without affecting the truth value of the formula. When a node is constructed apply the following simplification rules.

1. Subsumption: $X \vee A \vee B \longrightarrow X \vee A$ iff $\vdash B \Rightarrow A$
2. Complementary literals in different disjuncts (known as *left hand side resolution* later in the thesis): $X \vee (A \wedge l) \vee (B \wedge \neg l) \longrightarrow X \vee (A \wedge l) \vee (B \wedge \neg l) \vee (A \wedge B)$ where the conjunction $(A \wedge B)$ may need further simplification.

3.3.6 Example

Breadth-First Search is used to detect the loop in the set of rules given in Section 3.2.9. Recall we are trying to resolve with the rule

$$\mathcal{L}Q \Rightarrow \Diamond l$$

and we repeat the rules for reference.

$$\begin{array}{llll} 1. & \bigcirc x & \Rightarrow & \neg l \\ 2. & \bigcirc a & \Rightarrow & \neg l \\ 3. & \bigcirc b & \Rightarrow & c \vee d \\ 4. & \bigcirc c & \Rightarrow & a \end{array} \qquad \begin{array}{llll} 5. & \bigcirc d & \Rightarrow & a \\ 6. & \bigcirc a & \Rightarrow & x \\ 7. & \bigcirc x & \Rightarrow & b \end{array}$$

To create a new node H_i, we examine each rule in turn, disjoining the conjunction of literals on the left hand to the new node if the rule satisfies the criteria given. The new node is simplified when we have examined every rule. To avoid confusion, as the node is being constructed we will call it H_i^*, only labelling the complete simplified node by H_i.

1. The rules 1 and 2 both 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

   $$H_0 = x \vee a$$

   and note that $\mathrm{lits}(H_0) = \{x, a\}$.

2. To build the next node, H_1, we examine each rule (other than rules 1 and 2) in turn using the algorithm given in Section 3.3.3.

   Rule 3 As $c \vee d \not\Rightarrow H_0$ and $\mathrm{lits}(c \vee d) \not\subseteq \mathrm{lits}(H_0)$ rule 3 is discarded.

Rule 4 As $a \Rightarrow H_0$ but $c \not\Rightarrow H_0$ we combine rule 4 with rules 1 and 2 to give

$$\begin{array}{lrcl} 1+4. & \circledcirc(x \wedge c) & \Rightarrow & a \wedge \neg l \\ 2+4. & \circledcirc(a \wedge c) & \Rightarrow & a \wedge \neg l. \end{array}$$

Now $(a \wedge \neg l) \Rightarrow H_0$ in each case and both $(x \wedge c) \Rightarrow H_0$ and $(a \wedge c) \Rightarrow H_0$ so the left hand sides are disjoined and added to the new node to give

$$H_1^* = (x \wedge c) \vee (a \wedge c).$$

Rule 5 Similarly, $a \Rightarrow H_0$ but $d \not\Rightarrow H_0$ so rule 5 is combined with rules 1 and 2 to give

$$\begin{array}{lrcl} 1+5. & \circledcirc(x \wedge d) & \Rightarrow & a \wedge \neg l \\ 2+5. & \circledcirc(a \wedge d) & \Rightarrow & a \wedge \neg l. \end{array}$$

The left hand sides are disjoined and added to the new node to give

$$H_1^* = (x \wedge c) \vee (a \wedge c) \vee (x \wedge d) \vee (a \wedge d).$$

Rule 6 Here we have $x \Rightarrow H_0$ and $a \Rightarrow H_0$ so we can add the left hand side to our node directly to give

$$H_1^* = (x \wedge c) \vee (a \wedge c) \vee (x \wedge d) \vee (a \wedge d) \vee a.$$

This step means we can perform some simplification on the new node but we will delay this until we have processed all the rules.

Rule 7 This is similar to rule 3 as $b \not\Rightarrow H_0$ and $\text{lits}(b) \nsubseteq \text{lits}(H_0)$, so rule 7 is discarded.

We simplify H_1^* giving

$$H_1 = (x \wedge c) \vee (x \wedge d) \vee a$$

and note that $\text{lits}(H_1) = \{a, c, d, x\}$

3. We now start trying to build node H_2.

Rule 3 Using rule 3 $(c \vee d) \not\Rightarrow H_1$ but $\text{lits}(c \vee d) \subseteq \text{lits}(H_1)$ so we add rule 3 to a list to be used for combinations.

Rule 4 We have $a \Rightarrow H_1$ but $c \not\Rightarrow H_0$ so as previously rule 4 is combined with rules 1 and 2 and the left hand sides of rules $1+4$ and $2+4$ are disjoined to form the new node, i.e.

$$H_2^* = (x \wedge c) \vee (a \wedge c).$$

Rule 5 Similarly, $a \Rightarrow H_1$ but $d \not\Rightarrow H_0$ so we combine rule 5 with rules 1 and 2 and add the left hand sides of the combined rules to give

$$H_2^* = (x \wedge c) \vee (a \wedge c) \vee (x \wedge d) \vee (a \wedge d)$$

Rule 6 As $x \not\Rightarrow H_1$ but $\text{lits}(x) \subseteq \text{lits}(H_1)$ we save rule 6 for combinations.

Rule 7 As $b \not\Rightarrow H_1$ and $\text{lits}(b) \nsubseteq \text{lits}(H_1)$ we discard rule 7.

We have rules 3 and 6 remaining to be used for combinations. Combining them we obtain

$$3+6 \qquad \circledcirc(a \wedge b) \Rightarrow (c \wedge x) \vee (d \wedge x).$$

Now $(c \wedge x) \vee (d \wedge x) \Rightarrow H_1$ and $(a \wedge b) \Rightarrow H_0$ so we add $(a \wedge b)$ as a disjunct to the new node and simplify to give

$$H_2 = (x \wedge c) \vee (a \wedge c) \vee (x \wedge d) \vee (a \wedge d) \vee (a \wedge b)$$

and note that $\text{lits}(H_2) = \{a, b, c, d, x\}$.

4. We start trying to build the next node, H_3. For each of rules 3–7 of the form $\bigcirc X \Rightarrow Y$, $Y \not\Rightarrow H_2$ but $\text{lits}(Y) \subseteq \text{lits}(H_2)$, so all are used for combinations.
   (a) To try to match the first disjunct $(x \wedge c)$ of node H_2 we look for rules with either x or c on the right hand side to combine together. We find that rules 3 and 6 are suitable and combining them obtain
   $$3+6 \quad \bigcirc(a \wedge b) \Rightarrow (c \wedge x) \vee (d \wedge x)$$
   as before. Now $(c \wedge x) \vee (d \wedge x) \Rightarrow H_2$ and $(a \wedge b) \Rightarrow H_0$ so $(a \wedge b)$ is added as a disjunct of the new node giving
   $$H_3^* = (a \wedge b).$$
   (b) Taking the second disjunct of node H_2, $(a \wedge c)$, we try combining the rules $3+4$ and $3+5$.
   $$\begin{array}{lllll} 3+4 & \bigcirc(b \wedge c) & \Rightarrow & (a \wedge c) \vee (a \wedge d) \\ 3+5 & \bigcirc(b \wedge d) & \Rightarrow & (a \wedge c) \vee (a \wedge d) \end{array}$$
   For these two rules (in the form $\bigcirc X \Rightarrow Y$) $Y \Rightarrow H_2$ but $X \not\Rightarrow H_0$ so we combine both with rules 1 and 2 and add the combined left hand sides to the new node to give
   $$H_3^* = (a \wedge b) \vee (b \wedge c \wedge x) \vee (b \wedge c \wedge a) \vee (b \wedge d \wedge x) \vee (b \wedge d \wedge a)$$
   (c) For the next disjunct of H_2, $(x \wedge d)$, we combine rules 3 and 6 as we have done in (a) above.
   (d) For the disjunct $(a \wedge d)$ we combine rules $3+4$ and $3+5$ and then combine each with rules 1 and 2 as we have done in step (b) above.
   (e) For $(a \wedge b)$ we combine rules $4+7$ and $5+7$ to give
   $$\begin{array}{llll} 4+7 & \bigcirc(c \wedge x) & \Rightarrow & (a \wedge b) \\ 5+7 & \bigcirc(d \wedge x) & \Rightarrow & (a \wedge b) \end{array}$$
   so for both rules $Y \Rightarrow H_2$ and $X \Rightarrow H_0$ and the new node becomes
   $$\begin{array}{rl} H_3^* = & (a \wedge b) \vee (b \wedge c \wedge x) \vee (b \wedge c \wedge a) \vee (b \wedge d \wedge x) \\ & \vee (b \wedge d \wedge a) \vee (c \wedge x) \vee (d \wedge x) \end{array}$$

   We simplify this to give
   $$H_3 = (a \wedge b) \vee (c \wedge x) \vee (d \wedge x)$$
   and note $\text{lits}(H_3) = \{a, b, c, d, x\}$.

5. We now try to build node H_4. Again we must use all rules for combinations.
   (a) For the first disjunct, $(a \wedge b)$, from node H_3 we combine the rules $4+7$ and $5+7$ (as in 4(e)) to give
   $$\begin{array}{llll} 4+7 & \bigcirc(c \wedge x) & \Rightarrow & (a \wedge b) \\ 5+7 & \bigcirc(d \wedge x) & \Rightarrow & (a \wedge b) \end{array}$$
   so for both rules $Y \Rightarrow H_3$ and $X \Rightarrow H_0$ and the new node becomes
   $$H_4^* = (c \wedge x) \vee (d \wedge x)$$
   (b) For the disjunct $(c \wedge x)$, from node H_3, we combine the rules $3+6$ as previously in 4(a) and add the left hand side as a new disjunct to the node giving
   $$H_4^* = (c \wedge x) \vee (d \wedge x) \vee (a \wedge b)$$
   (c) The same rules as in 4(c), $3+6$ need to be combined for the disjunct $(d \wedge x)$.

   Simplifying the node we obtain
   $$H_4 = (c \wedge x) \vee (d \wedge x) \vee (a \wedge b).$$

As $H_4 \Leftrightarrow H_3$ we terminate having detected a loop. The Breadth-First loop we have found is $(c \wedge x) \vee (d \wedge x) \vee (a \wedge b)$. The graph constructed using Breadth-First Search for this set of rules is shown in Figure 3.5.

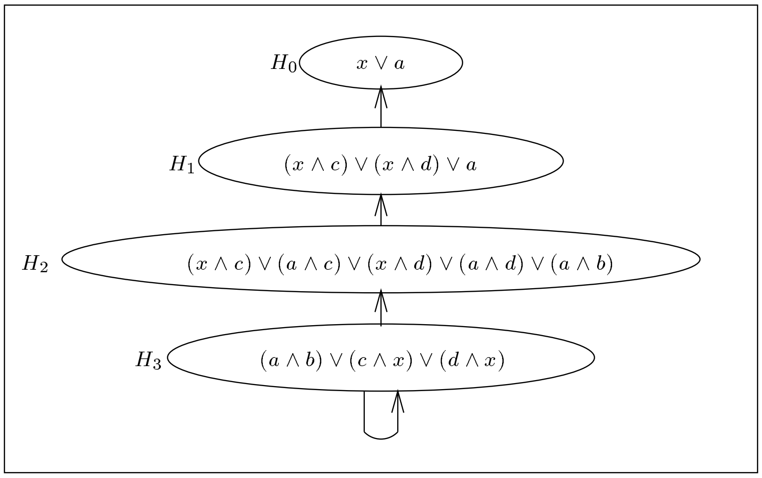

Figure 3.5: An Example Using Breadth-First Search

3.3.7 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.

$$\bigcirc \bigvee_i D_i \Rightarrow \Box \neg l.$$

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

$$(c \wedge x) \vee (d \wedge x) \vee (a \wedge b)$$

and the resolvents obtained are

$$\begin{array}{rcl} \bullet\,\mathbf{true} & \Rightarrow & \neg Q \vee (\neg(a \wedge b) \wedge \neg(c \wedge x) \wedge \neg(d \wedge x)) \\ \mathcal{L}Q & \Rightarrow & (\neg(a \wedge b) \wedge \neg(c \wedge x) \wedge \neg(d \wedge x))\, \mathcal{W}\, l \end{array}$$

and must be further translated into SNF. We note that in this example the resolvents are identical to those produced during Depth-First Search but this may not necessarily be the case (see Section 3.4.1 for more details).

3.3.8 Inclusion in the Main Processing Cycle

Breadth-First search finds *all* loops (see Section 3.4.1) 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). Criteria to test whether loops provide no new information are given in Section 5.3.2.

3.4 Comparison of Searches

Here we compare Breadth and Depth-First Searches. Section 3.4.1 describes the differences between their operation and output and Section 3.4.2 gives their relative timings. Section 3.4.3 examines which search performs better on two different rule-sets and why.

3.4.1 Description of Similarities and Differences.

Structures Produced

The paths constructed using the Depth-First Algorithm has simpler nodes (nodes are labelled with sets of literals) whereas the Breadth-First Algorithm produces nodes that are labelled with formulae in DNF. However, the structure of the graph produced using the Depth-First Algorithm is more complex, requiring branching to represent disjunctions on the right hand side of formulae in SNF_m. Further two search modes, forwards and backwards, are required to construct paths through such graphs. The graph produced from the Breadth-First Search, although having more complicated nodes, has no branching and does not require several search modes. Using Depth-First Search we need to backtrack when the expansion of the path using a particular choice of rules has proved to be fruitless, however using Breadth-First Search we have no branching and therefore do not need to backtrack.

Addition of Rules to the Rule-Set

As mentioned in Section 3.2.8 when using the Depth-First Algorithm we must add a rule

$$\bigcirc\mathbf{true} \Rightarrow a \vee \neg a$$

before each cycle of temporal resolution for any pair of complementary literals, a and $\neg a$, occurring on the left hand sides of any rules. We realise that this is far from ideal as we are trying to use strategies that, in general, keep the number of rules down to a minimum rather than adding extra rules.

However, when using Breadth-First Search such extra rules do not need to be added to the rule-set. Now complementary literals occurring on the left hand sides of rules do not cause problems as the simplification of the (disjunctive) nodes achieves the same result through left hand side resolution and rewriting $a \vee \neg a$ as **true**.

Example The example given in Section 3.2.8 considers the detection of the loop

$$\begin{array}{rcl} \bigcirc\neg a & \Rightarrow & \neg l \\ \bigcirc a & \Rightarrow & \neg l. \end{array}$$

when resolving with $\bullet\mathbf{false} \Rightarrow \Diamond l$. Invoking Breadth-First Search the top node is set to $\neg a \vee a$ which is simplified to **true** and returned.

In Depth-First Search we would start a backwards search from either $\{\neg a\}$ or $\{a\}$. Here let us take $\{\neg a\}$ (see Figure 3.6a). We may expand back by combining the added rule $\bigcirc\mathbf{true} \Rightarrow a \vee \neg a$ with $\bigcirc\neg a \Rightarrow \neg l$ to give $\bigcirc\neg a \Rightarrow (a \wedge \neg l) \vee (\neg a \wedge \neg l)$ (see Figure 3.6b). Now expanding forward from $\{a, \neg l\}$ we combine $\bigcirc\mathbf{true} \Rightarrow a \vee \neg a$ with $\bigcirc a \Rightarrow \neg l$ to give $\bigcirc a \Rightarrow (a \wedge \neg l) \vee (\neg a \wedge \neg l)$ and replace $\{a, \neg l\}$ by $\{a\}$ to complete the loop and return $\{\{\neg a\}, \{a\}\}$ (see Figure 3.6c).

Size of Loops Detected

The Breadth-First Algorithm finds *all* the loops for a particular $\Diamond$–rule but the Depth-First Search finds loops one at a time. We will illustrate with a small example.

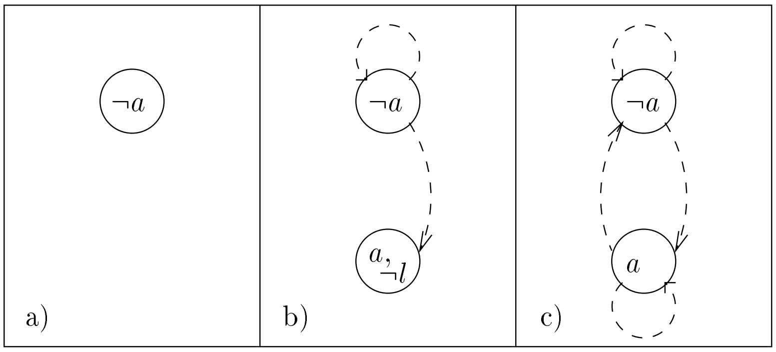

Figure 3.6: Depth-First on an Example with Rules Added

Example Given that we are trying to resolve with $\mathcal{L}Q \Rightarrow \Diamond l$ and we have rules

$$\begin{array}{rcl} \circledcirc a & \Rightarrow & a \\ \circledcirc a & \Rightarrow & \neg l \\ \circledcirc b & \Rightarrow & b \\ \circledcirc b & \Rightarrow & \neg l \end{array}$$

it is clear that the three SNF$_m$ rules

$$\begin{array}{rcl} \circledcirc a & \Rightarrow & a \wedge \neg l \\ \circledcirc b & \Rightarrow & b \wedge \neg l \\ \circledcirc (a \wedge b) & \Rightarrow & (a \wedge b \wedge \neg l) \end{array}$$

all satisfy our criteria for a loop separately (and in any combination). Performing Breadth-First Search we return the formula $a \vee b$ (because of simplification we will never return $a \vee b \vee (a \wedge b)$). Performing a Depth-First Search we may return $\{\{a\}\}$ or $\{\{b\}\}$ or $\{\{a, b\}\}$ or any other set of sets of literals drawn from the union of the members of these three sets. However, attempting to expand with individual rules before combined rules means that $\{\{a\}\}$ or $\{\{b\}\}$ will always be detected before $\{\{a, b\}\}$. See Section 5.2.1 for more detail.

So, by using the Breadth-First Algorithm we may run the algorithm once, rewrite the resolvents into SNF and obtain a contradiction during step resolution. However running the same example using Depth-First Search we may have to run the Depth-First Algorithm more than once (i.e. cycle round the main processing loop of SNF, step resolution and temporal resolution more than once) to obtain a contradiction. This is because we need to detect both the loops $\{\{a\}\}$ and $\{\{b\}\}$ to produce the same resolvents as those obtained from the output of Breadth-First Search $a \vee b$. Detecting the loop $\{a, b\}$ with Depth-First Search, for example would produce resolvents equivalent to detecting $a \wedge b$ in Breadth-First Search i.e. resolvents that are weaker than for $a \vee b$. See Section 5.2.3 for an example of the resolvents obtained from loops $\{a\}$ and $\{a, b, c\}$.

The 'Lead in' to the Loop

The Breadth-First Algorithm returns a DNF formula, part of which may correspond to rules that can be thought of as forming a path that *lead into* the Depth-First loop proper.

Example Consider the following set of rules illustrated in Figure 3.7 as a Depth-First Search style graph.

$$\begin{array}{rrcl}
1 & \bullet\,\mathbf{false} & \Rightarrow & \Diamond l \\
2 & \circledcirc a & \Rightarrow & \neg l \\
3 & \circledcirc b & \Rightarrow & \neg l \\
4 & \circledcirc a & \Rightarrow & a \\
5 & \circledcirc b & \Rightarrow & a
\end{array}$$

Output from Breadth-First Search for this set of rules would be the DNF-formula $a \vee b$ whereas output

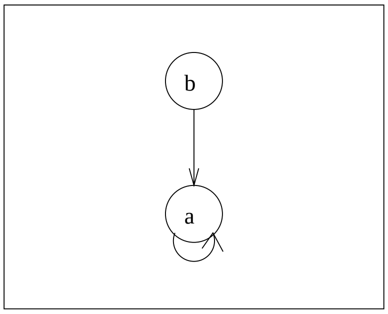

Figure 3.7: An Example Showing the Lead in to a Loop

from Depth-First Search would be the set of nodes $\{\{a\}\}$. The rules representing the loop detected using Breadth-First Search are

$$\begin{array}{rrcl}
2+4 & \circledcirc a & \Rightarrow & a \wedge \neg l \\
3+5 & \circledcirc b & \Rightarrow & a \wedge \neg l
\end{array}$$

but we would only find the first of these rules to represent the Depth-First loop. This is because the second of these SNF$_m$ rules represents the path into the loop (there are no rules to take us back to the $\{b\}$ node). Both systems are still complete but having found the Depth-First loop we will have to carry out further applications of the step resolution rule.

For example, if we add the two initial $\square$–rules

$$\begin{array}{rrcl}
6 & \bullet\,\mathbf{false} & \Rightarrow & \neg l \\
7 & \bullet\,\mathbf{false} & \Rightarrow & b
\end{array}$$

to our set so that we can obtain a contradiction, the loop from the above two rules output by Breadth-First Search will produce the resolvent

$$\bullet\,\mathbf{false} \Rightarrow (\neg a \wedge \neg b)\,\mathcal{W}\,l$$

Rewriting this into SNF we obtain the following rules, where t is a new proposition symbol.

$$\begin{array}{rrcr}
8 & \bullet\,\mathbf{false} & \Rightarrow & \neg a \vee l \\
9 & \bullet\,\mathbf{false} & \Rightarrow & \neg b \vee l \\
10 & \bullet\,\mathbf{false} & \Rightarrow & t \vee l \\
11 & \circledcirc t & \Rightarrow & \neg a \vee l \\
12 & \circledcirc t & \Rightarrow & \neg b \vee l \\
13 & \circledcirc t & \Rightarrow & t \vee l
\end{array}$$

To obtain a contradiction we apply the step resolution rule twice as follows.

$$\begin{array}{rrcll}
14 & \bullet\,\mathbf{false} & \Rightarrow & \neg b & [6+9 \text{ step resolution}] \\
15 & \bullet\,\mathbf{false} & \Rightarrow & \mathbf{false} & [7+14 \text{ step resolution}]
\end{array}$$

If we had done a Depth-First Search however we would obtain the resolvent

$$\bullet\,\mathbf{false} \Rightarrow \neg a\,\mathcal{W}\,l$$

and rewriting this into SNF would produce rules 8, 10, 11, and 13 (but not 9 and 12). In this case the proof leading to a contradiction is

$$\begin{array}{llcll}
14 & \bullet\,\mathbf{false} & \Rightarrow & t & [6{+}10 \text{ step resolution}] \\
15 & \circledcirc(t \wedge b) & \Rightarrow & l & [5{+}11 \text{ step resolution}] \\
16 & \circledcirc(t \wedge b) & \Rightarrow & \mathbf{false} & [3{+}15 \text{ step resolution}] \\
17 & \circledcirc\,\mathbf{true} & \Rightarrow & \neg t \vee \neg b & [16 \text{ rewrite to SNF}] \\
18 & \bullet\,\mathbf{false} & \Rightarrow & \neg t \vee \neg b & [16 \text{ rewrite to SNF}] \\
19 & \bullet\,\mathbf{false} & \Rightarrow & \neg b & [14{+}18 \text{ step resolution}] \\
20 & \bullet\,\mathbf{false} & \Rightarrow & \mathbf{false} & [7{+}19 \text{ step resolution}]
\end{array}$$

that is we have used five applications of the step resolution rule to reach a contradiction where as previously we only required two.

3.4.2 Timings

A prototype implementation performing Fisher's temporal resolution method has been built. A subprogram performs each of the three parts of the temporal resolution method. Two different loop search programs have been written, one for Depth-First Search and one for Breadth-First Search. The programs are written in SICStus Prolog [CW91] running under UNIX and timings have been carried out on a SPARCstation 1 using compiled Prolog code. The full set of results are given in Appendix B.1 however we present a summary of the data here. Table 3.1 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 looking at these values we hope to eliminate inaccuracies from very low timings.

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

Table 3.1: Summary of Timing from Depth and Breadth-First Search

Observations

The figures in Table 3.1 indicate that Breadth-First Search performs better in more examples that 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 that Breadth-First Search does better on larger examples.

Some of the timings have values of 0 which seems to indicate that the accuracy of the smaller timings is not very good. This is why we have excluded the examples with small timings in Table 3.1. The timings for Depth-First Search have a larger range of values (from 0 to 11038 milliseconds) than Breadth-First Search (from 0 to 1859 milliseconds). 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 lot of time is spent exploring fruitless paths then the time may be far greater than that for Breadth-First Search.

The number of calls to each eventuality in each example (i.e. the number of times we look for loops for that eventuality) is 1 for Breadth-First Search except for Example 39 $\Diamond w_1$ and Example 47 $\Diamond t_3$ where it is 2. This is what we would expect as Breadth-First Search finds *all* loops for an eventuality. We may have to search for loops for an eventuality more than once in Breadth-First Search when loops for the eventuality depend on resolvents from resolution with this or other eventualities. Depth-First Search however has examples that enter the loop search up to 5 times. Table 3.2 shows the number of examples for each number of calls to the loop search by the Depth-First Search Algorithm. The number of examples and percentage of each group is given where the time for Breadth-First Search is less than the time for Depth-First Search. The table indicates that for this set of examples the more

	1	2	3	4	5
Total	25	22	0	2	1
BFS < DFS (No.)	13	14	-	2	1
BFS < DFS (%)	52%	64%	-	100%	100%

Table 3.2: Summary by Number of Calls Depth-First Search

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 the 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.

3.4.3 A Comparison via Two Rule-Sets

Given the two loop search algorithms described earlier in the chapter it may be useful to analyse what type of loops each performs best on. Then, one loop search algorithm could be used in preference to another on a particular set of rules.

Here, we take two loops L_1 and L_2 produced from rules of different structures and investigate whether one loop is detected quicker by Depth-First Search than Breadth-First Search and vice versa for the other. For discussion purposes we will divide the SNF rules that we merge to make a loop into two sets, those that generate the literal we require and those that provide the looping structure—the latter being known as rules that *propagate* the loop. Here the SNF rules, R_1, that *propagate* the loop for L_1 are rules with a single literal on the left hand side and a disjunction of n literals on the right hand side for some $n \geqslant 1$, and the SNF rules, R_2, that *propagate* the loop for L_2 are rules with a single literal on the right hand side and a conjunction of m literals on the left hand side for $m \geqslant 1$.

Rules Used

Let us assume we are looking for a loop in $\neg l$ and both sets of rules, R_1 and R_2 include the following rules, each of which generate the literal $\neg l$ we require.

$$\begin{array}{rcl} \circledcirc\, l_1 & \Rightarrow & \neg l \\ \circledcirc\, l_2 & \Rightarrow & \neg l \\ \dots & & \dots \\ \circledcirc\, l_{11} & \Rightarrow & \neg l \end{array}$$

In addition the set of rules R_1 that will give us loop L_1 will be composed of the rules

$$\begin{array}{rlcl} 1. & \bigcirc l_1 & \Rightarrow & l_2 \vee l_3 \vee \ldots \vee l_{10} \vee l_{11} \\ 2. & \bigcirc l_2 & \Rightarrow & l_3 \vee l_4 \vee \ldots \vee l_{11} \vee l_1 \\ & \ldots & & \ldots \\ 11. & \bigcirc l_{11} & \Rightarrow & l_1 \vee l_2 \vee \ldots \vee l_9 \vee l_{10} \end{array}$$

and the set of rules R_2 that will give us loop L_2 will be composed of the rules

$$\begin{array}{rrcl} 1. & \bigcirc(l_1 \wedge l_2 \wedge \ldots \wedge l_{10}) & \Rightarrow & l_{11} \\ 2. & \bigcirc(l_2 \wedge l_3 \wedge \ldots \wedge l_{11}) & \Rightarrow & l_1 \\ & \ldots & & \ldots \\ 11. & \bigcirc(l_{11} \wedge l_1 \wedge \ldots \wedge l_9) & \Rightarrow & l_{10} \end{array}$$

Proposition 1 *Breadth-First Search will find loop L_1 more efficiently than Depth-First Search and Depth-First Search will find loop L_2 more efficiently than Breadth-First.*

Timings

The two rule-sets were processed using both Depth and Breath-First Searches and the time (in milliseconds) for each is given in Table 3.3.

Algorithm	R_1	R_2
Depth	450	1959
Breadth	250	7630

Table 3.3: Timings (in milliseconds) of Two Loops with Different Algorithms

Analysis of Search

The timings we have obtained back up the proposition. To identify why this is the case, we examine the behaviour of each search algorithm with each rule-set.

Depth-First with Rule-Set R_1 Diagrams showing a Depth-First Search on rule-set R_1 are given in Figure 3.8.

1. First we search backwards from a node, say $\{l_1\}$, looking for a rule with l_1 on the right hand side, say rule 2. For this rule-set this requires the examination of at most 2 rules
2. Backwards search from $\{l_2\}$ will match a rule with l_2 on the left hand side with this, say rule 1, giving us $\{l_1\}$ through which we have searched already. Again this requires the examinatuin of at most 2 rules.
3. We then must search through all the remaining disjuncts until the loop is closed. As now we are doing forwards search this involves examining at worst 11 rules for each remaining disjunct (9 of them).

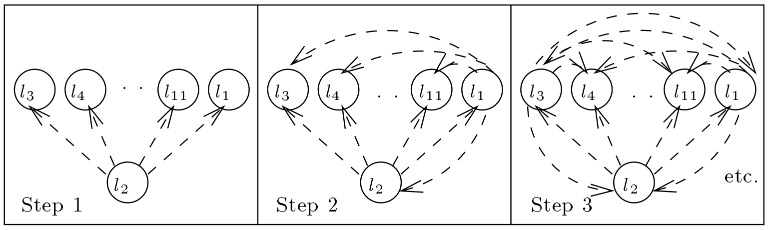

Figure 3.8: Depth-First Search Using Rule-Set R_1

Depth-First with Rule-Set R_2 Diagrams showing a Depth-First Search on rule-set R_2 are given in Figure 3.9.

1. The search starts from the left hand side of one of the rules that give us $\neg l$, say $\{l_1\}$. We match this to the right hand side of a rule, rule 2.
2. Then as no rule matches the new node

$$\{l_2, l_3, l_4, l_5, l_6, l_7, l_8, l_9, l_{10}, l_{11}\}$$

we combine rules 1, and 3–11 (i.e. we must do 9 combinations) to give us the merged SNF rule

$$\circledcirc (l_1 \wedge l_2 \wedge \ldots l_{10} \wedge l_{11}) \Rightarrow l_2 \wedge l_3 \wedge \ldots \wedge l_9 \wedge l_{10} \wedge l_{11}$$

the left hand side of which becomes the new node.

3. To match this we must combine rules 1–11 (i.e. 10 combinations) giving us the merged SNF rule

$$\circledcirc (l_1 \wedge l_2 \wedge \ldots \wedge l_{10} \wedge l_{11}) \Rightarrow l_1 \wedge l_2 \wedge \ldots \wedge l_{10} \wedge l_{11}$$

which completes our loop. We have had to perform 19 combinations in total.

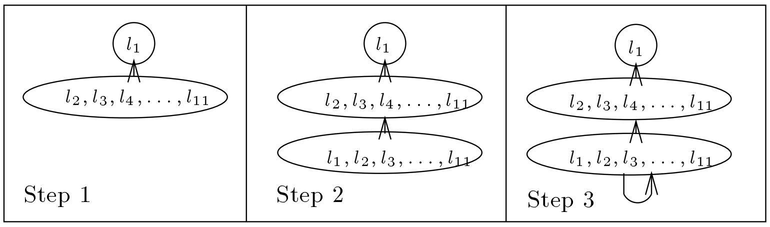

Figure 3.9: Depth-First Search Using Rule-Set R_2

Breadth-First with Rule-Set R_1 Diagrams showing a Breadth-First Search on rule-set R_1 are given in Figure 3.10.

1. The top node H_0 disjoins all the literals that give us a $\neg l$.
2. The second node is made by disjoining the left hand sides of all the rules whose right hand side imply the previous node, i.e. rules 1–11. This turns out to be the same as node H_0 so we terminate having found a loop. Here we examine each rule once, i.e. we have examined 11 rules.

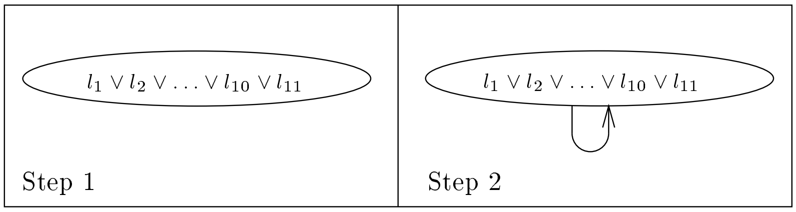

Figure 3.10: Breadth-First Search Using Rule-Set R_1

Breadth-First with Rule-Set R_2 Diagrams showing a Breadth-First Search on rule-set R_2 are given in Figures 3.11, 3.12 and 3.13.

1. The top node is constructed as previously. Again, the second node is generated by using rules whose right hand side implies the top node (and whose left hand side implies the top node). We use rules 1–11 of rule-set R_2 and disjoin their left hand sides noting that no combinations of rules were required to produce this node.

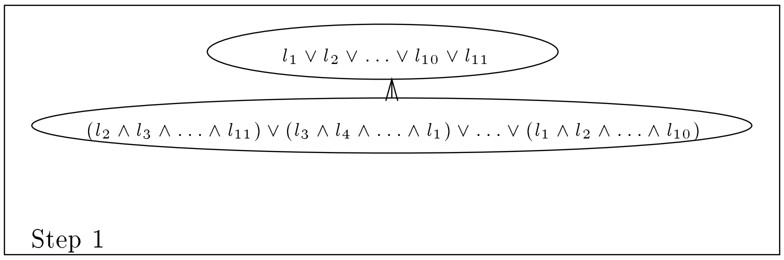

Figure 3.11: Breadth-First Search Using Rule-Set R_2 : Step 1

2. Now for each disjunct there are ten literals so we must combine the relevant ten rules for each disjunct. For example, for the disjunct $(l_2 \wedge l_3 \wedge l_4 \wedge l_5 \wedge l_6 \wedge l_7 \wedge l_8 \wedge l_9 \wedge l_{10} \wedge l_{11})$ we must combine rules 1 and 3–11 giving an SNF$_m$ rule

$$\circledcirc\,(l_1 \wedge l_2 \wedge \ldots \wedge l_{11}) \Rightarrow l_2 \wedge l_3 \wedge \ldots \wedge l_{11}.$$

This is done for each disjunct and we simplify the node to get $l_1 \wedge l_2 \wedge l_3 \wedge l_4 \wedge l_5 \wedge l_6 \wedge l_8 \wedge l_9 \wedge l_{10} \wedge l_{11}$. Here we have had to combine 10 rules (i.e. 9 combinations) for each of the 11 disjuncts making a total of 99 combinations.

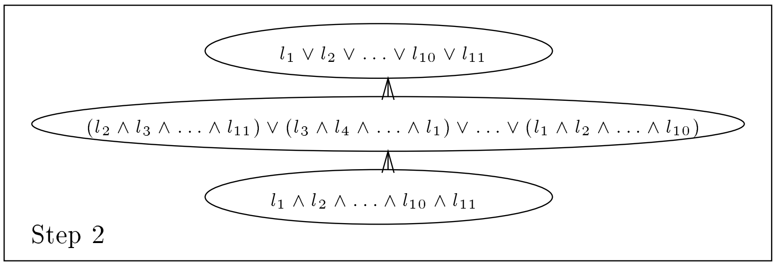

Figure 3.12: Breadth-First Search Using Rule-Set R_2 : Step 2

3. Finally all the rules 1–11 must be combined to detect this is the final node requiring 10 combinations. In total we have performed $0 + (9 \times 11) + 10 = 109$ combinations.

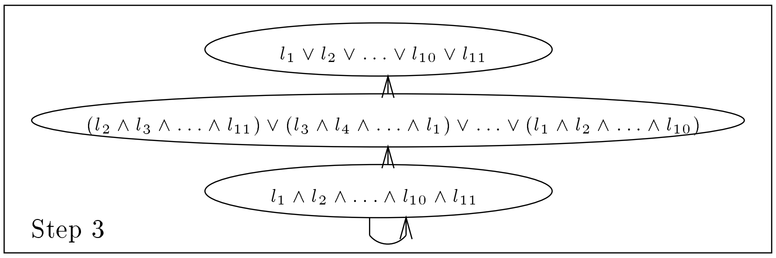

Figure 3.13: Breadth-First Search Using Rule-Set R_2 : Step 3

Extending the Example

The experiment was performed again using different numbers of rules. The results of the timings are given in Appendix B.2 in Table B.2 and graphs of the results are shown in Figures 3.14 and 3.15.

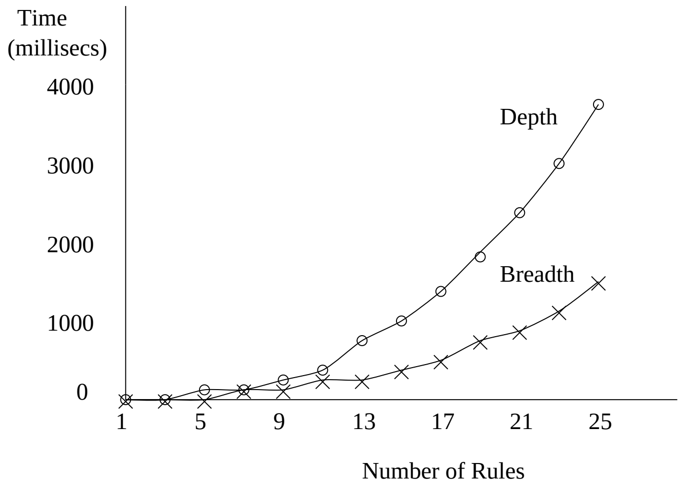

Figure 3.14: Timings for Loop Searches on Rule-Set R_1

Comment The results from increasing the number and size of the rules still back up the proposition.

Conclusions

These two rule-sets suggest that Breadth-First Search performs better on rules with disjuncts on their right hand sides (R_1). This is because the nodes in Breadth-First Search graphs are in DNF. Depth-First Search requires more time for such loops as rules with disjuncts on the right hand side require branching in the graph created and each branch must be checked to see whether it leads out of the loop. Further, we have to search the rule-set more times using Depth-First Search to find a rule that will match the conditions for forwards search. Breadth-First Search examines each rule just once to build the second node.

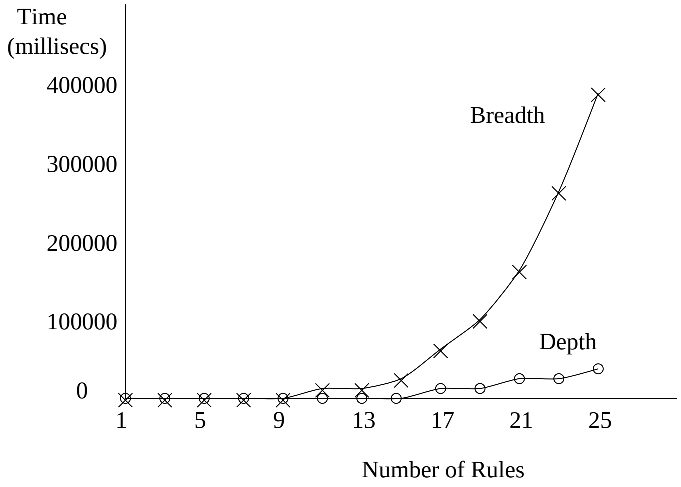

Figure 3.15: Timings for Loop Searches on Rule-Set R_2

Depth-First Search performs better than Breadth-First Search on the ruleset with conjunctive left hand sides, R_2, as no choice is required. The Depth-First Search Algorithm takes just one path to the solution rather than *every* path and thus has to perform fewer combinations (in our example Depth-First Search requires 19 combinations and Breadth-First Search requires 109).

Finally, although this section provides insight into which loops are detected more easily with each loop search and why, we note that the two loops analysed are extreme examples. Loops through sets of rules are likely to be a combination of rules some that have conjunctions on the left hand side, others having disjunctions on the right hand side and others having both. Further, other factors do come into play, discussed in Section 3.4.2, for example that Breadth-First Search finds all the loops for an eventuality where as Depth-First Search does not. Using Depth-First Search the same portion of the graph may be searched before any 'new' loops are detected.

Chapter 4

Correctness Issues

We begin this chapter with a summary of Peim's completeness proof for temporal resolution. Then, the three algorithms that detect formulae for resolution with a $\Diamond$–rule, given in Chapter 3, are shown to be sound and complete. Completeness for the MTSA is shown in relation to the graph built for this algorithm—the reduced behaviour graph for the set of global $\Box$–rules. Completeness for the Breadth-First Search Algorithm is proved by relating each step in the Breadth-First Search Algorithm to sets of nodes in the reduced behaviour graph for the set of global $\Box$–rules. Completeness for Depth-First Search is in turn shown by relating the search performed using the Depth-First Search Algorithm to the graph constructed using the Breadth-First Search Algorithm.

4.1 Correctness of Temporal Resolution

Completeness for Fisher's temporal resolution system is shown in [Pei94]. However Peim proves completeness for a more general system termed a *Clausal Transition System* (CTS). In a CTS, to prove properties about a program, rather than translating both the program control and data information into a normal form, the original program structure is retained by encoding the program control as an automaton and the data as PTL formulae related to transitions in the automaton. So, a CTS is an automaton with sets of SNF rules and an enabling condition (or *guard*) attached to the transitions. The guard, a PTL formula, determines which transition can be taken from a state and the SNF rules describe the change in the program variables.

We note that Fisher's system is merely a special case of this with one state and a transition from this state to itself labelled by the guard or enabling condition **true**. We give an outline of the proof modified for the special case we are considering here. This is useful as Peim's graph construction is the basis of both one of the loop search algorithms described in Section 3.1 and the completeness proofs for the loop search algorithms given later in this chapter. Further, it illustrates the relationship between Fisher's resolution method and the ability to construct (or not) a model for a set of SNF rules.

In Peim's original presentation, completeness is shown by induction on the size of a *behaviour graph* for a CTS. Given a CTS, nodes are triples (s, V, Y) where s is a state, V a valuation of all the propositions in the CTS and Y is a subset of the literals that occur on the right hand side of $\Diamond$–rules in the CTS. However, as stated above, for a set of SNF rules, T, the system we are considering has only one state so we remove the state component and our nodes are pairs (V, Y). Given a node (V, Y), let R' be the largest subset of the global $\Box$-rules such that the valuation, V, satisfies the literals on the left hand sides of each rule. Let C be the set of clauses on the right hand side of the rules in R' and V' be a valuation that satisfies these clauses. Let $Y' \subseteq Y$ be the set of literals not satisfied by V and Y'' be the set of literals from the right hand side of the global $\Diamond$–rules whose left hand sides are satisfied by the valuation V. Edges are constructed from (V, Y) to $(V', Y' \cup Y'')$ for each valuation V' and these are the only edges out of (V, Y).

As stated previously, this graph is similar to the one described in Section 3.1 in which loops are detected for resolution with a $\Diamond$–rule. However, the graph in Section 3.1 is built from the set of global $\Box$-rules alone as it is in this set of rules we look for loops.

The set of *initial nodes* is identified by those nodes (V, Y) where the valuation V satisfies the set of clauses on the right hand side of the set of initial $\Box$-rules and where Y is the set of literals from the right hand side of the initial $\Diamond$–rules. The *behaviour graph* for a set of SNF rules T is the set of nodes and edges reachable from the initial nodes.

Next, the following deletions are performed on the behaviour graph for T until no further deletions are possible.

- If a node has no edges leading from it delete this node and all edges into it.
- If a node (V, Y) contains an eventuality $l \in Y$ and l is neither satisfied by V nor is there a node reachable from (V, Y) whose valuation satisfies l then (V, Y) is deleted.

The resulting graph is known as the *reduced behaviour graph* for T. From [Pei94] we have the following proposition (adapted for our special case).

Proposition 2 *A set of SNF rules is unsatisfiable if and only if its reduced behaviour graph is empty.*

This is proved by showing that the deletions made to the (unreduced) behaviour graph cannot remove any models (infinite sequences of propositional valuations) so that an empty behaviour graph means that there are no models and the set of SNF rules must be unsatisfiable. Further, if the behaviour graph is not empty then there must be a model so that the set of SNF rules is not unsatisfiable.

Peim tackles the problem of the introduction of new variables required for translating resolvents into SNF by introducing a new variable relating to each eventuality before any resolution takes place. Thus given an eventuality $\Diamond l$, Peim suggests the introduction of w_l as a new proposition meaning informally *waiting for l*. As noted in Section 2.2.3, if we are resolving with $\mathcal{L}Q \Rightarrow \Diamond l$, the two resolvents

$$\begin{array}{rcl} \mathcal{L}Q & \Rightarrow & w_l \lor l \\ \circledcirc w_l & \Rightarrow & w_l \lor l \end{array}$$

are unrelated to specific loops in l and may be added to T before any resolution is carried out. Peim calls this *augmenting* the rule-set and shows it does not affect completeness. Further, as there may be models of T where w_l does not have the meaning we intend, Peim defines *normal* models of T as a model satisfying $\Box(w_l \Leftrightarrow \bigcirc\Diamond l)$ and states that T is *well-behaved* if it has either no models or a normal model. We note that the augmentation of a set of SNF rules T is well-behaved and has a model if and only if T does and that Fisher's resolution method preserves well-behavedness. Peim then proves completeness for CTSs which for Fisher's system is the following.

Theorem 2 *(Completeness of Temporal Resolution). If a well-behaved augmented set of SNF rules is unsatisfiable, then it has a refutation by Fisher's resolution procedure.*

The proof is by induction on the number of nodes in the (unreduced) behaviour graph for T, an augmented well-behaved set of SNF rules. If the behaviour graph is empty then the initial $\Box$–rules must be unsatisfiable so there is a refutation by applying step resolution to the initial clauses. If the behaviour graph is not empty a node must be able to be deleted from the graph as described above.

If a terminal node (V, Y) exists, consider the set of global $\Box$-rules R' whose left hand side satisfy the valuation V. Then from the construction of the graph the set of clauses from the right hand side of R' must be unsatisfiable. In Fisher's resolution system this represents a series of applications of step resolution which lead to a rule with **false** on the right hand side, i.e. $\circledcirc X \Rightarrow \mathbf{false}$ where $V \models X$. This is rewritten as

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

Adding these rules to T giving T' and constructing the behaviour graph for T', no edges will be incident on (V, Y) because we have added the rule $\circledcirc\,\mathbf{true} \Rightarrow \neg X$. So any edges out of any node must lead to a node that satisfies $\neg X$. As $V \models X$ there can be no edges into (V, Y) in T'. So (V, Y) becomes unreachable and the graph for T' is a strict subset of the graph for T. By induction we assume that T' has a refutation and so must T.

Otherwise, if no terminal node exists there must be a node n that contains an eventuality l, where l is not satisfied in any node reachable from n. If N is the set of nodes reachable from n then any edges out of a node in N lead to a node that is also in N. For each node $n = (V, Y)$ in N the set of SNF rules whose left hand side is satisfied by V are combined to give $\bigcirc A_n \Rightarrow B_n$ for $n \in N$. This set of SNF$_m$ rules is shown to form a loop in l (for a similar proof see Section 4.2 Lemma 1). Adding the resolvents from temporal resolution with this loop and the eventuality l to T giving T' means that the behaviour graph for T' will have no edges into N. The only way that n may exist is if it is an initial node. Then the eventuality must be an initial $\Diamond$–rule and some of the resolvents are initial rules. It is shown that these are not satisfied by V so n is eliminated. So, the behaviour graph for T' is a strict subset of that for T and by induction we assume that as T' has a refutation so must T.

4.2 Correctness of the MTSA

Given a set T of SNF rules let

- $I \subseteq T$ be the set of initial $\Box$–rules;
- $R \subseteq T$ be the set of global $\Box$–rules; and
- $S \subseteq T$ be the set of $\Diamond$–rules.

We construct the behaviour graph $G = (N, E)$, for R the set of global $\Box$-rules of T, where N is the set of nodes and E the set of edges, as described in Section 3.1. Recall nodes, $n \in N$, of G are valuations of all the propositions occurring in T.

Definition 1 *(The applicable rule-set to a node.) Let T be a set of SNF rules, R the set of global $\Box$-rules of T, and n a valuation of all the propositions occurring in T. The set of rules $R_n \subseteq R$ such that $R_n = \{ \bigcirc A \Rightarrow B \mid \bigcirc A \Rightarrow B \in R, n \in N$ and $n \models A\}$ are known as* the applicable rule-set to n.

Definition 2 *(The reduced behaviour graph for R.) Let T be a set of SNF rules, R the set of global $\Box$-rules of T, and G the behaviour graph for R. Delete any nodes in G that have no successors and delete any edges into those nodes. Continue deleting such nodes and edges until no more can be deleted. The graph obtained, G_R, is known as* the reduced behaviour graph for R.

Definition 3 *(Terminal subgraph where l holds). Let T be a set of SNF rules, and $G_R = (N_R, E_R)$ the reduced behaviour graph for R, the set of global $\Box$-rules in T. For l, a literal in T, $G_T = (N_T, E_T)$ is a* terminal subgraph where l holds *if and only if*

1. *$G_T \subseteq G_R$;*
2. *if $n \in N_T$ then $n \models l$, i.e. l holds at each node; and*
3. *if $n \in N_T$ then $\nexists e \in E_R$ such that $e = (n, n')$ and $n' \notin N_T$, i.e. G_T is terminal.*

Definition 4 *(A proper loop in l.) Let T be a set of SNF rules, R the set of global $\Box$-rules of T and L[1] a loop in l made by combining rules in R. For each rule $r \in L$ of the form $\bigcirc A \Rightarrow B$ find the set $L_r = \{ \bigcirc P \Rightarrow Q \mid \bigcirc P \Rightarrow Q \in R$ and $\vdash A \Rightarrow P\}$. For each r in L combine all the rules in L_r to give the set of SNF$_m$ rules L'. If L' contains no rules with* **false** *on the right hand side then L' is a* proper loop in l.

[1] L is a set of SNF$_m$ rules, combined from rules in R, which form a loop in l.

The definition of a proper loop in l requires us to combine as many global $\Box$–rules from R as possible (after all step resolution is carried out) without changing the left hand side of the rules in L. This is because the reduced behaviour graph represents the *full* set of global $\Box$–rules not just a few of them. For example the rules $\bigcirc a \Rightarrow l$ and $\bigcirc a \Rightarrow a$ may be combined to give $\bigcirc a \Rightarrow a \wedge l$, a loop in l. However if $\bigcirc a \Rightarrow \neg l$ was also in the rule-set, performing step resolution would generate the additional global $\Box$–rule, $\bigcirc \textbf{true} \Rightarrow \neg a$. Now by trying to construct a proper loop from the rule $\bigcirc a \Rightarrow (a \wedge l)$ that we had previously, by also combining these two new rules we obtain $\bigcirc a \Rightarrow \textbf{false}$. Note there will be no edges out of any node n, such that $n \models a$, in the reduced behaviour graph for this set of rules.

The following two lemmas establish the link between graphs and sets of rules. Lemma 1 shows that if the reduced behaviour graph for a set of rules contains a terminal subgraph then the rules contain a proper loop. Lemma 2 shows the converse, i.e. given a set of rules containing a proper loop then the reduced behaviour graph for this set of rules contains a terminal subgraph.

Lemma 1 *Let T be a set of SNF rules, R the set of global $\Box$-rules of T, and G_R the reduced behaviour graph for R. The set of SNF$_m$ rules, L, constructed by combining the applicable rule-set to each node in any terminal subgraph $G_T = (N_T, E_T)$ of G_R where l holds is a proper loop in l.*

Proof
For each node $n \in N_T$ let $R_n \subseteq R$ be the applicable rule-set to n and let $\bigcirc A_n \Rightarrow B_n$ be the rule formed by combining the rules in R_n. L, the set of rules $\{ \bigcirc A_n \Rightarrow B_n \mid n \in N_T\}$, is a loop in l. To show this we must show the following.

1. For each $n \in N_T$ and combined applicable rule-set $\bigcirc A_n \Rightarrow B_n$ we must have $\vdash B_n \Rightarrow l$. Let n' be a valuation that satisfies B_n. Then by construction of the reduced behaviour graph n has a successor n' such that $n' \models B_n$ and from our assumption that G_T is a terminal subgraph where l holds at each node then $n' \in N_T$ and hence $n' \models l$ and so $B_n \Rightarrow l$ as required.

2. For each $n \in N_T$ we must have
$$\vdash B_n \Rightarrow \bigvee_{n' \in N_T} A_{n'}.$$
Let n'' be a valuation that satisfies B_n. Then by construction of G_R, there is an edge from n'' to a node $n' \in N_T$. As $n' \in N_T$ then $n' \models A_{n'}$ for some $n' \in N_T$ so $B_n \Rightarrow \bigvee\limits_{n' \in N_T} A_{n'}$ as required.

L must also be a proper loop as in order to obtain the rules in L, we have combined the set of applicable rules for each node in N_T. By the definition of the set of applicable rules there are no further rules in R we could combine for each node. Further, as N_T is a set of nodes in the reduced behaviour graph, i.e. deletions of any nodes with no successor have already taken place, there is an edge out of every node $n \in N_T$ to $n' \in N_T$. From construction of the graph, n' satisfies the set of clauses on the right hand side of the applicable rule-set for n. So by combining the applicable rule-set for n into a single SNF$_m$ rule we have used all possible rules (without changing the left hand side) and we obtain a rule that does not simplify to **false** on the right hand side. So L must be proper. $\Box$

Lemma 2 *Let T be a set of SNF rules, R the set of global $\Box$-rules of T, and G_R the reduced behaviour graph for R. If rules in R can be combined to form L a proper loop in l, a set of SNF$_m$ rules of the form $\bigcirc A_i \Rightarrow B_i$ for $i = 0$ to m, then G_R contains a terminal subgraph $G_T = (N_T, E_T)$ where l holds at each node and for all $n \in N_T$, $n \models A_i$, for some $i = 0$ to m.*

Proof
Let L be of the form $\bigcirc A_i \Rightarrow B_i$ for $i = 0$ to m. As L is a loop in l, $B_i \Rightarrow l$ and $B_i \Rightarrow \bigvee\limits_{j=0}^{m} A_j$ for all $i = 0$ to m. Let $G_T = (N_T, E_T)$ be a subgraph of G_R such that $n \in N_T$ iff $n \models \bigvee\limits_{j=0}^{m} A_j$ and $n \models l$. Let n be a node in N_T such that $n \models A_i$ for some $i = 0$ to m and $n \models l$. Then from the construction of G_R

any edge from n leads to node n' such that $n' \models B_i$ and as we know that $B_i \Rightarrow l$ then $n' \models l$. Now from $B_i \Rightarrow \bigvee_{j=0}^{m} A_j$, the assumption that L is a loop in l, we know n' must satisfy the left hand side of one of the rules in L so $n' \in N_T$. The latter ensures a terminal subgraph and the former that l holds. Note all step resolution possible is carried out on the SNF rules. This ensures that when combining rules to obtain a proper loop in l, if no combined rules have **false** on the right hand side then the loop will be represented in the reduced behaviour graph by a terminal subgraph where l holds. $\square$

We claim that the MTSA finds *all* loops for a particular eventuality. The following definition, for the maximal terminal subgraph where l holds, is required to refer to *all* the terminal subgraphs where l holds, that occur in the reduced behaviour graph.

Definition 5 *(Maximal terminal subgraph where l holds). Let T be a set of SNF rules, R the set of global $\square$-rules of T, and G_R the reduced behaviour graph for R. A terminal subgraph $G_T = (N_T, E_T)$ of G_R is said to be a* maximal terminal subgraph where l holds *if and only if for all terminal subgraphs $G'_T = (N'_T, E'_T)$ of G_R where l holds $N'_T \subseteq N_T$ and $E'_T \subseteq E_T$.*

In the definition of a proper loop all possible rules are combined without changing the left hand side of a rule. The folowing definition of a maximal proper loop in l includes all such loops.

Definition 6 *(Maximal proper loop in l). Let T be a set of SNF rules and R the set of global $\square$-rules of T. L, a proper loop in l, formed from combining rules in R is said to be a* maximal proper loop in l *if and only if for all proper loops L' in l, from combining rules in R, $L' \subseteq L$.*

Lemmas 3 and 4 establish the relationship between a set of rules containing a maximal proper loop and the reduced behaviour graph for this set of rules containing a maximal terminal subgraph.

Lemma 3 *Let T be a set of SNF rules, R the set of global $\square$-rules of T, and G_R the reduced behaviour graph for R. If R contains rules that may be combined to form L a maximal proper loop in l, of the form $\bigcirc A_i \Rightarrow B_i$, for $i = 0$ to m, then G_R contains a maximal terminal subgraph $G_M = (N_M, E_M)$ where l holds and if $n \in N_M$ then $n \models A_i$ for some $i = 0$ to m.*

Proof
As L is a maximal proper loop in l it is also a proper loop in l, so from Lemma 2 the reduced behaviour graph G_R contains a terminal subgraph, $G_T = (N_T, E_T)$ where l holds such that $n \in N_T$ then $n \models A_i$ for some $i = 0$ to m. We need to show G_T is also maximal. Assume it is not, i.e. there is a terminal subgraph $G'_T = (N'_T, E'_T)$ where l holds such that $G'_T \not\subseteq G_T$. Then from Lemma 1 we can construct a set of rules L' from G'_T such that L' is a proper loop in l. Now, from the definition of a maximal proper loop in l, L must contain all proper loops in l so L' cannot exist and G_T is in fact maximal. $\square$

Lemma 4 *Let T be a set of SNF rules, R the set of global $\square$-rules of T, and G_R the reduced behaviour graph for R. If R contains rules that may be combined to form L a maximal proper loop in l, of the form $\bigcirc A_i \Rightarrow B_i$, for $i = 0$ to m, then G_R contains a maximal terminal subgraph $G_M = (N_M, E_M)$ where l holds and*

$$n \in N_M \textit{ iff } n \models \bigvee_{i=0}^{m} A_i \wedge l$$

Proof
From Lemma 3 if R contains rules that can be combined to form a maximal proper loop L then G_R contains a maximal terminal subgraph where l holds and if $n \in N_M$ then $n \models A_i$ for some $i = 0$ to m. So

$$n \models \bigvee_{i=0}^{m} A_i$$

and as N_M is a maximal terminal subgraph where l holds $n \models l$ so

$$n \in N_M \Rightarrow n \models \bigvee_{i=0}^{m} A_i \wedge l.$$

We must show

$$n \models \bigvee_{i=0}^{m} A_i \wedge l \Rightarrow n \in N_M.$$

Take a node $n \in N_R$ such that

$$n \models \bigvee_{i=0}^{m} A_i \wedge l.$$

Then as

$$n \models \bigvee_{i=0}^{m} A_i$$

all edges out of n must lead back into a terminal subgraph where l holds (because of the construction of the reduced behaviour graph and that L is a proper loop in l). As $n \models l$ then $n \in N_M$ as required. $\square$

We prove two lemmas that will be useful later. The first shows that at each application of the MTSA the set of nodes constructed is a subset of the previous set constructed. The second shows that if, at some stage in the algorithm, a node that was a member of the set of nodes built previously is deleted, then there is a path from that node to a node where the literal we require does not hold. We denote the set of nodes built at each iteration of the MTSA by $N_0, N_1, \ldots$ etc.

Lemma 5 *Let T be a set of SNF rules, R the set of global $\Box$-rules of T, $G_R = (N_R, E_R)$ the reduced behaviour graph for R and $N_0 \subseteq N_R$ a set of nodes in G_R. If the set of nodes $N_{i+1} \subseteq N_R$ is defined as $N_{i+1} = \{n \mid n \in N_0$ and $\{n' \mid n' \in N_R$ and $(n, n') \in E_R\} \subseteq N_i\}$ for $i \geqslant 0$ then*

$$N_{i+1} \subseteq N_i \textit{ for } i \geqslant 0.$$

Proof

By induction on the index i. The base case, $i = 0$, $N_1 \subseteq N_0$ holds trivially. Suppose the lemma holds for some $i = j > 0$, $N_{j+1} \subseteq N_j$ for $j \geqslant 0$ and we are trying to show it holds for $i = j + 1$, i.e. we must show $N_{j+2} \subseteq N_{j+1}$. Let $n \in N_{j+2}$ then $n \in N_0$ and $\{n' \mid n' \in N_R$ and $(n, n') \in E_R\} \subseteq N_{j+1} \subseteq N_j$ so $n \in N_{j+1}$ (as $n \in N_0$ and $\{n' \mid n' \in N_R$ and $(n, n') \in E_R\} \subseteq N_j$ then n must also be in the set N_{j+1}). $\square$

Lemma 6 *Let T be a set of SNF rules, R the set of global $\Box$-rules of T, and $G_R = (N_R, E_R)$ the reduced behaviour graph for R. If the MTSA deletes a node at the i^{th} step such that $n_{i-1} \in N_{i-1}$ and $n_{i-1} \notin N_i$ then there exists a path $(n_{i-1}, \ldots, n_1, n_0, m)$ out of N_0 of length i such that $(n_j, n_{j-1}) \in E_R$ for $j = i-1, \ldots 1$, $(n_0, m) \in E_R$, $n_j \in N_j$ for $j = i-1, \ldots 0$ and $m \notin N_0$.*

Proof

By induction on the length of the path (see Figure 4.1). For the base case, $i = 1$ let $n_0 \in N_0$ and $n_0 \notin N_1$. Then as n_0 has been deleted by the MTSA from step 2 of the definition of the MTSA (Section 3.1.2) there must be a node m such that $(n_0, m) \in E_R$ and $m \notin N_0$. So there exists a path of length 1 out of N_0. Suppose the lemma holds for $i = j > 1$, i.e for $n_{j-1} \in N_{j-1}$ and $n_{j-1} \notin N_j$ there exists a path of length j from n_{j-1} out of N_0. We are trying to show the lemma holds for $i = j + 1$, i.e. for $n_j \in N_j$ and $n_j \notin N_{j+1}$ there exists a path of length $j + 1$ out of N_0. Assume $n_j \in N_j$ and $n_j \notin N_{j+1}$ then from the definition of the MTSA there must be an edge (n_j, n_{j-1}) such that $n_{j-1} \notin N_j$. If $n_j \in N_j$ then from Lemma 5, $n_j \in N_{j-2}$ and as $n_j \in N_j$ from the definition of the MTSA, $n_{j-1} \in N_{j-1}$. So we have $n_{j-1} \notin N_j$ and $n_{j-1} \in N_{j-1}$ and from our assumption we know that there exists a path of length j from n_{j-1} out of N_0, so there exists a path from n_j out of N_0 that must be the length of the path of n_{j-1} out of N_0 plus 1, i.e. $j + 1$. $\square$

Theorem 3 *(Soundness of the Maximal Terminal Subgraph Algorithm.) If, given a set of SNF rules T, the MTSA terminates returning the set of rules L, of the form $\circledcirc A_i \Rightarrow B_i$, for $i = 0$ to m, for resolution with $\mathcal{L}Q \Rightarrow \Diamond \neg l$, a rule in T, then*

$$\circledcirc \bigvee_{i=0}^{m} A_i \Rightarrow \Box l.$$

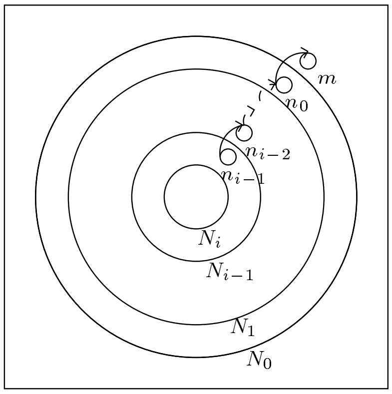

Figure 4.1: If node n_{i-1} is deleted then there is a path of length i out of N_0.

Proof
Let $G_R = (N_R, E_R)$ be the reduced behaviour graph built from R the set of global $\square$-rules of T. L has been constructed by running the MTSA so that in step 2 it detects a subgraph $G_T = (N_T, E_T)$ and then by combining the applicable rule-set for each $n \in N_T$ so that $n \in N_T$ then $n \models A_i$ for some $i = 0$ to m. For G_T to be a terminal subgraph where l holds we must establish the three conditions from the definition of a terminal subgraph where l holds (Definition 3).

1. $G_T \subseteq G_R$ must hold as the MTSA starts with $N_0 \subseteq N_R$ and from Lemma 5 $N_{i+1} \subseteq N_i$ for $i \geqslant 0$ so $N_T \subseteq N_R$.
2. The required literal l holds at each node. Step 2(a) of the algorithm ensures we take the set of nodes N_0 where l holds originally. We know for each subsequent set of nodes $N_{i+1} \subseteq N_i$ (by Lemma 5), so $N_T \subseteq N_0$ and thus if $n \in N_T$ then $n \models l$.
3. No edges lead out of G_T. If any nodes in G_T had edges leading out of G_T we would continue processing with step 2 of the algorithm and delete them. Thus G_T is terminal.

Thus G_T is a terminal subgraph where l holds and from Lemma 1 this represents a (proper) loop in l so

$$\bigcirc \bigvee_{i=0}^{m} A_i \Rightarrow \square l$$

as required. $\square$
The completeness theorem we prove is sufficient for completeness of the resolution system. Thus, rather than showing that a set of rules that form a loop in l in the rule-set is *the same* as the set of rules output by the MTSA, we simply check that if l holds then the disjunction of the left hand sides of the two sets of rules are equivalent. That is, if l holds, we ensure that the disjunction of the (conjunction of literals on the) left hand side of the rules in the loop that exist in the rule-set is equivalent to the disjunction of the (conjunction of literals on the) left hand sides of rules detected by applying the MTSA.

Ensuring that the left hand sides are equivalent is sufficient for completeness of temporal resolution because having detected a loop $\bigcirc A_i \Rightarrow B_i$, for $i = 0$ to m the resolvents for temporal resolution require

$$\bigwedge_{i=0}^{m} \neg A_i$$

or in the (DNF) form we have it here

$$\neg(\bigvee_{i=0}^{m} A_i)$$

as well as the $\Diamond$-rule to resolve the loop with. So ensuring the two left hand sides are equivalent is sufficient to retain completeness of the resolution system.

Further, we show the assumption that l holds is sufficient to retain completeness of temporal resolution. The completeness proof for the MTSA shows that

$$\models l \Rightarrow \left(\bigvee_{j=0}^{m_{L'}} A_{L',j} \Leftrightarrow \bigvee_{i=0}^{m_L} A_{L,i} \right)$$

where $A_{L,i}$ for $i = 0$ to m_L and $A_{L',j}$ for $j = 0$ to $m_{L'}$ are the left hand sides for the maximal proper loop L and the set of rules output from the MTSA, L', respectively. By soundness of the MTSA

$$\circledcirc \bigvee_{j=0}^{m_{L'}} A_{L',j} \Rightarrow \Box l$$

and by the assumption that L is a maximal proper loop where l holds

$$\circledcirc \bigvee_{i=0}^{m_L} A_{L,i} \Rightarrow \Box l$$

i.e. if either

$$\bigvee_{j=0}^{m_{L'}} A_{L',j} \text{ or } \bigvee_{i=0}^{m_L} A_{L,i}$$

holds in the next moment we can assume l holds. In particular if L contains rules with $\neg l$ as conjuncts on the left hand side of rules then l must still hold at the next moment in time. So any edge out of any node n in the reduced behaviour graph that satisfies the left hand side of a rule in L that contains $\neg l$ must lead to a node, n', that satisfies both l and

$$\bigvee_{i=0}^{m_L} A_{L,i}$$

and therefore by completeness

$$\bigvee_{j=0}^{m_{L'}} A_{L',j}.$$

Adding the resolvents from temporal resolution with L to the set of rules will remove any edges into the nodes that satisfy the disjunction of the left hand side of L where $\neg l$ is an outstanding eventuality. In particular the edge into n is removed. However, adding the resolvents from temporal resolution with the loop output by the MTSA, L', to the set of rules will remove any edges into the set of nodes n' where

$$n' \models \bigvee_{j=0}^{m_{L'}} A_{L',j}$$

and $\neg l$ is an outstanding eventuality. So n becomes a terminal node and must therefore be deleted from the reduced behaviour graph. This corresponds to step resolution in the resolution system. So assuming l holds and that the disjunction of the left hand sides of the two sets of rules are equivalent is sufficient for the completeness of temporal resolution.

Theorem 4 *(Sufficient Completeness of the Maximal Terminal Subgraph Algorithm). Given a set of SNF rules T where R the set of global $\Box$-rules of T contains a maximal proper loop in l, L, of the form $\circledcirc A_{L,i} \Rightarrow B_{L,i}$, for $i = 0$ to m_L, applying the MTS Algorithm will yield L' a proper loop in l of the form $\circledcirc A_{L',j} \Rightarrow B_{L',j}$ for $j = 0$ to $m_{L'}$ and*

$$\models l \Rightarrow \left(\bigvee_{j=0}^{m_{L'}} A_{L',j} \Leftrightarrow \bigvee_{i=0}^{m_L} A_{L,i} \right).$$

Proof

As R contains rules we may combine to form L, a maximal proper loop in l, by Lemma 3, the reduced behaviour graph $G_R = (N_R, E_R)$ of R contains a maximal terminal subgraph $G_M = (N_M, E_M)$ where l holds. For L of the form $\odot A_{L,i} \Rightarrow B_{L,i}$, for $i = 0$ to m_L,

$$n \in N_M \text{ iff } n \models \bigvee_{i=0}^{m_L} A_{L,i} \wedge l$$

(Lemma 4). Having constructed the reduced behaviour graph for R and decided to search for a loop in l the MTSA first constructs a set $N_0 \subseteq N_R$ such that $n \in N_0$ iff $n \models l$. To build N_{i+1} we delete any nodes from N_{i+1} that have edges that lead out of the set N_i, i.e. for any node $n \in N_{i+1}$ then $(n, n') \in E_R$ and $n' \in N_i$ and $N_{i+1} \subseteq N_0$. That is we remove nodes in stages firstly removing nodes that take us out of the set of nodes N_0 by traversing one edge, then by traversing two edges etc. until we end up with a subset of nodes of N_0 from which we cannot escape (or all nodes are deleted). This continues until $N_{i+1} = N_i$ (or $N_{i+1} = \emptyset$ but here we assume N_M exists so we exclude this case). Each set N_{i+1} that we build has the property $N_{i+1} \subseteq N_i$ for $i \geqslant 0$ (see Lemma 5), so as N_0 is finite the algorithm terminates.

Let $G_T = (N_T, E_T)$ be the subgraph constructed by step 2 of the MTSA[2] then as $N_T \subseteq N_0$ and l holds at each node in N_0, l holds at each node in N_T. Also $n \in N_T$ means $(n, n') \in E_R$ and $n' \in N_T$ (as the last set of nodes we have built is the same as its predecessor) so the graph is terminal. So by applying steps 1 and 2 of the MTSA we find a terminal subgraph G_T where l holds.

Both G_T and G_M are terminal subgraphs where l holds but G_M is also the maximal terminal subgraph, i.e. the union of all the terminal subgraphs where l holds so $G_T \subseteq G_M$ (from Definition 5 of a maximal terminal subgraph where l holds). Take any node $n \in N_M$ where $n \notin N_T$. From the definition of the maximal terminal subgraph we know that $n \models l$ and therefore when applying the MTSA, $n \in N_0$, so if $n \notin N_T$ we must have deleted n at some point. Without loss of generality assume that n was deleted at the i^{th} step. From Lemma 6 we know that there exists a path from n out of N_0 of length i. But from the definition of N_M, the maximal terminal subgraph, we know that no edges lead out of N_M so n cannot exist and therefore $N_M = N_T$ and $G_M = G_T$.

To obtain L' from G_T, step 3 of the MTSA, for each $n \in N_T$ we take the applicable rule-set and by Lemma 1, L' is a proper loop in l. Note for L' of the form $\odot A_{L',j} \Rightarrow B_{L',j}$ if $n \in N_T$ then

$$n \models \bigvee_{j=0}^{m_{L'}} A_{L',j}$$

and if $n \in N_T$ then $n \models l$ (Lemma 5). Further,

$$n \models \bigvee_{j=0}^{m_{L'}} A_{L',j} \wedge l \Rightarrow n \in N_T$$

as n would not be deleted using the MTSA as

$$n \models \bigvee_{j=0}^{m_{L'}} A_{L',j}$$

means that every edge out of n leads into N_T and as $n \models l$ then $n \in N_T$. Thus as $N_T = N_M$ for any node $n \in N_T$,

$$n \models \bigvee_{i=0}^{m_L} A_{L,i}, n \models \bigvee_{j=0}^{m_{L'}} A_{L',j}$$

and $n \models l$. Further, the set of nodes N_M satisfies

$$n \in N_M \text{ iff } n \models \bigvee_{i=0}^{m_L} A_{L,i} \wedge l$$

[2]It is easy to construct the subgraph given the set of nodes N_T as it is merely the projection of N_T on G_R

(Lemma 4) and the set of nodes N_T satisfies

$$n \in N_T \text{ iff } n \models \bigvee_{j=0}^{m_{L'}} A_{L',j} \wedge l.$$

Recall nodes in the reduced behaviour graph are valuations of all propositions in the set of SNF rules. As the set of nodes representing the loop L, N_M, is the same as the set of nodes, N_T, detected by step 2 of the MTSA from which we obtain loop L', and as the formulae

$$l \wedge \bigvee_{i=0}^{m_L} A_{L,i} \text{ and } l \wedge \bigvee_{j=0}^{m_{L'}} A_{L',j}$$

are both satisfied by exactly then same valuations then

$$\models l \wedge \bigvee_{i=0}^{m_L} A_{L,i} \Leftrightarrow l \wedge \bigvee_{j=0}^{m_{L'}} A_{L',j}$$

and

$$\models l \Rightarrow \left(\bigvee_{j=0}^{m_{L'}} A_{L',j} \Leftrightarrow \bigvee_{i=0}^{m_L} A_{L,i} \right).$$

as required. $\square$

4.3 Correctness of Breadth-First Search

We will deal with the correctness proofs of Breadth-First Search before Depth-First Search as the former is presented via reference to the construction of a reduced behaviour graph (as is done in the completeness proof for Fisher's resolution procedure [Pei94]) and the latter is shown with respect to the completeness of Breadth-First Search. The completeness of the MTSA is used to show the completeness of Breadth-First Search. We will denote sets of nodes built at the i^{th} step of the MTSA as $N_{M,i}$, the DNF formulae built at the i^{th} step of the Breadth-First Search as H_i and relate H_i to a set of nodes in the reduced behaviour graph, $G_R = (N_R, E_R)$, $N_{B,i} = \{n \mid n \in N_R \text{ and } n \models H_i\}$. First we will prove lemmas that link the sets of nodes built at corresponding steps of the MTSA and the Breadth-First Search Algorithm.

Lemma 7 *Let T be a set of SNF rules and $G_R = (N_R, E_R)$ the reduced behaviour graph for R the set of global $\square$-rules in T. Let $N_{B,i}$ and $N_{M,i}$ be the sets of nodes in G_R representing the i^{th} step in the Breadth-First Search Algorithm and MTSA respectively. If G_R contains a maximal terminal subgraph where l holds then*

$$(n \in N_{B,i} \wedge n \models l) \Rightarrow n \in N_{M,i}.$$

We are trying the show that if l holds at any node, n, in the set of nodes in the reduced behaviour graph representing the i^{th} step of the Breadth-First Search Algorithm, $N_{B,i}$, then n is also a member of the set of nodes in the reduced behaviour graph representing the i^{th} step of the MTSA, $N_{M,i}$.

Proof

By induction on the steps in each algorithm (see Figure 4.2). For the base case let $i = 0$ so we are trying to show $(n \in N_{B,0} \wedge n \models l) \Rightarrow n \in N_{M,0}$. This holds trivially from the definition of $N_{M,0}$. We assume the lemma holds for $i = j > 0$, i.e. $(n \in N_{B,j} \wedge n \models l) \Rightarrow n \in N_{M,j}$, and prove it holds for $i = j + 1$, i.e. $(n \in N_{B,j+1} \wedge n \models l) \Rightarrow n \in N_{M,j+1}$. Let $n \in N_{B,j+1}$ and $n \models l$. From the Breadth-First Algorithm we know that

$$n \models \bigvee_{k=0}^{m} P_k$$

where there are rules in R we can combine together to form $\bigcirc P_k \Rightarrow Q_k$ for $k = 0$ to m such that

1. $P_k \Rightarrow H_0$; and

2. $Q_k \Rightarrow H_j$.

Now from (1) if $n \in N_{B,j+1}$ and $(n, n') \in E_R$ then $n' \models l$ as every edge out of every node in $N_{B,0}$ leads to a node where l holds and from (2) $(n \in N_{B,j+1} \wedge (n, n') \in E_R) \Rightarrow n' \in N_{B,j}$. So as $n' \models l$ and $n' \in N_{B,j}$ then from our assumption $n' \in N_{M,j}$. So as $n \models l$ and for all n' such that $(n, n') \in E_R$ and $n' \in N_{M,j}$ then from the definition of the MTSA $n \in N_{M,j+1}$, as required. □

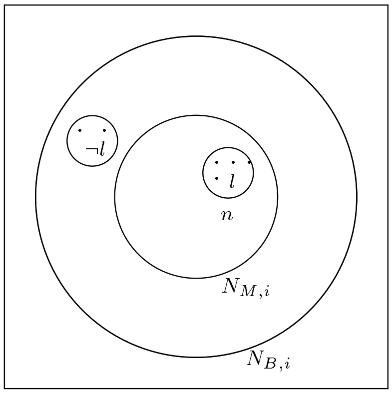

Figure 4.2: Any node in $N_{B,i}$ containing l is also in $N_{M,i}$.

Lemma 8 *Let T be a set of SNF rules and $G_R = (N_R, E_R)$ the reduced behaviour graph for R the set of global $\Box$-rules in T. Let $N_{B,i}$ and $N_{M,i}$ be the sets of nodes in G_R representing the i^{th} step in the Breadth-First Search Algorithm and MTSA respectively. If G_R contains a maximal terminal subgraph where l holds then*

$$(n \in N_{B,i} \wedge (n, n') \in E_R) \Rightarrow n' \in N_{M,i}.$$

We are trying the show that any edge out of any node, n, in the set of nodes in the reduced behaviour graph representing the i^{th} step of the Breadth-First Search Algorithm, $N_{B,i}$, leads to a node n' that is a member of the set of nodes in the reduced behaviour graph representing the i^{th} step of the MTSA, $N_{M,i}$.

Proof

By induction on the number of steps in the Breadth-First Search Algorithm and MTSA (see Figure 4.3). For the base case we take $i = 0$ and need to show $(n \in N_{B,0} \wedge (n, n') \in E_R) \Rightarrow n' \in N_{M,0}$. Now in the Breadth-First Search Algorithm, $N_{B,0}$ is built from detecting rules $\circledcirc X_h \Rightarrow l$, so $m \in N_{B,0}$ iff $m \models \bigvee_h X_h$. From the construction of the reduced behaviour graph if $n \in N_{B,0} \wedge (n, n') \in E_R$ then $n' \models l$. This is the criterion for $N_{M,0}$ so $n' \in N_{M,0}$ as required. We assume the lemma holds for $i = j > 0$, i.e. $(n \in N_{B,j} \wedge (n, n') \in E_R) \Rightarrow n' \in N_{M,j}$ and prove it holds for $i = j + 1$, i.e. $(n \in N_{B,j+1} \wedge (n, n') \in E_R) \Rightarrow n' \in N_{M,j+1}$. If $n \in N_{B,j+1}$ then we have SNF$_m$ rules $\circledcirc P_k \Rightarrow Q_k$ for $k = 0$ to m such that

1. $P_k \Rightarrow H_0$; and

2. $Q_k \Rightarrow H_j$

so $n \models \bigvee_{k=0}^{m} P_k$. From (2) $(n \in N_{B,j+1} \wedge (n, n') \in E_R) \Rightarrow n' \in N_{B,j}$ and by assumption $(n' \in N_{B,j} \wedge (n', n'') \in E_R) \Rightarrow n'' \in N_{M,j}$. From (1) if $n \in N_{B,j+1} \wedge (n, n') \in E_R$ then $n' \models l$. So by Lemma 7, $n' \in N_{M,j}$. Now from the definition of the MTSA if $n' \in N_{M,j}$ and for all n'' such that $(n', n'') \in E_R$ and $n'' \in N_{M,j}$ then $n' \in N_{M,j+1}$ as required. □

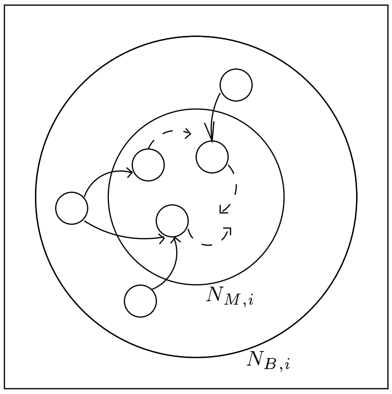

Figure 4.3: Any edge out of a node in $N_{B,i}$ leads to a node in $N_{M,i}$.

Lemma 9 *Let T be a set of SNF rules and $G_R = (N_R, E_R)$ the reduced behaviour graph for R the set of global $\square$-rules in T. Let $N_{B,i}$ and $N_{M,i}$ be the sets of nodes in G_R representing the i^{th} step in the Breadth-First Search Algorithm and MTSA respectively. If G_R contains a maximal terminal subgraph where l holds then*

$$\forall n'(n \in N_{M,i} \wedge (n, n') \in E_R \wedge n' \in N_{M,i}) \Rightarrow n \in N_{B,i}.$$

We are trying the show that if all edges out of any node, n, in the set of nodes in the reduced behaviour graph representing the i^{th} step of the MTSA, $N_{M,i}$, lead to some node n' that is also in $N_{M,i}$ then n is a member of the set of nodes in the reduced behaviour graph representing the i^{th} step of the Breadth-First Search Algorithm, $N_{B,i}$.

Proof

By induction on the steps in the Breadth-First Search Algorithm or MTSA. For the base case let $i = 0$ and show that $\forall n'(n \in N_{M,0} \wedge (n, n') \in E_R \wedge n' \in N_{M,0}) \Rightarrow n \in N_{B,0}$. If we assume $\forall n'(n \in N_{M,0} \wedge (n, n') \in E_R \wedge n' \in N_{M,0})$ then all edges (n, n') out of n must satisfy $n' \in N_{M,0}$, i.e. $n' \models l$. Now we build $N_{B,0}$ by searching for rules $\circledcirc X_h \Rightarrow l$ and set $N_{B,0} = \{m \mid m \in N_R \text{ and } m \models \bigvee_h X_h\}$. So from the Breadth-First Search Algorithm we know $m \models \bigvee_h X_h$ iff $m \in N_{B,0}$ and from the construction of the graph that $m \models \bigvee_h X_h$ iff for all $m'((m, m') \in E_R \wedge m' \models l)$. In this case we know for all n' $(n, n') \in E_R \wedge n' \models l$ so $n \in N_{B,0}$. So if we have $n \in N_{M,0}$ and all edges from n lead back into $N_{M,0}$ we know $n \in N_{B,0}$.

We assume the lemma holds for $i = j > 0$, i.e. $\forall n'(n \in N_{M,j} \wedge (n, n') \in E_R \wedge n' \in N_{M,j}) \Rightarrow n \in N_{B,j}$, and prove it holds for $i = j + 1$, i.e. $\forall n'(n \in N_{M,j+1} \wedge (n, n') \in E_R \wedge n' \in N_{M,j+1}) \Rightarrow n \in N_{B,j+1}$. Assume $\forall n'(n \in N_{M,j+1} \wedge (n, n') \in E_R \wedge n' \in N_{M,j+1})$. From the MTSA if $n \in N_{M,j+1}$ and $(n, n') \in E_R$ then $n' \in N_{M,j}$ and if $n' \in N_{M,j+1}$ and $(n', n'') \in E_R$ then $n'' \in N_{M,j}$. From our assumption if $\forall n'(n' \in N_{M,j} \wedge (n', n'') \in E_R \wedge n'' \in N_{M,j})$ then $n' \in N_{B,j}$. Now $n' \in N_{B,j}$ iff $n' \models H_j$ and to expand from H_j we search for SNF$_m$ rules $\circledcirc P_k \Rightarrow Q_k$ for $k = 0$ to r such that

1. $P_k \Rightarrow H_0$; and
2. $Q_k \Rightarrow H_j$.

The first rule ensures that l holds at the next moment in time, i.e. if $m \in N_{B,j+1}$ every edge out of m leads to a node where l holds. So n must lead to a node where l holds as we have $n \in N_{M,j+1} \wedge (n, n') \in E_R \wedge n' \in N_{M,j+1}$ and from Lemma 5 if $n' \in N_{M,j+1}$ then $n' \in N_{M,0}$ as $N_{M,j+1} \subseteq N_{M,0}$ so $n' \models l$. Also from (2) we require that every edge out of $m \in N_{B,j+1}$ leads to a node in $N_{B,j}$ and we know that this is satisfied by n as $(n, n') \in E_R \wedge n' \in N_{B,j}$. So any n that satisfies our assumptions shows that $n \in N_{B,j+1}$ as required. $\square$

Lemma 10 *Let T be a set of SNF rules and $G_R = (N_R, E_R)$ the reduced behaviour graph for R the set of global $\square$-rules in T. If G_R contains a maximal terminal subgraph where l holds, $G_M = (N_M, E_M)$, then the Breadth-First Search algorithm yields a DNF formula D such that if $N_B = \{n \mid n \in N_R$ and $n \models D\}$ then $N_M \subseteq N_B$.*

Proof
From Lemma 9 if for all n' such that $n \in N_{M,i} \wedge (n, n') \in E_R \wedge n' \in N_{M,i}$ then $n \in N_{B,i}$. As G_R contains a maximal terminal subgraph where l holds then at some point $N_{M,i} = N_{M,i+1}$ and $N_{B,i} = N_{B,i+1}$ so $n \in N_M \wedge (n, n') \in E_R \wedge n' \in N_M \Rightarrow n \in N_B$. Now as N_M is a maximal terminal subgraph any edge from n will lead back into N_M, i.e. $n' \in N_M$. So $n \in N_M \Rightarrow n \in N_B$ and $N_M \subseteq N_B$ as required. $\square$

Theorem 5 *(Soundness for the Breadth-First Search Algorithm.) If, given a set of SNF rules, T, the Breadth-First Search Algorithm terminates detecting a DNF-formula D for resolution with $\mathcal{L}Q \Rightarrow \diamondsuit\neg l$ in T then*

$$\circledcirc D \Rightarrow \square l.$$

Proof
Let $G_R = (N_R, E_R)$ be the reduced behaviour graph for R the set of global $\square$-rules of T. Let $N_B = \{n \mid n \in N_R$ and $n \models D\}$ and N_M is the set of nodes constructed by the MTSA then, from Lemma 10, $N_M \subseteq N_B$.

Now N_B is terminal as N_M is terminal and from Lemma 8 (with index i dropped as N_M and N_B have stabilised) we have $n \in N_B \wedge (n, n') \in E_R \Rightarrow n' \in N_M$, i.e. every edge out of every node in N_B leads to a node that is in N_M so N_B is also terminal. From the MTSA if $n \in N_M$ then $n \models l$ and as $n \in N_B$ iff $n \models D$, from Lemma 8 $(n \models D \wedge (n, n') \in E_R) \Rightarrow n' \in N_M$ then $D \Rightarrow \bigcirc l$. So as N_B is terminal and l holds at each next node in N_B then $D \Rightarrow \bigcirc \square l$ or $\circledcirc D \Rightarrow \square l$ as required. $\square$

Theorem 6 *(Sufficient completeness for the Breadth-First Search Algorithm). Given a set of SNF rules T where R the set of global $\square$-rules of T that contains rules that may be combined together to form a maximal proper loop in l, L, of the form $\circledcirc A_i \Rightarrow B_i$ for $i = 0$ to m then by applying the Breadth-First Search algorithm we can find a DNF formula D such that*

$$\models l \Rightarrow \left(D \Leftrightarrow \bigvee_{i=0}^{m} A_i \right).$$

Proof
As R contains a maximal proper loop in l, from Lemma 3, the behaviour graph for R, $G_R = (N_R, E_R)$, contains a maximal terminal subgraph where l holds, $G_M = (N_M, E_M)$. The Breadth-First Search Algorithm outputs D so let $N_B = \{n \mid n \in N_R$ and $n \models D\}$. We have the following

- $n \in N_M$ iff $n \models \bigvee_{i=0}^{m} A_i \wedge l$ (Lemma 4);
- N_M, the maximal terminal subgraph where l holds, is identical to the set of nodes constructed in step 2 of the MTSA (Completeness of MTSA);
- $N_M \subseteq N_B$ (Lemma 10);
- $(n \in N_B \wedge n \models l) \Rightarrow n \in N_M$, from Lemma 7 (extended to when no more deletions on each set of nodes may take place); and
- $n \in N_M \Rightarrow n \models l$ from the definition of the maximal terminal subgraph where l holds.

So we have

$$(n \in N_B \wedge n \models l) \Leftrightarrow n \in N_M,$$

i.e. the set of nodes in N_B where l also holds is identical to the set of nodes N_M. Recall each node in the reduced behaviour graph is merely a valuation so for all valuations n

$$n \models D \wedge l \text{ iff } n \models \bigvee_{i=0}^{m} A_i \wedge l$$

so

$$\models D \wedge l \Leftrightarrow \bigvee_{i=0}^{m} A_i \wedge l$$

and

$$\models l \Rightarrow \left(D \Leftrightarrow \bigvee_{i=0}^{m} A_i \right)$$

as required. $\square$

Before proving correctness of Depth-First Search we define a *core loop* for a loop L with respect to the reduced behaviour graph for a set of rules. We then go on to show that detecting the core loop for L rather than L does not affect the completeness of temporal resolution.

Definition 7 *(The core nodes of a loop) Let T be a set of SNF rules, R the set of global $\square$-rules of T, and $G_R = (N_R, E_R)$ the reduced behaviour graph for R. Let D be a DNF formula such that $\circledcirc D \Rightarrow \square l$, for resolution with $\mathcal{L}Q \Rightarrow \Diamond \neg l$ a rule in T and N_D the set of nodes such that $n \in N_D$ iff $n \models D$. Delete any nodes n from N_D where there is no edge $(n', n) \in E_R$ such that $n' \in N_D$ until we can delete no further nodes. The set of nodes remaining are the* core nodes of D

Definition 8 *(The core loop) Let T be a set of SNF rules, R the set of global $\square$-rules of T, and $G_R = (N_R, E_R)$ the reduced behaviour graph for R. Let D be a DNF formula such that $\circledcirc D \Rightarrow \square l$, for resolution with $\mathcal{L}Q \Rightarrow \Diamond \neg l$ a rule in T, and find the core nodes $N_{D'}$ for D. For each node $n \in N_{D'}$ find the applicable rule-set for n and combine together the rules into an SNF$_m$ rule. This set of SNF$_m$ rules is known as the* core loop *for D.*

Note, we can be sure that the core loop for D is actually a loop as the core nodes for D form a terminal subgraph where l holds and from Lemma 1 the rules constructed from taking the applicable ruleset for each node in such a subgraph form a loop in l. We use the behaviour graph constructed by Peim for the complete set of SNF rules in the following proof. Recall Peim's completeness proof for temporal resolution (described in Section 4.1) builds a a behaviour graph where nodes are pairs (V, Y) and completeness depends on the fact that a set of rules is unsatisfiable if and only if the *reduced* behaviour graph for this set of rules is empty. The reductions performed to obtain a reduced behaviour graph are the deletions of nodes n if there are no edges leading out of n and if there is an eventuality in node n that is not satisfied in any node reachable from n (call this set N_L) then this node is also deleted. The former corresponds with a series of applications of the step resolution rule deriving a rule with **false** on the right hand side followed by the use of the rule

$$\frac{\circledcirc P \;\Rightarrow\; \mathbf{false}}{\bullet\,\mathbf{true} \;\Rightarrow\; \neg P}$$

which removes edges into any nodes n such that $n \models P$. The latter corresponds with temporal resolution as the rules that have been used to build edges between the nodes in N_L will form a loop in the literal we require. So the addition of the resolvents from detecting this loop means that edges into the subgraph N_L are removed.

Lemma 11 *Let T be a set of SNF rules, D the DNF formula such that $\circledcirc D \Rightarrow \square l$ for resolution with r, rule in T of the form $\mathcal{L}Q \Rightarrow \Diamond \neg l$. All the resolvents from D and r can be generated from the resolvents from the core loop of D and r and step resolution between these rules and rules in T.*

Proof
Build the behaviour graph, $G_T = (N_T, E_T)$ for T. Find the set of nodes $N_D \subseteq N_T$ such that $(V, Y) \in N_D$ iff $V \models D$ and $\neg l \in Y$, i.e. $\neg l$ is an outstanding eventuality. Then from the completeness of temporal resolution any node in N_D can be deleted from G_T corresponding to the addition of the resolvents from D and r [Pei94]. Let D' be the DNF formula obtained from disjoining the conjunctions of literals on the left hand side of the core loop for D. Let $N_{D'} \subseteq N_T$ such that $(V, Y) \in N_{D'}$ iff $V \models D'$ and $\neg l \in Y$ be the corresponding set of nodes in G_T for D'. From the definition of the core nodes of D (Definition 7), $N_{D'} \subseteq N_D$. We must show that any node $n = (V, Y)$ such that $n \in N_D$ and $n \notin N_{D'}$ can be deleted from G_T as it becomes a terminal node. From [Pei94] this corresponds with the application of step resolution and we are done.

Given that $m \geqslant 0$ deletion steps are performed to obtain the core nodes of D let $N_{D,0} = N_D$ and $N_{D,i}$ for $i > 0$ be the sets of nodes remaining in N_D following the i^{th} deletion step. So, $N_{D,1}$ is the set of nodes following the first set of deletions etc. and $N_{D,m} = N_{D'}$. If $m = 0$ no deletions are performed then $N_D = N_{D'}$ and we are done. Otherwise any node n deleted from the set of nodes $N_{D,i}$, for $0 \leqslant i < m$, satisfies

- no node in $N_{D,i}$ leads into n; and
- any edge out of n leads into $N_{D,i+1}$.

Deleting any nodes in $N_{D'}$ (due to resolution with r) means that nodes deleted at the m^{th} step of deletions ($N_{D,m-1}$) become terminal nodes. Similarly deleting the set of nodes $N_{D,i} - N_{D,i+1}$ for $0 < i < m$ because they have become terminal mean that the set $N_{D,i-1} - N_{D,i}$ become terminal. This continues until we have deleted the nodes $N_{D,1}$ so any nodes in $N_{D,0} - N_{D,1}$ are terminal and are deleted. Thus all the nodes in $N_{D,0} = N_D$ have been deleted corresponding to temporal resolution between D' and r and followed by step resolution. $\square$

4.4 Correctness of Depth-First Search

The Soundness of Depth-First Search is shown directly. Correctness is proved by relating a Depth-First Search path to a particular route through the Breadth-First Search graph built for the same set of rules.

Theorem 7 *(Soundness for the Depth-First Search Algorithm) Let T be a set of SNF rules, R the set of global $\square$-rules in T and $\mathcal{L}Q \Rightarrow \Diamond\neg l$ a rule in T. If when resolving with $\mathcal{L}Q \Rightarrow \Diamond\neg l$ the Depth-First Search Algorithm terminates detecting either $\{\mathbf{true}\}$ or the set of nodes $\{lits(A_0), \ldots lits(A_n)\}$ then either $\bigcirc\mathbf{true} \Rightarrow \square l$ or*

$$\bigcirc \bigvee_{i=0}^{n} A_i \Rightarrow \square l$$

respectively.

Proof
The Depth-First Search Algorithm returns either $\{\mathbf{true}\}$ or the set of nodes $\{\text{lits}(A_0), \ldots, \text{lits}(A_n)\}$ for resolution with $\mathcal{L}Q \Rightarrow \Diamond\neg l$. The only way the former may be returned is by detecting the rule $\bigcirc\mathbf{true} \Rightarrow l$ in R. Thus as l holds at every moment in time (except possibly the beginning of time) we have

$$\bigcirc\mathbf{true} \Rightarrow \square l$$

as required.

Otherwise we assume that the Depth-First Search Algorithm returns $M_L = \{\text{lits}(A_0), \ldots, \text{lits}(A_n)\}$ a set of nodes. Recall that nodes in Depth-First Search are the sets of literals occurring on the left hand side of rules. Each node $\text{lits}(A_i)$ for $i = 0$ to n and edges leading from it represent the use of an SNF_m rule

$$\bigcirc A_i \Rightarrow \bigvee_{j=0}^{m_i} C_{ij}$$

formed from combining rules in R. For each such rule, for all C_{ij} for $j = 0$ to m_i there exists a node $\text{lits}(A_k) \in M_L$, for $k = 0$ to n such that $\text{lits}(A_k) \subseteq \text{lits}(C_{ij})$, i.e. there is a node in the list of nodes returned by Depth-First Search that is a subset of the set of literals for each disjunct on the right hand side of the rule. So each disjunct on the right hand side implies (the conjunction of literals in) a node in M_L.

Assume the edges constructed by each expansion of the Depth-First Search Algorithm have been labelled by the set of SNF rules combined to obtain the SNF_m rule applied, so that the rules, L, which represent M_L are

$$\left\{ \begin{array}{lcl} \circledcirc A_0 & \Rightarrow & B_0 \\ \dots & & \dots \\ \circledcirc A_n & \Rightarrow & B_n \end{array} \right\}$$

To determine whether this set of rules implies the invariance of l we must check whether the two side conditions imposed on the set of rules for resolution with a $\diamondsuit$–rule, in the definition of the resolution rule given in Section 2.2.3, hold for L. We must test

1. $\vdash B_i \Rightarrow l$
2. $\vdash B_i \Rightarrow \bigvee_{j=0}^{n} A_j$

for all $0 \leqslant i \leqslant n$. To ensure the first condition initially we start the search from a node that is the set of literals from the left hand side of a rule of the form $\circledcirc X \Rightarrow l$ and when applying the forwards and backwards search algorithms we always check that l will hold in the next moment in time. Specifically during backwards search we check that the left hand side of a rule being used to expand the graph and all the disjuncts on the right hand side imply the left hand side of one of the start rules. With forwards search all the disjuncts on the right hand side of the rule being considered must imply the left hand side of one of the start rules. We need not check the left hand side is covered as we know the current node implies it which itself has been shown will give an l at the next moment.

For the second condition, during backwards search, we check that one disjunct on the right hand side implies the previous node (a left hand side of a rule) and the other disjuncts are stored to be solved using forwards search. With forwards search we test that the current node implies the left hand side of a rule where the current node is a disjunct from the right hand side of a rule thus ensuring

$$B_i \Rightarrow \bigvee_{j=0}^{n} A_j.$$

The algorithm only terminates when there are no disjuncts remaining to be solved and there is a looping path from each disjunct back to a node already visited.

In summary, the matching of the left or right hand sides of rules to the current node, only stopping when we have reached a node we have visited before and when there are no remaining disjuncts means that we will always be brought back to a rule in the set. The restriction of only allowing the expansion of nodes using rules that will generate a l means that l is ensured at each node. So once one of the left hand sides of the rules in L is satisfied, no edge can lead out of the loop and l will always be generated, i.e.

$$\circledcirc \bigvee_{i=1}^{n} A_i \Rightarrow \square l$$

as required. $\square$

To prove completeness of Depth-First Search we relate the expansion of the graph built during Depth-First Search to a particular route through the Breadth-First Search Graph. Note in this section when referring to the Breadth-First Search Graph we mean the graph described in Section 3.3 that is a sequence of nodes containing formulae in DNF. The graph built during Depth-First Search is very closely related to the set of SNF rules being considered, so that the construction of a new node directly represents the use of an SNF_m rule. Similarly, we require each disjunct in each node in Breadth-First Search also to be directly related to the use of an SNF_m rule. Three conditions are imposed so that the graph produced during Breadth-First Search is directly related to the SNF rule-set and therefore is more easily compared to that built during Depth-First Search. These conditions are shown not to affect the completeness of Breadth-First Search. Edges are then added to the Breadth-First Search graph between a disjunct in one node and disjuncts in the previous node to indicate the use of an SNF_m rule.

Definition 9 *(A restricted Breadth-First Search graph) Let T be a set of SNF rules, R the set of global $\square$-rules of T, and r, a $\diamondsuit$-rule in T for resolution. The graph G built using the Breadth-First Search Algorithm on R for resolution with r is known as a* restricted Breadth-First Search Graph *if the following conditions are imposed.*

1. *No left hand side resolution on pairs of SNF_m rules that have been used to build a particular node is allowed.*
2. *We add a (valid) rule of the form $\bigcirc\mathbf{true} \Rightarrow l \vee \neg l$ for every pair of complementary literals l that occur on the left hand side of rules.*
3. *Implication and equivalence testing are carried out by set comparison in order to test that either the SNF_m rules meet the expansion conditions, or that the last two nodes meet the termination conditions.*

By left hand side resolution we mean the repeated application of the rule

$$Y \vee (A \wedge l) \vee (B \wedge \neg l) \longrightarrow Y \vee (A \wedge l) \vee (B \wedge \neg l) \vee (A \wedge B)$$

where Y is a DNF formula, A and B are conjunctions of literals and l is a literal. The application of this rule is performed as part of the simplification of DNF nodes built in Breadth-First Search. This condition is enforced so that each disjunct in each node in Breadth-First Search relates exactly to the use of an SNF_m rule rather than a set of SNF_m rules on which left hand side resolution has been performed.

The second condition allows the addition of (valid) rules, as is done in Depth-First Search and is necessary if the first and third conditions are imposed.

The third condition ensures that *each* disjunct on the right hand side of each SNF_m rule, used to build a node in Breadth-First Search, must imply a disjunct in the previous node. We want to be able to compare (the literals from) each disjunct in nodes built during Breadth-First Search to nodes in Depth-First Search. As each node in Depth-First Search is considered individually not as part of a disjunction, this can be enforced in Breadth-First Search by ensuring implication tests are done by set comparison (as well as allowing no left hand side resolution).

We explain in more detail what we mean by stating that implication and equivalence tests are done by set comparison. Given a Breadth-First Search node in DNF,

$$H_i = \bigvee_{j=0}^{m_i} D_j$$

to test whether A, a conjunction of literals, implies H_i we must check whether $\exists j.\mathrm{lits}(D_j) \subseteq \mathrm{lits}(A)$. When A is a disjunction,

$$A = \bigvee_k C_k$$

we check that $\forall k\ \exists j.\mathrm{lits}(D_j) \subseteq \mathrm{lits}(C_k)$. Similarly when testing equivalences for the termination of Breadth-First Search, given two DNF nodes,

$$H_i = \bigvee_{j=0}^{m_i} D_{i,j} \text{ and } H_{i+1} = \bigvee_{k=0}^{m_{i+1}} D_{i+1,k}$$

we check $\forall j\ \exists k.\mathrm{lits}(D_{i+1,k}) \subseteq \mathrm{lits}(D_{i,j})$ and $\forall k\ \exists j.\mathrm{lits}(D_{i,j}) \subseteq \mathrm{lits}(D_{i+1,k})$. Note if we apply conditions 1 and 3 alone (without 2) we don't get the correct results for implication tests in all cases. For example given the DNF formula $(a\wedge\neg l)\vee(a\wedge l)$ to test whether a implies this formula we test either $\{a,\neg l\}\subseteq\{a\}$ or $\{a,l\}\subseteq\{a\}$ as $a\Rightarrow(a\wedge\neg l)\vee(a\wedge l)$. If a is the right hand side of an SNF$_m$ rule being used to expand the graph we can combine the rule with $\bigcirc\mathbf{true}\Rightarrow l\vee\neg l$ and then test whether $(a\wedge\neg l)\vee(a\wedge l)$ rather than a implies $(a\wedge\neg l)\vee(a\wedge l)$. Now the set comparison tests become $\{a,\neg l\}\subseteq\{a,\neg l\}$ and $\{a,l\}\subseteq\{a,l\}$ as required.

We show that these conditions make no difference to the completeness of Breadth-First Search.

Lemma 12 *Let T be a set of SNF rules, R the set of global $\square$-rules of T, and $\mathcal{L}Q\Rightarrow\Diamond\neg l$ a $\Diamond$-rule in T for resolution. If G and G' are the Breadth-First Search graph and restricted Breadth-First Search graph for R for resolution with $\mathcal{L}Q\Rightarrow\Diamond\neg l$ and allowing the terminal node in each graph to be repeated infinitely, then the nodes $H_i\in G$ and $H'_i\in G'$ are equivalent.*

Proof
By induction on the number of nodes in G and G'. Note that if we reach the terminal node in G before we do in G', i.e. for the terminal nodes H_T and $H'_{T'}$, $T<T'$, then set node $H_{T+j}=H_T$ for $j>0$ and vice versa.

For the base case, $i=0$, we show $H_0\Leftrightarrow H'_0$. If we are resolving with $\mathcal{L}Q\Rightarrow\Diamond\neg l$ to build the first node in each case we search for rules $\bigcirc X_h\Rightarrow l$ in R and disjoin the conjunctions on their left hand sides. Now conditions (2) and (3) play no part in building the top node of G' but condition (1) does. In H_0 we simplify by carrying out both subsumption and left hand side resolution, however in H'_0 we simplify with subsumption only. The application of left hand side resolution produces an equivalent formula so $H_0\Leftrightarrow H'_0$ as required.

We assume that the theorem holds for $i=j$ for $j>0$, i.e. $H_j\Leftrightarrow H'_j$ and show it holds for $i=j+1$, i.e. $H_{j+1}\Leftrightarrow H'_{j+1}$. To apply the Breadth-First Search Algorithm from H_j look for rules, r of the form

$$\bigcirc P\Rightarrow\bigvee_k F_k$$

that satisfy

1. $P\Rightarrow H_0$; and
2. $\bigvee_k F_k\Rightarrow H_j$.

As $H_0\Leftrightarrow H'_0$ then $P\Rightarrow H'_0$ and from our assumption $H_j\Leftrightarrow H'_j$ so $\bigvee_k F_k\Rightarrow H'_j$. However we must also ensure that we satisfy condition (3), i.e. for

$$H'_j=\bigvee_{m=0}^{w} d'_{jm}\quad\forall k\ \exists m\ .F_k\Rightarrow d'_{jm}$$

where d'_{jm} is a disjunct of H'_j. Now as $H_j\Leftrightarrow H'_j$ the only simplification performed on H_j but not on H'_j is left hand side resolution. As

$$\bigvee_k F_k\Rightarrow H'_j$$

holds we may combine the rule r with the rule $\bigcirc\mathbf{true}\Rightarrow p\vee\neg p$ for any complementary literals p and $\neg p$ in H'_j to obtain the rule

$$\bigcirc P\Rightarrow\bigvee_j F_j\wedge(p\vee\neg p)$$

ensuring

$$\bigvee_j F_j\wedge(p\vee\neg p)\Rightarrow H'_j.$$

This continues until a rule for each pair of complementary literals has been combined with r to form r'. Thus as we have added in all the literals that left hand side resolution has removed, each disjunct on the right hand side of r' implies a disjunct in H'_j thus satisfying condition (3). Now as each rule added to R in condition (2) has **true** on the left hand side the left hand side of r' remains unchanged as P and so $P \Rightarrow H'_0$ still holds. So r' satisfies the conditions for restricted Breadth-First Search. The left hand side of rules used to expand nodes H_j and H'_j respectively are disjoined and again the only difference is from carrying out left hand side resolution on H_{j+1} rather than H'_{j+1}. We know each application of left hand side resolution retains equivalence so $H_{j+1} \Leftrightarrow H'_{j+1}$ as required. $\square$

Definition 10 *(An annotated restricted Breadth-First Search graph) Let T be a set of SNF rules, R the set of global $\square$-rules of T and G a restricted Breadth-First Search Graph for R for resolution with a rule in T. Add directed edges between every disjunct in every node H_i for $i > 0$ and disjuncts in the previous node H_{i-1} in G as follows. For the disjunct $d_{i,j}$ in node H_i, in G, constructed by using the SNF$_m$ rule*

$$\circledcirc A \Rightarrow \bigvee_{k=0}^{m} B_k$$

such that $A \Leftrightarrow d_{i,j}$ and for all k, $k = 0$ to m there exists an h such that $B_k \Rightarrow d_{i-1,h}$, for $d_{i-1,h}$ a disjunct in H_{i-1}, add a directed arc between $d_{i,j}$ and each $d_{i-1,h}$. Label these arcs with the set of SNF rule numbers that have been combined to make the SNF$_m$ rule. Add directed edges similarly between the final node and itself. The remaining graph is known as an annotated restricted Breadth-First Search Graph.

Note as we have assumed the condition (3), the final node in the restricted Breadth-First Graph and its successor must be the same up to the ordering of literals so we can add edges between the final node and itself.

In Lemma 13 we consider the properties of the subgraph in the terminal node of the annotated restricted Breadth-First Search Graph. Then parts of this subgraph are removed that are not part of the core loop.

Lemma 13 *Let T be a set of SNF rules, R the set of global $\square$-rules of T and G an annotated restricted Breadth-First Search Graph constructed from rules in R for resolution with a $\diamondsuit$-rule in T. Let D_T be the set of disjuncts in the terminal node and E_T be the set of edges between these disjuncts as defined above, then $G_T = (D_T, E_T)$ satisfies the following*

1. *every node in D_T has an edge leading from it back into G_T, i.e. for every $d \in D_T$ there exists an edge $e = (d, d')$ where $e \in E_T$ and $d' \in D_T$;*
2. *the graph is* complete *for sets of disjunctive edges, i.e. if $d \in D_T$ for all edges $E_A = \{e_0, e_1, \ldots, e_m\}$ built from the rule*

$$\circledcirc A \Rightarrow \bigvee_{k=0}^{m} B_k$$

where $A \Leftrightarrow d$ and for all $k = 0$ to m, $e_k = (d, d')$ for some $d' \in D_T$ where $B_k \Rightarrow d'$ then $E_A \subseteq E_T$.

Proof
If we build the Breadth-First node, H_{T+1}, *following* the final Breadth-First node, H_T, every disjunct in H_{T+1} has at least one edge leading from it back into H_T representing the use of a rule

$$\circledcirc A \Rightarrow \bigvee_{k=0}^{m} B_k.$$

From the definition of an annotated restricted Breadth-First Search Graph there is a disjunct in H_{T+1} equivalent to A and each B_k implies a disjunction in H_T. However as H_T and H_{T+1} are the same up to the ordering of literals (as we know $H_T \Leftrightarrow H_{T+1}$ and by assumption 3 they are the same up to the ordering of literals) then every disjunct in H_T has an edge leading back into H_T. Thus there is an edge into every node in D_T from a node in D_T and if there is an edge representing the use of a rule with two or more disjuncts on the right hand side in E_T then edges for every other disjunct in this rule are in E_T.

Definition 11 *(Breadth compact loop) Let T be a set of SNF rules, R the set of global $\Box$-rules of T and G an annotated restricted Breadth-First Search Graph constructed from rules in R for resolution with a $\Diamond$-rule in T. Let D_T be the set of disjuncts in the terminal node and E_T be the set of edges between these disjuncts then $\forall d \in D_T$ if $\nexists d' \in D_T$ such that $(d', d) \in E_T$ then delete d from D_T and all the edges leading from d. Repeat until we cannot delete any more disjuncts. The remaining graph (D'_T, E'_T) is known as a* breadth compact loop, *(BCL).*

Lemma 14 *A BCL exists for every Breadth-First Search that terminates having detected a Breadth-First loop.*

Proof
For every Breadth-First Search that terminates detecting a loop we can build an annotated restricted Breadth-First Search Graph G where the final node is H_T. Let D_T be the set of disjuncts in the final node. We can construct a subgraph $G_T = (D_T, E_T)$ in the final node, by Lemma 13. The definition for a BCL means we recursively delete any nodes (and associated edges) with no edges leading into them. As we have shown that every disjunct in D_T has at least one edge leading from it and back into D_T, and D_T is finite then we must have at least one node with edges leading back into it. Thus not all nodes will be deleted and a BCL exists. $\Box$

Lemma 15 *Let T be a set of SNF rules, R the set of global $\Box$-rules of T and G an annotated restricted Breadth-First Search Graph constructed from rules in R for resolution with r, a $\Diamond$-rule in T of the form $\mathcal{L}Q \Rightarrow \Diamond\neg l$. Let D be the DNF formula in the terminal node of G. If disjuncts are removed from D to create a BCL, returning the remaining DNF formula D' for resolution with r we can generate the same resolvents from these rules and step resolution with rules in T as from the resolvents of D and r.*

Proof
Let $G_R = (N_R, E_R)$ be the reduced behaviour graph of R and N_D, $N_{D'}$ and $N_{L'}$ be sets of nodes in G_R where $n \in N_D$ iff $n \models D$, $n \in N_{D'}$ iff $n \models D'$ and $n \in N_{L'}$ iff n is a core node of D. From Lemma 11 all the resolvents from D and r can be generated by resolution with the core loop of D and r and step resolution between these rules and rules in T. For this lemma to hold we must show that $N_{L'} \subseteq N_{D'} \subseteq N_D$. This is sufficient as Lemma 11 shows that if by deleting nodes $N_{L'}$ from temporal resolution, through step resolution we can delete all the nodes in N_D so if $N_{L'} \subseteq N_{D'}$ and the nodes in $N_{D'}$ are deleted through temporal resolution then the remainder can similarly be deleted through resolution steps. We have $N_{L'} \subseteq N_D$ (from the definition of the core nodes of a loop) and $N_{D'} \subseteq N_D$ as $D' \Rightarrow D$.

For any node $n \in N_{L'}$ the following holds from Definition 7 of the core nodes for D

- $N_{L'} \subseteq N_D$;
- $n \models l$;
- if $(n, n') \in E_R$ then $n' \in N_{L'}$, i.e. any edge out of any node in $N_{L'}$ leads to a node also in $N_{L'}$; and
- there exists $(n', n) \in E_R$ such that $n' \in N_{L'}$, i.e. there is an edge into every node in $N_{L'}$ from a node in $N_{L'}$.

We must show that any $n \in N_{L'}$ is also in $N_{D'}$, i.e. $n \models N_{D'}$. The first condition states that $n \in N_D$ which certainly holds for $n \in N_{D'}$. The second states that l holds at n. This holds in $N_{D'}$ if we take any node $n' \in N_{D'}$, and edge $(n', n) \in E_R$ then $n \models l$ as required. The third condition states that $N_{L'}$ is a terminal subgraph and this is the case for $n \in N_{D'}$ as any disjunct in the BCL has an edge out of it, representing the use of an SNF rule, to other disjuncts in the BCL so $n \models D'$ and for node $(n, n') \in E_R$, $n' \models D'$. Finally there is an edge into every node in $N_{L'}$ from a node in $N_{L'}$. For any node $n \in N_{D'}$ we have $n \models D'$. We have assumed $n \models l$ as all nodes in $N_{L'}$ have l holding. Every disjunct remaining in the BCL has an edge back into it so for every node $n \models D' \wedge l$ in $N_{D'}$ has an edge into it from a node $n' \in N_{D'}$ so any $n \in N_{L'}$ must also be in $N_{D'}$.

Definition 12 *(A breadth construct path.) Let T be a set of SNF rules, R the set of global $\Box$-rules of T and G an annotated restricted Breadth-First Search Graph constructed from rules in R for resolution with r a $\Diamond$-rule in T. A* breadth construct path *(BCP) from node H_j in G is a sequence of disjuncts $[d_{j,i_j}, d_{j-1,i_{j-1}}, \dots d_{0,i_0}]$ where*

1. *for all $k = 0$ to j, d_{k,i_k} is a disjunct in H_k;*
2. *every adjacent pair have an edge between them; and*
3. *for any disjunct d_{k,i_k} in H_k, for $k = 0$ to j, in the BCP there is a disjunct from each previous node in order.*

Lemma 16 *Let T be a set of SNF rules, R the set of global $\Box$-rules of T and G an annotated restricted Breadth-First Search Graph constructed from rules in R for resolution with r, a $\Diamond$-rule in T. Given any disjunct, d_{j,i_j} from node H_j in G we can find a BCP of length j from $d_{j,i_j} \in H_j$ to $d_{0,i_0} \in H_0$.*

Proof
By induction on the length of the path. For the base case let $j = 0$ so the BCP is $[d_{0,i_0}]$ and the length of the path is 0. Assume the lemma holds for $j = k > 0$, i.e. there is a path from d_{k,i_k} to d_{0,i_0} of length k and show it holds for $j = k + 1$. Any disjunct $d_{k+1,i_{k+1}}$ in the Breadth-First node H_{k+1} in G is only created by the use of an SNF$_m$ rule $\circledcirc A \Rightarrow \bigvee_h B_h$ where $A \Leftrightarrow d_{k+1,i_{k+1}}$ and $\forall h\ \exists m\ B_h \Rightarrow d_{k,m}$ for $d_{k,m}$ a disjunct in node H_k. We have drawn edges between $(d_{k+1,i_{k+1}}, d_{k,m})$ for all h. Thus for any disjunct $d_{k+1,i_{k+1}}$ we pick in node H_{k+1} we will be able to follow an edge out of this into some disjunct d_{k,i_k} in node H_j and we know there is a BCP from d_{k,i_k} to d_{0,i_0} of length k so there is a path of length $k+1$ from $d_{k+1,i_{k+1}}$. $\Box$

Lemma 17 *Let T be a set of SNF rules, R the set of global $\Box$-rules of T and G an annotated restricted Breadth-First Search Graph constructed from rules in R for resolution with r, a $\Diamond$-rule in T. For any BCP from a disjunct $d_{T,k}$ on a BCL constructed from the terminal node of G we can use the Depth-First Search Algorithm to reach $d_{T,k}$ and each node visited will be equivalent to the next disjunct from the BCP taken in reverse order.*

Proof
Every pair of adjacent disjuncts $[d_{T,i_T}, \dots, d_{j,i_j}, d_{j-1,i_{j-1}}, \dots, d_{0,i_0}]$ on a BCP are related by an edge by the definition of a BCP. This edge represents the use of a rule

$$\circledcirc A \Rightarrow \bigvee_k B_k$$

where $A \Leftrightarrow d_{j,i_j}$ for d_{j,i_j} a disjunct in H_j and $B_k \Rightarrow d_{j-1,i_{j-1}}$ for $d_{j-1,i_{j-1}}$ a disjunct in H_{j-1} and some k. Recall in Depth-First Search backwards given some Depth-First Search node n_{i-1} we look for an SNF$_m$ rule

$$\circledcirc A \Rightarrow \bigvee_k B_k$$

where $B_k \Rightarrow \mathrm{con}(n_{i-1})$ for some k, all the other disjuncts on the right hand side imply the left hand side of one of the start rules, A implies the left hand side of one of the start rules, and set the next Depth-First Search node $n_i = \mathrm{lits}(A)$. This is precisely what we have above if $n_{i-1} = \mathrm{lits}(d_{j-1,i_{j-1}})$ then $n_i = \mathrm{lits}(d_{j,i_j})$. Thus we can find this path in Depth-First Search by using precisely the same SNF$_m$ rules labelling the edge in our BCP (in reverse order), providing we start from a node equivalent to d_{0,i_0}.

In Breadth-First Search d_{0,i_0} is just the left hand side of some rule $\circledcirc X \Rightarrow \neg l$ in R, assuming we are resolving with $\Diamond l$ as we have disallowed any left hand side simplification. When using the Depth-First Algorithm we start the search off from a start node that is just the literals from the left hand side of a rule of the form $\circledcirc X \Rightarrow \neg l$ so at some point we will start from a left hand side equivalent to d_{0,i_0}. $\Box$

Definition 13 *(A minimal breadth compact loop) Let T be a set of SNF rules, R the set of global $\Box$-rules of T and G an annotated restricted Breadth-First Search Graph constructed from rules in R for resolution with a $\Diamond$-rule in T. Take any disjunct, d from a BCL constructed from the terminal node in G and set $L = \{d\}$. For one set of edges E_d leaving d labelled by the same rules, for each $(d, d') \in E_d$ if $d' \notin L$ add d' to L and repeat for all d'. If $d' \in l$ then terminate for this branch. Remove the leads in to L as described in Definition 11. The remaining graph is a* minimal BCL *(MBCL).*

Lemma 18 *Let T be a set of SNF rules, R the set of global $\Box$-rules of T and G an annotated restricted Breadth-First Search Graph constructed from rules in R for resolution with a $\Diamond$-rule in T. Let D be the DNF formula from the terminal node of G and D' be the DNF formula remaining after deleting the disjuncts that are not part of any MBCL in the terminal node of G. The resolvents obtained from D and r can be generated from resolving with D' and r and step resolution between these rules and rules in T.*

Outline Proof
This is similar to the proof for Lemma 15. We try to split up the loops and then remove individual lead in. $\Box$

Lemma 19 *Let T be a set of SNF rules, R the set of global $\Box$-rules of T and G an annotated restricted Breadth-First Search Graph constructed from rules in R for resolution with a $\Diamond$-rule in T. For any minimal BCL (D_M, E_M) constructed from the terminal node of G we can find a Depth-First loop (N_D, E_D) such that $N_D = D_M$ and $E_D = E_M$.*

Every edge in E_M can be made in Depth-First Search backwards by using the SNF_m rule that the edge is labelled with as described in the proof for Lemma 17. In the definition of the MBCL we start with a disjunct and recursively follow directed edges forwards adding the nodes we arrive at until we reach nodes that are already in the set on all branches of the recursion. If instead of following *all* disjunctive branches leading from a node we just choose one and note that there are remaining branches to be solved, until the path selected returns to a node we have already visited we build something equivalent to what has been called a partial loop in the Depth-First Algorithm but having worked forwards not backwards. Therefore following the same edges backwards must build the same partial loop. Now all the disjuncts we reach following the disjunctive edges we did not select in building the MBCL equivalent of the partial loop have been saved and will be identical to those remaining to be solved from the Depth-First Search as we have used exactly the same SNF_m rules to build the partial loop. In the MBCL we have added these nodes into a set and follow down all branches from each node. In Depth-First Search we do a forward search from a node n_i that looks for rules of the form

$$\circledcirc A \Rightarrow \bigvee_k B_k$$

where $n_i \Rightarrow A$ and use B_k for some k as the next node, saving the other B_k's still to be solved. Again this is what we do in the construction of the MBCL but here we take just one disjunct at a time. $\Box$

Theorem 8 *(Sufficient Completeness of Depth-First Search.) Let T be a set of SNF rules, where R the set of global $\Box$-rules of T contains rules that may be combined to form a maximal proper loop in l, L, of the form $\circledcirc A_i \Rightarrow B_i$ for $i = 0$ to n for resolution with r, a rule in T. Every resolvent from L and r can be generated by detecting a set of Depth-First loops, resolving with r and step resolution between these rules and rules in T.*

Proof
Construct a Breadth-First Search Graph, G, for R for resolution with r. From the completeness of Breadth-First Search we have shown that D the DNF-formula in the terminal node of G satisfies

$$\models l \Rightarrow \left(D \Leftrightarrow \bigvee_{i=0}^{n} A_i \right).$$

By building an annotated restricted Breadth-First Search graph, G' rather than G, D', the DNF formula in the terminal node of G' satisfies $D' \Leftrightarrow D$ (Lemma 12). Construct the BCL in the terminal node of G', i.e. we remove disjuncts from D' that have no edges into them, leaving the DNF formula D''. Lemma 14 shows that a BCL exists and Lemma 15 shows that resolution with D' and r produces the same resolvents as from resolution with D'' and r and step resolution between these rules and rules in T. Construct the set of MCBLs in the terminal node of G'. For each MBCL we can construct an equivalent Depth-First loop (Lemma 19) and an appropriate lead in for any disjunct on the MBCL (Lemma 17). Let D''' be the DNF formula from the remaining disjuncts. From Lemma 18 resolution between D''' and r and step resolution between these rules and rules in T, can generate all the resolvents obtained from r and D as required. $\square$

Chapter 5

Heuristics for Temporal Resolution

Here we look at a variety of heuristics to be applied before or during the loop search algorithms discussed in Chapter 3. In Section 5.1 we describe a way of reducing the number of rules in which we must search for a loop without affecting the completeness of the loop finding algorithms. More specifically, some preprocessing of the data is carried out before the loop search algorithms are applied. In Sections 5.2 and 5.3 we examine strategies used to reduce the search space required for Depth and Breadth-First Search algorithms.

5.1 Reducing the Rule-Set

5.1.1 Overview

When looking at non-trivial examples we may have to search for loops in large rule-sets. The algorithm proposed below aims to reduce the size of rule-sets in which we must search without removing any rules that are necessary for the loops. In the following text we shall refer to this removal of rules as *pruning*. First we look at different SNF rules that may be combined to make a loop and use our observations to develop an algorithm to remove rules that we know will not form part of the loop.

Let us partition the literals, from the set of rules that form a loop, into those for *loop propagation* and those that provide the *required literal*. For example if we are resolving with $\mathcal{L}Q \Rightarrow \Diamond l$ and have the following rules

$$\begin{array}{rcl} \bigcirc a & \Rightarrow & a \\ \bigcirc a & \Rightarrow & \neg l \end{array}$$

that may be combined to make the loop

$$\bigcirc a \Rightarrow a \wedge \neg l$$

then the literal *a propagates the loop* and $\neg l$ is the *required literal*. Recall we can delete the lead into a loop without affecting the completenss of the loop search algorithm (see Section 4.4 Lemma 15). So if we ignore the *lead into the loop*, for example the SNF$_m$ rule $\bigcirc b \Rightarrow a \wedge \neg l$ in the loop

$$\begin{array}{rcl} \bigcirc b & \Rightarrow & a \wedge \neg l \\ \bigcirc a & \Rightarrow & a \wedge \neg l, \end{array}$$

then for any literals *propagating the loop* occurring on the left hand side of a rule there must be some rules with this literal occurring on the right hand side so that this rule will *fire*. The set of SNF rules

$$\begin{array}{rcl} \bigcirc a & \Rightarrow & \neg b \\ \bigcirc \neg b & \Rightarrow & a \\ \bigcirc a & \Rightarrow & \neg l \\ \bigcirc \neg b & \Rightarrow & \neg l \end{array}$$

may be combined to form the loop

$$\begin{array}{rcl} \bigcirc a & \Rightarrow & \neg b \wedge \neg l \\ \bigcirc \neg b & \Rightarrow & a \wedge \neg l \end{array}$$

if we are resolving with $\mathcal{L}Q \Rightarrow \Diamond l$. We note that a and $\neg b$, the literals for *loop propagation*, each occur on both the left hand side of some rules and the right hand side of some rules.

Loops we detect may involve rules that have a literal that is positive (i.e. a proposition that occurs in the scope of (zero or) an even number of negations) on the left of one of the rules and negative (i.e. a proposition that occurs in the scope of an odd number of negations) on the left of another, even though the loop does not depend on this literal (in the terminology we have used above this literal does not propagate the loop). Note when discussing positive and negative literals we are not looking at their polarity in the rule itself as described in [Mur82] but regarding the conjunction on the left and disjunction on the right of the rule on their own. For example, in the rules below, the literals b and $\neg b$ do not propagate the loop but we require all three of these rules to detect the loop.

$$\begin{array}{rcl} \bigcirc (a \wedge \neg b) & \Rightarrow & a \\ \bigcirc (a \wedge b) & \Rightarrow & a \\ \bigcirc a & \Rightarrow & \neg l \end{array}$$

When we carry out Breadth-First Search, b and $\neg b$ are removed through simplification and subsumption when generating the (disjunctive) nodes. This "matching" of positive and negative literals on the left hand side of rules has been called *left hand side resolution* previously in this thesis. When carrying out Depth-First Search we add the (valid) rule $\bigcirc\, \mathbf{true} \Rightarrow b \vee \neg b$ to enable us to detect this loop.

If we have not performed all the step resolution possible then SNF rules that can be combined to form loops may have a literal that is positive on the right hand side of one and negative on the right hand side of the other, even though the loop does not depend on that literal. For example with the following rule-set

$$\begin{array}{rcl} \bigcirc a & \Rightarrow & a \vee \neg b \\ \bigcirc a & \Rightarrow & a \vee b \\ \bigcirc a & \Rightarrow & \neg l \end{array}$$

performing step resolution we would resolve the first two rules to obtain $\bigcirc a \Rightarrow a$ which would then subsume these two rules.

Finally the *literal we require*, $\neg l$ may just occur on the right hand side of SNF rules merged to form loops or may be also used to *propagate the loop*, i.e. appear on the left hand side. Its negation however, the literal l, will not appear on its own on the right hand side as when we combine it with a start rule we obtain a rule with **false** on the right hand side. Similarly if l occurs as a disjunct on the right hand side of a rule then we may always resolve it with a start rule to remove the l. Finally we may have l occurring on the left hand side of a rule for example in the following rule-set

$$\begin{array}{rcl} \bigcirc l & \Rightarrow & \neg l \\ \bigcirc l & \Rightarrow & a \\ \bigcirc a & \Rightarrow & \neg l \\ \bigcirc a & \Rightarrow & a \end{array}$$

that gives the loop

$$\begin{array}{rcl} \bigcirc l & \Rightarrow & a \wedge \neg l \\ \bigcirc a & \Rightarrow & a \wedge \neg l. \end{array}$$

As previously mentioned the first of these SNF_m rules is the lead into the loop and can be deleted without affecting the completeness of resolution. We try to use the above observations to delete rules that will not form part of the loop.

5.1.2 An Algorithm to Reduce the Rule-Set

Let T be a set of SNF rules and R the set of global $\square$-rules in T. Assuming we are resolving with the rule $\mathcal{L}Q \Rightarrow \Diamond l$ in T, two sets of propositions are calculated:-

$\mathfrak{L}$ the set of propositions occuring positively on the left hand side of rules in R or negatively on the right hand side of rules in R; and

$\mathfrak{R}$ the set of propositions occuring positively on the right hand side of rules in R or negatively on the left hand side of rules in R.

Recall that the translation to SNF will have pushed all negations through to the propositions and carried out any simplification necessary leaving conjunctions of literals on the left hand side of rules and disjunctions of literals on the right hand side of rules. By a proposition occuring positively we mean that there are no negation signs directly preceeding the proposition and by occuring negatively we mean that there is a single negation sign preceding the proposition.

As $\neg l$ is the required literal this is deleted from each set. The two sets are compared and if a proposition appears in one set and not the other then any rules with this proposition in it are deleted from the rule-set. Effectively we are retaining propositions if we have

1. a positive p on the left side of a rule and a negative p on the left hand side of a rule : left hand side resolution
2. a positive p on the left side of a rule and a positive p on the right hand side of a rule : positive loop propagation
3. a positive p on the right side of a rule and a negative p on the right hand side of a rule : right hand side resolution
4. a negative p on the left side of a rule and a negative p on the right hand side of a rule : negative loop propagation

This process is continued until no more rules may be deleted and a loop search algorithm is used to find loops amongst the remaining rules. Note we just remove some of the global $\square$–rules for the purpose of detecting the next loop and require them to be added back into the set of rules for further rounds of step and temporal resolution.

Algorithm Assume we are resolving with $\mathcal{L}Q \Rightarrow \Diamond l$, that T is the set of SNF rules and $R \subseteq T$ is the set of global $\square$-rules before pruning takes place.

1. Delete any rules containing a positive occurrence of l either on the left or right hand sides.
2. Calculate the sets $\mathfrak{L}$ and $\mathfrak{R}$ from R.
3. Delete l from both sets.
4. If $\mathfrak{L} = \mathfrak{R}$ terminate returning R otherwise perform step 5.
5. Repeat until $\mathfrak{L} = \mathfrak{R}$.
   (a) If $p \in \mathfrak{L}$ and $p \notin \mathfrak{R}$ delete all rules in R containing p or $\neg p$ and set R to be the set of SNF rules following deletions.
   (b) If $p \in \mathfrak{R}$ and $p \notin \mathfrak{L}$ delete all rules in R containing p or $\neg p$ and set R to be the set of SNF rules following deletions.
6. Recalculate the sets $\mathfrak{L}$ and $\mathfrak{R}$ from R and proceed to step 4.

Example

Here we show the use of the algorithm on a set of rules assuming we are resolving with $\mathcal{L}Q \Rightarrow \Diamond l$. Note we can combine rules 1 and 3, 2 and 4 together to give a loop and also use rules 6, 7 and 8 to give a loop but rules 5, 9 and 10 are not required.

$$
\begin{array}{rrcl}
1 & \bigcirc a & \Rightarrow & \neg b \\
2 & \bigcirc \neg b & \Rightarrow & a \\
3 & \bigcirc a & \Rightarrow & \neg l \\
4 & \bigcirc \neg b & \Rightarrow & \neg l \\
5 & \bigcirc x & \Rightarrow & d \\
6 & \bigcirc (c \wedge \neg d) & \Rightarrow & c \\
7 & \bigcirc (c \wedge d) & \Rightarrow & c \\
8 & \bigcirc c & \Rightarrow & \neg l \\
9 & \bigcirc \neg b & \Rightarrow & \neg y \vee \neg b \vee x \\
10 & \bigcirc l & \Rightarrow & a
\end{array}
$$

1. Delete rule 10 as it contains l.
2. We build sets $\mathfrak{L}$ and $\mathfrak{R}$ as described and having deleted l from both sets we obtain $\mathfrak{L} = \{a, b, c, d, x, y\}$ and $\mathfrak{R} = \{a, b, c, d, x\}$.
3. We have $y \in \mathfrak{L}$ but $y \notin \mathfrak{R}$ so we delete any rules with y positive on the left hand side or negative on the right hand side (rule 9).
4. Next we recalculate $\mathfrak{L}$ and $\mathfrak{R}$, using rules 1–8 and obtain $\mathfrak{L} = \{a, b, c, d, x\}$ and $\mathfrak{R} = \{a, b, c, d\}$.
5. Again we have $x \in \mathfrak{L}$ but $x \notin \mathfrak{R}$ so we delete any rules with x positive on the left hand side or negative on the right hand side (rule 5) and remain with rules 1–4 and 6–8.
6. Again recalculating $\mathfrak{L}$ and $\mathfrak{R}$ we get $\mathfrak{L} = \{a, b, c, d\}$ and $\mathfrak{R} = \{a, b, c, d\}$, so $\mathfrak{L} = \mathfrak{R}$ and the algorithm returns with rules 1–4 and 6–8.

5.1.3 Improvements to The Algorithm

Related to Step Resolution

We assume that we have carried out all the step resolution possible so retaining rules on the grounds of right hand side resolution is unnecessary. As an improvement on our previous algorithm we define four sets of propositions so that rules would not be allowed to remain if the only reason was because of right hand side resolution.

The improved algorithm creates sets of propositions $\mathfrak{L}^+$,$\mathfrak{L}^-$,$\mathfrak{R}^+$ and $\mathfrak{R}^-$ defined as follows.

$\mathfrak{L}^+$ is the set of propositions that occur positively on the left hand side of the set of global $\Box$-rules.

$\mathfrak{L}^-$ is the set of propositions that occur negatively on the left hand side of the set of global $\Box$-rules.

$\mathfrak{R}^+$ is the set of propositions that occur positively on the right hand side of the set of global $\Box$-rules.

$\mathfrak{R}^-$ is the set of propositions that occur negatively on the right hand side of the set of global $\Box$-rules.

These sets are compared with each other and if, for example, a proposition occurs in $\mathfrak{L}^+$ and not in either $\mathfrak{L}^-$ or $\mathfrak{R}^+$ then any rules with this proposition occurring positively on the left hand side are deleted and the proposition is deleted from $\mathfrak{L}^+$. If a proposition p did occur in both $\mathfrak{L}^+$ and $\mathfrak{R}^+$ then we do not want to delete rules with p occuring positively on the left or positively on the right, as p might be a literal that is propagating the loop and these rules may be the ones combined to form the loops. Similarly if p is a member of both $\mathfrak{L}^+$ and $\mathfrak{L}^-$ then although p may not propagate the loop the rules with p (and $\neg p$) on the left hand side may be required to build the loop although p and $\neg p$ may be removed through some form of left hand side resolution.

Implementation Improvements

If we store the propositions only we must recalculate the members of each set after all deletions have been made as we may end up with a proposition occurring in one of the sets that belonged to a rule that has been deleted. However to prevent the need for recalculation we can, instead of building sets of propositions, build sets of pairs (p, c) where p, as previously, is the proposition and c is the number of occurrences with p in that position (i.e. right or left hand side of the rule) and sign (i.e. positive or negative). So after each cycle of deletions from the set of global $\Box$-rules, rather than having to recalculate the four sets of propositions, the pair $(p, c) \in \mathfrak{L}^+$, for example, indicates that there are 'c' occurences of p in a left hand side positive position left in the ruleset. When a rule is deleted, any literal occurring in that rule must have its count decremented in the relevant proposition set. If for a pair (p, c), c reaches 0, then this pair may be deleted from its respective set as p now does not occur in the set of rules in this position and sign (i.e. positive or negative).

5.1.4 An Improved Algorithm

Let us assume we are trying to resolve with

$$\mathcal{L}Q \Rightarrow \Diamond l$$

and the set of global-$\Box$ rules in SNF having performed all the step resolution possible is R.

1. If l occurs in a rule on the right or left hand side delete the rule.
2. Create the empty sets $\mathfrak{L}^+$,$\mathfrak{L}^-$,$\mathfrak{R}^+$ and $\mathfrak{R}^-$.
3. For each rule $r \in R$ in turn, increase the count in the relevant rule-set for each proposition occurring in r as follows.
   (a) If p is a proposition occuring positively on the left hand side of r, if p is not in $\mathfrak{L}^+$ then add $(p, 1)$ to $\mathfrak{L}^+$, otherwise search for (p, n) in $\mathfrak{L}^+$ and increment n by 1.
   (b) If p is a proposition occuring negatively on the left hand side of r, if p is not in $\mathfrak{L}^-$ then add $(p, 1)$ to $\mathfrak{L}^-$, otherwise search for (p, n) in $\mathfrak{L}^-$ and increment n by 1.
   (c) If p is a proposition occuring positively on the right hand side of r, if p is not in $\mathfrak{R}^+$ then add $(p, 1)$ to $\mathfrak{R}^+$, otherwise search for (p, n) in $\mathfrak{R}^+$ and increment n by 1.
   (d) If p is a proposition occuring negatively on the left hand side of r, if p is not in $\mathfrak{R}^-$ then add $(p, 1)$ to $\mathfrak{R}^-$, otherwise search for (p, n) in $\mathfrak{R}^-$ and increment n by 1.
4. Remove the proposition l from any of the sets $\mathfrak{L}^+$, $\mathfrak{L}^-$,$\mathfrak{R}^+$ or $\mathfrak{R}^-$.
5. Remove rules as follows.
   (a) If (p, m) is in $\mathfrak{L}^+$, for $m > 0$, and neither $\mathfrak{L}^-$ or $\mathfrak{R}^+$ have (p, n) as a member for $n > 0$ then delete all rules with p occurring positively on the left hand side, delete (p, m) from $\mathfrak{L}^+$ and decrement the count of the other propositions occurring in the rule in their respective sets.
   (b) If (p, m) is in $\mathfrak{L}^-$, for $m > 0$, and neither $\mathfrak{L}^+$ or $\mathfrak{R}^-$ have (p, n) as a member for $n > 0$ then delete all rules with p occurring negatively on the left hand side, delete (p, m) from $\mathfrak{L}^-$ and decrement the count of the other propositions occurring in the rule in their respective sets.
   (c) If (p, m) is in $\mathfrak{R}^+$, for $m > 0$, and (p, n) is not a member of $\mathfrak{L}^+$ for $n > 0$ then delete all rules with l occurring positively on the right hand side, delete (p, m) from $\mathfrak{R}^+$ and decrement the count of the other propositions occurring in the rule in their respective sets.
   (d) If (p, m) is in $\mathfrak{R}^-$, for $m > 0$, and (p, n) is not a member of $\mathfrak{L}^-$ for $n > 0$ then delete all rules with p occurring negatively on the right hand side, delete (p, m) from $\mathfrak{R}^-$ and decrement the count of the other propositions occurring in the rule in their respective sets.
6. Delete any members of the sets $\mathfrak{L}^+$,$\mathfrak{L}^-$,$\mathfrak{R}^+$ or $\mathfrak{R}^-$ whose count has got to 0.
7. Repeat steps 5 and 6 until no more rules can be deleted.

5.1.5 Example

Assume we are resolving with $\mathcal{L}Q \Rightarrow \Diamond l$ and we have the following global $\Box$-rules.

$$\begin{array}{lrcl}
1. & \bigcirc a & \Rightarrow & w \\
2. & \bigcirc b & \Rightarrow & \neg w \\
3. & \bigcirc \mathbf{true} & \Rightarrow & \neg a \vee \neg b \\
4. & \bigcirc (c \wedge \neg a) & \Rightarrow & c \\
5. & \bigcirc (c \wedge \neg b) & \Rightarrow & c \\
6. & \bigcirc c & \Rightarrow & \neg l
\end{array}$$

1. Firstly we build the sets $\mathfrak{L}^+$, $\mathfrak{L}^-$, $\mathfrak{R}^+$, and $\mathfrak{R}^-$ taking each rule in turn. In rule 1 a occurs positively on the left hand side so we add $(a, 1)$ to the set $\mathfrak{L}^+$ and w occurs positively on the right hand side so we add $(w, 1)$ to $\mathfrak{R}^+$. The four sets are $\mathfrak{L}^+ = \{(a, 1)\}$, $\mathfrak{L}^- = \{\}$, $\mathfrak{R}^+ = \{(w, 1)\}$, and $\mathfrak{R}^- = \{\}$.

2. In rule 2, b occurs positively on the left hand side so we add $(b, 1)$ to the set $\mathfrak{L}^+$ and w occurs negatively on the right hand side so we add $(w, 1)$ to $\mathfrak{R}^-$. The four sets are $\mathfrak{L}^+ = \{(a, 1), (b, 1)\}$, $\mathfrak{L}^- = \{\}$, $\mathfrak{R}^+ = \{(w, 1)\}$, and $\mathfrak{R}^- = \{(w, 1)\}$.

3. In rule 3 we have **true** on the left hand side so we don't add anything to the sets and a and b occur negatively on the right hand side so we add $(a, 1)$ and $(b, 1)$ to $\mathfrak{R}^-$. The four sets are $\mathfrak{L}^+ = \{(a, 1), (b, 1)\}$, $\mathfrak{L}^- = \{\}$, $\mathfrak{R}^+ = \{(w, 1)\}$, and $\mathfrak{R}^- = \{(w, 1), (a, 1), (b, 1)\}$.

4. We continue as above until we have processed all the rules, and having deleted any occurances of l we obtain the four sets $\mathfrak{L}^+ = \{(a, 1), (b, 1), (c, 3)\}$, $\mathfrak{L}^- = \{(a, 1), (b, 1)\}$, $\mathfrak{R}^+ = \{(w, 1), (c, 2)\}$, and $\mathfrak{R}^- = \{(w, 1), (a, 1), (b, 1)\}$.

5. When doing deletions, starting with $\mathfrak{L}^+$ we see a and b are both in $\mathfrak{L}^-$ and c is in $\mathfrak{R}^+$. Similarly for $\mathfrak{L}^-$ a and b are both in $\mathfrak{L}^+$. Looking at $\mathfrak{R}^+$ the proposition c occurs in $\mathfrak{L}^+$ but w does not occur in $\mathfrak{L}^+$ so we need to remove all the rules with w in the $\mathfrak{R}^+$ position and adjust all the other proposition counts. We see rule 1 is the only rule containing w in a $\mathfrak{R}^+$ position so remove this rule, delete w from $\mathfrak{R}^+$ and decrease the count of a, that occurs on the left hand side of rule 1, from $\mathfrak{L}^+$. The rules remaining in our set are rules 2–6 and our four sets (having deleted the propositions whose count have reach zero) are $\mathfrak{L}^+ = \{(b, 1), (c, 3)\}$, $\mathfrak{L}^- = \{(a, 1), (b, 1)\}$, $\mathfrak{R}^+ = \{(c, 2)\}$, and $\mathfrak{R}^- = \{(w, 1), (a, 1), (b, 1)\}$.

6. Looking at $\mathfrak{R}^-$ w doesn't occur in $\mathfrak{L}^-$ so we delete any rules with w in a right hand side negative position and decrease the count of any other propositions in these rules. Looking at the rule-set rule 2 is the only rule with $\neg w$ on the right hand side so we delete this rule leaving rules 3–6, and decrease the count for b occuring in a left hand side positive position in this rule. The sets of propositions, having deleted the propositions whose count has reached zero, become $\mathfrak{L}^+ = \{(c, 3)\}$, $\mathfrak{L}^- = \{(a, 1), (b, 1)\}$, $\mathfrak{R}^+ = \{(c, 2)\}$, and $\mathfrak{R}^- = \{(a, 1), (b, 1)\}$.

7. We cannot delete any more rules so the pruned rule-set becomes rules 3–6. We may now use a loop finding algorithm to detect any loops. In this example the loop is

$$\begin{array}{rcll}
\bigcirc (c \wedge \neg a) & \Rightarrow & (c \wedge \neg a \wedge \neg l) \vee (c \wedge \neg b \wedge \neg l) & [\mathrm{SNF}_m 3, 4, 6] \\
\bigcirc (c \wedge \neg b) & \Rightarrow & (c \wedge \neg a \wedge \neg l) \vee (c \wedge \neg b \wedge \neg l) & [\mathrm{SNF}_m 3, 5, 6]
\end{array}$$

5.1.6 The Example Using the Old Algorithm

The example we gave above is now carried out using the old algorithm and we show that with the newer algorithm we delete more of the unwanted rules.

1. First we build the sets $\mathfrak{L}$ and $\mathfrak{R}$ which after the deletion of l turn out to be $\mathfrak{L} = \{a, b, c, w\}$ and $\mathfrak{R} = \{a, b, c, w\}$.

2. As there are no propositions that occur in one set and not the other we cannot delete any rules from the rule-set so our pruned rule-set contains all the rules.

5.1.7 Correctness

Definition 14 *(The rules necessary for a loop.) Take a loop L of the form $\bigcirc A_i \Rightarrow B_i$ for $i = 0$ to m and for any rule $\bigcirc A_j \Rightarrow B_j$, for some $j = 0$ to m, such that $A_j \Rightarrow A_k$ for some $k = 0$ to m, $j \neq k$, delete $\bigcirc A_j \Rightarrow B_j$ from L. Continue until no more deletions are possible. The remaining set of rules is known as* the rules necessary for L

Note given a loop L of the form $\bigcirc A_i \Rightarrow B_i$ for $i = 0$ to m taking the rules necessary for L, L', of the form $\bigcirc A'_j \Rightarrow B'_j$ for $j = 0$ to m' then

$$\bigvee_{i=0}^{m} A_i \Leftrightarrow \bigvee_{j=0}^{m'} A'_j.$$

As we use the negation of the disjunction of these left hand sides to produce resolvents it makes no difference whether we use L or L' for resolution.

Lemma 20 *Let T be a set of SNF rules and R the set of global $\Box$-rules in T that contain a maximal proper loop L for resolution with $\mathcal{L}Q \Rightarrow \Diamond l$ a rule in T. Let D be the disjunction of the conjunction of literals on the left hand side of rules in L and L' be the core loop for D. Applying pruning before loop search will not remove any rules from R necessary for L'.*

Proof
Let L' be a set of SNF$_m$ rules of the form $\bigcirc A_i \Rightarrow B_i$ for $i = 0$ to m then as L' is a loop in $\neg l$ the following holds

- for all $i = 0$ to m, $B_i \Rightarrow \neg l$; and
- for all $i = 0$ to m, $B_i \Rightarrow \bigvee_{j=0}^{m} A_j$.

Let $G_R = (N_R, E_R)$ be the reduced behaviour graph for R and $N_{L'}$ be the set of nodes such that $n \in N_{L'}$ if and only if n is a member of the core nodes for D. From the definition of the core nodes for D if $n \in N_{L'}$ then

- for all $(n, n') \in E_R$, $n' \in N_{L'}$, i.e. all edges out of n lead to a node in $N_{L'}$;
- there exists $n' \in N_{L'}$ such that $(n', n) \in E_R$, i.e. n has an edge leading into it from a node in $N_{L'}$; and
- $n \models \neg l$, i.e. $\neg l$ holds at every node in $N_{L'}$.

Let us assume a rule, $r \in R$, has been deleted by pruning but is required to be combined for a rule, $\bigcirc A_h \Rightarrow B_h$ in L' and examine each case in turn. We assume all the step resolution has been carried out on the global $\Box$–rules.

1. Let l be a literal on the left or right hand side of r. The former cannot happen as we know that $\neg l$ holds in every node in $N_{L'}$ and the rules in L' have been built from the applicable ruleset for each node in $N_{L'}$ (by Definition 8 of a core loop). If l is a literal on the right hand side of r then as $B_h \Rightarrow \neg l$ holds and all step resolution has been carried out we must have combined a rule of the form $\bigcirc X \Rightarrow \neg l$ with the rules combined to give $\bigcirc A_h \Rightarrow B_h$. By performing step resolution between r and $\bigcirc X \Rightarrow \neg l$ we can remove l from the right hand side of r to give the rule r'. If we combine r' rather than r with the other rules we still obtain $\bigcirc A_h \Rightarrow B_h$. We have assumed that all step resolution has been carried out so deleting r will make no difference to L'.

2. Let x be a positive literal on the left hand side of r then we can delete this rule if
   (a) there is no rule with x on the right hand side $(\mathfrak{L}^+, \mathfrak{R}^+)$; and
   (b) there is no rule with $\neg x$ on the left hand side $(\mathfrak{L}^+, \mathfrak{L}^-)$.

The literal x will be a conjunct on the left hand side of $\bigcirc A_h \Rightarrow B_h$. From the second condition for being a loop we know each right hand side in L' must imply the disjunction of the left hand sides. Further as there is an edge into every node in $N_{L'}$ we can't delete any rules $\bigcirc A_j \Rightarrow B_j$ for some $j = 0$ to m from L' to give L'', where $\bigcirc A_j \Rightarrow B_j$ itself is not a loop and L'' is still a loop i.e. we have no leads into the loop. For r either

(a) to make r 'fire' there must be some B_i with an x as a conjunct as one of the disjuncts of B_i, so there must be an SNF rule with x on the right hand side to build this SNF$_m$ rule; or

(b) as for all $i = 0$ to m, $B_i \Rightarrow \bigvee_{j=0}^{m} A_j$ and no rules can be deleted we must be able to do some left hand side resolution so there must be some A_i where $\neg x$ is one of the conjuncts of A_i, but then $\neg x$ must appear on the left hand side of one of the SNF rules.

Either way we have got a contradiction with our assumptions so this rule cannot be part of the loop. The argument is similar for $\neg x$ a negative literal on the left hand side of r and deals with the cases $(\mathfrak{L}^-, \mathfrak{R}^-)$ and $(\mathfrak{L}^-, \mathfrak{L}^+)$.

3. Let x be a positive literal on the right hand side of r then we can delete r if there is no rule with x on the left hand side $(\mathfrak{R}^+, \mathfrak{L}^+)$.

   Then either we must have an A_i with x as a conjunct on the left hand side, i.e. there must be an SNF rule with x on the left hand side or we have combined the rules too many times and we can delete this from the loop and still obtain a loop. The argument is similar for $\neg x$ a negative literal on the right hand side of r giving the case $(\mathfrak{R}^-, \mathfrak{L}^-)$.

Theorem 9 *(Completeness of the Pruning Algorithm) Let T be a set of SNF rules and R the set of global $\square$-rules in T that contain a maximal proper loop L for resolution with $\mathcal{L}Q \Rightarrow \Diamond l$ a rule in T. If the pruning algorithm is applied before searching for a loop in $\neg l$ then any resolvents produced from resolution with L can be derived by applying pruning, loop search and step resolution.*

From Lemma 20 we show that applying pruning will not delete any rules necessary for L' the core loop for L. Lemma 11 shows that the resolvents from L and $\mathcal{L}Q \Rightarrow \Diamond l$ can be generated by resolution between the core loop L' for L and $\mathcal{L}Q \Rightarrow \Diamond l$ and by step resolution between these rules and rules in T. So by applying pruning before loop search we will still find a loop from which we can derive the same rules through temporal and then step resoulution as we could if no pruning had been applied. $\square$

5.1.8 Complexity of Pruning

Here we consider the complexity of various parts of the pruning algorithm. The cost of building the four sets of propositions $\mathfrak{R}^+$,$\mathfrak{R}^-$,$\mathfrak{L}^+$ and $\mathfrak{L}^-$, will be linear in the length of (the conjunction of) the global $\square$-rules. Given a set of m global $\square$-rules with n propositions, the worst case is if all n propositions can be deleted from the four sets of propositions. We must consider the cost of ensuring a proposition occurs in one set and not another. This will be of order n, so for n propositions will be of order n^2. Further, for each proposition deleted from $\mathfrak{R}^+$, $\mathfrak{R}^-$, $\mathfrak{L}^+$ and $\mathfrak{L}^-$, we will have to search through m rules. So if n propositions are to be deleted we will have to the search of the order of $n \times m$ rules.

5.1.9 Results

Statistics relating to the number of rules pruned and the time taken to detect loops for the examples given in Appendix A are given in Appendix B.3. Where pruning was employed on the rule-set, the loop times given include the time for the pruning. Examples where there are different eventualities shown mean that loops are detected for each eventuality and examples where an eventuality occurs more than once means that temporal resolution must be applied more than once for this eventuality. The loop search used in each case is Breadth-First Search.

Table B.3 shows the number of global $\square$–rules before temporal resolution takes place, the number global $\square$–rules after the rule-set has been pruned, timings for loop detection with and without pruning and the time for pruning alone for various examples.

5.1.10 Comments on Results

The timings of the loop searches are a trade off between the overhead required to prune rules from the rule-set plus the loop finding times on a potentially smaller data set, against the time for finding the loop in the full rule-set. Table 5.1 gives a summary of the results given in Table B.3 indicating the number of eventualities where the time for pruning followed by loop finding on the resulting rule-set is less than or equal to, or greater than the original. The figures in brackets are the percentages for these values. The first row of the table gives the figures for the full data set, subsequent rows take subsets of the data. The second row contains the data for all examples where the original rule-set contains more than 10 rules. The third row contains data for all examples where at least one of the timings is greater than 60 milliseconds. The reason we have chosen to isolate these subsets is that for the former we expect pruning to be particularly beneficial where the examples are not trivially small, and the latter to exclude any timing inaccuracies that may occur for timings that are very small.

Subset of Data	Pruning $\leqslant$ Original	Pruning $>$ Original
(i) Complete data set	39 (75%)	13 (25%)
(ii) No. rules > 10	26 (87%)	4 (13%)
(iii) Time > 60	21 (84%)	4 (16%)

Table 5.1: Summary of Results from Pruning

We note it is the same four examples where the time for pruning and loop finding is greater than the time for loop finding alone in the subsets of the data labelled (ii) and (iii). The times for these examples for the former are not larger than 25% more than the timings of the latter (in fact the actual figures are 13%, 1%, 1% and 22%) whereas the gain from using pruning on some examples is much greater. For the eventuality in Example 23 the timings for finding the loop with pruning is $3\frac{1}{2}$ times better than the time for finding the loop without pruning.

The column that gives the time for pruning alone is there to illustrate the time that pruning takes as compared to the number of rules. For all examples that have greater than 10 rules (to avoid the inaccuracies of the very low timings) the time for pruning lies between 1.2 and 3.2 times the number of rules in the rule set before pruning takes place.

The timings we have obtained suggest that pruning does help on larger, more difficult rule-sets.

5.2 Strategies for Depth-First Searches

Here we examine several heuristics to direct the search specific to the Depth-First Algorithm. Section 5.2.1 adopts a strategy aimed at reducing the search by trying to find the 'simplest' loops first. Section 5.2.2 cuts down the search by recording nodes through which we have searched before and know do not lead to or form part of a loop. Section 5.2.3 examines loop subsumption, i.e. when a new loop provides no new information over and above a loop already found.

5.2.1 Find Simple Loops First

In Chapter 1 the unit preference strategy for applying resolution to classical logic was described. To apply this strategy, unit clauses (i.e. clauses of just a single literal) are resolved with other clauses before any resolution between two non-unit clauses. One motivation for this is that the resolution of a unit and non-unit clause will result in a clause that is shorter in length than the non-unit clause. Another advantage of this strategy is its simplicity.

We can try to apply this type of strategy to temporal resolution by trying to detect *simple* loops first for an eventuality. Here by a simple loop we mean a loop that can be described using one SNF_m rule. Further, we would like to detect simple loops involving the smallest number of literals on the left hand side first. Examples of *simple* loops, assuming we are resolving with $\mathcal{L}Q \Rightarrow \Diamond \neg l$ are

$$\bigcirc a \Rightarrow a \wedge l$$

or

$$\bigcirc (b \wedge c) \Rightarrow (b \wedge c \wedge l)$$

where we would hope to detect the former before the latter.

As with the unit preference strategy in classical resolution, one motivation for this approach is its simplicity. The approach is easy to understand and implement, and by looking for a simple loop at each node of the Depth-First Graph we may avoid having to build a large portion of the graph. Secondly, by attempting to detect simple loops with the fewest literals on the left hand side first, we are searching for the simplest loops with the strongest resolvents first. For example the resolvents obtained from the loop $\bigcirc a \Rightarrow a \wedge l$ are stronger than from a loop with a larger number of literals, including a, on the left, $\bigcirc (a \wedge b \wedge c) \Rightarrow (a \wedge b \wedge c \wedge l)$. Therefore the resolvents of the former,

$$\begin{array}{rcl}
\bigcirc t & \Rightarrow & t \vee \neg l \\
\mathcal{L}Q & \Rightarrow & t \vee \neg l \\
\bigcirc t & \Rightarrow & \neg a \vee \neg l \\
\mathcal{L}Q & \Rightarrow & \neg a \vee \neg l \\
\bullet\, \mathbf{false} & \Rightarrow & \neg a \vee \neg Q \\
\bigcirc\, \mathbf{true} & \Rightarrow & \neg a \vee \neg Q
\end{array}$$

will subsume resolvents of the latter

$$\begin{array}{rcl}
\bigcirc t & \Rightarrow & t \vee \neg l \\
\mathcal{L}Q & \Rightarrow & t \vee \neg l \\
\bigcirc t & \Rightarrow & \neg a \vee \neg b \vee \neg c \vee \neg l \\
\mathcal{L}Q & \Rightarrow & \neg a \vee \neg b \vee \neg c \vee \neg l \\
\bullet\, \mathbf{false} & \Rightarrow & \neg a \vee \neg b \vee \neg c \vee \neg Q \\
\bigcirc\, \mathbf{true} & \Rightarrow & \neg a \vee \neg b \vee \neg c \vee \neg Q
\end{array}$$

where t is a new propositional variable.

In the original Depth-First Algorithm the order in which loops are detected depends on which rules or combinations of rules are used to expand the graph with at each step. The rules were used in the order they appeared in the database. Here, the proposed algorithm searches for a simple loop from each of the start nodes before attempting to build the graph from each of the start nodes.

Algorithm

Assume that we are resolving with $\mathcal{L}Q \Rightarrow \Diamond \neg l$

1. Search the rule-set for the rule $\bigcirc\, \mathbf{true} \Rightarrow l$. If we find it terminate returning $\{\mathbf{true}\}$.
2. Find the start nodes.
3. Order the start nodes according to the number of literals in each node, least first.
4. Look for a simple loop for each start node, i.e.
   (a) For a start node $\{x\}$ that is just a single literal search the rule-set for the rule $\bigcirc x \Rightarrow x$. If detected we terminate with a loop.
   (b) For a start node n, of more than one literal search for rules $\bigcirc P \Rightarrow x$ such that x is a single literal, $x \in n$ and $\mathrm{lits}(P) \subseteq n$. Combine these together and test to see whether we have a loop, if so we terminate with this loop.
5. Otherwise perform a backwards search from each start node in turn as usual.

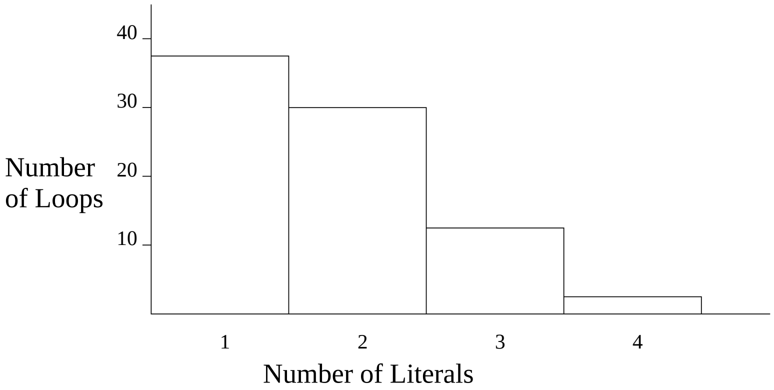

Figure 5.1: Bar Chart Showing the Size of Simple Loops Found

Extension of the Algorithm

Presently the algorithm we have suggested is only applied to each start node. We could, however, extend this approach to any node in the graph we build. To do this for backwards search, given a node n we look for the set of global $\square$-rules, $\bigcirc P \Rightarrow x$ such that $x \in n$ and $\text{lits}(P) \subseteq n$, as described in step 4 part b of the algorithm above. Combining this set, we will find a simple loop before doing any other combinations, if one exists.

The extended simple loops algorithm for Depth-First Search has been tried out on the examples given in Appendix A.

Results

By using the extended simple loops algorithm on the examples in Appendix A we can detect all the loops necessary to find a contradiction by searching for simple loops. The timings for Depth-First Search detecting simple loops first are given in Appendix B.1. We illustrate the type of loops found with the bar chart in Figure 5.1. The 'Number of Literals' axis refers to the number of literals on the left hand side of the loop detected. As all the examples we have tried can be solved by looking for simple loops this strategy looks promising. Figure 5.1 illustrates that we need to find more of the smaller simple loops than the larger ones. This may be because during the translation to normal form, the translation of a $\square$-formula, for example $\square P$, gives rules of the form $\bigcirc t \Rightarrow t$ and $\bigcirc t \Rightarrow P$ where t is a new proposition and $\bigcirc t \Rightarrow t$ is almost in simple loop form already. After step resolution takes place $\bigcirc t \Rightarrow t$ may be combined with one or more rules to produce an simple loop.

Completeness

Testing to see whether a simple loop exists from each node of the search graph is an ordering strategy, i.e. if detecting a simple loop fails for a particular node then the search proceeds as usual. So this strategy is complete due to the completeness of Depth-First Search.

5.2.2 Reducing the Search Graph

During Backwards Search

Recall during backwards search we match the right hand side of a rule with the current node and make the new node the set of literals on the left hand side of this rule (checking that the left hand side and each disjunct on the right hand side implies the left hand side of one of the start rules). Building up the graph in this manner we may use an SNF_m rule r to expand from node n and build a new node (n') that we cannot expand from, i.e. there are no rules that satisfy the conditions for backwards search. At this point in the algorithm we must backtrack to a previous node (here n) and select an alternative SNF_m rule with which to expand the graph. We may backtrack (several times) and build a new portion of the graph again generating node n'. Here the original Depth-First Algorithm would again check the rule-set for rules with which to expand n'. However as this has been done previously we know already that n' has no (backwards) successors and could save processing time by recording this. To do this we build a list of *failed nodes backwards*. When we commence a cycle of temporal resolution this list is empty. A node is added to the failed nodes backwards list if it has no backwards successors, or all its (backwards) successors are failed nodes backwards. So the list of failed nodes backwards represents sets of nodes that do not lead to a loop using the backwards search algorithm.

During Forwards Search

The same may happen during forwards search although this time we are expanding the graph in the direction of the arrows, i.e. we are looking for rules to build an edge (and node) out of the node where we are currently. Here we build a list of *failed nodes forwards*. Nodes are added to this list if for a node n' there are no rules we can apply to expand the graph that satisfy the conditions in the forwards search algorithm or all its forward successors are failed nodes forward.

Changes to the Algorithm

Before each cycle of temporal resolution we must create two empty lists F_B (failed nodes backwards) and F_F (failed nodes forwards). When we apply backwards search we must now check that the set of literals, from the left hand side of the rule, is not a member of F_B or any of the disjuncts on the right hand side are not members of F_F. That is for an SNF_m rule

$$\odot \bigwedge_k l_k \Rightarrow \bigvee_j D_j$$

we must make sure that

$$\begin{gathered}\nexists n \in F_B \text{ such that} \\ \mathrm{lits}(\textstyle\bigwedge_k l_k) = n \text{ and } \forall j \ \nexists n' \in F_F \text{ such that } n' = \mathrm{lits}(D_j).\end{gathered}$$

When applying forwards search we must check all disjuncts on the right hand side of the rule are not members of F_F, i.e.

$$\forall j \ \nexists n \in F_F \text{ such that } n = \mathrm{lits}(D_j)$$

and we must add the relevant nodes to F_B and F_F as explained above.

Example If we are resolving with $\mathcal{L}Q \Rightarrow \Diamond\neg l$, given the set of rules

$$\begin{array}{cccc} 1 & \odot a & \Rightarrow & l \\ 2 & \odot b & \Rightarrow & l \\ \\ 3 & \odot b & \Rightarrow & a \\ 4 & \odot x & \Rightarrow & b \end{array}$$

we start Depth-First Search from one of the start nodes, say $\{a\}$. We use rule 3 to expand the graph and set $\{b\}$ as the current node. Expanding from $\{b\}$ we combine rules 1 and 4 giving

$$\odot (a \wedge x) \Rightarrow b \wedge l$$

and set the current node to be $\{a, x\}$. There are no successors to $\{a, x\}$, as no rules have x on the right hand side, so we add $\{a, x\}$ to F_B. Looking for another rule to apply we combine rules 2 and 4 to get

$$\odot (b \wedge x) \Rightarrow b \wedge l$$

but again fail and add $\{b, x\}$ to F_B, so F_B is now $F_B = \{\{a, x\}, \{b, x\}\}$. We also fail when we combine rules 1, 2 and 4

$$\circledcirc (a \wedge b \wedge x) \Rightarrow b \wedge l$$

(as we would expect) and add $\{a, b, x\}$ to F_B. No other rules satisfy our backwards search expansion criterion (as for any other combination of rules the set of literals from the left hand side are alredy in F_B) so we backtrack adding $\{b\}$ to F_B. We have $F_B = \{\{a, x\}, \{b, x\}, \{a, b, x\}, \{b\}\}$. We look for alternative rules to expand backwards from $\{a\}$ and we can combine rules 1 and 3 to get

$$\circledcirc (a \wedge b) \Rightarrow a \wedge l.$$

Searching backwards from $\{a, b\}$ the only successors are $\{b, x\}$ (from combining rules 2, 3 and 4) or $\{a, b, x\}$ (from combining rules 1, 3 and 4 or 1, 2, 3 and 4) which are already in F_B. So $\{a, b\}$ also fails and the only other possible successor from $\{a\}$ is by combining all four rules giving $\{a, b, x\}$ which is also a member of F_B thus $\{a\}$ also fails. Now we try to begin the search from $\{b\}$ but notice that $\{b\}$ is a member of F_B so we terminate without finding a loop. A diagram of the full search path from each of nodes $\{a\}$ and $\{b\}$ are given in Figure 5.2. Note in the diagrams of Depth-First Search we have given previously, search paths that have reached dead ends have not been illustrated and only the portions that form part of the loop are depicted. Here we try to illustrate the full search.

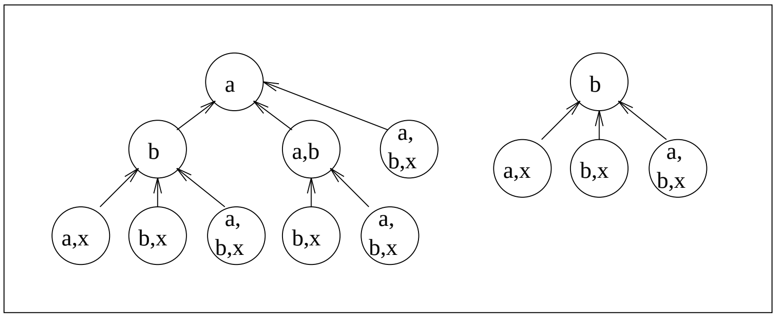

Figure 5.2: Full Depth-First Search Paths for a Set of Rules

In our example without a list containing failed nodes we would have to perform a backwards search from all 13 nodes shown in Figure 5.2. However using the idea of saving failed nodes we have only to perform the search from 6 nodes.

Improvements

We can refine our strategy to remove unsuccessful search paths further. Recall, for us to use a rule

$$\circledcirc \bigwedge_k l_k \Rightarrow \bigvee_j D_j$$

in backwards search requires $D_j \Rightarrow \mathrm{con}(n)$ for some j for n the current node and $\mathrm{con}(n)$ the conjunction of literals in the current node. So for some other node n' such that $n \subseteq n'$ if we can't find a successor for n then we won't be able to find a backwards successor for n' either. Rather than checking that the set of literals n' on the left hand side of a rule is a member of F_B we can check whether $\exists n \in F_B$ and $n \subseteq n'$. Further, to keep the length of F_B as short as possible, if we add a node n'' to F_B and $\exists n \in F_B$ such that $n'' \subseteq n$ then we may delete n from F_B.

With forwards search to expand using the rule

$$\circledcirc \bigwedge_k l_k \Rightarrow \bigvee_j D_j$$

we require $\mathrm{con}(n) \Rightarrow \bigwedge_k l_k$ for n the current node. If we can't find a forward successor for n then if for some other node n' and $n' \subseteq n$ then we won't be able to find a forwards successor for n' either. So, as above, rather than checking the set of literals n' constructed from each disjunct on the right hand side of a new rule is a member of F_F we can check whether $\exists n \in F_F$ such that $n' \subseteq n$. Further if we add a node n'' to F_F as $\exists n \in F_F$ such that $n \subseteq n''$ then n may be deleted from F_F.

Results

Rather than illustrate this strategy with timings for examples we note that we would expect such a strategy to have most noticable improvements on examples that were satisfiable rather than unsatisfiable. This is because a full Depth-First Search must be carried out from each start node for each eventuality, detecting no new loops, before the main processing algorithm can terminate. This generally involves exploring the same subgraph more than once as parts of the graph may be reached by several routes.

The storage of a list of nodes that do not lead to a loop does reduce the search that we have to perform if there is any chance of searching the same subgraph more than once. A disadvantage, however, to this strategy is the cost of storing and maintaining this list. If the list of nodes is long it may be quicker to build the *complete* search graph rather than check this list every time a new node is built. Again the strategy is a trade-off between the cost of building the search graph and the cost of searching and maintaining the lists of failed nodes. So, we will now consider how large the length of the lists of failed nodes forward and backwards could potentially reach.

If F_B and F_F are kept as small as possible, so for F_B any nodes n' are deleted if $n \in F_B$ and $n \subseteq n'$ and for F_F any nodes n' are deleted if $n \in F_B$ and $n' \subseteq n$, as described previously we consider the worst possible length of F_B and F_F dependant on the number of literals in the rule-set. Given a set of global $\square$-rules R containing m different literals an upper bound for the length of F_B and F_F can be calculated, by taking the length of the list containing sets of subsets of the literals in R such that none are subsets of each other. This is calculated by the number of combinations of size r taken from m or

$$mCr = \frac{m!}{r!(m-r)!}$$

and is largest for $r = m/2$ for m even and either $r = (m-1)/2$ or $r = (m+1)/2$ for m odd [And87].[1]

5.2.3 Loop Subsumption

Overview

The aim of this section is to identify the interaction between loops found in Depth-First Search used for temporal resolution with a $\Diamond$–rule. Then loops may be rejected as we know that the resolvents from a previous loop or the resolvents and a series of applications of the step resolution rule will give us no new information. So rather than adding the new resolvents to the rule-set, increasing the size of the rule-set without providing any new information and requiring processing to remove some or all of them through subsumption we can detect this as the loop is identified.

Given the following (satisfiable) set of rules

$$\begin{array}{rrcl}
1. & \bigcirc(a \wedge b) & \Rightarrow & a \\
2. & \bigcirc(b \wedge c) & \Rightarrow & b \\
3. & \bigcirc(c \wedge a) & \Rightarrow & c \\
4. & \bigcirc\mathbf{true} & \Rightarrow & a \\
5. & \bigcirc\mathbf{true} & \Rightarrow & b \\
6. & \bigcirc\mathbf{true} & \Rightarrow & c \\
7. & \bigcirc a & \Rightarrow & l \\
8. & \bullet\,\mathbf{false} & \Rightarrow & \Diamond\neg l
\end{array}$$

[1]Known as Sperner's Theorem.

the Breadth-First Search Algorithm would return (the DNF formula) (a). Simplification on (DNF) nodes in Breadth-First Search keeps nodes in their simplest forms, so for example we wouldn't return $(a \vee (a \wedge b))$ in this case.

With Depth-First Search the loop we would like to return is (the set of nodes) $\{\{a\}\}$. Assume this is the loop we detect on the first cycle of Depth-First Search (by combining rules 4 and 7 to give the SNF$_m$ rule $\circledcirc a \Rightarrow a \wedge l$), then from our Breadth-First Search solution we know we have no more loops and that this rule-set is satisfiable. However we must continue to do more cycles of Depth-First Search to look for more loops before we can assume the rule-set is satisfiable. The original Depth-First Algorithm may return the loops $\{\{a,b\}\}$ (SNF$_m$ 1,5,7), $\{\{a,c\}\}$ (SNF$_m$ 3,4,7), $\{\{a,b,c\}\}$ (SNF$_m$ 1,2,3,7) or even a loop that is a set made up from elements of these sets. However having found the loop $\{\{a\}\}$ none of the others will give us any extra information. To illustrate this we give the resolvents from $\{\{a\}\}$ and from $\{\{a,b,c\}\}$ where t is a new proposition

1. Loop is $\{\{a\}\}$

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & \neg a \vee \neg l \\ \bullet\,\mathbf{false} & \Rightarrow & t \vee \neg l \end{array}$$

$$\begin{array}{rcl} \circledcirc t & \Rightarrow & \neg a \vee \neg l \\ \circledcirc t & \Rightarrow & t \vee \neg l \end{array}$$

2. Loop is $\{\{a,b,c\}\}$

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & \neg a \vee \neg b \vee \neg c \vee \neg l \\ \bullet\,\mathbf{false} & \Rightarrow & t \vee \neg l \\ \circledcirc t & \Rightarrow & \neg a \vee \neg b \vee \neg c \vee \neg l \\ \circledcirc t & \Rightarrow & t \vee \neg l \end{array}$$

and note that each rule in 2 is subsumed by a rule in 1.

Definition 15 *A Depth-First loop L' is subsumed by L a set of nodes from previously detected loops for an eventuality if and only if for all nodes $n' \in L'$ there exists a node $n \in L$ and $n \subseteq n'$.*

That is, every node in the new loop is a superset of a node in some loop previously detected for this eventuality.

Lemma 21 *If a Depth-First loop L' is subsumed, resolvents obtained from L' would have been subsumed by rules already in the rule-set.*

Proof The resolvents generated from loops, written in SNF have two rules that are unrelated to the particular loops detected, i.e.

$$\begin{array}{rcl} \circledcirc t & \Rightarrow & t \vee \neg l \\ \mathcal{L}Q & \Rightarrow & t \vee \neg l \end{array}$$

if we are resolving with $\mathcal{L}Q \Rightarrow \Diamond\neg l$ and t is a new proposition symbol. It makes no difference to the completeness of temporal resolution if we use the same new variable (here t) for all the resolvents for an eventuality [Pei94]. In fact in [Pei94] this is assumed as part of the completeness proof and is known as *augmenting* the rule-set, as adding such rules for each eventuality may be done before any processing has taken place. So we can ignore these two rules and assume they are already in the rule-set if we have detected a loop previously.

The other resolvents, for each node in the loop n' where $\mathrm{con}(n')$ is the conjunction of literals in n', are of the form

$$\begin{array}{rcl} \mathcal{L}Q & \Rightarrow & \neg(\mathrm{con}(n')) \vee \neg p \\ \bullet\,\mathbf{false} & \Rightarrow & \neg(\mathrm{con}(n')) \vee \neg Q \\ \circledcirc\,\mathbf{true} & \Rightarrow & \neg(\mathrm{con}(n')) \vee \neg Q \\ \circledcirc t & \Rightarrow & \neg(\mathrm{con}(n')) \vee \neg p \end{array}$$

where, as Q and $\mathrm{con}(n')$ are conjunctions of literals their negations will be disjuncts of literals. In each case if we have had a node in a loop n and $n \subseteq n'$ then each of these resolvents will be subsumed by a rule generated by resolvents from a previous loop.

Completeness

Completeness follows directly from Lemma 21.

Complexity

Given L' a set of nodes from a 'new' loop and L a set of nodes obtained from loops derived previously we could test for loop subsumption by checking that

$$\bigvee_{n' \in L'} \text{lits}(n') \Rightarrow \bigvee_{n \in L} \text{lits}(n).$$

See Section 5.3.2 for a discussion of the complexity of this situation. Note, however that we must check that no 'new' loops can be detected for each eventuality anyway before the resolution procedure can terminate. So, some processing is required to establish this whether we are ensuring that each node in each new loop has been detected previously or that loop subsumption applies.

Example

Now for the set of rules given in Section 5.2.3 if we find loop $\{\{a\}\}$ first then the loops $\{\{a, b\}\}$, $\{\{a, c\}\}$, $\{\{a, b, c\}\}$ or any set made up of members of these sets (or $\{a\}$) will be subsumed as $\{a\} \subseteq \{a, b\}$ etc.

5.3 Strategies for Breadth-First Search

5.3.1 Left Hand Side Resolution

Overview

During step resolution the main inference rule described in Section 2.2.2 allows resolution between its two rules if we have a proposition and its negation occurring on the right hand side of a rule. Now we could add a second inference rule of the form

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

without affecting the soundness of the temporal resolution procedure. We shall call this *left hand side resolution*. For an informal justification we use the temporal equivalence $\circledcirc(A \wedge B) \Leftrightarrow \circledcirc A \wedge \circledcirc B$ and classical logic to rewrite the rule as

$$\begin{array}{c} \circledcirc \neg A \vee \circledcirc \neg r \vee F \\ \circledcirc \neg B \vee \circledcirc r \vee G \\ \hline \circledcirc \neg A \vee \circledcirc \neg B \vee F \vee G \end{array}$$

and can then rewrite this back into our rule format. Note we are allowed to resolve $\circledcirc r$ and $\circledcirc \neg r$ as both formulae refer to the same point in time (i.e. the previous moment). However, in general, we choose not to do this as in practice the application of such a rule produces too much information and we generate far too many new rules.

In the description of the Breadth-First Search Algorithm given in Section 3.3 we do use this inference rule as part of our simplification of nodes. Recall to expand from a node, H_i, (in DNF) we search for rules $\circledcirc A \Rightarrow B$ such that $B \Rightarrow H_i$ and $A \Rightarrow H_0$. Now given two rules

$$\begin{array}{rcl} \circledcirc(A_1 \wedge r) & \Rightarrow & B_1 \\ \circledcirc(A_2 \wedge \neg r) & \Rightarrow & B_2 \end{array}$$

and we know that

$$\begin{array}{rcl} (A_1 \wedge r) & \Rightarrow & H_0 \\ (A_2 \wedge \neg r) & \Rightarrow & H_0 \end{array}$$

and

$$\begin{array}{rcl} B_1 & \Rightarrow & H_i \\ B_2 & \Rightarrow & H_i. \end{array}$$

By applying the left hand side resolution rule we obtain the rule

$$\bigcirc(A_1 \wedge A_2) \Rightarrow B_1 \vee B_2$$

where

$$(A_1 \wedge A_2) \Rightarrow H_0$$

and

$$B_1 \vee B_2 \Rightarrow H_i.$$

So to build node H_{i+1} we have the left hand side of the two original rules

$$H_{i+1} = ((A_1 \wedge r) \vee (A_2 \wedge \neg r))$$

and by left hand side resolution can add the disjunct $(A_1 \wedge A_2)$ to get

$$H_{i+1} = ((A_1 \wedge r) \vee (A_2 \wedge \neg r) \vee (A_1 \wedge A_2)).$$

This means we can implement implication tests using set inclusion (see Section 4.4) avoiding having to add rules such as $\bigcirc\mathbf{true} \Rightarrow l \vee \neg l$ for every pair of literals on the left hand sides of rules.

Note left hand side resolution is not applied between all possible global $\square$-rules because, a noted previously too many rules are generated. We use it to 'simplify' nodes built in Breadth-First Search, i.e. we limit the application of left hand side resolution to between rules used to expand nodes in the Breadth-First Search graph.

Example

If we are resolving with $\mathcal{L}Q \Rightarrow \Diamond l$ and we search for loops in the rules

$$\begin{array}{rcl} \bigcirc a & \Rightarrow & \neg l \\ \bigcirc(a \wedge b) & \Rightarrow & a \\ \bigcirc(a \wedge \neg b) & \Rightarrow & a \end{array}$$

by Breadth-First Search we obtain (a) as the top node. We look for rules to satisfy the expansion criteria and find rules 2 and 3. Now adding in these we get $H_1 = ((a \vee b) \vee (a \wedge \neg b))$. Left hand side resolution gives us an extra disjunct a which subsumes the first two disjuncts. Now $H_0 = H_1$ so we terminate with (a).

Without left hand side resolution from node $H_1 = ((a \wedge b) \vee (a \wedge \neg b))$, if we are performing implication tests by set inclusion, we would have to add the rule $\bigcirc\mathbf{true} \Rightarrow b \vee \neg b$ and combine it with rule 2 and then 3 to get $H_2 = ((a \wedge b) \vee (a \wedge \neg b))$ and we terminate as $H_1 = H_2$ with $((a \wedge b) \vee (a \wedge \neg b))$ and note that this is equivalent to (a).

Completeness

If X is a DNF formula and $\sigma(X)$ is X after performing left hand side resolution then $X \Leftrightarrow \sigma(X)$. This is straight from classical logic as by applying left hand side resolution once on

$$Y \vee (A \wedge r) \vee (B \wedge \neg r)$$

where Y is a DNF formula, A and B are conjunctions of literals and r is a proposition we obtain

$$Y \vee (A \wedge r) \vee (B \wedge \neg r) \vee (A \wedge B)$$

which is equivalent. Nodes built in Breadth-First Search are used for testing implications (checking that the left hand side of rules imply H_0 and the right hand side of rules imply the previous node) and equivalences (to test for termination we must ensure $H_i \Leftrightarrow H_{i+1}$). If the formulae $H_i \Leftrightarrow \sigma(H_i)$ then the outcome of implication and equivalence tests should be then same with and without left hand side resolution. So as Breadth-First Search is complete then Breadth-First Search using left hand side resolution is also complete.

5.3.2 Loop Subsumption

Having detected a loop for an eventuality it is stored as a DNF formula. The next time resolution is performed on this eventuality even if the same DNF formula (up to the ordering of literals) is not detected we may be able to reject the new loop as it doesn't provide any new information.

For example having detected the loop

$$(a \vee b \vee c)$$

we would like to reject loops such as

$$(a \vee b)$$

or

$$((a \wedge d) \vee b \vee c).$$

In these cases every rule generated from the resolvents of the latter two loops will already exist in the rule-set or be subsumed by a rule already there (as long as the same new variable is used to translate all these loops into SNF). By rejecting such loops at the end of the temporal resolution step we save the effort required to translate the resolvents into SNF, and for them to be subsumed during step resolution as well as avoiding a further round of temporal resolution being done for this eventuality to again ensure no new loops are detected. So rather than ensuring the new loop L is the same as a previous loop L_i for $i = 1$ to n we may simply ensure

$$L \Rightarrow \bigvee_{i=1}^{n} L_i$$

to safely reject L as providing no new information. Note, we must check that no 'new' loops can be detected for each eventuality anyway before the resolution procedure can terminate. So some processing is required to establish this using loop subsumption or otherwise.

Completeness

As for loop subsumption in Depth-First Search (Section 5.2.3). The use of the same new variable to write resolvents into SNF each time the same eventuality is processed does not affect completeness of the resolution system [Pei94]. Hence when a new loop L satisfies

$$L \Rightarrow \bigvee_{i=1}^{b} L_i$$

for L_i for $i = 1$ to n the previous loops detected for this eventuality all the SNF rules it will be translated into will be subsumed by rules already in the rule-set.

Complexity

From above for each new loop we must test

$$L \Rightarrow \bigvee_{i=1}^{b} L_i.$$

The problem of validity of classical propositional logic is co-NP-complete [Hak85]. For an implication with 'n' propositions if we build the truth table for all valuations of the propositions then we must test 2^n valuations, i.e. an exponential algorithm.

Chapter 6

Strategies for Step Resolution

In this chapter we discuss strategies that can be applied during step resolution. As step resolution is similar to classical resolution most of the strategies suggested in this chapter are directly related to those found in classical resolution. Section 6.1 examines how several strategies for classical resolution might be applied to step resolution. Section 6.2 explains how a classical resolution theorem prover has been used to implement step resolution.

6.1 The Application of Classical Strategies

As we have seen in Section 2.2.2 the step resolution inference rule is very similar to the classical resolution inference rule except that it is applied to formulae that hold at the same moment in time. A number of strategies to guide classical resolution theorem provers have been developed over the years, see for example [CL73, Lov78, WOLB84] for a full discussion of these. Here we will mention some of the more widely used strategies and discuss their applicability to step resolution.

6.1.1 Set of Support

This strategy was introduced in [WCR64] and discussed further in [WRC65]. For a set of clauses, C, a non-empty subset of these C' is chosen as the *Set of Support*. Resolution is only permitted between two clauses if at least one of these clauses is a member, or derived from a member, of C'. The Set of Support (or SoS as we shall abbreviate to) is known as a restriction strategy as it restricts the application of the resolution rule, only allowing resolution between a member of the set of support (or clauses derived from this) and any other clause. The motivation behind the SoS strategy is to reduce the search space required to produce a refutation by preventing resolution between clauses that will not help produce a refutation. The SoS strategy is refutation complete if we choose the SoS (C') such that the set $C - C'$ is satisfiable. Two ways of choosing a SoS syntactically and semantically to satisfy this condition are given below.

1. **Syntactic**. If C is an unsatisfiable set of clauses choose C' as either the set of all the clauses containing no positive literals or all the clauses containing no negative literals. In the first case $C - C'$ is satisfiable, as if we assign true to all the positive literals then $C - C'$ must be true as each clause in $C - C'$ has at least one positive literal.

2. **Semantic**. Given a consistent database C if we are trying to prove C', i.e. $C \Rightarrow C'$, we negate and try to show $C \wedge \neg C'$ gives us a contradiction. If we put clauses relating to $\neg C'$ in the SoS then resolution is only allowed between clauses where at least one is from or derived from $\neg C'$. So the refutation is guided by restricting the application of the resolution inference rule to drawing conclusions that depend on the specific theorem to be proved.

The SoS idea is easily applied to step resolution. After a cycle of temporal resolution we have a set $I \cup R$ of rules previously derived, and a set of new rules $I' \cup R'$ from the translation of the resolvents of temporal resolution into SNF. By taking $I' \cup R'$ as the set of support, step resolution is only carried out with at least one of the parent rules being derived from the new temporal resolvents. This has been incorporated as an obvious strategy to prevent resolution between the same rules several times. However it is not clear how the SoS strategy could be applied on the first cycle of step resolution apart from putting all the $\square$–rules into the set of support. Certainly we could choose the SoS semantically if we are given such a formula to validate but for completeness the algorithms currently developed for temporal resolution require all step resolution to have been carried out before temporal resolution can take place.

6.1.2 Unit Resolution, Unit Preference Strategy

Unit resolution [WCR64], again a restriction strategy, requires that at least one of the parent clauses is a single literal (known as a *unit clause*). In general it is not refutation complete as can be seen from the following set of unsatisfiable clauses $\{p \vee q, p \vee \neg q, \neg p \vee q, \neg p \vee \neg q\}$. However, it is complete for particular sets of clauses for example Horn Clauses.[1] The unit preference strategy was the first refinement on basic resolution to appear [WCR64] and is an ordering strategy where unit resolution is performed before it would normally be performed in the search plan. As unit preference is an ordering strategy (rather than a restriction strategy) it is refutation complete as it merely provides an ordering on its application. These strategies are motivated by the idea that the final parts of refutations deal with short and unit clauses (in fact the last application of the resolution rules must be two unit clauses to gain a contradiction). Also resolution between a clause and unit clause will produce a new clause that is shorter than the non-unit clause. A further advantage is the simplicity of these strategies.

The unit preference strategy can be applied in a general way to step resolution by

1. first carrying out all the resolution between initial $\square$–rules (i.e. those with $\bullet$**false** on the left hand side); and
2. then carrying out resolution between all the global $\square$–rules (i.e. those with $\circledcirc \bigwedge_i l_i$ on the left hand side).

As it will be the resolution of two initial $\square$–rules that will give us the final contradiction, performing step resolution in this order will certainly save time on the final round of step resolution by applying the resolution rules where we know the contradiction will occur. Secondly, within these two sets we can resolve the rules with a single literal on the right hand side first, for example the initial $\square$–rule $\bullet$**false** $\Rightarrow a$ will be chosen to resolve with the other initial $\square$–rules before say $\bullet$**false** $\Rightarrow b \vee c \vee d$.

6.1.3 Hyperresolution

With hyperresolution [Rob65a] we take a clause C that is mixed or negative (i.e. has at least one negative literal) and resolve it with a set of positive clauses K (i.e. clauses that only contain positive literals). Each negative literal from C is resolved with a positive literal from a member of K. For Negative Hyperresolution we take a mixed or positive clause and resolve it with a set of negative clauses.

Hyperresolution considers simultaneously two or more clauses rather than just a pair in an aim to increase efficiency. It can be thought of as a succession of binary resolutions that produce no intermediate clauses. Hyperresolution is refutation complete when unrestricted but refutation completeness may be lost when constrained by the set of support.

A hyperresolution style rule could be formulated for rules in SNF. For example the clauses on the right hand side of rules with $\circledcirc$**true** on the left hand side could be resolved using the hyperresolution rule already described. If the resolvent of (some of) these clauses is X then the rule $\circledcirc$**true** $\Rightarrow X$ can be added to the set of SNF rules. A similar rule could be applied for SNF rules with conjunctions of literals

[1] Horn Clauses are clauses that contain at most one positive literal.

on their left hand sides. The advantage of reducing the number of intermediate resolvents created may not be an advantage when applying hyperresolution style rules to step resolution because currently the loop finding algorithms require all step resolution to be carried out before the loop finding algorithms are applied.

6.1.4 Weighting

Weighting, discussed in standard texts for example [WOLB84] is a mechanism to instruct a theorem prover to consider certain concepts or facts more important than others, i.e. it allows different degrees of significance to be assigned to different concepts. This type of strategy is used to assist the theorem prover by using knowledge or intuition about how to solve a problem.

Weighting can be applied to step resolution by giving the rules weights according to the number of propositions and connectives they contain to give the rules an ordering producing shorter simpler rules before longer more complicated ones. The ordering will be an extension of what was described in Section 6.1.2 as the complete rule-set will be ordered rather than just picking rules with a single literal on the right hand side for resolution first. It is unclear exactly what order will produce the best results. One suggestion is to partition the initial and global $\square$-rules according to their left hand sides, i.e. rules with the following on their left hand sides:-

- $\bullet\,\mathbf{false}$;
- $\circledcirc\,\mathbf{true}$;
- $\circledcirc \bigwedge_i c_i$.

Order the three groups as given here. Rules in each group can be ordered further dependent on the number of literals on their right hand sides. Rules in the third group can be ordered even further so that for all rules with n literals on the right hand side, rules with fewer literals on the left hand side should appear before rules with a larger number of literals, so that $\circledcirc\, a \Rightarrow c \vee d$ will appear before $\circledcirc\,(b \wedge c) \Rightarrow c \vee d$ for example. In this way the rules with the shortest right hand side in each group would be resolved during step resolution before longer ones, and rules with $\bullet\,\mathbf{false}$ on the left hand side resolved before rules with $\circledcirc\,\mathbf{true}$ on the left hand side etc. This type of ordering could also be used for the global $\square$-rules during temporal resolution so simpler rules would be chosen first for expanding the graphs built. This will help detect the simpler loops first (using the Depth-First Search Algorithm) and make simplification of nodes quicker and easier (if using the Breadth-First Algorithm).

6.1.5 Other Strategies

Other well known strategies for classical resolution for example input resolution, linear resolution, $\mathrm{P}_1/\mathrm{N}_1$ resolution and UR-resolution described in [CL73, Lov78, WOLB84] could also be applied to step resolution. However as previously mentioned we may encounter difficulties when applying the temporal resolution algorithms as their completeness depends on all the step resolution being carried out.

6.2 Utilising a Classical Resolution Theorem Prover to Implement Step Resolution

A prototype temporal resolution theorem prover was developed using SICStus Prolog [CW91]. Rather than invest time improving the programs performing step resolution we incorporated a resolution theorem prover for classical logics to perform step resolution. Then efficiency advantages would be gained, by utilising a well known, well tested program, as well as the having the potential to experiment with classical resolution strategies. The theorem prover chosen was OTTER [McC90]. OTTER (Organized Techniques for Theorem-proving and Effective Research) is a resolution theorem prover, for classical propositional and first-order logics, allowing the use of resolution inference rules and strategies, mentioned in this chapter and in Chapter 1. Implemented in C, input and output to OTTER is via files that set and clear flags relating to these inference rules and strategies, enable the input of the list of clauses (or formulae) and identify a Set of Support. In OTTER a clause

$$a \lor \neg b$$

is written as

$$a \mid -b.$$

6.2.1 Identifying Required Behaviour

As step resolution is similar but not identical to classical logic resolution we first examine the components of step resolution with the potential use of OTTER in mind. We have to rewrite rules in the temporal resolution framework into a format acceptable to OTTER, such that OTTER will only perform resolution between rules that we require step resolution between and will carry out as much simplification and subsumption as possible in the process. Specifically the problems we need to consider are identified below.

Relating to Step Resolution

1. **Preservation of left hand sides of rules.** The step resolution rule allows pairs of initial $\Box$–rules, or global $\Box$–rules, to be resolved using the following (step resolution) inference rule

$$\begin{array}{rcl} \mathcal{L}P & \Rightarrow & A \lor r \\ \mathcal{L}Q & \Rightarrow & B \lor \neg r \\ \hline \mathcal{L}(P \land Q) & \Rightarrow & A \lor B \end{array}$$

   where $\mathcal{L}$ is one of the last-time operators. If we use OTTER we need to preserve these left hand sides but must not do resolution on them.

2. **Only allow resolution between rules with the same last-time operator.** During step resolution we can only resolve pairs of initial $\Box$–rules, or global $\Box$–rules and may not resolve an initial $\Box$–rule with a global $\Box$–rule. When using OTTER we must either input the global $\Box$–rules together and then the initial $\Box$–rules, or rewrite the rules in such a way that resolution between the two sets cannot occur.

3. **Subsumption.** The subsumption rule is given as

$$\begin{array}{rcl} P & \Rightarrow & A \\ Q & \Rightarrow & B \\ \hline Q & \Rightarrow & B \end{array} \qquad \text{where } \vdash P \Rightarrow Q \text{ and } \vdash B \Rightarrow A.$$

   The side conditions $\vdash P \Rightarrow Q$ and $\vdash B \Rightarrow A$ must hold before this subsumption step can be applied. We need to be careful how we represent the left hand sides of rules in OTTER to make sure this is done correctly as OTTER performs subsumption in the usual way.

4. **Termination.** The step resolution process terminates when either no new resolvents are generated or **false** is derived by generating one of the following unsatisfiable formulae

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

   We would like the detection of the empty clause by OTTER to represent the derivation of one of the above rules.

5. **Removal of valid subformulae.** This is achieved by the removal of the following formulae.

$$\begin{array}{rcl} \bigcirc\,\mathbf{false} & \Rightarrow & A \\ P & \Rightarrow & \mathbf{true} \end{array}$$

We would hope that the translation we use into OTTER syntax would enable the normal removal of such rules through simplification in the OTTER system.

6. **Rewriting rules which imply false.** This is done by applying the following rule.

$$\frac{\bigcirc P \;\;\Rightarrow\;\; \mathbf{false}}{\bullet\,\mathbf{true} \;\;\Rightarrow\;\; \neg P}$$

This rule allows the addition of extra constraints, representing $\Box\neg P$, if by satisfying P in the last moment in time we get a contradiction. This rule is very different to anything we carry out in classical resolution and therefore we expect it to be difficult to replicate using OTTER.

Relating to OTTER

SoS OTTER uses two lists *usable* and *sos* to input its sets of clauses. We must decide which clauses should be included in each set so we do not repeat resolution steps carried out on previous cycles, but do attempt to generate all possible resolvents as currently our loop finding algorithms depend on this.

The OTTER File and Output. Input to OTTER is via a file that sets and clears various flags depending on the output and processing required. We must choose these settings to make OTTER do as much work as possible.

6.2.2 Obtaining Required Behaviour

Step Resolution

If the rules in SNF are transformed in the following way then some of the problems listed above will be avoided.

$$\begin{array}{rclcl} \bullet\,\mathbf{false} & \Rightarrow & \bigvee_{j=1}^{n} l_j & \longrightarrow & R_1(l_1) \mid R_1(l_2) \mid \ldots \mid R_1(l_n) \\ \bigcirc\,\mathbf{true} & \Rightarrow & \bigvee_{j=1}^{n} l_j & \longrightarrow & R_2(l_1) \mid R_2(l_2) \mid \ldots \mid R_2(l_n) \\ \bigcirc \bigwedge_{i=1}^{m} k_i & \Rightarrow & \bigvee_{j=1}^{n} l_j & \longrightarrow & -L(k_1) \mid -L(k_2) \mid \ldots \mid -L(k_m) \\ & & & & \mid R_2(l_1) \mid R_2(l_2) \mid \ldots \mid R_2(l_n) \end{array}$$

Where the literals on the left hand side of rules are rewritten as propositions as follows

$$\begin{array}{rcl} L(l) & \longrightarrow & slast\_l \\ L(\neg l) & \longrightarrow & slast\_neg\_l \end{array}$$

and literals on the right hand side of rules are rewritten as the following propositions

$$\begin{array}{rcl} R_1(l) & \longrightarrow & l \\ R_1(\neg l) & \longrightarrow & -l \end{array}$$

for initial $\Box$–rules and as

$$\begin{array}{rcl} R_2(l) & \longrightarrow & s\_l \\ R_2(\neg l) & \longrightarrow & -s\_l \end{array}$$

for global $\Box$–rules. Rewriting in this way solves problems 1–5 given in Section 6.2.1 as we explain below.

1. We have preserved the literals on the left hand side of the global $\square$–rules by splitting up the conjuncts and prefixing them with "*slast_neg_*" when they are preceeded by a negation and "*slast_*" otherwise. In this way the literals and their sign are preserved but resolution between these literals will never take place as we further prefix with a '$-$' sign so OTTER will assume they all are negative literals.

2. By prefixing the propositions occurring in the right hand sides of the global $\square$–rules with an "s_" we will only resolve together rules that are either global $\square$–rules or initial $\square$–rules. Note, in order that OTTER resolves positive and negative literals as we require negative literals have an outermost "$-$" sign (negation in OTTER syntax) while positive literals do not. This is different to how we translate literals from the left hand side of rules above.

3. Subsumption should work now in the classical propositional way due to prefixing the literals on the right hand side of global rules by "*s_*", rewriting conjuncts on the left hand side of rules separately, prefixing them by "*–slast_*", and removing left hand sides which are $\bullet$**false** and $\circledcirc$**true**. For example previously we might have had the rule $\circledcirc(a \wedge b) \Rightarrow p \vee q$ which is subsumed by $\circledcirc a \Rightarrow p$. Now in the OTTER format we will have the clause $-slast\_a \mid -slast\_b \mid p \mid q$ which is subsumed by $-slast\_a \mid p$.

4. Termination will be detected by OTTER. Pairs of resolvents e.g. $\bullet$**false** $\Rightarrow a$ and $\bullet$**false** $\Rightarrow \neg a$ which may be resolved to give us $\bullet$**false** $\Rightarrow$ **false**, will now be the clauses a and $-a$ in OTTER which when resolved give us the empty clause. Similarly $\circledcirc$**true** $\Rightarrow a$ and $\circledcirc$**true** $\Rightarrow \neg a$ will now be clauses s_a and $-s\_a$ in OTTER again producing the empty clause.

5. The removal of *some* valid subformulae will be carried out automatically. Any resolution step in OTTER that makes a clause become true (i.e. $P \Rightarrow$ **true** in our SNF rule form) will make that clause be removed from the set of rules so $P \Rightarrow$ **true** is removed automatically. However, we may perform resolution steps that allow $\circledcirc$**false** $\Rightarrow A$ to occur for example by resolving $\circledcirc a \Rightarrow p \vee q$ and $\circledcirc \neg a \Rightarrow \neg p$ we obtain $\circledcirc$**false** $\Rightarrow q$. In OTTER format this is attempting to resolve $-slast\_a \mid s\_p \mid s\_q$ with $-slast\_neg\_a \mid -s\_p$ and will give $-slast\_a \mid -slast\_neg\_a \mid s\_q$. Such formulae must be removed when rewriting back from OTTER format into Prolog for use in the temporal resolution process.

6. The removal of rules that imply false is more difficult and involves returning to Prolog code to search for such rules, rewriting them and re-inputting to OTTER. We wish to achieve the rewrite expressed by the following rule in OTTER syntax

$$-L(k_1) \mid \ldots \mid -L(k_m) \quad \longrightarrow \quad \left\{ \begin{array}{l} R_{f1}(k_1) \mid \ldots \mid R_{f1}(k_m) \\ R_{f2}(k_1) \mid \ldots \mid R_{f2}(k_m) \end{array} \right\}$$

   where

$$\text{if } L(k_i) = slast\_k_i \text{ then } R_{f1}(k_i) = -k_i \text{ and } R_{f2}(k_i) = -s\_k_i$$
$$\text{if } L(k_i) = slast\_neg\_k_i \text{ then } R_{f1}(k_i) = k_i \text{ and } R_{f2}(k_i) = s\_k_i.$$

However the rules we wish to rewrite will be obvious as there will be no literals on the right hand side.

Relating to OTTER

SoS OTTER uses a SoS strategy and to generate all the required clauses we must put all the clauses into the SoS. OTTER's main loop for inferring and processing clauses [McC90] is:-

```
While (sos is not empty and no refutation has been found)
    1. Select ‘‘next’’ clause in sos;
    2. Move from sos to usable;
    3. Infer and process new clauses using the inference rules
          in effect; each new clause must have this clause as
          one of its parents and members of usable as its
```

```
            other parents;  new clauses that pass the retention
            tests are appended to sos;
    End of while loop.
```

If we put all our clauses in the SoS the first time we use OTTER then we will generate all the clauses as required, however if we put all of the clauses into the usable list then we will get no resolvents at all. Following that we must use the previous rule-set as the *usable* rules and any new resolvents as the *sos*.

The OTTER File and Output The template for the OTTER file we used and explanation of the terms is given in Appendix C

6.2.3 The Use of OTTER in the Main Processing Loop

Previously, all the subprograms were written in Prolog. In the new version, after the translation to SNF, the set of rules is output in OTTER format. This output is parsed using Emacs Lisp, which creates the correct input for, and runs OTTER. The set of clauses generated by running OTTER are output in Prolog syntax and processing continues in Prolog.

Main Algorithm

Set the lists *usable* and *sos* to empty.

1. Input formula, translate it to SNF and store the resulting initial and global $\square$-rules in *sos*.
2. Output lists *usable* and *sos* in OTTER syntax.
3. Generate input for and run OTTER with the two lists of rules *usable* and *sos*.
4. Parse OTTER output searching for
   (a) a proof by detecting the text `----> UNIT CONFLICT` ; the procedure terminates having detected a contradiction.
   (b) the new list of usable rules, which will now contain the *usable* and *sos* lists input to OTTER and any new resolvents. Translate them into Prolog syntax and output.
5. Set *usable* to be the rules output from OTTER and *sos* to be empty.
6. (a) If *usable* contains rules of the form $\bigcirc P \Rightarrow \mathbf{false}$ delete from *usable*, add the rules $\bullet\, \mathbf{false} \Rightarrow \neg P$ and $\bigcirc \mathbf{true} \Rightarrow \neg P$ to *sos* and continue processing at part 2;
   (b) otherwise perform temporal resolution on *usable*.
7. (a) If no new resolvents have been returned from temporal resolution then we terminate having found the formula satisfiable;
   (b) otherwise translate the new resolvents into SNF, store in *sos* and continue processing at part 2.

6.2.4 Problems with the Translation to OTTER

1. The use of the flag **process_input** means that no subsumption needs to be done in Prolog at all (even after translation of the formula into SNF, or when we translate rules that imply false for further input to OTTER), but also means that every time we use OTTER many rules will be checked for subsumption between which we know no subsumption can possibly take place. Thus, this involves redundancy.

2. **Termination of the Prolog-OTTER cycle.** Incorporating the use of OTTER into the main algorithm we look for rules that imply **false**, rewrite them into the form we require, add them to the set of support, delete the original from the set of usable rules and input the two sets of rules to OTTER. The two rules that were resolved together to produce a rule with **false** on the right hand side cannot be resolved together again because both will be moved into the *usable* set of rules and resolution must be between a rule that is in the *sos* and another rule. However, if by performing different resolution steps, we re-generate the same rule that implies **false** we need an "extra" input to OTTER to detect this (see the example below). One way to prevent continual regeneration of rules that imply **false** is to keep them in the rule-set but then we cannot tell whether further cycles of step resolution are required. We will not continue to generate the same rule with **false** on its right hand side and continue passing control from Prolog to OTTER because when two rules are resolved together that imply **false** in the next moment in time, those two rules are moved into the set of usable rules and the rewritten rules are put in the set of support. However, we may resolve other rules that give us the same rule implying false in the next moment in time. Again this rule is rewritten and put in the set of support. When the input to OTTER is processed the two rules in the set of support are subsumed by the pair of identical rules already in the usable list.
   Example
   Given the rule-set
$$
\begin{array}{lllll}
1 & [] & \bigcirc a & \Rightarrow & \neg z \\
2 & [] & \bigcirc b & \Rightarrow & z
\end{array}
$$
$$
\begin{array}{lllll}
3 & [] & \bigcirc b & \Rightarrow & a \\
4 & [] & \bigcirc a & \Rightarrow & b
\end{array}
$$
   this will be used as the SoS for input to OTTER. Output from OTTER, translated into the usual syntax, will be as follows.
$$
\begin{array}{lllll}
1 & [] & \bigcirc a & \Rightarrow & \neg z \\
2 & [] & \bigcirc b & \Rightarrow & z \\
3 & [] & \bigcirc b & \Rightarrow & a \\
4 & [] & \bigcirc a & \Rightarrow & b \\
5 & [1,2] & \bigcirc a \wedge b & \Rightarrow & \mathbf{false}
\end{array}
$$
   The first four rules will be input as the usable rules. Rule 5 will be deleted and rewritten as the following two rules which will be re-input to OTTER as the SoS.
$$
\begin{array}{lllll}
6 & [5] & \bigcirc \mathbf{true} & \Rightarrow & \neg a \vee \neg b \\
7 & [5] & \bullet \mathbf{false} & \Rightarrow & \neg a \vee \neg b
\end{array}
$$
   On the second input to OTTER we will obtain the output
$$
\begin{array}{lllll}
1 & [] & \bigcirc a & \Rightarrow & \neg z \\
2 & [] & \bigcirc b & \Rightarrow & z \\
3 & [] & \bigcirc b & \Rightarrow & a \\
4 & [] & \bigcirc a & \Rightarrow & b \\
6 & [5] & \bigcirc \mathbf{true} & \Rightarrow & \neg a \vee \neg b \\
7 & [5] & \bullet \mathbf{false} & \Rightarrow & \neg a \vee \neg b \\
8 & [6,4] & \bigcirc a & \Rightarrow & \neg a \\
9 & [6,3] & \bigcirc b & \Rightarrow & \neg b \\
10 & [8,3] & \bigcirc a \wedge b & \Rightarrow & \mathbf{false}
\end{array}
$$
   and have derived $\bigcirc a \wedge b \Rightarrow \mathbf{false}$ for the second time. However, on the next input to OTTER rules 1–4 and 6–9 will be the usable rules and the SoS will have rules that are rewrites of rule 10 (and are identical to rules 6 and 7).
$$
\begin{array}{lllll}
11 & [10] & \bigcirc \mathbf{true} & \Rightarrow & \neg a \vee \neg b \\
12 & [10] & \bullet \mathbf{false} & \Rightarrow & \neg a \vee \neg b
\end{array}
$$

When the input to OTTER is processed rules 11 and 12 are subsumed by rules 6 and 7 making the SoS empty, no resolution can take place and the rules output are 1–4, 6–9. No further inputs to OTTER are attempted as the rules output do not contain any rules of the form $\bigcirc P \Rightarrow \mathbf{false}$.

6.2.5 Results

Using the new system linked to OTTER, examples that had previously run out of memory during step resolution (implemented in Prolog), examples 16, 22, 23, 24, 39 and 42, could now perform all the step resolution required and (using the Breadth-First loop finding algorithm for the temporal resolution) continued to produce the expected contradiction.

Chapter 7

Efficient Translation to Normal Form

7.1 Different Translations to SNF

In this chapter we consider how changes in the translation to SNF affect the size and simplicity of the SNF rules produced. After looking at some of the factors that may affect the size of SNF formula produced, slightly different ways of translating formulae into SNF are tried. Statistics for three different translations to SNF are given for the formulae in Appendix A and conclusions drawn. The advantage of translating arbitrary temporal formulae to a normal form is that most of the temporal operators are removed and we must then only define inference rules necessary for the remaining temporal operators. However, for this gain in simplicity of formulae we must sacrifice a potential increase in the number of proposition symbols required (recall that new propositions are added, for example, during the removal of temporal operators and due to renaming) and an increase in the overall length of the formula.

The order in which certain transformations are carried out, the use of temporal equivalences as rewrite rules and the choice of which transformation to carry out may affect the length of the conjunction of the rules in SNF and number of new propositions required.

Example Given the formula

$$\newcommand{\start}{\bullet} \start\,\mathbf{false} \Rightarrow \Box\Diamond x$$

in the required $past \Rightarrow future$ form we may choose either to rename the subformula '$\Diamond x$' or directly remove the $\Box$–operator. With the former we first rewrite by renaming $\Diamond x$ where all t_i for $i = 1, \ldots$ are new proposition symbols.

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & \Box t_1 \\ t_1 & \Rightarrow & \Diamond x \end{array}$$

Next we 'unwind' the $\Diamond x$ once and remove the $\Box$–formula.

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & t_1 \wedge t_2 \\ \circledcirc\, t_2 & \Rightarrow & t_1 \wedge t_2 \\ t_1 & \Rightarrow & x \vee \bigcirc\Diamond x \end{array}$$

Then, finally we rewrite the last rule and split up the conjunctions on the right hand sides of the first two rules.

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & t_1 \\ \bullet\,\mathbf{false} & \Rightarrow & t_2 \\ \circledcirc\, t_2 & \Rightarrow & t_1 \\ \circledcirc\, t_2 & \Rightarrow & t_2 \\ \circledcirc\, (t_1 \wedge \neg x) & \Rightarrow & \Diamond x \end{array}$$

With the second alternative we directly apply the transformation to remove the $\square$–operator.

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & \Diamond x \wedge t_1 \\ \circledcirc\, t_1 & \Rightarrow & \Diamond x \wedge t_1 \end{array}$$

Then we split up conjunctions on the right hand sides of rules to obtain the final rule-set.

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & \Diamond x \\ \bullet\,\mathbf{false} & \Rightarrow & t_1 \\ \circledcirc\, t_1 & \Rightarrow & \Diamond x \\ \circledcirc\, t_1 & \Rightarrow & t_1 \end{array}$$

Thus, in this example the removal of the $\square$–operator directly gives us a smaller number of rules, smaller total number of propositions and takes fewer steps to obtain. However we do end up with two copies of the eventuality which may be even worse. This is because we may then have to duplicate complex temporal resolvents when resolving with $\Diamond x$.

7.1.1 Replacement of Duplicate Subformulae

A potential improvement on the transformation to SNF that has been previously described is to search for subformulae that are the same and replace them by the same new variable. The time spent searching for duplicate subformulae will be increased but if duplicate subformulae are found the length of the resulting SNF formula and number of new propositions required will be less than if this is not done. There is a trade-off between the pre-processing time required to detect duplicate subformulae and the time saved during temporal resolution because there are fewer rules.

7.1.2 Unwinding

The technique of unwinding, for example using the following rewrite rules

$$\begin{array}{rcl} \{P \Rightarrow \square A\} & \longrightarrow & \{P \Rightarrow A \wedge \bigcirc \square A\} \\ \{P \Rightarrow \Diamond A\} & \longrightarrow & \{P \Rightarrow A \vee \bigcirc \Diamond A\} \end{array}$$

derived from the temporal equivalences

$$\begin{array}{rcl} \square A & \equiv & A \wedge \bigcirc \square A \\ \Diamond A & \equiv & A \vee \bigcirc \Diamond A \end{array}$$

may be useful as they introduce a $\bigcirc$-operator in front of the temporal operator we are considering. If we then rename the whole of the formula with the $\bigcirc$-operator preceeding it, i.e. $\bigcirc \square A$ or $\bigcirc \Diamond A$ (rather than $\square A$ or $\Diamond A$) it is easy to produce the $\circledcirc$-operator on the left hand side required for SNF, i.e. from $t_1 \Rightarrow \bigcirc \Diamond A$ to $\circledcirc\, t_1 \Rightarrow \Diamond A$

7.2 Using Three SNF Translations

Here we describe experiments where three different translations using a combination of the above effects are considered in order to see how the formulae in SNF differ in terms of various statistics. Translation (i) does not look for duplicate subformulae and unwinds formulae once before carrying out any subformulae replacement. Translation (ii) identifies duplicate subformulae and replaces them with one variable and replaces subformulae at the deepest level of nesting (without unwinding) but may use unwinding later. Translation (iii) is the same as (i) except that duplicate subformulae are replaced.

7.2.1 Example

Here we illustrate the use of each translation on an example formula. In Section 7.2.2 statistics for all the examples given in Appendix A are given for each translation. The example formula to be translated to SNF using each of the three variations described with negations pushed through to propositions is

$$\bullet\,\mathbf{false} \Rightarrow ((\square w \wedge \Diamond\Diamond\neg w) \vee (\square\square w \wedge \Diamond\neg w)).$$

(i) Translation with unwinding and no replacement of duplicates. See translation (iii) but don't allow the replacement of duplicates.

(ii) Translation without unwinding. Replacement of duplicate subformulae at the deepest level of nesting, i.e. t_1 for $\square w$ and t_2 for $\Diamond\neg w$.

$$\begin{array}{rcl}
\bullet\,\mathbf{false} & \Rightarrow & (t_1 \wedge \Diamond t_2) \vee (\square t_1 \wedge t_2) \\
t_1 & \Rightarrow & \square w \\
t_2 & \Rightarrow & \Diamond\neg w
\end{array}$$

Removal of the remaining temporal subformulae from the original formula.

$$\begin{array}{rcl}
\bullet\,\mathbf{false} & \Rightarrow & (t_1 \wedge t_3) \vee (t_4 \wedge t_2) \\
t_1 & \Rightarrow & \square w \\
t_2 & \Rightarrow & \Diamond\neg w \\
t_3 & \Rightarrow & \Diamond t_2 \\
t_4 & \Rightarrow & \square t_1
\end{array}$$

Unwinding temporal formulae once and splitting up formulae with conjuncts on the right hand side.

$$\begin{array}{rcl}
\bullet\,\mathbf{false} & \Rightarrow & (t_1 \wedge t_3) \vee (t_4 \wedge t_2) \\
t_1 & \Rightarrow & w \\
t_1 & \Rightarrow & \bigcirc\square w \\
t_2 & \Rightarrow & \neg w \vee \bigcirc\Diamond\neg w \\
t_3 & \Rightarrow & t_2 \vee \bigcirc\Diamond t_2 \\
t_4 & \Rightarrow & t_1 \\
t_4 & \Rightarrow & \bigcirc\square t_1
\end{array}$$

Removal of $\bigcirc\square$ and $\bigcirc\Diamond$ and rewriting to *past* $\Rightarrow$ *future* format.

$$\begin{array}{rclcrcl}
\bullet\,\mathbf{false} & \Rightarrow & t_1 \vee t_4 & \quad & \circledcirc t_1 & \Rightarrow & w \\
\bullet\,\mathbf{false} & \Rightarrow & t_1 \vee t_2 & & \circledcirc(t_2 \wedge w) & \Rightarrow & \Diamond\neg w \\
\bullet\,\mathbf{false} & \Rightarrow & t_3 \vee t_4 & & \circledcirc(t_3 \wedge \neg t_2) & \Rightarrow & \Diamond t_2 \\
\bullet\,\mathbf{false} & \Rightarrow & t_3 \vee t_2 & & \bullet\,\mathbf{false} & \Rightarrow & \neg t_4 \vee t_1 \\
\bullet\,\mathbf{false} & \Rightarrow & \neg t_1 \vee w & & \circledcirc\,\mathbf{true} & \Rightarrow & \neg t_4 \vee t_1 \\
\circledcirc\,\mathbf{true} & \Rightarrow & \neg t_1 \vee w & & \circledcirc t_4 & \Rightarrow & t_4 \\
\circledcirc t_1 & \Rightarrow & t_1 & & \circledcirc t_4 & \Rightarrow & t_1
\end{array}$$

(iii) Translation with unwinding. Unwinding temporal formulae once at deepest nesting and then replacing t_1 for $\bigcirc\square w$ and t_2 for $\bigcirc\Diamond\neg w$.

$$\begin{array}{rcl}
\bullet\,\mathbf{false} \Rightarrow ((w \wedge t_1) \wedge \Diamond(\neg w \vee t_2)) & \vee & (\square(w \wedge t_1) \wedge (\neg w \vee t_2)) \\
\circledcirc t_1 & \Rightarrow & t_1 \\
\circledcirc t_1 & \Rightarrow & w \\
\circledcirc t_2 & \Rightarrow & \Diamond\neg w
\end{array}$$

Unwinding second layer of nesting.

$$
\begin{array}{rcl}
\bullet\,\mathbf{false} & \Rightarrow & ((w \wedge t_1) \wedge ((\neg w \vee t_2) \vee \bigcirc \Diamond(\neg w \vee t_2))) \\
 & \vee & ((w \wedge t_1) \wedge \bigcirc \Box (w \wedge t_1) \wedge (\neg w \vee t_2))
\end{array}
$$

$$
\begin{array}{rcl}
\odot\, t_1 & \Rightarrow & t_1 \\
\odot\, t_1 & \Rightarrow & w \\
\odot\, t_2 & \Rightarrow & \Diamond \neg w
\end{array}
$$

Removal of $\bigcirc$–subformulae.

$$
\begin{array}{rcl}
\bullet\,\mathbf{false} & \Rightarrow & ((w \wedge t_1) \wedge ((\neg w \vee t_2) \vee t_3)) \\
 & \vee & ((w \wedge t_1) \wedge t_4 \wedge (\neg w \vee t_2))
\end{array}
$$

$$
\begin{array}{rcl}
\odot\, t_1 & \Rightarrow & t_1 \\
\odot\, t_1 & \Rightarrow & w \\
\odot\, t_2 & \Rightarrow & \Diamond \neg w \\
\odot\, t_3 & \Rightarrow & \Diamond(\neg w \vee t_2) \\
\odot\, t_4 & \Rightarrow & t_4 \\
\odot\, t_4 & \Rightarrow & w \\
\odot\, t_4 & \Rightarrow & t_1
\end{array}
$$

Removal of $\Diamond$ applied to a disjunction and rewriting the rules into the correct format.

$$
\begin{array}{rclcrcl}
\bullet\,\mathbf{false} & \Rightarrow & w & \quad & \odot\, t_3 & \Rightarrow & \Diamond t_5 \\
\bullet\,\mathbf{false} & \Rightarrow & t_1 & & \odot\, t_4 & \Rightarrow & t_4 \\
\bullet\,\mathbf{false} & \Rightarrow & \neg w \vee t_2 \vee t_3 & & \odot\, t_4 & \Rightarrow & w \\
\odot\, t_1 & \Rightarrow & t_1 & & \odot\, t_4 & \Rightarrow & t_1 \\
\odot\, t_1 & \Rightarrow & w & & \bullet\,\mathbf{false} & \Rightarrow & \neg t_5 \vee \neg w \vee t_2 \\
\odot\, t_2 & \Rightarrow & \Diamond \neg w & & \odot\,\mathbf{true} & \Rightarrow & \neg t_5 \vee \neg w \vee t_2
\end{array}
$$

7.2.2 Statistics for the Different Translations

Formulae were translated using the three translations outlined and statistics compiled relating to the SNF rules obtained. The examples used are valid temporal formulae taken from [MP81] and are given in Appendix A, labelled by the numbers to which we refer.

Both the translations (ii) and (iii) identified duplicate subformulae and replaced them by a single new variable. This was possible in formulae 1, 2, 3, 12, 13, 16, 22, 40, 41 and 42. Apart from this translation (iii) was the same as translation (i) which *unwinds* formulae once before carrying out any subformula replacement. Translation (ii) replaces subformulae immediately by new variables without *unwinding* them once. In the cases of no duplication of subformulae translations (i) and (iii) are identical. The following statistics are given for comparison of the different SNF translations.

1. Rules : The number of $\Box$–rules after translation to SNF and one round of simplification and subsumption has taken place.
2. Variables : The number of new proposition variables created during the SNF translation.
3. Eventualities : The number of $\Diamond$–rules.
4. Total Props (TP): The total number of propositions in the set of rules calculated by summing the value for each rule where

$$
\begin{array}{rcl}
TP(\bullet\,\mathbf{false} \Rightarrow \Diamond l) & = & 1 \\
TP(\odot\, P \Rightarrow \Diamond l) & = & |\mathrm{lits}(P)| + 1 \\
TP(\bullet\,\mathbf{false} \Rightarrow Q) & = & |\mathrm{lits}(Q)| \\
TP(\odot\,\mathbf{true} \Rightarrow Q) & = & |\mathrm{lits}(Q)| + 1 \\
TP(\odot\, P \Rightarrow Q) & = & |\mathrm{lits}(P)| + |\mathrm{lits}(Q)|
\end{array}
$$

for l a literal, P a conjunction of literals and Q a disjunction of literals.

Statistic	Tran (i)	Tran (ii)	Tran (iii)
Rules	19	14	12
Variables	7	4	5
Eventualities	3	2	2
Total Props	47	32	26

Table 7.1: Statistics for Translating Example 12 into SNF

The figures for this example (example 12) are given in Table 7.1. A table with these statistics for each of the three translations is given in Appendix B.4.

7.2.3 Observations

Translation of $\square$-subformulae and $\diamondsuit$-subformulae. When translating $\square$ or $\diamondsuit$-formulae the same number of new proposition sysmbols are created using each translation. However the number of rules and total number of propositions depends on the context the formulae appear in. With the other temporal formulae, more new proposition symbols are created with the second translation, as well as many more rules. In the following illustration of this observation $\mathcal{F}$ is a future-time context and P is assumed to be a CNF or DNF formula

Example: SNF Translation of $\square P$: Unwinding First (Translations (i) and (iii))

$$\bullet\,\mathbf{false} \Rightarrow \mathcal{F}(\square P)$$

Unwinding once.

$$\bullet\,\mathbf{false} \Rightarrow \mathcal{F}(P \wedge \bigcirc \square P)$$

Replacing $\bigcirc \square P$ by a new variable

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & \mathcal{F}(P \wedge t_1) \\ \circledcirc\, t_1 & \Rightarrow & t_1 \\ \circledcirc\, t_1 & \Rightarrow & P \end{array}$$

SNF Translation of $\square P$: Renaming first (Translation ii)

$$\bullet\,\mathbf{false} \Rightarrow \mathcal{F}(\square P)$$

Replacing $\square P$ by a new variable.

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & \mathcal{F}(t_1) \\ t_1 & \Rightarrow & \square P \end{array}$$

Unwinding once.

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & \mathcal{F}(t_1) \\ t_1 & \Rightarrow & P \wedge \bigcirc \square P \end{array}$$

Rewriting.

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & \mathcal{F}(t_1) \\ \circledcirc\, t_1 & \Rightarrow & t_1 \\ \circledcirc\, t_1 & \Rightarrow & P \\ \circledcirc\,\mathbf{true} & \Rightarrow & \neg t_1 \vee P \\ \bullet\,\mathbf{false} & \Rightarrow & \neg t_1 \vee P \end{array}$$

All three translations are the same for examples 19, 20 and 21. This is because '$\bigcirc$' is the only temporal operator and the translations we have described all translate $\bigcirc$-operators in the same way. Translations (ii) and (iii) are the same for examples 3 and 22 as $\bigcirc$ is the only temporal operator but we can do some replacement of duplicate subformulae. Translations (i) and (iii) are the same where there are no duplicate subformulae (all examples except 1, 2, 3, 12, 13, 16, 22, 40, 41 and 42).

7.2.4 Conclusions

The best results for the translation to SNF Overall the translation (iii) seems to produce the best results from our data. In all examples except 11, 14, and 15 all columns for translation (iii) give at least as good results as the other two translations. The reason that these do better with translation (ii) is that they contain subformulae of the form $\Diamond\Box w$ or $\Diamond\Diamond w$. Examining the former, when the $\Box$-operator is unwound once and the next operator replaced by a new proposition t_1 we obtain the subformulae $\Diamond(w \wedge t_1)$ and $⦿ t_1 \Rightarrow \Box w$. The latter is easily translated but when the former is unwound the conjunct in the brackets is duplicated $((w \wedge t_1) \vee \bigcirc\Diamond(w \wedge t_1))$ and when the $\bigcirc$-formula is renamed we obtain $(w \wedge t_1) \vee t_2$ and $⦿ t_2 \Rightarrow \Diamond(w \wedge t_1)$. The number of rules the former will create depends on its context. The latter will need the formula inside the $\Diamond$-operator renaming and will create four additional rules to link the new proposition symbol and this formula.

How the second translation may help us The second SNF translation may help us by retaining the structure of the original clauses, the main goal of [PG86]. If we were given an unsatisfiable formula

$$\bullet\,\mathbf{false} \Rightarrow \Box\Phi \wedge \Diamond\neg\Phi$$

where Φ was some (possibly complicated) arbitrary formula, the replacement of Φ by a new variable t_1 would give the following clauses

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & \Box t_1 \wedge \Diamond\neg t_1 \\ t_1 & \Rightarrow & \Phi \end{array}$$

translating the first clause into SNF by introducing a new propositional variable, t_2, to translate $\Box t_1$, we get

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & t_2 \\ ⦿ t_2 & \Rightarrow & t_2 \\ ⦿ t_2 & \Rightarrow & t_1 \\ \bullet\,\mathbf{false} & \Rightarrow & \neg t_2 \vee t_1 \\ ⦿\,\mathbf{true} & \Rightarrow & \neg t_2 \vee t_1 \\ \bullet\,\mathbf{false} & \Rightarrow & \Diamond\neg t_1 \end{array}$$

To resolve with $\bullet\,\mathbf{false} \Rightarrow \Diamond\neg t_1$ we may use the loop in t_1 made from combining the second and third rules into SNF_m to give

$$⦿ t_2 \Rightarrow t_1 \wedge t_2.$$

This can be carried out without reference to the clause with Φ in it and by adding the resolvents from temporal resolution we could obtain a contradiction during the next round of step resolution.

Chapter 8

Related Work

In this chapter we look at other systems for theorem proving in temporal logics using resolution, tableau and automata based approaches. We describe the systems and explain the similarities to and differences from our approach.

8.1 Other Resolution Systems

The resolution system we have described in this thesis is clausal, that is we translate formulae to a (clausal) normal form before applying any resolution rules. Both clausal and non-clausal resolution based systems have been developed for temporal logics, some of which are described, and compared with Fisher's system in the following sections.

8.1.1 Clausal

Peim

Fisher's resolution system is generalised to prove properties of finite labelled transition systems in [Pei94] (see also Section 4.1). Program control is encoded as an automaton and data encoded by SNF (global $\square$) rules attached to transitions. SNF rules that apply to *all* transitions, known as *global transition rules*, are separated out for efficiency. As well as having sets of SNF rules associated with them, transitions are conditional with guards as non-temporal formulae. The initial state of the program is represented by an initial state in the automaton and a related set of (non-temporal) formulae that must hold (equivalent to the initial $\square$–rules in Fisher's system). There are also sets of initial and global eventualities (corresponding to our initial and global $\Diamond$–rules). Such a system is known as a Clausal Transition System or CTS.

Given a CTS, resolution is split into step and loop resolution. Step resolution takes place between clauses in the initial state or between global transition rules, or between a rule associated to a transition t and either a global transition rule, or another rule associated with t. In Fisher's system, rules of the form $\bigcirc P \Rightarrow \mathbf{false}$ are rewritten as constraints on states, i.e. as $\bigcirc\mathbf{true} \Rightarrow \neg P$ and $\bullet\mathbf{false} \Rightarrow \neg P$. Similarly, in Peim's system the detection of such a rule in the global transition rules entails the addition of $\bigcirc\mathbf{true} \Rightarrow \neg P$ to the global transition rules and $\neg P$ to the initial clauses. However by obtaining $\bigcirc P \Rightarrow \mathbf{false}$ as a rule in the set associated to a transition t we must conjoin $\neg P$ to the guard related to t as t must not be taken if P holds. Further, if t is a transition out of state s, to ensure that at least one transition may be taken out of s, ϕ, the disjunction of the guards of all the transitions out of s are recomputed and $\bigcirc\mathbf{true} \Rightarrow \phi$ added to all transitions into s. If s is initial we add ϕ to the set of initial clauses.

Loop resolution is defined in a similar way to the definition of the temporal resolution rule we give except rules are related to transitions. Thus, a loop in l is a set of SNF$_m$ rules

$$\begin{array}{rcl} \circledcirc A_{t,0} & \Rightarrow & B_{t,0} \\ \dots & & \dots \\ \circledcirc A_{t,n_t} & \Rightarrow & B_{t,n_t} \end{array}$$

for each transition t formed by merging global transition rules or SNF rules associated with t such that

1. for each t and i $\vdash B_{t,i} \Rightarrow l$; and
2. for each t and i if s is the destination state of transition t, t' is a transition out of s and $C_{t'}$ the enabling condition or guard for t' then

$$\vdash B_{t,i} \Rightarrow \bigwedge_{t' \text{ out of } s} (C_{t'} \Rightarrow \bigvee_{j=0}^{n_{t'}} A_{t',j}).$$

The first side condition ensures that the required literal holds (as with Fisher's system) and the second that no rule leads out of the loop, which now must take account of the transitions and their enabling conditions.

As the loop definition for a CTS is more complicated than for Fisher's system so are the resolvents. Peim presents the resolvents for a CTS in a form similar to the rule form where as Fisher uses the $\mathcal{W}$-operator that needs further translation to the normal form. Recall (Section 4.1) rather than generating new variables each time a loop is detected for an eventuality Peim suggests associating a new proposition to each eventuality before any resolution takes place. For example if we have $\Diamond l$ as the right hand side of a $\Diamond$–rule then w_l would be the new proposition, meaning informally *waiting for l*. Also he suggests adding in the resolvents $\circledcirc w_l \Rightarrow l \vee w_l$ and either $l \vee w_l$ if $\Diamond l$ is the right hand side of an initial rule or $\circledcirc Q \Rightarrow l \vee w_l$ if the $\Diamond$–rule is global, i.e. $\circledcirc Q \Rightarrow \Diamond l$, known as *augmenting* the rule-set, before any resolution takes place. For the other resolvents, a formula F_s is defined for each state s

$$F_s = \bigwedge_{t \text{ out of } s} (C_t \Rightarrow \bigvee_{i=0}^{n_t} A_{t,i})$$

so whenever F_s holds in a state the negation of the eventuality holds for the next step onwards. Thus, if the $\Diamond$–rule is initial we add the resolvent $l \vee \neg F_{s_0}$ to the initial clauses and for each state s add $\circledcirc w_l \Rightarrow l \vee \neg F_s$ to each transition into s. If the $\Diamond$–rule is of the form $\circledcirc Q \Rightarrow \Diamond l$, for each s we add the resolvents $\circledcirc Q \Rightarrow l \vee \neg F_s$ and $\circledcirc w_l \Rightarrow w_l \vee \neg F_s$ to each transition into s. In both cases informally this means that once the $\Diamond$–rule has been 'fired' the eventuality (l) must be satisfied before we enter the loop. Further if the rule is a global $\Diamond$–rule we also add $\circledcirc \mathbf{true} \Rightarrow \neg Q \vee \neg F_s$ to all transitions into s and conjoin $\neg Q \vee \neg F_s$ to the enabling condition for each transition out of s.

The resolvents are similar to those in Fisher's system. The last two mentioned correspond (assuming we are resolving $\circledcirc Q \Rightarrow \Diamond l$ with a set of rules $\circledcirc A_i \Rightarrow B_i$ that satify the side conditions) with Fisher's resolvent

$$\bullet\, \mathbf{true} \Rightarrow \neg Q \vee \bigwedge_i \neg A_i.$$

The pair of rules augmenting the rule-set and the other pair dependent on whether the $\Diamond$–rule is initial or global correspond to

$$\circledcirc Q \Rightarrow \bigwedge_i \neg A_i \,\mathcal{W}\, l.$$

Peim proves the correctness of his resolution procedure. This has been outlined in Section 4.1. He notes that Fisher's system is merely a special case of a CTS with one state and a transition from this state to itself with the guard or enabling condition **true**. Peim is investigating ways of adapting the Breadth-First Search Algorithm, described in this thesis, to detect loops for CTSs and thus enable mechanisation. Even though the system is clausal, the program structure is retained as an automaton with SNF rules attached to the transitions. Thus Peim's system is suitable for program verification and does look promising for tackling more complex systems.

Venkatesh

Venkatesh [Ven86] describes a clausal resolution method for PTL for future-time operators including $\mathcal{U}$. First formulae are translated into a normal form containing a restricted nesting of temporal operators. The normal form is

$$\bigwedge_{i=1}^{n} c_i \wedge \bigwedge_{j=1}^{m} \Box c_j^{'}$$

where each c_i and $c_j^{'}$ (known as *clauses*) is a disjunction of formulae of the form $\bigcirc^k l$, $\bigcirc^k \Box l$, $\bigcirc^k \Diamond l$ or $\bigcirc^k (l' \mathcal{U} l)$ (known as *principal terms*) for l and l' literals, $k \geqslant 0$ and $\bigcirc^k$ denoting a series of k $\bigcirc$–operators. To translate into the normal form temporal equivalences are used to ensure that negations are applied only to propositions, the $\bigcirc$–operator is distributed over conjunctions and disjunctions, and rules are applied to ensure a CNF-like structure. Renaming is carried out, similar to our approach in translation to SNF, to remove the nesting of temporal operators not allowed in the normal form. For example to translate $\bigcirc^i(\Box F)$ into the normal form, where F is a temporal formula, then we can replace it by $\bigcirc^i(\Box t) \wedge \Box(t \Leftrightarrow F)$ where t is a new proposition symbol. The translation to the normal form is shown to preserve the satisfiability of the temporal formula.

The clauses in the normal form therefore either apply to the first moment in time or to every moment in time (those enclosed in a $\Box$–operator). Thus resolution proofs are displayed in columns separating the clauses that hold in each state. To determine unsatisfiability, the principal terms (except $\bigcirc^k l$) in each clause are *unwound* to split them into present and future parts. For example the clause $F \vee \Diamond l$ is replaced by $F \vee l \vee \bigcirc \Diamond l$ and similarly for $\Box$ and $\mathcal{U}$. Next, classical style resolution is carried out between complementary literals relating to the present parts of the clauses in each column or state. Then, any clauses in a state that contain only principal terms with one or more next operators are transferred to the next state and the number of next operators attached to each term is reduced by one. This is known as the SKIP operation.

This process is shown to be complete for clauses that contain no eventualities. Formulae that contain eventualities that are delayed indefinitely due to unwinding are eliminated and this process is shown to be complete.

As already mentioned, Venkatesh makes use of temporal equivalences and renaming, requiring the introduction of new variables, to translate formulae into the required normal form. These are both techniques used in Fisher's system. The normal form is relatively simple and has removed all nesting of formulae but allows $\Box$, $\Diamond$ and $\mathcal{U}$ operators applied to literals as disjuncts in the clauses as well as an external $\Box$–operator around the clauses. Fisher removes most temporal operators except $\bullet$, $\circledcirc$ and $\Diamond$ as well as flattening formulae.

Ventatesh requires that the unwinding process takes place before any resolution is carried out. Fisher uses a process similar to unwinding that introduces new proposition symbols during the translation to SNF to remove operators that are not required as part of the normal form. The resolution between complementary literals in the first state, performed by Venkatesh, is similar to the resolution between initial $\Box$–formulae described here. Likewise, resolution between clauses enclosed in the $\Box$–operator that apply to each state is similar to step resolution between global $\Box$–rules.

The main difference is the treatment of persistent eventualities. Fisher's system looks for the set of rules with which we can resolve an eventuality where as Venkatesh eliminates those that persist. Further, Fisher's system reasons *backwards* so contradictions within states generate constraints on previous states until a contradiction in the initial state is obtained. However, Venkatesh starts by processing in the initial states and using the SKIP operation carries information to the following state. If **false** is generated by resolution of clauses within a state then the original formula was unsatisfiable.

The use of the SKIP operator to carry information into the next state and the unwinding of principal terms into their present and future parts is similar to the execution of METATEM [BFG+89a] programs (see Section 8.4). For example in Venkatesh's system a clause $\bigcirc a \vee \bigcirc b$ in state i for $i \geqslant 0$ will be transferred into state $i+1$ as $a \vee b$. Similarly in a (propositional) METATEM program at state $i+1$ the rule $\circledcirc\mathbf{true} \Rightarrow a \vee b$ imposes the constraint $a \vee b$ on the present state (as **true** was satisfied in state i).

Cavalli and Fariñas del Cerro

A clausal resolution method for PTL is outlined in [CdC84]. The temporal operators defined in the logic include $\bigcirc$, $\Box$, and $\Diamond$ but do not include $\mathcal{U}$ or any past-time temporal operators. The method described rewrites formulae to a complicated normal form and then applies a series of temporal resolution rules.

A formula, F, is said to be in Conjunctive Normal Form (CNF), if it is of the form

$$F = C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

where each C_j is called a *clause* and is of the following form.

$$\begin{array}{rcl} C_j & = & L_1 \vee L_2 \vee \ldots \vee L_n \vee \Box D_1 \vee \Box D_2 \vee \ldots \vee \Box D_p \\ & \vee & \Diamond A_1 \vee \Diamond A_2 \vee \ldots \vee \Diamond A_q \end{array}$$

Here each L_i is a literal preceded by a string of zero or more $\bigcirc$–operators, each D_i is a disjunction of the same general form as the clauses and each A_i is a conjunction where each conjunct possesses the same general form as the clauses. The rules required to rewrite formulae into the normal form depend on temporal theorems and classical methods. Renaming and the introduction of new propositions is not required.

The resolution rules are split into three types

1. classical operators
2. temporal operators
3. transformation operators

denoted by Σ_1, Σ_2, and Σ_3 respectively. Resolution rules are of the form that $\Box x$ and $\Diamond y$ can be resolved if x and y are resolvable and the resolvent will be the resolvent of x and y with a $\Diamond$–operator in front.
Classical operations allow classical style resolution to be performed, for example

$$\begin{array}{ll} \Sigma_1(p, \neg p) = \emptyset & \text{(where } \emptyset \text{ denotes the empty set or } \textbf{false}\text{)} \\ & \text{And } (p, \neg p) \text{ is resolvable.} \end{array}$$

The temporal resolution rules allow resolution between formulae in the context of certain operators, for example

$$\begin{array}{c} \Sigma_2(\Box E, \triangle F) = \triangle \Sigma_i(E, F) \text{ (provided that } \triangle \text{ is one of } \Box, \Diamond, \text{ or } \bigcirc) \\ \text{And if } (E, F) \text{ is resolvable then } (\Box E, \triangle F) \text{ is resolvable;} \end{array}$$

where Σ_i denotes that an operation of type i is being applied where $i = 1, 2$ or 3.
The transformation rules allow rewriting of some formulae, to enable the continued application of the resolution rules, for example

$$\begin{array}{c} \Sigma_3(\Box E, F) = \Sigma_i(\Box \Box E, F) \\ \text{And if } (\Box \Box E, F) \text{ is resolvable then } (\Box E, F) \text{ is resolvable.} \end{array}$$

There are three rules of inference given where $R(C_1, C_2)$ (or $R(C_1)$) is a resolvent of C_1 and C_2 $((C_1))$. If $C_1 \vee C$ and $C_2 \vee C'$ are clauses then the resolution inference rules are

$$\frac{C_1 \vee C \quad C_2 \vee C'}{R(C_1, C_2) \vee C \vee C'}$$

if C_1 and C_2 are resolvable and

$$\frac{C_1 \vee C}{R(C_1) \vee C}$$

if C_1 is resolvable. The following inference rule can also be applied (for $E(D \vee D \vee F)$ a clause) to carry out simplification.

$$\frac{E(D \vee D \vee F)}{E(D \vee F)}$$

Formulae are refuted by translation to clausal form and repeated application of the inference rules. Resolution only takes place between clauses in the context of certain operators outlined in the resolution rules.

The method is only similar to the Fisher method as it uses translation to a clause form, although the normal form is much more complicated, and attempts to resolve $\Box$ and $\Diamond$–rules by using the above rules. The number of temporal operators allowed in the formulae for refutation is much more limited though as in this resolution method past-time operators are not allowed, and the resolution rules are much more complicated. Also the completeness proof is only given for the $\Box$, $\Diamond$, and $\bigcirc$ operators.

8.1.2 Non-Clausal

Non-clausal temporal resolution systems are developed for propositional and then first-order temporal logics that are discrete and linear and have finite past and infinite future [Aba87]. The systems are developed first for fragments of the logic including the temporal operators $\bigcirc$, $\Box$, and $\Diamond$ and then extended for $\bigcirc$, $\Box$, $\Diamond$, $\mathcal{W}$ [1] and $\mathcal{P}$. The binary operator $\mathcal{P}$ is known as *precedes* where $u \mathcal{P} v = \neg((\neg u) \mathcal{W} v)$.

The propositional system for $\bigcirc$, $\Box$, and $\Diamond$ has rules for simplification, distribution, cut, resolution, modality and induction. Simplification rules include rules that apply negations to formulae, rules that simplify formulae containing **false**, and the weakening rule that allows the deletion of a conjunct that is considered useless. The distribution rule allows the distribution of $\wedge$ over $\vee$. The cut rule allows the introduction of rules of the form $u \vee \neg u$ and is not necessary for the completeness of the propositional system (but is necessary for the first-order system). The resolution rule is of the form

$$A < u, \ldots, u >, B < u, \ldots, u > \longrightarrow A < \mathbf{true} > \vee B < \mathbf{false} >$$

where $A < u, \ldots, u >$ denotes that u occurs one or more times in A. Here occurrences of u in A and B are replaced with **true** and **false** respectively. To ensure the rule is sound each u that is replaced must be in the scope of the same number of $\bigcirc$-operators, and must not be in the scope of any other modal operator in A or B, i.e. they must apply to the same moment in time. The modality rules apply to formulae in the scope of the temporal operators. For example the $\Box$-rule allows any formula $\Box u$ to be rewritten as $u \wedge \bigcirc \Box u$. The induction rule deals with the interaction between $\bigcirc$ and $\Box$ and is of the form

$$w, \Diamond u \longrightarrow \Diamond(\neg u \wedge \bigcirc(u \wedge \neg w)) \text{ if } \vdash \neg(w \wedge u).$$

Informally this means that if w and u cannot both hold at the same time and if w and $\Diamond u$ hold now then there must be a moment in time (now or) in the future when u does not hold and at the next moment in time u holds and w does not. A proof editor has been developed for the propositional system with the $\bigcirc$, $\Box$, and $\Diamond$ operators. The resolution system is then extended to allow for the operators $\mathcal{W}$ and $\mathcal{P}$ also. A decision procedure based on tableau is given and completeness for the resolution systems are shown by proving that if a formula $\neg u$ is found unsatisfiable by the tableau decision procedure then there is a refutation for $\neg u$.

Although a proof editor has been developed for the restricted propositional system it seems unlikely that Abadi's system lends itself to a fully automatic implementation. This is because of the large number of rules that may be applied. The modality rules for example deal with subformulae in the scope of modal operators, required as the system is non-clausal. Fisher's system does use a version of the modality rule for $\Box u$ and $\Diamond u$, for example for $\Box u$ we have

$$\Box u \longrightarrow u \wedge \bigcirc \Box u$$

[1] Abadi denotes $\mathcal{W}$, *unless* (or *weak until*), as $\mathcal{U}$.

in the translation to the normal form but the modality rules are not required elsewhere. Further, the induction rule requires a proof as a side condition to its usage which will make automatic proofs difficult.

The system for $\bigcirc$, $\square$, $\diamondsuit$, $\mathcal{W}$ and $\mathcal{P}$ is extended for first-order temporal logic (FTL). The notion of *rigid* and *flexible* symbols is introduced for time-independent and time-dependent symbols. Rules for skolemization are given based on skolemization in classical logics. Restrictions relating to the use of universal and existential operators in the scope of certain temporal operators to ensure the soundness of skolemization rules are enforced. The resolution rule is based on that given for PTL allowing for unification and again restrictions are imposed relating to quantification and ensuring that resolution is performed on formulae that occur in the same moment in time. Notions of completeness are discussed. It is shown that while all effective proof systems for FTL are incomplete, a slight extension to the resolution system is as powerful as Peano arithmetic.

The resolution system is used in program verification and it is suggested how a limited version of the system could be used to interpret programs in a temporal logic programming language. Finally non-clausal resolution systems based on those that have been developed for temporal logics are given for modal logics.

8.2 Systems Based on Temporal Tableaux

Here we describe some of the decision procedures for temporal logics that have been based on the semantic tableau method [Smu68, Fit83]. Using semantic tableau methods a formula f is shown to be valid if it is not possible to construct a model for $\neg f$.

DP and DPP

DP and DPP [Gou84] are decision procedures for propositional linear temporal logic. DP is for future-time temporal logic over infinite models of time allowing the five future-time temporal operators described in Chapter 2. DPP allows past-time operators also and uses models of time that extend infinitely into the future and past.

To test the unsatisfiability of a formula f, DP uses an algorithm that tries to construct a model for f. The formula is decomposed into a set of states containing formulae relating to the current moment in time, or the next moment in time (i.e. formulae which have $\bigcirc$ as their main operator). A graph is built where each node represents a state and is labelled by formulae that hold in that state and edges between states lead to possible *next* states. Nodes are deleted if they contain an eventuality that is not satisfied in any node reachable from that node or if they have no successors. This structure is called an H-graph. A model for f can be constructed from the H-graph if an infinite path from a node labelled by f or equivalent formulae can be found. As nodes are labelled by formulae other than propositions this path is known as a Hintikka structure and it is shown that a model can be constructed for f if and only if it has a Hintikka Structure.

The H-graph constructed is related to the the behaviour graph described in Chapter 3 except nodes contain formulae other than propositions. Further, in Chapter 3 we have chosen to build the behaviour graph for the set of global $\square$–rules rather than the complete set of SNF rules, although we indicate how Peim builds such a graph for the full rule-set in Section 4.1. The construction of the H-graph uses all the information in the formula f. In the H-graph nodes are deleted that contain unsatisfiable eventualities. In the behaviour graph we look for terminal subgraphs where the negation of an eventuality holds at each node. Adding new rules from temporal resolution means that edges into this subgraph, where the eventuality is outstanding, are deleted. The Depth-First and Breadth-First Searches described attempt to identify these subgraphs without the need to construct the rest of the graph.

Wolper

Wolper presents a tableau based decision procedure for PTL in [Wol85] and extends it to ETL (an extension of PTL with operators corresponding to arbitrary right-linear grammars) in [Wol83]. In [Wol85] a formula f to be tested for satisfiability is broken down into sets of formulae by marking and repeatedly applying decomposition rules (similar to rules used in [Gou84]) such as

$$\Diamond P \longrightarrow \{\{P\}, \{\bigcirc\Diamond P\}\}$$

until each set contains only marked formulae, atomic propositions and their negations and formulae whose main connective is the $\bigcirc$–operator. Each set is known as a *state*. A graph is constructed by building nodes labelled by these sets. The child of a state s (known as a *pre-state*) is created and labelled by the $\bigcirc$–formulae in s with the outermost $\bigcirc$–operator removed. The decomposition rules are applied again and this process is repeated until no new states or pre-states are created. Nodes are deleted if they contain a proposition and its negation, if all successors of a node have been eliminated or they contain an unsatisfiable eventuality. After all possible deletions have taken place, if the initial node has been deleted f is unsatisfiable otherwise it is satisfiable. In [Wol83] this is extended to deal with grammar operators to allow the expression of properties such as even(p). Many of the similarities that apply between this system and Fisher's resolution method are the same as those mentioned for DP already.

8.3 Systems Based on Automata

The semantics of propositional linear temporal logic is just an infinite sequence of choices of truth values for propositions ($\mathcal{P}$). An interpretation of a PTL formula then can be viewed as an infinite word over the alphabet $2^{\mathcal{P}}$. Thus finite automata over infinite words (Büchi Automata [Büc62]) can be used to represent temporal logic formulae. Translations from PTL (and ETL [Wol83]) into Büchi Automata have been developed [Wol87, GW89, SVW87, WVS83]. To determine the validity of a PTL formula, f, we negate the formula, construct a Büchi Automaton for $\neg f$ and see whether it is empty. If the Büchi Automaton is empty then it accepts no infinite words so $\neg f$ must be unsatisfiable and f valid.

In the next sections we describe the translation from PTL to Büchi Automata in more detail. Also we compare the loop search described in this thesis with the deletion of states in the Büchi Automaton.

8.3.1 Relationship to Büchi Automata

A Büchi Automaton, [Büc62],

$$A = (\Sigma, S, \rho, S_0, F)$$

is a non-deterministic finite automaton on infinite words where

- Σ is an alphabet;
- S is a set of states;
- ρ is a non-deterministic transition function $\rho : s \times \Sigma \rightarrow 2^s$;
- S_0 is a set of initial states $S_0 \subseteq S$; and
- F is a set of accepting states $F \subseteq S$.

Translation from PTL to Büchi Automata

There are procedures for PTL (or ETL formulae) φ, that produce a finite-state Büchi automaton A_φ such that A_φ accepts precisely the computations that satisfy φ [SVW87, WVS83]. Thus to determine whether φ is satisfiable one can check whether A_φ accepts some word, i.e. A_φ is non-empty. The following theorem concerning non-emptiness is taken from [GW89].

Theorem 10 *A Büchi Automaton $A = (\Sigma, S, \rho, S_0, F)$ is non-empty if and only if there is a state $s \in F$ accessible from a state $s_0 \in S_0$ and also accessible from s itself.*

Thus to test the validity of a formula φ, it is negated, a Büchi Automaton, $A_{\neg\varphi}$, constructed for $\neg\varphi$ and if $A_{\neg\varphi}$ is empty then $\neg\varphi$ is unsatisfiable and φ valid.

Following Wolper's approach in [Wol87, GW89] to build a Büchi Automaton for a PTL formula φ we first build the closure of φ ($cl(\varphi)$) which is the set of all subformulae and their negations. From this set we construct two automaton, a *local automaton* and an *eventuality automaton* that are then composed to make A_φ. The local automaton checks that sequences satisfy φ without considering the eventualities. States are maximal propositionally consistent subsets of the closure of φ. Initial states are those that contain φ and all states are accepting states. The transition function ρ_L must ensure that the symbol being read is compatible with the state of the automaton and that the next state is compatible with the semantics of the temporal operators. The former is ensured by labelling transitions with the state they lead from, i.e. $\rho_L(s,a)$ is only non-empty when $s = a$. The latter, i.e. that state t is in $\rho_L(s,a)$, is checked by ensuring rules of the form

- for every $\bigcirc\varphi_1 \in cl(\varphi)$, we have $\bigcirc\varphi_1 \in s$ if and only if $\varphi_1 \in t$

are satisfied where $s, t \in S_L$ and S_L is the set of states for the local automaton. The eventuality automaton checks that all the eventualities are realised where the set of states is the set of subsets of eventualities in φ. If a state is labelled by $\emptyset$ then there are no eventualities outstanding otherwise the set contains the eventualities still to be satisfied. The only initial state and accepting state is that labelled by $\emptyset$. Transitions keep track of unresolved eventualities for example $t \in \rho_E(s,a)$ if and only if $s \neq \emptyset$ and

- for every $\diamondsuit\varphi_1 \in s$, we have $\diamondsuit\varphi_1 \in t$ if and only if $\varphi_1 \notin a$

where $s, t \in S_E$ for S_E the set of states for the eventuality automaton, ρ_E is the transition function and a is the label of a transition from s.

The full Büchi Automaton, A_φ is the composition of the local and eventuality automata. Here start states are those that both contain φ and have no outstanding eventualities. The accepting states are those which contain no outstanding eventualities. Any states that have no successors or cannot be reached from a start state can be deleted as they cannot be part of an accepting run.

Loop Search in the Behaviour Graph and in Büchi Automata

In the behaviour graph we look for loops in l by identifying a terminal subgraph where l holds at each node to resolve with an eventuality $\diamondsuit\neg l$. By adding the resolvents from temporal resolution any edges into the terminal subgraph are deleted, i.e. we try to avoid the *bad* loops. In the Büchi automaton we look for a state in the set of accepting states where we can find a path back to itself, i.e. here we are looking for *good* loops.

8.4 MetateM

SNF grew from work on the transformation of MetateM programs [FN92, Fis92b]. MetateM is a framework for executable temporal logics [BFG+89a]. To show the validity of φ, $\neg\varphi$ is converted into the MetateM rule form

$$\square \bigwedge_{i=1}^{n} P_i \Rightarrow F_i$$

where P_i is a past-time formula and F_i is a (non strict) future-time formula. Execution of the set of rules

$$\{P_i \Rightarrow F_i \mid i \leqslant 1 \leqslant n\}$$

is carried out by examining in which rules the past-time component are satisfied (declaratively). Then the future-time formulae associated with these rules must be satisfied (imperatively). This is known as the Imperative Future Paradigm [Gab87].

The rules are interpreted at every moment in time and a set of constraints on the present and future execution of the program is generated at every state. To ensure that eventualities are not postponed indefinitely a loop checking mechanism is employed. If the loop checker detects the indefinite postponement of an eventuality then backtracking to a previous choice point takes place. Thus if execution fails the original formula (φ) is valid. Note execution will fail if it is going to within a bounded time [BFG$^+$89a].

Chapter 9

Conclusions and Future Work

Here we summarise our work and draw conclusions. We also outline future areas of work including the possibilities of applying Fisher's temporal resolution method to other forms of temporal logic.

9.1 Conclusions

We have described the approach to clausal resolution in PTL as suggested by Fisher and given different algorithms with which to implement the temporal resolution rule. Given an eventuality, $\Diamond l$, the temporal resolution rule requires that we find a set of rules where once the left hand side of one of the set is satisfied then $\neg l$ always holds. One algorithm we describe, the MTSA, achieves this by representing rules in the normal form, that apply to every moment in time, as a graph and detecting the largest terminal subgraph where $\neg l$ holds at each node. The other two algorithms avoid having to build the graph of *all* such rules and work with the rules directly. The Breadth-First Search Algorithm constructs a series of DNF formula such that if the DNF formula is satisfied then $\neg l$ holds at the next moment in time, terminating when the last formula in the series is equivalent to its predecessor. Completeness is proved for both the MTSA and Breadth-First Search Algorithm and shows that *all* possible loops are detected. Depth-First Search, the other algorithm described, uses rules one at a time until a set of nodes is constructed such that any edge from one of this set leads back to another node in the set. Completeness is shown and illustrates that the Depth-First Search Algorithm may have to be run several times to detect *all* possible loops. Soundness for each of the algorithms is also shown.

A prototype temporal resolution theorem prover using the Breadth-First and Depth-First Search Algorithms has been implemented and tested on examples. Strategies have been suggested for choosing between loop search algorithms, improving the efficiency of loop search algorithms, for translating temporal formulae to a normal form, applying the step resolution rule and reducing the number of rules during loop search.

Our approach is resolution rather than tableau based so we are not limited to finite state problems and can use strategies to guide the application of the resolution rules. Although other temporal resolution systems have been outlined in Chapter 8 we feel our system has the following advantages. It is applicable to 'full' PTL with the usual range of temporal operators, i.e. not just to $\bigcirc$, $\Diamond$ and $\Box$ as [CdC84] is whilst remaining complete. Also it is suitable for automatic implementation as there is just one temporal resolution rule and does not suffer from problems with proof direction due to the number of resolution rules as [Aba87].

Although subsumption and simplification rules are used to remove redundant information, i.e. formulae that provide no 'new' information, a weakness with the approach is, as with classical resolution theorem provers, that much irrelevant information is generated, i.e. information that does not play any part in the resolution proof. However, we have suggested strategies to alleviate this problem. The pruning strategy discards information irrelevant for loop search to enable the loop finding algorithm to search for loops in a smaller set of rules. Further, we have suggested how classical strategies could be adopted for application with the step resolution rule and how to translate into the normal form to minimize the number of rules and new propositions required.

Within the loop search algorithms the expensive part is the requirement to combine rules together. Careful implementation of this crucial part of the algorithm could improve performance. Further, although one temporal resolution rule is desirable for ease of implementation, it does seem inefficient to spend time rewriting any $\square$-formulae into SNF and then detect (possibly the same) $\square$-formulae during temporal resolution. Maybe a more pragmatic approach would be to try and resolve any suitable $\square$-formulae with $\Diamond$-rules before entering the main step and temporal resolution cycle.

We have advanced theorem proving for PTL by producing a prototype fully automated temporal resolution theorem prover. However an efficient implementation, in a *fast* language is required before a fair comparison with other temporal theorem provers can be made. Further, ideas to parallelise temporal resolution and the algorithms developed here may make such systems increasingly efficient. Finally, the extension of Fisher's basic system applied to transition systems developed by Peim [Pei94] aiming to improve efficiency by separating program control and data has based its loop search algorithm on the Breadth-First Algorithm described in this thesis.

9.2 Future Work

9.2.1 Strategies

In the thesis we have investigated a range of strategies useful for the temporal resolution method. Here we note some ideas that have not been mentioned but that may develop into useful strategies.

Using semantic information to guide the loop search

Writing formulae into a normal form before applying resolution rules means that formulae are of a uniform shape and so fewer resolution rules are required. The disadvantage of this clausal approach is that as well as increasing the length of the formula, the structure of the formula is lost [PG86]. However it may be possible to use semantic information from the original formula or problem to guide the loop search, or direct the application of step resolution. Recall that using the Set of Support Strategy (see Section 6.1.1) for classical logics, the SoS can be chosen semantically by selecting the set of clauses derived from the negation of the property we are trying to prove. Thus the application of the classical resolution rule is restricted and can only be applied between rules where a contradiction may exist by the use of semantic information. We may be able to use similar ideas in the temporal resolution method by isolating the rules derived from the negation of the proposition we are trying to prove and limiting the step and temporal resolution allowed. However with the SoS strategy, completeness is only guaranteed if the set of clauses minus the SoS is satisfiable. Similarly we must make sure that adopting an SoS type strategy does not affect the completeness of the system.

Secondly when translating formulae into SNF any $\square$–operators are removed by rewriting. During temporal resolution we are then required to search for sets of formulae that together imply a $\square$–formula. It may be useful to store information about the $\square$–operators we have removed and then resolve them directly with $\Diamond$–rules.

Reuse of Loops Found

Rather than look for loops for each eventuality in isolation it may be useful to check whether a loop detected for one eventuality can be used for resolution with another. For example consider the two $\Diamond$–rules

$$\begin{array}{lcl} \mathcal{L}Q & \Rightarrow & \Diamond\neg l \\ \mathcal{L}R & \Rightarrow & \Diamond\neg m \end{array}$$

and that we detect the loop

$$\bullet a \Rightarrow a \wedge l$$

for resolution with the first $\diamondsuit$–rule that is made up from combining the rules

$$\begin{array}{rcl} \bigcirc a & \Rightarrow & a \\ \bigcirc a & \Rightarrow & l. \end{array}$$

If we search the rule-set for the rule $\bigcirc a \Rightarrow m$ then we can resolve the "a" loop with both eventualites. Here we need to evaluate the cost of detecting loops with the cost of checking whether the loop is applicable to any other eventuality.

Scheduling Step and Temporal Resolution

In the strategies discussed so far we have assumed that all step resolution has been carried out before any temporal resolution is performed. However, performing some step resolution followed by some temporal resolution may be advantageous. This idea may link in with the idea of using semantic information to guide the refutation, particularly related to the set of support strategy. Again we must consider completeness issues carefully.

Choice of $\diamondsuit$–formula for Resolution

If a set of rules contains several $\diamondsuit$–rules, the order they are resolved in may make a difference to the efficiency of the search. For example the contradiction may depend on one $\diamondsuit$–rule and none of the others, or rules that make up loops for resolution with one eventuality may be resolvents from resolution with another $\diamondsuit$–rule. If we can develop a heuristic to prioritise $\diamondsuit$–rules usefully for resolution then we may be able to detect a contradiction more quickly. A suggested heuristic would be to choose the $\diamondsuit$–rule with the largest number of start rules (i.e. if the $\diamondsuit$–rule is $\mathcal{L}Q \Rightarrow \diamondsuit\neg l$ the start rules are of the form $\bigcirc A \Rightarrow l$) because, as there are more rules involing l we may be more likely to find a loop in l. A further idea is to choose the $\diamondsuit$–rule related to semantic information and the set of support already discussed.

9.2.2 An Efficient Implementation

Currently subprograms for the prototype temporal resolution theorem prover for the translation to the normal form, step resolution and temporal resolution (both Depth and Breadth-First Searches) are written in SICStus Prolog [CW91]. In Chapter 6 we described how we used Otter [McC90], a classical resolution theorem prover written in C, to implement step resolution. However an efficient implementation for the full theorem prover in a *fast* language with a user friendly interface is desirable. Then a range of more challenging examples can be tried to obtain a more realistic comparison with other temporal resolution theorem provers.

9.2.3 Parallel Temporal Resolution

Here, we have described strategies to guide and implement Fisher's temporal resolution system sequentially. An overview to how this may be extended to use parallel implementation techniques is given in [DFJ95]. Although parallelising the algorithms will not affect the complexity of the underlying proof procedure (recall the complexity of satisfiablity of PTL is PSPACE-complete [SC85]), it may mean that larger, more complex, formulae may be handled.

We are primarily interested in applying parallelism to the step and temporal resolution parts of the main algorithm as these are the most costly parts of the method. As previously mentioned step resolution is similar to classical resolution so it is likely that techniques developed to parallelise classical resolution can be utilised, for example [BdKH$^+$88, BG87, Clo87, LM90, War87]. Temporal resolution can be parallelised by applying loop search algorithms to all eventualities in parallel as well as during loop search itself. To parallelise the search for loops for a particular eventuality we must consider how to parallelise the different loop search algorithms we have discussed.

For the Depth-First Search Algorithm a parallel Depth-First Search could be spawned for each start node. However, some communication would be needed between branches to avoid exploring the same part of the graph several times. In Breadth-First Search the top node could be split into parts and each part run in parallel. However to retain completeness a full Breadth-First Search would have to be carried out on one processor. Further, although we have not chosen to implement the MTSA here, if parallel techniques could be applied to the construction of the graph then this may prove a useful method for applying temporal resolution in a parallel temporal resolution theorem prover. Finally we may be able to apply parallelism to the algorithm as a whole by having processes performing

- translation to SNF;
- application of the step resolution rule;
- simplification and subsumption during step resolution;
- temporal resolution

in parallel taking advantage of the idea that not all step resolution has to be performed before any temporal resolution can be carried out as mentioned earlier.

9.2.4 Application to Other Temporal Logics

It may be possible to use Fisher's resolution method for PTL as the basis for developing resolution methods for other temporal logics. The difficulties anticipated for μ-TL, Branching Time Temporal Logics, Temporal Logics of Knowledge and Belief and First-Order Temporal Logic are documented below.

Application to Temporal Logic with Fixpoint Operators

Wolper extended PTL in [Wol83] to increase its expressiveness, discussed above, allowing properties defineable by right linear grammars, to form a family of languages known as ETL. However, for each grammar, a grammar operator is required for each non-terminal symbol in the grammar. νTL [BB86] is an extension of PTL by maximal and minimal fixpoints. Only one temporal connective, $\bigcirc$, is required and νTL is shown to be equivalent to ETL. μTL [Var88] is similar to νTL but allows both past-time temporal operators and fixpoint operators.

The translation of PTL formulae to SNF removes the future-time temporal formulae $\Box$ and $\mathcal{W}$ (corresponding to maximal fixpoints) and rewrites them as a pair of mutually recursive rules. For example

$$\bullet\,\mathbf{false} \Rightarrow a\,\mathcal{W}\,b$$

is rewritten as

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & b \lor (a \land t) \\ \circledcirc\, t & \Rightarrow & b \lor (a \land t) \end{array}$$

where t is a new proposition symbol. However the future-time formulae $\Diamond$ and $\mathcal{U}$ (corresponding to minimal fixpoints) are rewritten as a pair of mutually recursive rules and a $\Diamond$–rule. For example

$$\bullet\,\mathbf{false} \Rightarrow a\,\mathcal{U}\,b$$

is rewritten as

$$\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & b \lor (a \land t) \\ \circledcirc\, t & \Rightarrow & b \lor (a \land t) \\ \bullet\,\mathbf{false} & \Rightarrow & \Diamond b. \end{array}$$

Now, if any *arbitary* maximal fixpoint can be rewritten into SNF and any *arbitrary* minimal fixpoint can be rewritten as a set of recursive rules and a $\Diamond$–rule then SNF is powerful enough to translate μTL formulae. So, *if* any μTL formula can be translated *directly* into SNF then Fisher's temporal resolution system can still be applied.

Application to Branching Time

Throughout the thesis we have considered linear time temporal logics and models of such logics have been a series of states. If we relax the forwards linearity constraint to consider branching time temporal logics [Lam80, EH86, ES88], models will have a tree-like structure that branches into the future (if the backwards linearity constraint is maintained). Thus, the syntax of branching time logics have operators related to paths, $\forall$, for all paths and $\exists$, there exists a path, and operators relating to states, $\Box$, $\Diamond$, $\bigcirc$ and $\mathcal{U}$ usually written in this context as G, F, X and $\mathcal{U}$ respectively. How and which operators may be combined depends on the particular branching time logic we are considering, see [BAMP81, CE81, CES86] for example. Thus, when trying to adapt Fisher's temporal resolution method to apply to branching time logics we must adjust the normal form and both step and temporal resolution rules to deal with two possible $\Box$, $\Diamond$ and $\bigcirc$–operators. Consequently both the normal form and the resolution rules will be more complicated.

Application to Temporal Logics of Belief and Knowledge

Temporal logics enriched with modal operators to deal with belief and knowledge, [HV88], have been useful for the specification and verification of distributed and multi-agent systems [WF92, FW93]. As both types of system are becoming more important in computer science and artificial intelligence, proof methods for such logics are desirable. Tableau based decision procedures for such logics have been proposed [WF94] and it may be possible to extend Fisher's temporal resolution method for these logics too.

Application to First-Order Temporal Logic

Fisher's resolution system has only been described for PTL however extensions for first-order logic could be investigated. A first-order normal form, SNF_f, has been proposed in [Fis92b] and translation into SNF_f described. However, as *full* First-Order Temporal Logic is obviously undecidable [Hus87, Sza87, Aba88], we would first have to look for subsets of First-Order Temporal Logic to which this form of temporal resolution could be successfully applied.

Appendix A

Examples

A set of examples taken from [MP81] used throughout the thesis.

No	Example
1	$\Box\neg w \Leftrightarrow \neg\Diamond w$
2	$\Diamond\neg w \Leftrightarrow \neg\Box w$
3	$\bigcirc\neg w \Leftrightarrow \neg\bigcirc w$
4	$w \Rightarrow \Diamond w$
5	$\Box w \Rightarrow w$
6	$\bigcirc w \Rightarrow \Diamond w$
7	$\Box w \Rightarrow \bigcirc w$
8	$\Box w \Rightarrow \Diamond w$
9	$\Box w \Rightarrow \bigcirc\Box w$
10	$w_1\,\mathcal{U}\,w_2 \Rightarrow \Diamond w_2$
11	$\Diamond\Box w \Rightarrow \Box\Diamond w$
12	$\Box w \Leftrightarrow \Box\Box w$
13	$\Diamond w \Leftrightarrow \Diamond\Diamond w$
14	$\Box\bigcirc w \Leftrightarrow \bigcirc\Box w$
15	$\Diamond\bigcirc w \Leftrightarrow \bigcirc\Diamond w$
16	$((\bigcirc w_1)\,\mathcal{U}\,(\bigcirc w_2)) \Leftrightarrow \bigcirc(w_1\,\mathcal{U}\,w_2)$
17	$\Box(w_1 \wedge w_2) \Leftrightarrow (\Box w_1 \wedge \Box w_2)$
18	$\Diamond(w_1 \vee w_2) \Leftrightarrow (\Diamond w_1 \vee \Diamond w_2)$
19	$\bigcirc(w_1 \wedge w_2) \Leftrightarrow (\bigcirc w_1 \wedge \bigcirc w_2)$
20	$\bigcirc(w_1 \vee w_2) \Leftrightarrow (\bigcirc w_1 \vee \bigcirc w_2)$
21	$\bigcirc(w_1 \Rightarrow w_2) \Leftrightarrow (\bigcirc w_1 \Rightarrow \bigcirc w_2)$
22	$\bigcirc(w_1 \Leftrightarrow w_2) \Leftrightarrow (\bigcirc w_1 \Leftrightarrow \bigcirc w_2)$
23	$((w_1 \wedge w_2)\,\mathcal{U}\,w_3) \Leftrightarrow ((w_1\,\mathcal{U}\,w_3) \wedge (w_2\,\mathcal{U}\,w_3))$
24	$(w_1\,\mathcal{U}\,(w_2 \vee w_3)) \Leftrightarrow ((w_1\,\mathcal{U}\,w_2) \vee (w_1\,\mathcal{U}\,w_3))$
25	$\Box w_1 \vee \Box w_2 \Rightarrow \Box(w_1 \vee w_2)$
26	$\Diamond(w_1 \wedge w_2) \Rightarrow \Diamond w_1 \wedge \Diamond w_2$
27	$((w_1\,\mathcal{U}\,w_3) \vee (w_2\,\mathcal{U}\,w_3)) \Rightarrow (w_1 \vee w_2)\,\mathcal{U}\,w_3$
28	$(w_1\,\mathcal{U}\,(w_2 \wedge w_3)) \Rightarrow ((w_1\,\mathcal{U}\,w_2) \wedge (w_1\,\mathcal{U}\,w_3))$
29	$\Box(w_1 \Rightarrow w_2) \Rightarrow (\Box w_1 \Rightarrow \Box w_2)$
30	$\Box(w_1 \Rightarrow w_2) \Rightarrow (\Diamond w_1 \Rightarrow \Diamond w_2)$
31	$\Box(w_1 \Rightarrow w_2) \Rightarrow (\bigcirc w_1 \Rightarrow \bigcirc w_2)$
32	$\Box(w_1 \Rightarrow w_2) \Rightarrow ((w_1\,\mathcal{U}\,w_3) \Rightarrow (w_2\,\mathcal{U}\,w_3))$
33	$\Box(w_1 \Rightarrow w_2) \Rightarrow ((w_0\,\mathcal{U}\,w_1) \Rightarrow (w_0\,\mathcal{U}\,w_2))$
34	$(\Box w_1 \wedge \bigcirc w_2) \Rightarrow \bigcirc(w_1 \wedge w_2)$
35	$(\Box w_1 \wedge \Diamond w_2) \Rightarrow \Diamond(w_1 \wedge w_2)$

36	$(\Box w_1 \wedge (w_2 \mathcal{U} w_3)) \Rightarrow (w_1 \wedge w_2) \mathcal{U} (w_1 \wedge w_3)$
37	$(w \wedge \Box(w \Rightarrow \bigcirc w)) \Rightarrow \Box w$
38	$(w \wedge \Diamond\neg w) \Rightarrow \Diamond(w \wedge \bigcirc\neg w)$
39	$(\Diamond w_1 \wedge \Diamond w_2) \Rightarrow (\Diamond(w_1 \wedge \Diamond w_2) \vee \Diamond(w_2 \wedge \Diamond w_1))$
40	$\Box w \Leftrightarrow w \wedge \bigcirc \Box w$
41	$\Diamond w \Leftrightarrow w \vee \bigcirc \Diamond w$
42	$w_1 \mathcal{U} w_2 \Leftrightarrow w_2 \vee (w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2))$
43	$(\neg w \mathcal{U} w) \Rightarrow \Diamond w$
44	$(\Box w_1 \wedge \Diamond w_2) \Rightarrow (w_1 \mathcal{U} w_2)$
45	$((w_1 \Rightarrow w_2) \mathcal{U} w_3) \Rightarrow ((w_1 \mathcal{U} w_3) \Rightarrow (w_2 \mathcal{U} w_3))$
46	$((w_1 \mathcal{U} w_2) \wedge (\neg w_2 \mathcal{U} w_3)) \Rightarrow (w_1 \mathcal{U} w_3)$
47	$(w_1 \mathcal{U} (w_2 \wedge w_3)) \Rightarrow ((w_1 \mathcal{U} w_2) \mathcal{U} w_3)$
48	$((w_1 \mathcal{U} w_2) \mathcal{U} w_3) \Rightarrow ((w_1 \vee w_2) \mathcal{U} w_3)$
49	$(\Diamond w_1 \wedge \Diamond w_2) \Rightarrow ((\neg w_1 \mathcal{U} w_2) \vee (\neg w_2 \mathcal{U} w_1))$

Table A.1: Formulae used in Examples

Appendix B

Statistics

B.1 Timings for the two Search Algorithms

Timings for the theorem prover using the Depth and Breadth-First Loop Searching Algorithms (see Section 3.4.2). The number in the "Example" column refers to the examples given in Appendix A. The timings are the total amount of time spent in the loop finding section for each eventuality. All timings are in milliseconds and are obtained using the Prolog predicate *statistics(X, Y)* with X bound to *runtime* and run on a SPARCstation 1 using compiled SICStus Prolog code. "Calls" is the number of times the program attempted loop search for each eventuality.

Example	Eventuality	Depth-First		Breadth-First	
		Calls	Time	Calls	Time
1	$\Diamond w$	1	0	1	9
2	$\Diamond \neg w$	1	0	1	10
9	$\Diamond \neg w$	1	0	1	9
10	$\Diamond w_2$	1	0	1	9
11	$\Diamond t_5$	2	624	1	39
	$\Diamond t_6$	1	20	1	89
12	$\Diamond \neg w$	2	30	1	20
	$\Diamond t_5$	2	39	1	60
13	$\Diamond w$	2	40	1	30
	$\Diamond t_5$	2	29	1	60
14	$\Diamond \neg w$	2	230	1	70
	$\Diamond t_4$	1	19	1	79
15	$\Diamond w$	2	248	1	60
	$\Diamond t_1$	1	20	1	79
16	$\Diamond w_2$	5	11038	1	1310
	$\Diamond t_2$	4	2536	1	550
17	$\Diamond \neg w_1$	2	51	1	30
	$\Diamond \neg w_2$	2	69	1	69
	$\Diamond t_7$	2	139	1	120
18	$\Diamond w_1$	2	59	1	40
	$\Diamond w_2$	2	89	1	60
	$\Diamond t_7$	2	130	1	110
23	$\Diamond w_3$	4	3030	1	1309
24	$\Diamond t_7$	2	1170	1	1859
	$\Diamond w_2$	2	528	1	1370
	$\Diamond w_3$	2	589	1	910

25	$\Diamond t_4$	2	28	1	20
26	$\Diamond t_4$	2	29	1	19
27	$\Diamond w_3$	2	178	1	79
28	$\Diamond t_4$	2	109	1	110
29	$\Diamond \neg w_2$	1	29	1	29
30	$\Diamond w_1$	1	40	1	30
32	$\Diamond w_3$	1	129	1	89
33	$\Diamond w_1$	1	230	1	69
35	$\Diamond w_2$	1	40	1	30
36	$\Diamond w_3$	1	260	1	90
37	$\Diamond \neg w$	1	50	1	40
38	$\Diamond \neg w$	1	50	1	29
39	$\Diamond w_1$	2	1418	2	1289
	$\Diamond w_2$	1	20	1	170
40	$\Diamond \neg w$	1	0	1	10
41	$\Diamond w$	1	0	1	0
42	$\Diamond w_2$	1	60	1	40
43	$\Diamond w$	1	9	1	10
44	$\Diamond w_2$	1	50	1	29
45	$\Diamond w_3$	1	139	1	79
46	$\Diamond w_2$	1	269	1	99
47	$\Diamond t_3$	2	259	2	339
48	$\Diamond w_3$	1	789	1	109
49	$\Diamond w_1$	1	60	1	39

Table B.1: Times for Depth and Breadth-First Searches

B.2 Timings of Loops of Different Structures

Table B.2 shows the time that Depth and Breadth-First Search takes to find loops in two sets of rules R_1 and R_2. All times are given in milliseconds, and the examples were tested on a *SPARC*station 1 using compiled SICStus Prolog code. Both sets of rules R_1 and R_2 have n start rules with a single literal on the left and the negation of the eventuality on the right. Also the loop in R_1 has n rules with a single literal on the left hand side and $n-1$ (for n ¿ 1) literals disjoined on the right hand side. The loop R_2 has a single literal on the right hand side and $n-1$ (for n ¿ 1) literals conjoined on the left hand side. For $n = 1$ we have one rule with a single literal on the left and right hand sides. The value n is shown in the column "rules" and varies from 1 to 25. For more details see Section 3.4.3.

	Rule-set R_1		Rule-set R_2	
Rules	DFS	BFS	DFS	BFS
1	10	9	0	10
3	39	19	60	100
5	79	39	170	380
7	149	99	440	1220
9	289	170	980	3489
11	450	250	1959	7630
13	699	339	3420	17900
15	980	439	5660	34590
17	1350	559	8930	62280
19	1789	770	13480	104919
21	2349	960	19710	169599
23	2980	1210	27669	260570
25	3740	1469	37390	389300

Table B.2: Timings of Two Loops with Different Algorithms

B.3 Timings using the Pruning Algorithm

Table B.3 shows timings for the theorem prover using the Breadth-First Search (BFS) Algorithm with and without pruning. All times are given in milliseconds, and the examples were tested on a *SPARC*station 1 using compiled SICStus Prolog code. Note, however, to give a better comparison only the time spent carrying out temporal resolution is given (i.e. the time spent translating to SNF or performing step resolution is not included). The examples that have been omitted from the table are examples that involve no temporal resolution. The number in the "Eg." (example) column refers to the examples given in Appendix A. The number of rules is given before the loop search is carried out with and without pruning. The timings are the time spent in the loop finding section for each eventuality ($\diamondsuit$) and where pruning has been carried out this time includes the time required for pruning the rules. The column labelled "Pruning Only" gives the time taken just to prune the rules (without any loop finding). All timings have been obtained using the Prolog predicate *statistics(X,Y)* with X bound to *runtime*.

		Number of Rules		Time		
Eg.	$\diamondsuit$	BFS	BFS with Pruning	BFS	BFS with Pruning	Pruning Only
1	$\diamondsuit w$	2	2	9	20	0
2	$\diamondsuit \neg w$	2	2	10	9	0
9	$\diamondsuit \neg w$	2	2	9	9	0
10	$\diamondsuit w_2$	6	3	9	29	9
11	$\diamondsuit t_5$	16	4	39	39	29
	$\diamondsuit t_6$	32	6	89	70	49
12	$\diamondsuit \neg w$	8	5	20	30	19
	$\diamondsuit t_5$	24	5	60	59	29
13	$\diamondsuit w$	8	5	30	39	0
	$\diamondsuit t_5$	24	5	60	50	30
14	$\diamondsuit \neg w$	10	5	70	60	10
	$\diamondsuit t_4$	32	6	79	69	59
15	$\diamondsuit w$	10	5	60	49	10
	$\diamondsuit t_1$	32	5	70	79	49
16	$\diamondsuit w_2$	62	16	1310	1130	159
	$\diamondsuit t_2$	107	13	550	459	340
17	$\diamondsuit \neg w_1$	12	5	30	29	20
	$\diamondsuit \neg w_2$	26	5	69	70	39
	$\diamondsuit t_7$	40	5	120	100	70
18	$\diamondsuit w_1$	12	5	40	29	19
	$\diamondsuit w_2$	26	5	60	59	39
	$\diamondsuit t_7$	40	5	110	109	69
23	$\diamondsuit w_3$	42	12	1309	369	89
24	$\diamondsuit t_7$	47	20	1859	1850	89
	$\diamondsuit w_2$	166	20	1370	1220	500
	$\diamondsuit w_3$	172	16	910	690	519
25	$\diamondsuit t_4$	8	4	20	20	9
26	$\diamondsuit t_4$	8	4	19	30	10
27	$\diamondsuit w_3$	13	5	79	80	19
28	$\diamondsuit t_4$	16	7	110	80	19
29	$\diamondsuit \neg w_2$	5	3	29	30	10
30	$\diamondsuit w_1$	5	3	30	30	0
32	$\diamondsuit w_3$	12	4	89	70	29
33	$\diamondsuit w_1$	13	4	69	89	19

35	$\Diamond w_2$	5	3	30	30	10
36	$\Diamond w_3$	14	4	90	70	20
37	$\Diamond \neg w$	4	4	40	50	0
38	$\Diamond \neg w$	4	4	29	30	9
39	$\Diamond w_1$	10	6	30	30	0
	$\Diamond w_2$	30	9	170	109	60
	$\Diamond w_1$	90	12	1259	370	229
40	$\Diamond \neg w$	2	2	10	10	10
41	$\Diamond w$	2	2	0	9	0
42	$\Diamond w_2$	7	3	40	30	19
43	$\Diamond w$	4	4	10	10	10
44	$\Diamond w_2$	5	3	29	19	0
45	$\Diamond w_3$	14	4	79	50	30
46	$\Diamond w_2$	14	4	99	79	39
47	$\Diamond t_3$	19	6	200	130	39
	$\Diamond t_3$	31	5	139	99	70
48	$\Diamond w_3$	13	3	109	60	30
49	$\Diamond w_1$	6	3	39	30	19

Table B.3: Times for Depth and Breadth-First Search

B.4 Statistics Related to Translation to SNF

Here the statistics for three different translations to SNF described in Chapter 7 are given for the examples given in Appendix A. Column names have been abbreviated as follows.

1. R - Rules: The number of $\square$–rules after translation to SNF and one round of simplification and subsumption has taken place.
2. V - Variables: The number of new variables created during the SNF translation.
3. $\diamondsuit$ - Eventualities: The number of $\diamondsuit$–rules.
4. P - Total Propositions: The total number of propositions in the set of rules.

	Tran. 1				Tran. 2				Tran. 3			
No.	R	V	$\diamondsuit$	P	R	V	$\diamondsuit$	P	R	V	$\diamondsuit$	P
1	11	4	2	24	7	2	1	14	6	2	1	10
2	11	4	2	24	7	2	1	14	6	2	1	10
3	8	4	0	12	4	2	0	6	4	2	0	6
4	4	1	0	6	6	1	0	11	4	1	0	6
5	4	1	0	6	6	1	0	11	4	1	0	6
6	6	2	0	9	7	2	0	13	6	2	0	9
7	6	2	0	9	7	2	0	13	6	2	0	9
8	8	2	0	12	10	2	0	20	8	2	0	12
9	6	2	1	9	7	2	1	13	6	2	1	9
10	9	2	1	18	13	2	1	34	9	2	1	18
11	18	6	2	40	12	4	2	26	18	6	2	40
12	19	7	3	47	14	4	2	32	12	5	2	26
13	19	7	3	47	14	4	2	32	12	5	2	26
14	14	6	2	31	14	6	2	30	14	6	2	31
15	14	6	2	31	14	6	2	30	14	6	2	31
16	32	9	2	95	26	10	2	68	23	8	2	61
17	20	7	3	52	26	7	3	66	20	7	3	52
18	20	7	3	52	26	7	3	66	20	7	3	52
19	13	6	0	30	13	6	0	30	13	6	0	30
20	13	6	0	30	13	6	0	30	13	6	0	30
21	13	6	0	30	13	6	0	30	13	6	0	30
22	27	10	0	80	19	6	0	52	19	6	0	52
23	37	8	3	108	48	12	3	145	37	8	3	108
24	29	7	3	88	50	13	3	151	29	7	3	88
25	15	4	1	32	15	4	1	34	15	4	1	32
26	15	4	1	32	15	4	1	34	15	4	1	32
27	16	3	2	41	25	6	2	74	16	3	2	41
28	21	4	1	53	31	7	1	89	21	4	1	53
29	10	3	1	18	12	3	1	27	10	3	1	18
30	10	3	1	18	12	3	1	27	10	3	1	18
31	8	3	0	14	9	3	0	19	8	3	0	14
32	13	3	1	28	20	5	1	54	13	3	1	28
33	13	3	1	28	20	5	1	54	13	3	1	28
34	8	3	0	13	9	3	0	17	8	3	0	13
35	10	3	1	18	12	3	1	27	10	3	1	18
36	13	3	1	30	20	5	1	57	13	3	1	30
37	8	3	1	15	9	3	1	20	8	3	1	15
38	8	3	1	15	9	3	1	20	8	3	1	15

39	20	6	2	42	22	6	2	52	20	6	2	42
40	11	4	2	24	13	4	2	30	6	2	1	10
41	11	4	2	24	13	4	2	30	6	2	1	10
42	16	4	2	44	24	6	2	69	12	3	1	28
43	7	2	1	13	10	3	1	22	7	2	1	13
44	10	3	1	18	14	4	1	32	10	3	1	18
45	14	3	2	34	23	6	2	67	14	3	2	34
46	14	3	2	32	23	6	2	64	14	3	2	32
47	21	4	1	52	31	7	1	88	21	4	1	52
48	16	3	2	41	25	6	2	73	16	3	2	41
49	12	4	2	24	18	6	2	44	12	4	2	24

Table B.4: Statistics for Different SNF Translations

Appendix C

Using OTTER for Step Resolution

The template for the OTTER file is given below.

```
set(binary_res).
set(factor).
set(print_lists_at_end).
set(process_input).
set(prolog_style_variables).
clear(print_kept).
clear(print_given).
clear(print_back_sub).
clear(print_proofs).
assign(max_mem, 0).
assign(max_seconds, 0).
set(free_all_mem).

list(usable).
end_of_list.
list(sos).
end_of_list.
```

The rules from the Prolog lists *usable* and *sos* are inserted after

```
list(usable)
```

and

```
list(sos)
```

respectively. The other flags that have been set or cleared are described below:-

- **binary_res** Use the inference rule binary resolution.
- **factor** Not really necessary as factoring is only really required for first-order classical logic. However setting this flag prevents OTTER outputting the following warning message.

  ```
  WARNING, BINARY_RES is set, but FACTOR is clear.
  ```

- **print_lists_at_end** This prints the usable and sos lists out at the end of processing. OTTER's main loop takes an element of the sos, applies the inference rule being used (in this case binary resolution), appends any resolvents that pass all retention tests to the sos, and appends the rule itself to usable. This loop continues until a refutation has been found or the sos is empty. If we print usable out at the end this will contain all the rules in the current rule-set (without any subsumed rules).

- **process_input** The usable rules and sos input to OTTER are processed as if they have been generated by an inference rule. This means that simplification and subsumption will be done on input clauses by OTTER and it need not be done by Prolog.
- **prolog_style_variables** A name is a variable if and only if it starts with an upper case letter or '_'. If this flag is not set all names in clauses that start with the letters **u**, **v**, **w**, **x**, **y** or **z** are considered to be variables by OTTER.
- **print_kept** If set this flag outputs new clauses if they pass all retention tests.
- **print_given** If this flag is set clauses are output as soon as they become given clauses (i.e. the clause from the sos we are resolving with).
- **print_back_sub** If set this flag outputs clauses when they are back subsumed.
- **print_proof** If set this flag prints any proofs to the output file.
- **max_mem n** If n is not 0 OTTER will terminate the search before more than n Kbytes have been dynamically allocated.
- **max_seconds n** If n is not 0 OTTER will terminate the search after about n CPU seconds.
- **free_all_mem** If set return all free memory back to the memory manager at the end of the run.

Bibliography

[Aba87] M. Abadi. *Temporal-Logic Theorem Proving.* PhD thesis, Department of Computer Science, Stanford University, March 1987.

[Aba88] M. Abadi. The Power of Temporal Proofs. Technical Report 30, DEC Systems Research Center, Palo Alto, California, August 1988.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[And87] I. Anderson. *Combinatorics of Finite Sets.* Oxford University Press, 1987.

[BAMP81] M. Ben-Ari, Z. Manna, and A. Pnueli. The Temporal Logic of Branching Time. In *Proceedings of the Eighth ACM Symposium on the Principles of Programming Languages*, pages 164–176, Williamsburg, Virginia, January 1981. Also in Acta Informatica, volume 20, number 3, pages 207-226.

[Bar87] 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.

[BB86] B. Banieqbal and H. Barringer. A Study of an Extended Temporal Logic and a Temporal Fixed Point Calculus. Technical Report UMCS-86-10-2, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K., October 1986.

[BCL91] J. R. Burch, E. M. Clarke, and D. E. Long. Representing Circuits More Efficiently in Symbolic Model Checking. In *Design Automation Conference 1991*, 1991.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} states and beyond. In *Proceedings of the 5th Symposium on Logic in Computer Science, Philadelphia*, pages 428–439, June 1990.

[BdKH+88] U. Baron, J. de Kergommeaux, M. Hailpern, M. Ratcliffe, M. Roberts, J-Cl. Syre, and H. Westphal. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *Proceedings of the International Conference of Fifth Generation Computer Systems*, pages 841–850, Tokyo, Japan, 1988.

[BdlTC90] T. Boy de la Tour and G. Chaminade. The Use of Renaming to Improve the Efficiency of Clausal Theorem Proving. In Ph. Jorrand and V. Sgurev, editors, *Artificial Intelligence IV: Methodology, Systems, Applications*, pages 3–12. Elsevier Science Publishers B.V. (North-Holland), 1990.

[BFG+89a] 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*, volume 430 of *Lecture Notes in Computer Science*, Mook, Netherlands, June 1989. Springer-Verlag.

[BFG89b] H. Barringer, M. Fisher, and G. Gough. Fair SMG and Linear Time Model Checking. In *Proceedings of Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, Grenoble, France, June 1989. Springer-Verlag.

[BG85] J. P. Burgess and Y. Gurevich. The Decision Problem for linear Temporal Logics. *Notre Dame Journal of Formal Logic*, 26(2):115–128, April 1985.

[BG87] J. Beer and W. Giloi. POPE - A Parallel-Operating Prolog Engine. *Future Generations Computer Systems*, 3(2):83–92, 1987.

[BKP84] 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.

[BKP86] H. Barringer, R. Kuiper, and A. Pnueli. A Really Abstract Concurrent Model and its Temporal Logic. In *Proceedings of the Thirteenth ACM Symposium on the Principles of Programming Languages*, St. Petersberg Beach, Florida, January 1986.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Büc62] J. R. Büchi. On a Decision Method in Restricted Second Order Arithmetic. *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science 1960*, pages 1–12, September 1962.

[Bur84] J. P. Burgess. Basic Tense Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic (II)*, volume 165 of *Synthese Library*, chapter II.2, pages 89–134. Reidel, 1984.

[CdC84] 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.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronisation Skeletons Using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on the Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CL73] C. L. Chang and R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[Clo87] W. Clocksin. Principles of the DelPhi Parallel Inference Machine. *The Computer Journal*, 30(5):386–392, October 1987.

[CW91] M. Carlsson and J. Widen. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, Kista, Sweden, September 1991.

[DFB94] 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.

[DFJ95] C. Dixon, M. Fisher, and R. Johnson. Parallel Temporal Resolution. In *Proceedings of TIME-95 the Second International Workshop on Temporal Representation and Reasoning*, April 1995. Pre-symposium workshop held in conjunction with the Eighth Florida Artificial Intelligence Research Symposium (FLAIRS-95), Melbourne Beach, Florida.

[Dix92] C. L. Dixon. A Graph-Based Approach to Resolution in Temporal Logic. Master's thesis, Department of Computer Science, University of Manchester, October 1992. Also University of Manchester, Department of Computer Science, Technical Report UMCS-93-9-2.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time Temporal Logic. *ACM Journal*, 33(1):151–178, January 1986.

[ES88] E. A. Emerson and J. Srinivasan. Branching Time Temporal Logic. *Lecture Notes in Computer Science*, 354:123–172, 1988.

[Fis91] 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.

[Fis92a] M. Fisher. A Model Checker for Linear Time Temporal Logic. *Formal Aspects of Computing*, 4(3):299–319, June 1992.

[Fis92b] 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.

[Fis92c] M. Fisher. A Resolution Method for Temporal Logic. (Notes provided for the Fourth European Summer School in Logic, Language and Information, University of Essex, August 1992), 1992.

[Fis95] M. Fisher. A Normal Form for Temporal Logic and its Applications in Theorem-Proving and Execution. (Submitted), 1995.

[Fit83] M. Fitting. *Proof Methods for Modal and Intuitionistic Logics*, volume 169 of *Synthese Library*. D. Reidel, 1983.

[FN92] M. Fisher and P. Noël. Transformation and Synthesis in MetateM – Part I: Propositional MetateM. Technical Report UMCS-92-2-1, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K., February 1992.

[FW93] M. Fisher and M. Wooldridge. Specifying and Verifying Distributed Intelligent Systems. In Miguel Filgueiras and Luís Damas, editors, *Portuguese Conference on Artificial Intelligence (EPIA)*, volume 727 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, October 1993.

[Gab79] D. M. Gabbay. The Separation Property of Tense Logics. Unpublished manuscript, September 1979.

[Gab87] D. M. Gabbay. Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of Colloquium on Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 402–450, Altrincham, U.K., 1987. Springer-Verlag.

[GB88] G. D. Gough and H. Barringer. A Semantics Driven Temporal Verification System. In H. Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming (ESOP '88)*, volume 300 of *Lecture Notes in Computer Science*, Nancy, France, March 1988. Springer-Verlag.

[Gou84] G. D. Gough. Decision Procedures for Temporal Logic. Master's thesis, Department of Computer Science, University of Manchester, October 1984. Also University of Manchester, Department of Computer Science, Technical Report UMCS-89-10-1.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. The Temporal Analysis of Fairness. In *Proceedings of the Seventh ACM Symposium on the Principles of Programming Languages*, pages 163–173, Las Vegas, Nevada, January 1980.

[GW89] P. Gribomont and P. Wolper. *From Modal Logic To Deductive Databases*, chapter 4. John Wiley and Sons, 1989. Edited by A. Thayse.

[Hai82] B. T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic*, volume 129 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[Hak85] A. Haken. The Intractability of Resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[Har87] D. Harel. *Algorithmics: The Spirit of Computing.* Addison-Wesley, 1987.

[HC68] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* Methuen (UP), 1968.

[HMM83] J. Halpern, Z. Manna, and B. Moszkowski. A Hardware Semantics Based on Temporal Intervals. In J. Diaz, editor, *Proceedings of the 10th International Colloquim on Automata Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[Hus87] W. Hussak. Decidability in Temporal Presburger Arithmetic. Master's thesis, Department of Computer Science, University of Manchester, February 1987.

[HV88] J. Y. Halpern and M. Y. Vardi. Reasoning about Knowledge and Time in Asynchronous Systems. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 53–65, Chicago, Illinois, May 1988.

[Kow74] R. Kowalski. Predicate Logic as a Programming Language. In Jack L. Rosenfield, editor, *Proceedings of the International Federation for Information Processing*, pages 569–574. North Holland, 1974.

[Kow79] R. Kowalski. *Logic for Problem Solving.* North Holland, 1979.

[Lam80] L. Lamport. Sometime is Sometimes Not Never. In *Proceedings of the Seventh ACM Symposium on the Principles of Programming Languages*, pages 174–185, Las Vegas, Nevada, January 1980.

[Lam83] L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[LM90] E. Lusk and W. McCune. Experiments with ROO, a Parallel Automated Deduction System. In B. Fronhöfer and G. Wrightson, editors, *Proceedings of the International Workshop on Parallelization in Inference Systems*, volume 590 of *Lecture Notes in Artificial Intelligence*, pages 139–162. Springer-Verlag, 1990.

[Lov78] D. W. Loveland. *Automated Theorem Proving: a Logical Basis.* North-Holland, Inc., 1978.

[McC90] W. W. McCune. *OTTER 2.0 Users Guide.* Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439-4801, March 1990. ANL-90/9.

[Mos83] B. Moszkowski. *Reasoning about Digital Circuits.* PhD thesis, Department of Computer Science, Stanford University, 1983.

[MP81] 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.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1992.

[Mur82] N. V. Murray. Completely Non-Clausal Theorem Proving. *Artificial Intelligence*, 18:67–85, 1982.

[OL82] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[Pei94] M. Peim. Propositional Temporal Resolution Over Labelled Transition Systems. (Unpublished Technical Note), 1994.

[PG86] D. A. Plaisted and S. A. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the Eighteenth Symposium on the Foundations of Computer Science*, Providence, November 1977.

[Pnu81] A. Pnueli. The Temporal Semantics of Concurrent Programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pnu86] 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.

[Pop94] S. Popkorn. *First Steps in Modal Logic.* Cambridge University Press, 1994.

[Rob65a] J. A. Robinson. Automatic Deduction with Hyper-resolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.

[Rob65b] J. A. Robinson. A Machine–Oriented Logic Based on the Resolution Principle. *ACM Journal*, 12(1):23–41, January 1965.

[SC85] A. P. Sistla and E. M. Clarke. Complexity of Propositional Linear Temporal Logics. *ACM Journal*, 32(3):733–749, July 1985.

[SMSV83] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An Interval Based Temporal Logic. In Edmund Clarke and Dexter Kozen, editors, *Proceedings of the Workshop on The Logic of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 443–457. Springer-Verlag, 1983.

[Smu68] R. M. Smullyan. First order logic. *Ergebnisse der Mathematik*, 43, 1968.

[SVW87] A. P. Sistla, M. Vardi, and P. Wolper. The complementation problem for büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[Sza87] A. Szalas. Arithmetical Axiomatisation of First-Order Temporal Logic. *Information Processing Letters*, 26:111–116, November 1987.

[Var87] M. Y. Vardi. Unified Verification Theory. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification,Proceedings*, Lecture Notes in Computer Science, volume 398, pages 202–212. Springer-Verlag, April 1987.

[Var88] M. Y. Vardi. A Temporal Fixpoint Calculus. In *Proceedings of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 250–259, San Diego, California, January 1988. (Extended Abstract).

[Ven86] G. Venkatesh. A Decision Method for Temporal Logic based on Resolution. *Lecture Notes in Computer Science*, 206:272–289, 1986.

[VW86] 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.

[War87] D. Warren. The SRI model for OR-parallel execution of Prolog-abstract design and implementation issues. In *Proceedings of the International Symposium on Logic Programming*, pages 92–102, 1987.

[WCR64] L. Wos, D. Carson, and G. Robinson. The Unit Preference Strategy in Theorem Proving. In *Proceedings of AFIPS Fall Joint Computer Conference*, pages 615–621. Thompson Book Company, 1964.

[WF92] M. Wooldridge and M. Fisher. A First-Order Branching Time Logic of Multi-Agent Systems. In *10th European Conference on Artificial Intelligence (ECAI)*, Vienna, Austria, August 1992. Wiley and Sons.

[WF94] M. Wooldridge and M. Fisher. A Decision Procedure for a Temporal Belief 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.

[Wol83] P. Wolper. Temporal Logic Can Be More Expressive. *Information and Control*, 56, 1983.

[Wol85] P. Wolper. The Tableau Method for Temporal Logic: An overview. *Logique et Analyse*, 110–111:119–136, June-Sept 1985.

[Wol87] P. Wolper. On the Relation of Programs and Computations to Models of Temporal Logic. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification, Proceedings*, Lecture Notes in Computer Science, volume 398, Altrincham, UK, April 1987. Springer-Verlag.

[WOLB84] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning – Introduction and Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[WRC65] L. Wos, G. Robinson, and D. Carson. Efficiency and Completeness of the Set of Support Strategy in Theorem Proving. *ACM Journal*, 12:536–541, October 1965.

[WVS83] P. Wolper, M. Vardi, and A. P. Sistla. Reasoning about Infinite Computation Paths. In *Proceedings of the Twentyfourth Symposium on the Foundations of Computer Science*. IEEE, 1983.