Computer Science
University of Manchester

A Graph-Based Approach To Resolution In Temporal Logic

Clare Dixon

Technical Report UMCS–93–9–2

A Graph-Based Approach To Resolution In Temporal Logic*

Clare Dixon

Department of Computer Science
University of Manchester
Oxford Road, Manchester, U.K.
dixonc@cs.man.ac.uk

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE

Department of Computer Science
October 1992

Technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from ftp.cs.man.ac.uk in the directory /pub/TR. The files are stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

Abstract

An approach to the mechanisation of temporal logics, based on a form of clausal resolution has been developed [Fis91, Fis93]. Temporal formulae incorporating both past-time and future-time temporal operators are converted into Separated Normal Form, and then both non-temporal and temporal resolution rules are applied. The temporal resolution rule attempts to match conditions which must be eventually satisfied with sets of formulae which together imply that the condition will never be satisfied.

In this dissertation the sets of formulae which are used in the application of the temporal resolution rule are identified using graph theoretic techniques. This is achieved by first extracting formulae or combinations of formulae that could possibly stop a condition being satisfied and then using them to build a graph. Secondly any loops identified during the construction of the graph will mean that the condition can never be satisfied, and are used in the application of the temporal resolution rule thus deriving further formulae.

This temporal resolution method has been combined with sub-programs performing translation to normal form and non-temporal resolution to produce an integrated resolution based temporal theorem prover. The main algorithm cycles through these three procedures until false is derived, (the original formula was unsatisfiable) or no new formulae can be generated (the original formula was satisfiable).

Acknowledgements

I would like to thank my supervisors Prof. Howard Barringer and Dr. Michael Fisher for their support, guidance, and encouragement during the year. I would also like to thank Graham Gough for the helpful comments and suggestions relating to the dissertation most of which have been incorporated in this version, as well as the multitude of examples he provided for testing the theorem prover. The financial support from the Science and Engineering Research Council is gratefully acknowledged. My thanks are extended also to my family for their continued support, and the staff and students on the M.Sc. 'System Design' course for an enjoyable and rewarding year.

Contents

1 Introduction **6**
- 1.1 Overview . . . 6
- 1.2 Modal Logic . . . 7
- 1.3 Temporal Logic . . . 7
  - 1.3.1 Discrete Time . . . 7
  - 1.3.2 Linear Time . . . 8
  - 1.3.3 Branching Time . . . 8
- 1.4 Resolution . . . 9
  - 1.4.1 Classical Resolution . . . 9
  - 1.4.2 Applying Resolution to Temporal Logics . . . 9
- 1.5 Graph Theoretic Techniques . . . 10
- 1.6 Description of the Dissertation . . . 11

2 Overview of the Resolution Procedure **12**
- 2.1 Propositional Temporal Logic . . . 12
  - 2.1.1 Syntax . . . 12
  - 2.1.2 Semantics . . . 14
- 2.2 The Temporal Resolution Method . . . 15
  - 2.2.1 Translation to a normal form . . . 16
  - 2.2.2 Non-Temporal Resolution . . . 17
  - 2.2.3 Temporal Resolution . . . 18

3 Implementing Temporal Resolution **21**
- 3.1 Graphical Representation of Rules . . . 21
- 3.2 The Graph Search Algorithm . . . 24
  - 3.2.1 Backward Search Methods . . . 25
  - 3.2.2 Forward Search Methods . . . 25
  - 3.2.3 A Combined Approach . . . 26
- 3.3 The Detailed Algorithm . . . 27
  - 3.3.1 Definitions of Terms . . . 27
  - 3.3.2 Graph Construction and Loop Detection . . . 27
  - 3.3.3 Backward and Forward Search Algorithms . . . 28
  - 3.3.4 Prefix to the Loop . . . 30

3.3.5 Combinations of Rules . . . 30
3.4 Examples . . . 31
3.5 Formal Analysis of the Algorithm . . . 35
3.5.1 Soundness . . . 35
3.5.2 Completeness . . . 37
3.5.3 Complexity . . . 39
3.6 Enhancements to the Algorithm . . . 39
3.6.1 Storage of the Path . . . 39
3.6.2 Detection of loops . . . 41
3.6.3 Avoiding Repetition of Search Paths . . . 42

4 The Complete Resolution Theorem Prover 43
4.1 Implementation . . . 43
4.2 Translation to SNF . . . 43
4.3 Non-Temporal Resolution . . . 44
4.3.1 Overview . . . 44
4.3.2 Detailed Algorithm . . . 45
4.4 The Full Theorem Prover . . . 46
4.4.1 Integrating the Temporal Resolution Programs . . . 46
4.4.2 Self-Contained Sub-Loops . . . 47
4.4.3 Generation of New Resolvents . . . 48
4.4.4 Storage of Loops . . . 49
4.4.5 Termination . . . 49
4.4.6 The Detailed Algorithm . . . 50
4.5 Examples . . . 51

5 Related Work 55
5.1 Methods Based on Resolution . . . 55
5.1.1 Clausal Methods . . . 55
5.1.2 Non-Clausal Methods . . . 57
5.2 Tableaux . . . 58
5.3 Timings . . . 59

6 Further Work and Conclusions 60
6.1 Possibilities for Further Work . . . 60
6.1.1 Further Work Related to the Described Implementation . . . 60
6.1.2 Extensions to the Theorem Prover . . . 61
6.2 Conclusions . . . 63

A The SNF Translation 65

B Sample Output **66**
B.1 Example 1a 66
B.2 Example 1b 67
B.3 Example 2 69
B.4 Example 3 71
B.5 Example 4 73

C Formulae used in Additional Timings **76**

Bibliography **77**

Chapter 1

Introduction

1.1 Overview

Resolution is a proof technique and, as such, can be used to test the validity of logical formulae. The validity of a formula Φ can be checked by repeatedly applying the resolution inference rule (see below) to the negated formula, $\neg\Phi$. If a contradiction (false) is derived then $\neg\Phi$ is unsatisfiable and so the original formula Φ must be valid. The process of determining the unsatisfiability of the negation of a formula rather than determining the validity of the 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. The full resolution procedure, for both propositional and first order logics, was introduced in [Rob65] and strategies to direct the application of the resolution rules have been explained in standard texts for example [CL73, Lov78, WOLB84].

Temporal logic, a variety of modal logic, has been used to provide a framework for specifying and verifying programs and may be regarded as classical logic extended with various modalities. The use of temporal logics for the specification of programs has been described in [Bar87], while examples of their use in verification may be found in [MP81, Pnu81], and a survey of their use in both areas may be found in [Pnu86]. The mechanisation of proofs in temporal logic has predominantly been carried out using the semantic tableau method [Wol83, Gou84]. Resolution methods have been developed for temporal logics based on both clausal forms [CdC84] and non-clausal forms [AM90].

In this dissertation an implementation of the resolution method for temporal logics described by Fisher [Fis91, Fis92b, Fis93] is described. This clausal resolution method incorporates classical resolution, together with a new type of temporal resolution rule. The implementation of this proof procedure has focused particularly on an *efficient* implementation of the application of the temporal resolution rule. This has involved the development of graph-theoretic techniques in order to identify the occasions when the temporal resolution rule should be applied.

1.2 Modal Logic

Modal logic is concerned with the truth of logical formulae over networks of worlds. Propositional (first order, etc.) modal logic may be regarded as propositional (first order, etc.) logic enriched with the modalities $\Box$, and $\Diamond$, meaning *necessarily* and *possibly* respectively. A formula in modal logic is a statement made in the context of a particular world, so the formula $\Box p$ is true if p is true in all worlds *accessible* from the current world, and $\Diamond p$ is true if p is true in some world *accessible* from the current world. Propositional modal logic, therefore, consists of a language, a set of worlds (or states) each being an assignment of truth values to propositions of the language, and a binary relation R defining the accessibility on the set of worlds. The many different forms of modal logic are dependent on the nature of the structural properties of the network. More detailed coverage of modal logics may be found in [HC68, HC84], for example.

1.3 Temporal Logic

A temporal logic can be regarded as a modal logic where the accessibility relation between worlds is interpreted in a temporal sense. Thus the meaning of the "$\Box$" operator becomes *always in the future*, the "$\Diamond$" operator *sometime in the future*, and the relation R becomes an ordering on time. Different temporal logics may be constructed by imposing restrictions on the accessibility relation R between worlds, and may be, for example, continuous or discrete, linear or branching. The two most common are temporal logics based on linear time, and those based on branching time, both of which assume that time is *backward linear*. If W denotes the set of possible worlds, and R is the relation on W then the backward linearity of a model is represented by the following constraint on all $w, w_1, w_2 \in W$:

$$\textbf{if } (w_1, w) \in R \text{ and } (w_2, w) \in R \textbf{ then } \text{either } (w_1, w_2) \in R \text{ or } (w_2, w_1) \in R.$$

The operators $\blacksquare$ and $\blacklozenge$ meaning *always in the past* and *sometime in the past* respectively are analogous to the two future-time operators and allow reasoning backwards into the past. Temporal systems may be further enriched with operators such as $\bigcirc$ (next), $\bullet$ (last), $\mathcal{U}$ (until), $\mathcal{S}$ (since), $\mathcal{W}$ (unless), and $\mathcal{Z}$ (weak since) for example.

1.3.1 Discrete Time

Discreteness is assumed in both the linear time and branching time logics described below. A second relation, N, is assumed, known as the immediate accessibility relation, where N is taken as an irreflexive relation satisfying the discreteness property on all $w_1, w_2 \in W$:

$$\textbf{if } (w_1, w_2) \in N \textbf{ then } \text{there exists no } w \text{ such that } (w_1, w) \in N \text{ and } (w, w_2) \in N.$$

The relation R is the reflexive, transitive, closure of N. If $(w_1, w_2) \in N$ then w_2 is thought of as *the next moment in time* to w_1.

1.3.2 Linear Time

Models of discrete linear-time temporal logics may be viewed as the set of worlds where each world, w_i, is indexed by one of the integers, i.e. $i \in \mathbf{Z}$, creating a series of worlds, (or states) in a discrete linear order

$$\ldots, w_{-(k+1)}, w_{-k}, \ldots, w_{-1}, w_0, w_1, \ldots, w_l, w_{l+1}, \ldots .$$

For a particular world, w_i, worlds w_n for $n > i$ are said to be in the future of w_i and worlds w_n for $n < i$ are said to be in the past of w_i. Linear time is forwards as well as backwards linear, where forwards linearity is represented by the following constraints on all $w, w_1, w_2 \in W$:

if $(w, w_1) \in R$ and $(w, w_2) \in R$ **then** either $(w_1, w_2) \in R$ or $(w_2, w_1) \in R$.

Different models of time may be obtained by restricting R further to make these sequences finite or infinite, for example if R is restricted so that

for some w there is no w_1 such that $(w_1, w) \in R$

then the model of time will have a finite past. Similarly,

for all w there is a w_1 such that $(w, w_1) \in R$

gives a model of time with infinite future.

Discrete, linear-time temporal logic with finite past and infinite future is the model of time used in the rest of the dissertation, and can be represented by the series of worlds, w_i, where each world is indexed by one of the natural numbers, i.e. $i \in \mathbf{N}$

$$w_0, w_1, w_2, w_3 \ldots$$

where w_0 marks the initial world, giving a model of time with a finite past.

1.3.3 Branching Time

Discrete branching time temporal logics have a tree-like structure which will branch into the "future", if the backward linearity constraint is assumed. A particular world has a unique past, but may have many possible futures, i.e. the forward linearity constraint imposed upon R above has been dropped. Program verification and specification techniques have been developed using both approaches, and discussion has taken place comparing the two, for example see [EH86]. We will not join this discussion, as we only intend to show how a temporal resolution method for discrete, linear, temporal logic can be implemented.

1.4 Resolution

1.4.1 Classical Resolution

Classical resolution as applied to propositional and first order logic requires formulae to be in a particular form, Conjunctive Normal Form, before resolution rules may be applied. A formula in Conjunctive Normal Form may be represented as

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

where each C_i is in Disjunctive Normal Form and is known as a *clause*. A formula in Disjunctive Normal Form may be represented as

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

where each D_i is in Conjunctive Normal Form. Pairs of clauses are resolved using the classical (propositional) resolution rule

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

where $A \vee B$ is known as the *resolvent*, $A \vee p$ and $B \vee \neg p$ are called the *parents of the resolvent* and p and $\neg p$ are called the *complementary literals*. The resolution rule is applied until **false** has been derived or no further resolvents can be generated.

Although the basic resolution method is complete, i.e. when applied to an unsatisfiable set of clauses the procedure is guaranteed to produce **false**, it is not very 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. Unit resolution, for example requires that at least one of the parent clauses is a single literal (known as a *unit clause*), input resolution 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. Unit, and input resolution for example are not complete however linear resolution is.

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].

1.4.2 Applying Resolution to Temporal Logics

In some cases 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

$$\frac{\begin{array}{l} A \lor \Box p \\ B \lor \Box \neg p \end{array}}{A \lor B}$$

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 be resolved. For example, pairs of formulae including the $\Box$ and $\Diamond$ operators may be resolved using the following temporal resolution rule.

$$\frac{\begin{array}{l} A \lor \Box p \\ B \lor \Diamond \neg p \end{array}}{A \lor B}$$

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

Although it would appear sensible to be able to resolve clauses which have complementary literals enclosed in the $\Box$ and $\Diamond$ operators, further complications occur as the $\Box$-formulae may be hidden. Sets of formulae may together imply a $\Box$-formula, even though this may not be immediately obvious.

This dissertation identifies sets of clauses which together imply such a $\Box$-formula, and can then be used as part of a temporal resolution step.

1.5 Graph Theoretic Techniques

Graphs have proved to be a useful tool to represent, model, and solve several classes of problems arising in Computer Science. Informally, a graph may be thought of as a finite set of points some of which are connected by lines (an *undirected* graph), or a finite set of points some of which are connected by arrows (a *directed* graph).

A graph may be defined as a pair $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of vertices (or nodes) and $E = \{E_1, E_2, \ldots, E_m\}$ is a set of edges, with $E_i = \{v_j, v_k\}$, for some j and k such that $v_j \in V$, and $v_k \in V$. If the edges are ordered pairs (v_j, v_k), the graph is said to be *directed*, whereas if the edges are unordered pairs of distinct vertices the graph is said to be *undirected*.

Hypergraphs, a generalisation of graphs, sometimes called "labelled graphs" or "And-Or graphs", may be defined in a similar manner to standard graphs except the only constraint on edges (called hyperedges in the case of hypergraphs), is that $E_i \subseteq V$ for $i = 1, \ldots, m$. When $|E_i| = 2$ for $i = 1, \ldots, m$ the hypergraph is a standard graph.

Graphical representations and graph-search strategies have been utilised particularly in the field of Artificial Intelligence where problems typically involve finding a path from a node representing the initial state of the database, to one representing a database that satisfies some termination condition. Different search strategies have been developed and applied to problems. As a semantic model for temporal logic formulae can be represented as a graph, graphs will be used to identify the rules that together imply the $\Box$-formulae required for the temporal resolution rule. Definitions of standard graphs, and basic search strategies may be found in [AHU74, RK91]. Descriptions of terminology relating to hypergraphs, their definitions, algorithms upon them, and their applications many be found in [GLNP90, Nil82].

1.6 Description of the Dissertation

Chapter 2 introduces the logic used, Propositional Temporal Logic, and gives an overview of the Temporal Resolution method.
Chapter 3 explains the graph based approach to the implementation of the temporal resolution rule.
Chapter 4 describes the construction of the whole theorem prover, incorporating the graph-based approach.
Chapter 5 discusses related work.
Chapter 6 presents conclusions and considers possible areas for further work.

Chapter 2

Overview of the Resolution Procedure

In this chapter the logic used in the rest of the dissertation is described and the temporal resolution method that has been used to construct the theorem prover presented in Chapters 3 and 4 is explained. This chapter is a summary of the work presented in [Fis91, Fis92b, Fis93].

2.1 Propositional Temporal Logic

The logic used in this dissertation is Propositional Temporal Logic (PTL), which is based on a linear, discrete model of time with finite past and infinite future. This logic may be viewed as a classical propositional logic augmented with both future-time and past-time temporal operators. Future-time temporal operators include

$\Diamond$	sometime in the future
$\Box$	always in the future
$\bigcirc$	in the next moment in time
$\mathcal{U}$	until
$\mathcal{W}$	unless (weak until)

while past-time operators include

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

2.1.1 Syntax

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

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

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

- Past-time temporal operators

unary operators :	$\newmoon$, $\odot$, $\blacklozenge$, $\blacksquare$
binary operators :	$\mathcal{S}$, $\mathcal{Z}$

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

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

$\neg A$	$A \vee B$	$A \wedge B$	$A \Rightarrow B$	(A)	
$\Diamond A$	$\Box A$	$A\,\mathcal{U}\,B$	$A\,\mathcal{W}\,B$	$\bigcirc A$	
$\blacklozenge A$	$\blacksquare A$	$A\,\mathcal{S}\,B$	$A\,\mathcal{Z}\,B$	$\odot A$	$\newmoon A$

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 boolean 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 not in the context of any temporal operator 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, $\odot A$, $\newmoon A$, $\blacklozenge A$, $\blacksquare A$, $A\,\mathcal{S}\,B$, 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 based on a discrete, linear model of time with finite past and infinite future. 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 *ith* moment in time. As formulae in PTL are interpreted at a particular state, the truth of a formula f is denoted by

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

where σ is the model and i is the state index at which the temporal statement is to be interpreted. For any formula f, model σ and state index i, then either $(\sigma, i) \models f$ or $(\sigma, i) \not\models f$. 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 $\models f$.*

The semantics of a proposition is defined by the valuation given to it in the model

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

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 operators are defined as follows

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

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

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

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

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

*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.

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

$(\sigma, i) \models A \mathcal{S} B$	iff	there exists k s.t. $0 \leq k < i$ and $(\sigma, k) \models B$ and for all j s.t. $k < j < i$ then $(\sigma, j) \models A$.
$(\sigma, i) \models A \mathcal{Z} B$	iff	$(\sigma, j) \models A \mathcal{S} B$ or $(\sigma, i) \models \blacksquare A$.

For convenience there are two last-time operators "$\bullet$" and "$\odot$", but only one next operator "$\bigcirc$" as the model of time being used has finite past but infinite future. The use of two last time operators 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 operators are interpreted in a strict sense, that is the current index is not included whereas the future-time operators are non-strict and include the current index.

Equivalences

The two last time operators are related by the following equivalence

$$\models \odot \neg A \Leftrightarrow \neg \bullet A.$$

The $\Diamond$, $\Box$ and $\bigcirc$ operators may be derived from $\mathcal{W}$ and $\mathcal{U}$ as follows

$$\begin{array}{lcl} \models \Diamond A & \Leftrightarrow & \mathbf{true}\,\mathcal{U}\,A \\ \models \Box A & \Leftrightarrow & A\,\mathcal{W}\,\mathbf{false} \\ \models \bigcirc A & \Leftrightarrow & \mathbf{false}\,\mathcal{U}\,A \end{array}$$

and their past-time counterparts have the following equivalences

$$\begin{array}{lcl} \models \blacklozenge A & \Leftrightarrow & \mathbf{true}\,\mathcal{S}\,A \\ \models \blacksquare A & \Leftrightarrow & A\,\mathcal{Z}\,\mathbf{false} \\ \models \odot A & \Leftrightarrow & \mathbf{false}\,\mathcal{S}\,A. \end{array}$$

The $\Box$ and $\Diamond$ operators, and their past-time counterparts are related by the following equivalences

$$\begin{array}{lcl} \models \neg \Diamond \neg A & \Leftrightarrow & \Box A \\ \models \neg \blacklozenge \neg A & \Leftrightarrow & \blacksquare A. \end{array}$$

2.2 The Temporal Resolution Method

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

- translation to a normal form;
- non-temporal resolution;

- temporal resolution.

The original formula, once negated, is translated into a normal form to which resolution, both non-temporal and temporal, may be applied. Non-temporal resolution consists of the standard classical style resolution representing constraints at a particular moment in time. The 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 the original formula was unsatisfiable, or when no further resolution rules can be applied, showing the original formula is satisfiable.

2.2.1 Translation to a normal form

As with classical clausal resolution, translation to a normal form is carried out, before any resolution takes place. The normal form used is called Separated Normal Form (SNF) and was introduced in [Fis91] and has been extended in [FN92] and [Fis92a]. It has been shown [Gab87] that any arbitrary formula in PTL can be *separated* into its past, present and future-time components. It was initially from this 'separation result' that SNF was derived. Formulae in SNF are of the form

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

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

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

where k_a, l_b, and l are literals. Once in this form, both temporal and non-temporal resolution rules may be applied. However, it may be necessary to rewrite the resolvents produced into SNF. The translation of PTL formulae into SNF will not be described here although examples of some of the rewrite rules used in the SNF translation are given in Appendix A. For full details of SNF and its derivation see [FN92, Fis91].

It is assumed that each of the $P_i \Rightarrow F_i$ rules described here and in the rest of the dissertation are taken in the context of an external $\Box$-operator, although it will not be stated explicitly.

2.2.2 Non-Temporal Resolution

The non-temporal resolution process is basically classical resolution applied to $\square$-rules, which represent constraints applying to every moment in time. Standard simplification and subsumption rules may also be applied (see below).

Pairs of initial $\square$-rules, or global $\square$-rules, may be resolved using the following (non-temporal resolution) inference rules.

$$
\begin{array}{rcl}
\bullet\mathbf{false} & \Rightarrow & A \lor r \\
\bullet\mathbf{false} & \Rightarrow & B \lor \neg r \\
\hline
\bullet\mathbf{false} & \Rightarrow & A \lor B
\end{array}
\qquad\qquad
\begin{array}{rcl}
\odot P & \Rightarrow & A \lor r \\
\odot Q & \Rightarrow & B \lor \neg r \\
\hline
\odot(P \land Q) & \Rightarrow & A \lor B
\end{array}
$$

These are both instances of the following more general 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 $\mathcal{L}$ may be either of the last-time operators.

Simplification

The following rules can be removed during simplification as they represent valid subformulae.

$$\odot\mathbf{false} \;\Rightarrow\; A$$

$$P \;\Rightarrow\; \mathbf{true}$$

The first rule is valid as $\odot\mathbf{false}$ 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**

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

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

Subsumption

Subsumption also forms a part of the non-temporal 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

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

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

$$\begin{array}{rcl} \newcommand{\lastweak}{\mathbin{\bigcirc\!\!\!\!\cdot}} \bullet\,\mathbf{true} & \Rightarrow & \mathbf{false} \\ \bullet\,\mathbf{false} & \Rightarrow & \mathbf{false} \\ \odot\,\mathbf{true} & \Rightarrow & \mathbf{false}. \end{array}$$

2.2.3 Temporal Resolution

We would like to be able to apply a temporal resolution rule of the form

$$\frac{\begin{array}{c} P \vee \square p \\ Q \vee \lozenge \neg p \end{array}}{P \vee Q}$$

resolving together formulae with the $\square$ and $\lozenge$ operators. As $\lozenge\neg p$ means that $\neg p$ must be satisfied at some point in the future, $\lozenge\neg p$ is said to have an eventuality commitment $\neg p$, and so $\neg p$ is known as an *eventuality*. As the translation to SNF restricts the rules to be of a certain form, resolution will be between a $\lozenge$-rule and a $\square$-formula. This $\square$-formula may be "hidden", due to the interaction between $\bigcirc$ 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 $\lozenge\neg l$. The detection of such $\square$-formulae is crucial to the resolution process. From a set of rules in SNF, for every rule of the form

$$\mathcal{L}Q \Rightarrow \lozenge\neg p$$

where $\mathcal{L}$ may be either of the last-time operators, $\bullet$ or $\odot$, temporal resolution may be applied between this $\lozenge$-rule and a set of global $\square$-rules, which taken together force p always to be satisfied. Here if Q is satisfied then the rule is satisfiable unless the set of global $\square$-rules forces p to always be **true**. In this case, we must ensure that there is no situation where both Q and one of these rules are satisfied at the same moment. Similarly, once Q has been satisfied we must also ensure that none of the global $\square$-rules are satisfied *unless* $\neg p$ 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

$$P \Rightarrow \square p.$$

Although the $\lozenge$-rule may be either initial or global, i.e. the $\mathcal{L}Q$ above may be either $\bullet$**false**, or $\odot A$ where A is a conjunction of literals, the P may only be of the form $\odot A$. We must resolve

$$\bigcirc\!\!\!\bullet A \Rightarrow \Box p \qquad \text{with} \qquad \bigcirc\!\!\!\bullet Q \Rightarrow \Diamond\neg p$$

or

$$\bigcirc\!\!\!\bullet A \Rightarrow \Box p \qquad \text{with} \qquad \bullet\,\mathbf{false} \Rightarrow \Diamond\neg p.$$

So, in general

$$\bigcirc\!\!\!\bullet A \Rightarrow \Box p \qquad \text{with} \qquad \mathcal{L}Q \Rightarrow \Diamond\neg p$$

and the general resolution rule becomes

$$\left\{ \begin{array}{rcl} \bigcirc\!\!\!\bullet A & \Rightarrow & \Box p \\ \mathcal{L}Q & \Rightarrow & \Diamond\neg p \end{array} \right\} \longrightarrow \left\{ \begin{array}{rcl} \bullet\,\mathbf{true} & \Rightarrow & \neg A \vee \neg Q \\ \mathcal{L}Q & \Rightarrow & (\neg A)\,\mathcal{W}(\neg p) \end{array} \right\}$$

where the first resolvent shows that both A and Q cannot be satisfied together, and the second that once Q has occurred then $\neg p$ must occur (i.e. the eventuality must be satisfied) before A can be satisfied.

The full temporal resolution rule is given by the following

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

where the following side conditions must hold :-

$$\left\{ \begin{array}{rcl} \text{for all } 0 \leq i \leq n & \vdash & F_i \Rightarrow p \\ \text{and} & \vdash & F_i \Rightarrow \bigvee_{j=0}^{n} A_j \\ \text{and} & \vdash & (A_i \wedge p) \Rightarrow \bigvee_{k=0}^{n} F_k \end{array} \right\}$$

The side conditions in the rule above ensure that the rules of the form $\bigcirc\!\!\!\bullet A_i \Rightarrow F_i$ together imply $\Box p$. Thus if any of the A_i are satisfied then p will be *always* be satisfied, i.e.

$$\bigcirc\!\!\!\bullet \bigvee_{k=0}^{n} A_k \Rightarrow \Box p.$$

If a set of rules together imply a $\Box$-formula, for example $\Box p$, this is known as a *loop* in p . Examining the side conditions in more detail

1. $\vdash F_i \Rightarrow p$: each $\Box$-rule ensures that p is true.

2. $\vdash F_i \Rightarrow \bigvee_{j=0}^{n} A_j$: the right hand side of each rule will ensure that at least one of the left hand sides of the rules will be satisfied.

3. $\vdash (A_i \wedge p) \Rightarrow \bigvee_{k=0}^{n} F_k$: each of the left hand sides of the rules guarantees the satisfaction of at least one right hand side, ensuring there are no irrelevant rules. For example if we have the rules

$$\begin{array}{llcl} 1. & \bigcirc a & \Rightarrow & b \wedge p \\ 2. & \bigcirc b & \Rightarrow & a \wedge p \end{array}$$

we have a loop in p which satisfies all three side conditions. If we then consider the rule

$$\begin{array}{llcl} 3. & \bigcirc x & \Rightarrow & a \wedge p \end{array}$$

which does not form part of the loop in p, this still satisfies conditions 1 and 2 but does not satisfy condition 3 as

$$(x \wedge p) \Rightarrow (a \wedge p) \vee (b \wedge p)$$

does not hold.

The first of the new resolvents ensures that none of the rules which form a loop in p 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 p, i.e. A_i for $i = 0, 1, \ldots, n$ can be satisfied at the same time as Q. The second new resolvent ensures that once Q has been satisfied, meaning that the eventuality $\Diamond\neg p$ must be satisfied, none of the conditions for entering a loop is allowed to occur, i.e. A_i for $i = 0, 1, \ldots, n$ must not be satisfied until the eventuality $(\neg p)$ has been satisfied.

The new resolvents must be translated into SNF and another cycle of non-temporal and temporal resolution may be carried out. The whole procedure will terminate on derivation of **false** by generation of one of the previously mentioned unsatisfiable termination formulae, or when no new resolvents may be generated from either of the resolution steps. The soundness and completeness of the set of resolution rules has been shown in [Fis93].

Chapter 3

Implementing Temporal Resolution

In this chapter the implementation of the resolution rule described in Chapter 2 is described. To apply the temporal resolution rule, an appropriate implicit $\Box$-formula must be detected for resolution with one of the $\Diamond$-rules. As sets of rules which together imply such a $\Box$-formula may not be immediately apparent from inspection of the rule-set, graph-theoretic techniques have been used to identify them.

Sections 3.1 and 3.2 give an overview to the approach taken, Section 3.1 describing different ways the rules could be represented graphically, and Section 3.2 explaining some methods for searching the graphs. The full details of the algorithm are given in Section 3.3. Examples, analysis of the algorithm and potential enhancements are presented in Sections 3.4, 3.5 and 3.6 respectively.

3.1 Graphical Representation of Rules

We represent the global $\Box$-rules graphically as it is from these we hope to extract sets of rules which together imply a $\Box$-formula. A node n_i may be a single literal or a conjunction of literals and will be denoted using round brackets, for example, $n_j = (l_1)$, where n_j is made up of just a single literal, or $n_j = (l_1, l_2, \ldots, l_k)$ where n_j is made up of a conjunction of literals. A rule of the form

$$\bigcirc \bigwedge_{a=1}^{g} k_a \quad \Rightarrow \quad l$$

where each k_a and l, are literals, can be represented graphically as two nodes (or vertices), N, which are

$$N = \{(l), (k_1, \ldots, k_g)\}$$

and one edge E where

$$E = \{\{(k_1, \ldots, k_g), (l)\}\}.$$

This rule is shown graphically in Figure 3.1.

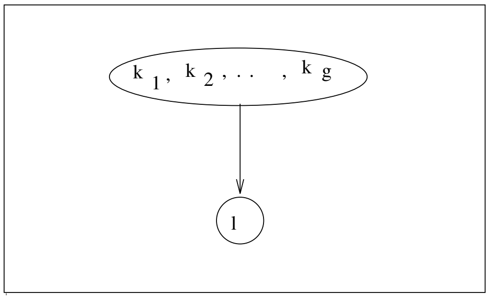

Figure 3.1: Representing a Rule Graphically

A rule of the form

$$\bigcirc \bigwedge_{a=1}^{g} k_a \;\Rightarrow\; \bigvee_{b=1}^{r} l_b$$

where each k_a and l_b are literals, and $b > 1$, which will be called a *disjunctive rule*, can be represented graphically as $r+1$ nodes (or vertices), N, which are

$$N = \{(l_1), (l_2), \ldots, (l_r), (k_1, \ldots, k_g)\}$$

and r edges E which are

$$E = \{\{(k_1, \ldots, k_g), (l_1)\}, \{(k_1, \ldots, k_g), (l_2)\}, \ldots, \{(k_1, \ldots, k_g), (l_r)\}\}.$$

This rule is shown graphically in Figure 3.2. In the rest of the dissertation, edges in graphs

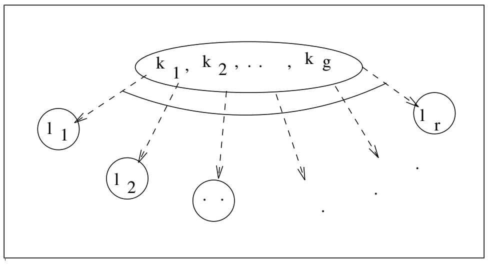

Figure 3.2: Representing a Disjunctive Rule Graphically

which have been formed from disjunctive rules will be shown using broken lines, whereas other edges will use solid lines. The arrows on the edges are as would be expected and indicate the time progression inherent in rules in SNF, from past to future, moving from the left to the right of a rule.

Sets of $\Box$-rules may be represented directly as either a hypergraph or by combining rules in SNF together as an ordinary graph. To represent the global $\Box$-rules as an ordinary graph, a variant on SNF, called merged-SNF (SNF_m), mentioned in [Fis92b], may be used. SNF_m may be characterised by the following rule

$$\frac{\begin{array}{rcl}\bigcirc A & \Rightarrow & F \\ \bigcirc B & \Rightarrow & G\end{array}}{\bigcirc(A \wedge B) \;\Rightarrow\; F \wedge G}$$

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

Example

Given the following rule-set

$$\begin{array}{rcl}\bigcirc x & \Rightarrow & y \vee z \\ \bigcirc x & \Rightarrow & w\end{array}$$

these rules can be represented directly by the hypergraph in Figure 3.3. To represent the rule-set

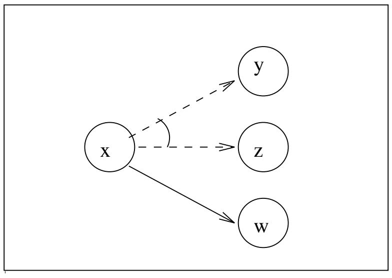

Figure 3.3: The Representation of a Set of Rules using a Hypergraph

as an ordinary graph the two rules are combined using SNF_m to give

$$\bigcirc x \Rightarrow (y \vee z) \wedge w$$

and the right hand side of the rule is rewritten into DNF to give

$$\bigcirc x \Rightarrow (y \wedge w) \vee (z \wedge w).$$

This is illustrated in Figure 3.4.

Representing the SNF rules directly as a hypergraph means that no further processing needs to be done to the rules themselves but building and searching the hypergraph will be more complicated. If the rules are first combined into SNF_m and represented as an ordinary graph, constructing and searching the ordinary graph will be less costly than for the hypergraph but the initial translation of the rules from SNF into SNF_m will create additional rules (for n rules in SNF up to $2^n - 1$ rules may be generated in SNF_m), making the translation into SNF_m the most costly part of the procedure.

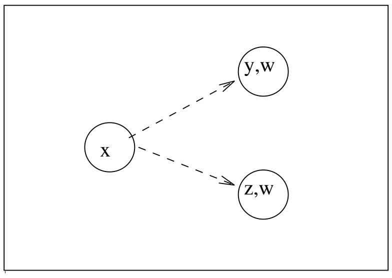

Figure 3.4: The Representation of a Set of Rules using an Ordinary Graph

The approach that has been adopted in this dissertation does not construct the complete graph from the full rule-set. Assuming that we want to resolve with $\mathcal{L}Q \Rightarrow \Diamond\neg p$, only the rules, or combinations of rules which will generate p are used to to build an ordinary graph.

This approach avoids both the expense of translating all the rules into SNF_m required in order to be able to represent the rules as an ordinary graph, and the complexity of building and searching a hypergraph if all the rules are represented directly. The algorithm developed to achieve this is described in detail in Sections 3.2 and 3.3 and examples of graphs that contain loops and the rules they have been built from are given in Section 3.4.

3.2 The Graph Search Algorithm

As mentioned previously, the motivation behind this method is to avoid the full translation of all the rules to SNF_m by using an algorithm which searches for the rules that can possibly generate the negation of the eventuality commitment of a $\Diamond$-rule, combining them together using the SNF_m translation where necessary, and using these rules to build a graph. Loops are detected during the process of graph construction. This avoids having to create the whole graph and subsequently search it.

Given that our aim is to resolve with $\mathcal{L}Q \Rightarrow \Diamond\neg p$, rules which imply p, i.e. have p as consequent, are sought and the literal or conjunction of literals from the antecedent of the rules are used as *initial nodes* in the graph. If a loop can be found from a graph of these nodes alone, or a graph built up from nodes generated from these rules combined with other rules, p will be generated at every node as required.

Having detected the initial nodes, a depth-first search is attempted on each one in turn to find a successor node which will still generate p. This process is continued using the new successor as the current node until a node is reached which has been searched through before and a loop has been detected (or no more successors can be found).

3.2.1 Backward Search Methods

Initial implementations attempted to use complete backward search to generate successors to the initial node. By backward search it is meant that the current node is matched with the consequent of a rule in SNF, or, if no such matching rule exists, a combination of rules in SNF_m. The antecedent of this rule would be then used as the next node.

Example

Given the following rules extracted from a rule-set

$$\begin{array}{lrcl} 1. & \bigcirc b & \Rightarrow & a \\ 2. & \bigcirc a & \Rightarrow & p \\ 3. & \bigcirc b & \Rightarrow & p \\ 4. & \bullet\,\mathbf{false} & \Rightarrow & \Diamond\neg p \end{array}$$

The initial nodes are (a), and (b), as $\neg p$ is the eventuality with which we wish to resolve (rule 4), and from rules 2 and 3 both a and b imply p in the next moment in time. Taking (a) as the current node, this is matched with the right hand side of rule 1. The propositions on the left hand side of rule 1 will still generate a p in the next moment in time, so backwards search continues with (b) as the new node. There are no rules with (b) on the right hand side so this search path fails. Backwards search would be attempted from the remaining initial nodes (in this example (b)) until a loop is detected or there are no remaining initial nodes from which to search. As previously mentioned, backwards search from (b) fails, so the procedure terminates finding no loops.

Backward search methods are advantageous as they make the search more directed, starting with the nodes we know will generate p as required. The problems associated with how the search should proceed having used a disjunctive rule invoked thoughts about alternative strategies, leading to the possibility of building the graph using forward search methods.

3.2.2 Forward Search Methods

Forward search methods attempt to match the current node with the left hand side, or antecedent of a rule in SNF, or if no such rule exists a combination of rules in SNF_m. The consequent of this rule is then used as the next node. Using forward search the graph will be constructed in the same direction as the directed arrows between nodes. Rules must first be combined into SNF_m, so all combinations of nodes which generate the negation of the eventuality can be used as initial nodes, which will be the starting nodes for the search.

Example

Using the partial rule-set given in Section 3.2.1, before a complete forward search can be carried out, rules must be combined using SNF_m to give the additional rules

$$
\begin{array}{llcll}
5. & \bigcirc(a \wedge b) & \Rightarrow & (a \wedge p) & [1 + 2 \text{ (or 1, 2, 3) merged}] \\
6. & \bigcirc b & \Rightarrow & (a \wedge p) & [1 + 3 \text{ merged}] \\
7. & \bigcirc(a \wedge b) & \Rightarrow & p & [2 + 3 \text{ merged}]
\end{array}
$$

In this case the initial nodes are (a), (b), and (a, b). Forward search from (a) would match a with the left hand side of rule 2, generating (p) as the next node. There are no rules which have p on the left hand side, so this search path fails and the current node is set to a new initial node (b). This is matched with the left hand side of rule 1 giving (a) as the new node, and forward search continues with (a) as the new current node. The forward search from (a) will fail as previously. A subsequent forward search from (b) will match b with the left hand side of rule 3 making (p) the new current node. This search path again fails and the new current node is set to the final initial node (a, b). After forward search fails having matched (a, b) with the left hand side of rule 5, and then fails having matched (a, b) with the left hand side of rule 7, the procedure terminates finding no loops.

Complete forward search methods need every combination of rules which may generate p to be computed as initial nodes, requiring the full translation to SNF_m, (which we were trying to avoid) thus making the method expensive. Its attraction stems from the fact that if a particular node is part of a loop then it must have a forward successor leading from it.

3.2.3 A Combined Approach

The search algorithm finally implemented uses a combination of both backwards and forwards search algorithms. At first the algorithm uses backwards search to construct a path from the initial nodes. If a disjunctive rule is used during the backwards search, the search continues in a backwards search manner as described, until the detection of a loop or failure. If a loop is detected which includes an edge which was part of a disjunctive rule the remaining disjuncts must also be solved and form part of the loop. The search to solve the remaining disjuncts is carried out using a forwards search method. The use of a search method combining both techniques preserves the directedness of the backwards search, yet applies forward search to disjuncts which must form part of the loop if the path is to form a loop.

If the algorithm is used on a rule-set which has a loop, but contains no disjunctive rules, backwards search alone will be used in detection of this loop. If a disjunctive rule has been used as part of the path, the remaining disjuncts are then solved using a forward search method, as if there are several disjuncts still to be solved then loops must be found through all these nodes if the disjunctive rule is to form part of the loop, i.e. the problem of taking every combination initially, mentioned above, if forward search was carried out solely does not occur. A second advantage of using forward search on disjuncts is that it naturally avoids searching down the same path. For example, if b was the current node which had been matched with the b on the right hand side of the rule

$$\bigcirc a \Rightarrow b \lor c$$

during backwards search, then a would be the new current node. If subsequently a loop had been found which involved this disjunctive link, the disjunct c would still have to be solved. A forward search from c ensures that a different rule must be used, whereas a backwards search may attempt to apply the same rule unless the rule was somehow marked to prevent this. A third advantage of using forward search to solve disjuncts is that having found a loop for part of a disjunct, there may be no further backwards search possible, even though there are loops on the other disjuncts.

If a disjunct is encountered while searching forwards all the disjuncts must be solved in a forwards search method.

3.3 The Detailed Algorithm

3.3.1 Definitions of Terms

In the description of the algorithm and its usage the following terms will be used. Let G be the directed graph $G = (N, E)$ where $N = \{n_1, n_2, \ldots, n_m\}$ is the set of nodes and $E = \{E_1, E_2, \ldots, E_r\}$ is the set of edges. As G is directed, each edge E_i for $i = 1, \ldots, r$ is an ordered pair (n_j, n_k), for $n_j, n_k \in N$. A *path* P through a graph is a sequence of nodes, $[n_1, n_2, \ldots, n_k]$, with each $n_j \in N$ for $j = 1, \ldots, k$, and each adjacent pair of nodes nj, n_{j+1} in this sequence for $j = 1, \ldots, k - 1$ such that $(n_j, n_{j+1}) \in E$. A *sub-path* P_s, of path P above, is a sequence of nodes $[n_p, \ldots, n_q]$ for $1 \leq p < q$ and $p < q \leq k$ which may be used to refer to a path which is a particular part of the main *path*, currently under consideration.

A directed graph is *strongly connected* if for each pair of vertices $n_i, n_j \in N$ there is a path from n_i to n_j. A *self-contained loop* is a path whose nodes form a *strongly connected* subgraph and where no disjunctive rules have been used. A *partial loop* is a path with at least one disjunctive edge (i.e. a disjunctive rule has been used to construct the graph) whose nodes form a *strongly connected* subgraph or lead to and then form a *strongly connected* subgraph. Thus a *loop* may be either a *self-contained loop* or made up from several *partial loops*.

3.3.2 Graph Construction and Loop Detection

Loops are detected as the graph is constructed which means that the complete graph may not need to be built. The algorithm used (given that we want to resolve with $\mathcal{L}Q \Rightarrow \Diamond\neg p$) is given below. The forwards and backwards search algorithms mentioned in points 4. and 6. are described in Section 3.3.3.

1. Search for all the rules of the form $\bigcirc R_i \Rightarrow p$
2. Use the literal (or conjunction of literals) R_i as the initial nodes in the graph.
3. Set the current node, n_1, equal to the next initial node, if one is available, and path equal to $[n_1]$, otherwise the algorithm terminates having found no new loops.

4. Perform a backwards search from n_1 until either
   (a) no loop has been detected from any of the successors to n_1 - repeat step 3;
   (b) a self-contained loop has been found - the algorithm terminates with this loop;
   (c) a partial loop has been found - continue processing with step 5.
5. Set n_1, the current node equal to a new disjunct if one is available, otherwise the algorithm terminates with a loop.
6. Perform a forward search from n_1 until either
   (a) no loop has been detected from any of the successors to n_1, 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 - the algorithm terminates with a loop;
   (c) a partial loop has been found - continue processing with step 5.

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

3.3.3 Backward and Forward Search Algorithms

A new node n_{i+1} may be generated using the backwards or forwards depth-first search algorithms, where applicable, from the current node n_i, where n_i represents a literal or conjunction of literals, and $n_i \Rightarrow p$, if there exists a global $\Box$-rule

$$\mathbf{\odot} \bigwedge_{a=1}^{g} k_a \quad \Rightarrow \quad \bigvee_{b=1}^{r} l_b$$

and the following conditions are satisfied.

1. The conditions for backwards search

$$\text{there exists a } b \text{ such that } 1 \leq b \leq r,\ n_i \Leftrightarrow l_b,$$

$$\text{and for all } b,\ l_b \Rightarrow p,\ \text{ and } \bigwedge_{a=1}^{g} k_a \Rightarrow p$$

$$\text{then } n_{i+1} \text{ will be } (k_1, k_2, \ldots, k_g).$$

   The current node is equivalent to one of the disjuncts from the right hand side of the rule, the other disjuncts all imply p, the new node, n_{i+1}, implies p, and will be the conjunction of literals from the left hand side of the rule.

2. The conditions for forwards search

$$n_i \Leftrightarrow \bigwedge_{a=1}^{g} k_a \text{ and for all } b \text{ such that } 1 \leq b \leq r,\ l_b \Rightarrow p$$

$$\text{then } n_{i+1} \text{ will be } l_b \text{ for some } b \text{ such that } 1 \leq b \leq r.$$

The current node is equivalent to the literal or conjunction of literals on the left hand side of the rule, all the disjuncts on the right hand side of the rule imply p and the next node, n_{i+1}, will be one of the disjuncts.

Having generated a new successor processing continues as follows.

1. For backwards search

   (a) If $r = 1$, i.e. the rule used is not a disjunctive rule, the new node is added to the head of the path.

   (b) If $r > 1$, i.e. a disjunctive rule has been used, the path is structured to indicate this, the remaining disjuncts are stored in a list, called the *disjunctive list*, still to be solved, and the new node is added to the head of the path.

2. For forwards search

   (a) If $r = 1$, i.e. the rule used is not a disjunctive rule, the new node is appended to the end of the sub-path.

   (b) If $r > 1$, i.e. a disjunctive rule has been used, the sub-path is structured to indicate this, the remaining disjuncts are stored in the *disjunctive list*, still to be solved, and the new node is appended to the end of the sub-path.

Successors to the current node are generated until the termination conditions of the main algorithm are achieved.

The tests of the form $L \Rightarrow p$ and $n_i \Leftrightarrow L$ required for the backwards and forwards search algorithms, for L a literal or conjunction of literals, p a literal, and n_i a node, have been computed using set theory. Let

$$L = \bigwedge_{j=1}^{n} l_j \text{ and } n_i = \bigwedge_{k=1}^{m} r_k$$

$$\begin{array}{rrl} \text{then} & L \Rightarrow p & \text{can be computed by ensuring that} \\ & & \{p\} \subseteq \{l_1, l_2, \ldots, l_n\} \\ \text{and} & n_i \Leftrightarrow L & \text{can be computed by ensuring that} \\ & & \{r_1, r_2, \ldots, r_m\} \subseteq \{l_1, l_2, \ldots, l_n\} \text{ and} \\ & & \{l_1, l_2, \ldots, l_n\} \subseteq \{r_1, r_2, \ldots, r_m\} \end{array}$$

where l_j, r_k and p are literals.

3.3.4 Prefix to the Loop

During backwards search, a loop is detected when a node, n_j is reached which has already been visited. The loop (or partial loop) will be delimited by nodes n_j and a node equivalent to it. In the case where no disjunctive edges have been used the path will be

$$[n_j, n_{j-1}, \ldots, n_{k+1}, n_k, n_{k-1}, \ldots, n_2, n_1]$$

where $n_j \Leftrightarrow n_k$. The loop delimited by nodes n_j and n_k will be

$$[n_j, n_{j-1}, \ldots, n_{k+1}, n_k].$$

and the remaining part of the path is

$$[n_{k-1}, \ldots, n_2, n_1].$$

The remaining part of the path must be removed from the main path and is known as the *prefix to the loop*. Similarly the prefix to the loop must still be removed if a structured path has been built up, and will need to be removed during forward search after finding a self-contained loop. When removing the prefix to the loop the nodes which form the prefix are treated as if they have never been visited.

3.3.5 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 the rules which may generate an appropriate successor are extracted from the rule-set for combination purposes.

For a node n_i such that $n_i = (k_1, k_2, \ldots, k_g)$, and a set of rules in SNF, or SNF_m, a rule

$$\bigcirc Q \Rightarrow \bigvee_{b=1}^{m} C_b$$

would only be selected for combination purposes, if during backwards search

$$\bigwedge_{a=1}^{g} k_a \Rightarrow C_b \text{ for some } b \text{ such that } 1 \leq b \leq m$$

or during forward search

$$\bigwedge_{a=1}^{g} k_a \Rightarrow Q.$$

This is because, if during backward search the current node is $n_i = (k_1, k_2, \ldots, k_g)$, where each k_i is a literal, then for a rule in SNF_m of the form

$$\bigcirc P \Rightarrow \bigvee_{b=1}^{r} D_b$$

to be used to generate a successor then $D_b \Leftrightarrow n_i$. So before the SNF$_m$ transformation can be applied, rules which have a disjunct, C_b, made up from literals which are a subset of $k_1, k_2, \ldots, k_g$, are searched for, which when combined *may* match $k_1 \wedge k_2 \wedge \ldots \wedge k_g$ exactly.

During forward search we are matching the left hand side of the rules so we must make sure that the left hand side of rules to be combined is made up from literals which are a subset of $k_1, k_2, \ldots, k_g$, so when they are combined they *may* match $k_1 \wedge k_2 \wedge \ldots \wedge k_g$ exactly.

These tests are again computed using set theory.

Example

If the current node is (a, b, c) and the rule-set is as follows

$$\begin{array}{rlcl} 1. & \odot a & \Rightarrow & b \\ 2. & \odot b & \Rightarrow & c \\ 3. & \odot x & \Rightarrow & a \vee x \\ 4. & \odot c & \Rightarrow & x \end{array}$$

rules 1, 2 and 3 would be selected to combine using the SNF$_m$ transformation during backwards search, and rules 1, 2, and 4 would be selected during forwards search.

3.4 Examples

Explanations of how the algorithm is used to detect loops from several rule-sets, are given below. It will be assumed that the original formulae have already been translated into SNF.

Example 1: A Simple Loop

$$\begin{array}{rrcl} 1. & \bullet\,\mathbf{false} & \Rightarrow & \Diamond\neg p \\ 2. & \odot p & \Rightarrow & p \end{array}$$

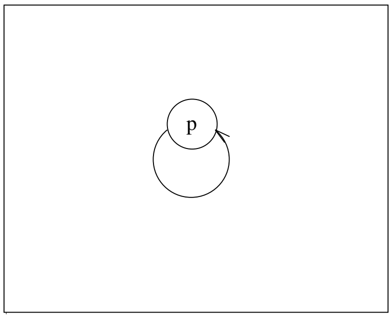

Figure 3.5: Example 1 – A Simple Loop

1. Rule 1 is the $\Diamond$-rule used as one of the resolvents.

2. The initial node is (p). The current node is set to (p), and the path is $[(p)]$.

3. Backwards search matches p with the right hand side of rule 2. The path is $[(p), (p)]$, and the new current node becomes (p). As this has already been visited the algorithm terminates with the path $[(p), (p)]$.

Example 2: A Loop From Pure Backwards Search

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

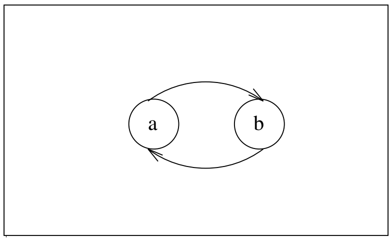

Figure 3.6: Example 2 – A Loop From Pure Backwards Search

1. Rule 5 contains the eventuality we wish to resolve with $(\neg l)$.

2. Initial nodes are (a) and (b), from rules 3 and 4. The current node is set to (a), and the path is $[(a)]$.

3. Backwards search matches (a) with the right hand side of rule 2. The new current node is (b), and the path is $[(b), (a)]$.

4. Backwards search matches (b) with the right hand side of rule 1. The new current node becomes (a), the new path becomes $[(a), (b), (a)]$, and the algorithm terminates as (a) has already been visited.

Example 3: A Disjunctive Loop with Prefix

$$
\begin{array}{rrcl}
1. & \bigcirc b & \Rightarrow & c \vee d \\
2. & \bigcirc c & \Rightarrow & b \\
3. & \bigcirc d & \Rightarrow & b \\
4. & \bigcirc b & \Rightarrow & a \\
5. & \bigcirc a & \Rightarrow & l \\
6. & \bigcirc b & \Rightarrow & l \\
7. & \bigcirc c & \Rightarrow & l \\
8. & \bigcirc d & \Rightarrow & l \\
9. & \bullet\,\mathbf{false} & \Rightarrow & \Diamond \neg l
\end{array}
$$

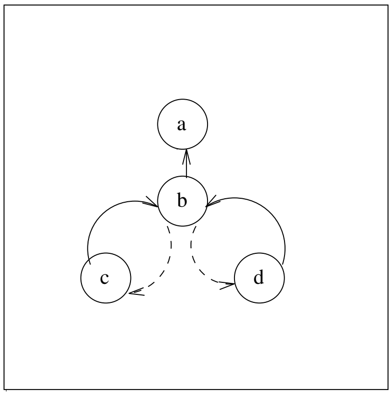

Figure 3.7: Example 3 – A Disjunctive Loop with Prefix

1. The $\Diamond$-rule is rule 9, and so we search for loops in l.
2. Initial nodes are (a), (b), (c), and (d), from rules 5, 6, 7, and 8. The current node is set to (a) and path to $[(a)]$.
3. Backwards search matches a with the right hand side of rule 4. The new current node is b, and the path is $[(b), (a)]$.
4. Backwards search matches (b) with the right hand side of rule 2. The new current node becomes (c), and new path $[(c), (b), (a)]$.
5. Backwards search matches (c) with the right hand side of the disjunctive rule 1. The new current node becomes (b), and (d) is stored on the disjunctive list, waiting to be solved. The path is $[(b)[(c), (b), (a)], [(d)]]$. As (b) has already been searched through we have found a partial loop, and need to solve the remaining disjuncts. The node (a) does not form part of this partial loop, so is the prefix to the loop, and is removed from the path. The path becomes $[(b)[(c), (b)], [(d)]]$.

6. The remaining disjunct d is solved using forward search. Forward search matches d with the left hand side of rule 3. The current node becomes (b), the sub-path for the disjunct becomes $[(d), (b)]$. As (b) has already been searched through before, we have found another partial loop, so the node (d) in the previous path is replaced by this sub-path to give a new path $[(b)[(c), (b)], [(d), (b)]]$.

7. There are no remaining disjuncts left to solve so the algorithm terminates.

If either of the other three initial nodes had been used to start the search, the algorithm would still have found the same loop.

Example 4: A Loop Using Combinations of Rules

$$
\begin{array}{lrcl}
1. & \bigcirc a & \Rightarrow & l \\
2. & \bigcirc b & \Rightarrow & l \\
3. & \bigcirc c & \Rightarrow & l \\
4. & \bigcirc(a \wedge b) & \Rightarrow & a \\
5. & \bigcirc(b \wedge c) & \Rightarrow & b \\
6. & \bigcirc(a \wedge c) & \Rightarrow & c \\
7. & \bullet\,\mathbf{false} & \Rightarrow & \Diamond\neg l
\end{array}
$$

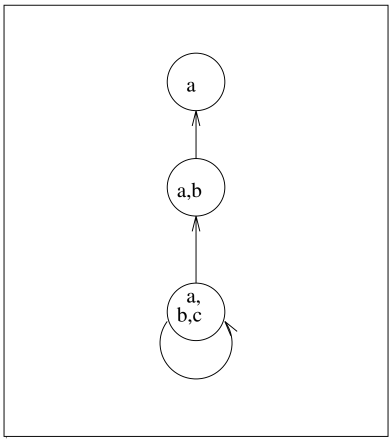

Figure 3.8: Example 4 – A Loop Using Combinations of Rules

1. The eventuality to be resolved with is $\neg l$, from rule 7.

2. Initial nodes are (a), (b), and (c), from rules 1, 2, and 3. The current node is set to be (a) and the path is $[(a)]$.

3. Backwards search from a matches a with the right hand side of rule 4. The new current node is set to (a, b), and the path is $[(a, b), (a)]$.

4. Backwards search is carried out attempting to match the right hand side of a rule with node (a, b). As no rule matches, rules 4 and 5 are extracted as they will give us an a, or a b, on the right hand side, and are combined using SNF$_m$ to give a new rule

$$\bigcirc(a \wedge b \wedge c) \Rightarrow a \wedge b$$

The right hand side of this new rule now matches the current node, so the new current node is set to (a, b, c) and the path becomes

$$[(a, b, c), (a, b), (a)]$$

5. Again no rule in SNF matches the current node (a, b, c). Rules 3, 4, and 5 are extracted as they will generate the required nodes on the right hand side to match with the current node, and combined using SNF$_m$ to give a new rule

$$\bigcirc(a \wedge b \wedge c) \Rightarrow (a \wedge b \wedge c)$$

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 (a, b, c), and the path becomes

$$[(a, b, c), (a, b, c), (a, b), (a)].$$

As we have already searched through node (a, b, c), we have found a loop. The prefix to the loop is removed, leaving the path as $[(a, b, c), (a, b, c)]$, and the algorithm terminates as there are no disjuncts to process.

If either of the other two initial nodes (b) and (c) had been had been used to start the search from, the algorithm would still have found the same loop. The graph of the search paths from all three initial nodes is shown in Figure 3.9.

3.5 Formal Analysis of the Algorithm

3.5.1 Soundness

To show the algorithm is sound we must show that if a loop is found using the algorithm then the set of rules which represents this, R, really does imply a $\square$-formula.

Proof

Assume that a loop has been found as shown in Figure 3.10 and may be represented by the path through nodes n_i

$$[n_1, n_2, \ldots, n_m, n_1].$$

Let the rules, R, in SNF$_m$ which represent this path be

$$\left\{ \begin{array}{ccc} \bigcirc A_0 & \Rightarrow & B_0 \\ \cdots & & \cdots \\ \bigcirc A_n & \Rightarrow & B_n \end{array} \right\}$$

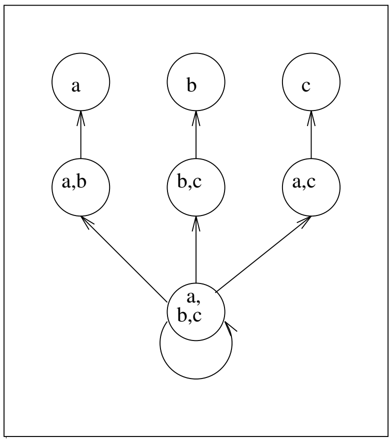

Figure 3.9: Example 4 – The Complete Graph

and assume we wish to resolve with

$$\mathcal{L}P \Rightarrow \Diamond\neg p.$$

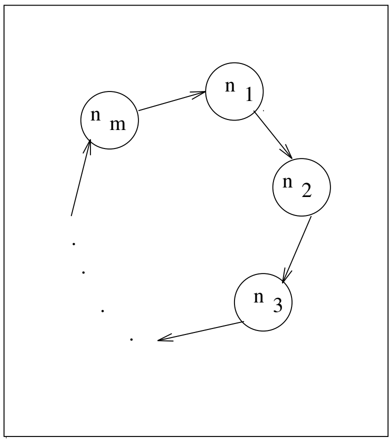

Figure 3.10: A Loop Found Through a Set of Rules

1. For the algorithm to terminate, either a loop has been found or all nodes have been searched through and no loop exists. If the algorithm terminates having found a loop, then a node is reached which has been visited before and there are no remaining disjuncts to be solved i.e. if a loop exists then the termination of the algorithm will establish the loop.

2. Each edge $\{n_i, n_j\}$ for $i, j = 1, \dots, m$ in the graph will correspond to one of the rules $\bigcirc A_i \Rightarrow B_i$ in SNF_m given how the $\bigcirc A_i \Rightarrow B_i$ rules have been defined. Let the global $\Box$-rules from the original translation to SNF be represented by the rules

$$\left\{ \begin{array}{ccc} \bigcirc X_0 & \Rightarrow & Y_0 \\ \dots & & \dots \\ \bigcirc X_p & \Rightarrow & Y_p \end{array} \right\}$$

then each of the $\bigcirc A_i \Rightarrow B_i$ will be made from combining one or more of the $\bigcirc X_j \Rightarrow Y_j$ rules used in the backwards/forwards search algorithm.

3. Each node will also have a p as one of its conjuncts, as the initial nodes searched from are of the form $X_j \Rightarrow p$ and the backwards and forwards search algorithms will only allow a rule to be picked if the next node generated will also generate a p.

So once one of the A_i rules is triggered, no edge can lead out of the loop and p will always be generated i.e.

$$\bigcirc \bigvee_{i=1}^{n} A_i \Rightarrow \Box p$$

so R does imply a $\Box$-formula.

3.5.2 Completeness

To show the algorithm is complete we must show that if there is a set of rules R of the form $\bigcirc A_i \Rightarrow B_i$ in SNF_m for $i = 0, \dots, n$ such that

$$\bigcirc \bigvee_{i=1}^{n} A_i \Rightarrow \Box p$$

then, by taking suitable choices the algorithm will find the loop corresponding to this set of rules.

Proof

The proof refers to the algorithm described on page 27, the numbers in brackets referring to the relevant part of the algorithm. Let the rules in SNF be

$$\left\{ \begin{array}{ccc} \bigcirc X_0 & \Rightarrow & Y_0 \\ \dots & & \dots \\ \bigcirc X_p & \Rightarrow & Y_p \end{array} \right\}$$

$$\left\{ \begin{array}{ccc} \bigcirc Z_0 & \Rightarrow & W_0 \\ \dots & & \dots \\ \bigcirc Z_q & \Rightarrow & W_q \end{array} \right\}$$

where the $p + 1$ rules $\bigcirc X_j \Rightarrow Y_j$ for $j = 0, \ldots, p$ are the rules which have been combined to make the $\bigcirc A_i \Rightarrow B_i$ above for $i = 0, \ldots, n$ and there are $q+1$ other rules of the form $\bigcirc Z_j \Rightarrow W_j$ for $j = 0, \ldots, q$.

Initially the algorithm searches for rules of the form $\bigcirc R_i \Rightarrow p$ (part 1 of the algorithm). Some of these are going to be in the set of rules $\bigcirc X_j \Rightarrow Y_j$ in order to generate p at every node of the loop. The search for a node always commences from one of these rules (parts 2 and 3 of the algorithm) so at some point we will begin the search for the loop from a rule of the form $\bigcirc R_i \Rightarrow p$ which is in our set $\bigcirc X_j \Rightarrow Y_j$. We will consider two cases

(i) a self-contained loop,

(ii) a loop made up from several partial loops.

For case (i) assume we've picked the rule $\bigcirc A_i \Rightarrow B_i$ in SNF_m, then to generate the next node we need to choose a rule of the form $\bigcirc C \Rightarrow A_i$ in SNF_m. There are two options

1. $\bigcirc C \Rightarrow A_i$ is made from combining at least one rule from $\bigcirc Z_j \Rightarrow W_j$ other rules with zero or more of the other global $\square$-rules.

2. $\bigcirc C \Rightarrow A_i$ is made by combining one or more rules from $\bigcirc X_j \Rightarrow Y_j$ (and none of the $\bigcirc Z_j \Rightarrow W_j$ rules) and is equivalent to one of the $\bigcirc A_i \Rightarrow B_i$ rules.

By guiding the search always to take the second option, rules selected will all be equivalent to one of the $\bigcirc A_i \Rightarrow B_i$ rules and will be made from combining one or more of the $\bigcirc X_j \Rightarrow Y_j$ rules until a the $\bigcirc A_i \Rightarrow B_i$ rule generated is already in the set and a loop has been detected (part 4(b) of the algorithm). As we know there is a loop, then this second choice is always available.

Considering case (ii) assume we've detected a partial loop from using rules $\bigcirc X_j \Rightarrow Y_j$ by using backwards search as described above but have also used a disjunctive rule (part 4(c) of the algorithm), the rule $\bigcirc A_i \Rightarrow B_i$ in SNF_m, where B_i is of the form $D_1 \vee \ldots \vee D_n$. Then we must still solve the remaining disjuncts. To generate the next node we need to choose a rule of the form $\bigcirc D_k \Rightarrow C$ in SNF_m for each D_k not yet searched through and where $k = 1, \ldots, n$. There are two options

1. $\bigcirc D_k \Rightarrow C$ is made from combining at least one rule from $\bigcirc Z_j \Rightarrow W_j$ other rules with zero or more of the other global $\square$-rules.

2. $\bigcirc D_k \Rightarrow C$ is made by combining one or more rules from $\bigcirc X_j \Rightarrow Y_j$ (and none of the $\bigcirc Z_j \Rightarrow W_j$ rules) and is equivalent to one of the $\bigcirc A_i \Rightarrow B_i$ rules.

Again by guiding the search always to take the second option, rules selected will all be equivalent to one of the $\bigcirc A_i \Rightarrow B_i$ rules and will be made from combining one or more of the $\bigcirc X_j \Rightarrow Y_j$ rules until a the $\bigcirc A_i \Rightarrow B_i$ rule generated is already in the set and a partial loop has been detected. (part 6(c) of the algorithm). Further partial loops are found from the remaining disjuncts until there are no disjuncts remaining (part 5 of the algorithm). As we know there is a loop, then by selecting rules from the $X_j \Rightarrow Y_j$ it is always possible to detect it.

3.5.3 Complexity

Let $\bigcirc P_i \Rightarrow F_i$ be the global $\Box$-rules in the SNF set for $i = 1, 2, \ldots, n$. Let the set R of all these rules combined into SNF$_m$ be

$$\left\{ \begin{array}{ccc} \bigcirc A_0 & \Rightarrow & B_0 \\ \ldots & & \ldots \\ \bigcirc A_r & \Rightarrow & B_r \end{array} \right\}$$

then in the worst case all these rules may need to be searched to establish whether a loop has been found or not. This is illustrated in Figures 3.11 and 3.12 where nodes are represented as $s_0, s_1, \ldots, s_r$ and s_r^*, and R_i denotes the use of the rule $\bigcirc A_i \Rightarrow B_i$. Figure 3.11 shows the case where a loop has been found and Figure 3.12 where a loop has not been found, where in both cases all the $\bigcirc A_i \Rightarrow B_i$ rules have been used. In the worst possible case all the combinations

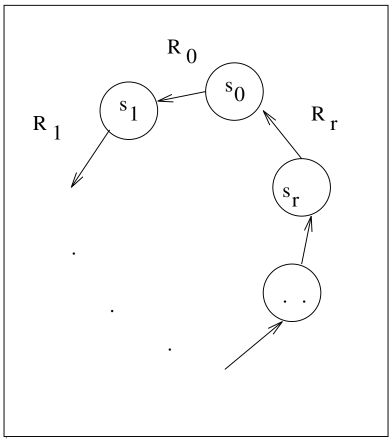

Figure 3.11: A Loop is Found by Using all the Rules

of the global $\Box$-rules must be taken to establish the existence (or not) of a loop. This will be $2^n - 1$ combinations for n global $\Box$-rules. The construction of the original rule-set from a formula is linear (see Section 6.1).

The worst case complexity is therefore exponential as combinations of all rules may have to be taken to detect the presence of a loop. The average case complexity should be better than this as the loop which generates the resolvents which are crucial the refutation, may be found without having to combine all (or possibly any) of the rules.

3.6 Enhancements to the Algorithm

3.6.1 Storage of the Path

Initially the path was stored as an unstructured list, solely recording which nodes had been visited. Later this was found to be unsatisfactory for two main reasons. The data structure

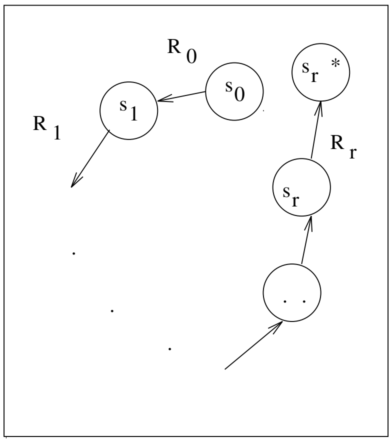

Figure 3.12: A Loop is not Found by Using all Rules

representing the path was supposed to record the route taken through the graph in order to identify a loop. The need to represent partial loops as part of the main loop made the flat structure unsuitable. Secondly the loop had to be stored in a form from which the rules used in its construction could be retrieved, so that they could be used to generate the new resolvents from the temporal resolution step. A more satisfactory solution was the storage of the path as a structured object, as it modelled the structure implicit in the graph itself. The representation of a path which has been formed using no disjunctive rules is just a list of nodes enclosed in square brackets. The use of a disjunctive rule is recorded by the opening of a left square bracket for each disjunct on the right hand side of the rule. The use of an internal right square bracket denotes that a partial loop has been found for that particular disjunct. The use of the structure is more easily understood by referring to the following example. The retrieval of the rules which had been originally used to construct the path was now possible, and the storage of disjuncts became part of the path structure.

Due to the nature of the forwards and backwards search mechanisms used, the path is built up adding new nodes onto the head of the path while using backwards search, but appending them to the end of the path while using forwards search. In this way, by reading the path from left to right, the direction of the arrows on the graph is maintained. When solving disjuncts (using forwards search) the sub-path for the partial loop for this particular disjunct is built up separately, and once it has been established to be a partial loop, it is inserted back into the main path structure.

Example

This may be illustrated by referring back to Example 3 on page 33. Initially the path was built up by adding nodes to the head of the path obtaining the paths

$$[(a)]$$
$$[(b), (a)]$$
$$[(c), (b), (a)]$$

each time a new successor is found. When the disjunctive rule 1 is used, this is shown by structuring the path using square brackets, to give

$$[(b), [(c), (b), (a)], [(d)]].$$

A partial loop has been found for one loop of the disjunct delimited by the nodes (b), so the prefix to the path $[(a)]$ is removed and the path becomes

$$[(b), [(c), (b)], [(d)]].$$

The disjunct (d) remains to be solved forwards. The sub-path is

$$[(d)]$$

which becomes

$$[(d), (b)]$$

having found (b) as the new successor, which is appended to the end of the sub-path. This sub-path is inserted back into the main path in place of the $[(d)]$ to give the solution path

$$[(b), [(c), (b)], [(d), (b)]].$$

3.6.2 Detection of loops

Having decided to store the path as a structured object, which for complicated graphs with many disjuncts would mean a high level of nesting, the detection of loops, by reaching a node previously searched through, became more expensive to test than when the path was unstructured. A structured path is more costly to search and, in order to be able the reconstruct the rules which had been used to generate the path in the first place, nodes need to be repeated to indicate the end of the partial loops. To ease this, nodes which had been searched through already were added to a flat list of *searched* nodes. This list contained fewer nodes than the path itself, as no repetition was necessary, and had no structure to complicate processing. Every time a new node was generated as a successor, the list of searched nodes could be checked to see whether it contained this new node, indicating a loop had been detected, more efficiently than searching through the path itself. However, additional work was required in deleting nodes from this list of *searched* nodes, when removing the prefix to the path.

Example

Returning to Example 3, to the point in the processing where the first partial loop has been found, but the prefix has not yet been removed the structured path is

$$[(b), [(c), (b), (a)], [(d)]]$$

and the list of solved nodes at this point would be

$$[(a), (b), (c)]$$

because at this point we have searched through all these nodes but have not yet searched from node (d). The partial loop found, delimited by (b) is

$$[(b), [(c), (b)]]$$

and the prefix to the loop is $[(a)]$ which is removed leaving the path as

$$[(b), [(c), (b)], [(d)]]$$

with the disjunct (d) remaining to be solved. As (a) has been removed from the path it must also be removed from the solved nodes which now becomes

$$[(b), (c)].$$

The processing continues as described previously.

3.6.3 Avoiding Repetition of Search Paths

Analogous to the storage of a list of *searched* nodes is the idea of storing a list of *failed* nodes. A *failed* node is defined as one which has no forwards successors. If a *failed* node is encountered during forward search no further processing is attempted from the node. In this way searches from nodes down paths through the graph which are known not to contain a loop can be avoided and so reduce the search space. This has not been incorporated in the current version of the program but is a possibility for future improvements to the implementation of the resolution step mentioned in Section 6.1.

Having implemented a graph search algorithm to detect loops through graphs this could be integrated with sub-programs to perform translation to SNF and non-temporal resolution to create a complete theorem prover.

Chapter 4

The Complete Resolution Theorem Prover

4.1 Implementation

The implementation of the temporal resolution step described in Chapter 3 has been integrated with other sub-programs necessary to perform the full resolution method described in Chapter 2 and to produce a complete resolution theorem prover. The resolution theorem prover has been implemented on Sun Workstations using SICSTUS PROLOG [CW91].

4.2 Translation to SNF

Existing sub-programs, performing the SNF translation, written in PROLOG, were available for inclusion in the complete resolution theorem prover. If the SNF translation is viewed as a set of rewrite rules, these may easily be encoded as predicates in a PROLOG program. These sub-programs were suitable for translating both Propositional and First Order Temporal Logic formulae, although the whole theorem prover has only been developed for Propositional Temporal Logic.

Formulae presented to the SNF translator may include any of the temporal operators in PTL, and must first be negated if validity rather than unsatisfiability is to be shown. It has been shown [Fis92b], if a formula A is satisfiable, then by anchoring it to the beginning of time i.e. $\square(\newmoon\,\mathbf{false} \Rightarrow A)$ has no effect on its satisfiability. Thus formulae accepted by the SNF translator are of the form

$$\newmoon\,\mathbf{false} \Rightarrow A$$

where A does not contain an implication sign, and the external $\square$-operator is assumed. Alternatively the SNF translator accepts a list of clauses separated by commas as input. The same program, therefore is suitable for both the translation of the original formula, and the new resolvents in clause form, from subsequent processing cycles.

An additional SNF style translation has been incorporated into the original programs for translation of rules of the form $\bigcirc\,\mathbf{true} \Rightarrow F$. This additional translation may be represented by the following rule

$$\frac{\bigcirc\!\!\!\!\bullet\,\mathbf{true} \Rightarrow \bigvee_{b=1}^{r} l_b}{\begin{array}{rcl} \bullet\,\mathbf{false} & \Rightarrow & t_1 \\ \bigcirc t_1 & \Rightarrow & t_1 \\ \bigcirc t_1 & \Rightarrow & \bigvee_{b=1}^{r} l_b \end{array}}$$

where each l_b is a literal and t_1 is a new proposition symbol.

This additional translation has been incorporated as there is no explicit representation of **true** in the graph constructed from the global $\square$-rules. As $\bigcirc$**true** is satisfied at every moment in time except the beginning, $\bigvee_{b=1}^{r} l_b$ must be satisfied at every moment in time except the beginning of time for the rule to be satisfied. The new constraints likewise make $\bigvee_{b=1}^{r} l_b$ true at every moment in time except the beginning of time by using a new propositional symbol t_1, which once satisfied will always generate itself and $\bigvee_{b=1}^{r} l_b$ in the next moment in time.

4.3 Non-Temporal Resolution

4.3.1 Overview

Sub-programs for carrying out non-temporal resolution were available for inclusion in the resolution theorem prover although some enhancements and additions were necessary particularly to perform further simplification, carry out subsumption and test for termination.

The algorithm used for performing non-temporal resolution, cycles around four main processes

1. Simplification
2. Subsumption
3. Resolution
4. Termination

which have been described previously in Section 2.2.2.

Recapping briefly, simplification carries out both standard classical, and additional temporal, simplification operations. Subsumption and resolution are similar to the classical cases but are applied to literals relating to the same moment in time. Termination is dependent on the generation of one to the following unsatisfiable formulae

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

or the generation of no new resolvents.

The algorithm operates using two separate sets of rules to avoid repeating simplification, subsumption, and resolution processing between pairs of rules where this has already been carried out. The two sets will be called *Old* and *New*, the *Old* set containing rules from previous cycles around the non-temporal algorithm, and the *New* set containing any new resolvents from the last resolution cycle.

4.3.2 Detailed Algorithm

Input to the algorithm is a set of rules which have been translated into SNF.

1. Set *Old* to be empty, and *New* to be the current rule-set.

2. Simplify *New*, i.e. perform standard, and additional temporal, simplification on each member of the rule-set. If simplification generates a termination formula, the algorithm terminates having generated **false**.

3. Take a rule R_i in *New*, compare it with each rule in *Old* in turn and

   (a) delete the rule, R_i in *New* if it is subsumed by the rule in *Old*

   (b) delete the current rule in *Old* if it is subsumed by the rule in *New*

   (c) leave both rules in their respective rule-sets if no subsumption may take place

   until either R_i has been deleted, or there are no more rules in *Old* to compare R_i with. If the rule has not been deleted compare it with the remaining rules in *New* deleting subsumed rules where necessary.

4. Repeat step 3 until all the rules in *New* have been processed. If *New* is now empty the algorithm terminates as no new resolvents have been found.

5. Take a rule R_i in *New* and attempt resolution between it and every rule in *Old* in turn, any new resolvents being stored in a separate list, *NextNew*. Append R_i to the end of *Old* and delete it from *New*.

6. Repeat 5 until *New* is empty, or a resolvent which is equivalent to a termination formula is generated and the algorithm terminates having generated **false**.

7. Set *New* equal to *NextNew*, and *NextNew* to empty.

8. If *New* is empty the algorithm terminates as no new resolvents have been found.

9. Repeat steps 2–8 until termination occurs, termination being the generation of **false** indicated by deriving one of the three unsatisfiable formulae previously mentioned, or by deriving no new resolvents.

4.4 The Full Theorem Prover

Having developed, or been able to utilise or adapt, existing software for the three main processes of the resolution prover, each of which had been tested individually and would work as a self contained program in its own right, the remaining task was to combine these programs to produce an integrated theorem prover.

4.4.1 Integrating the Temporal Resolution Programs

Exactly how the temporal resolution process should be incorporated into the complete theorem prover, appeared to be the main problem to be decided. More specifically what strategy should be adopted when more than one loop could be detected from a set of rules for resolution with a particular $\Diamond$-rule, or when there was more than one $\Diamond$-rule for resolution. Different options for integrating the resolution program into the whole theorem prover are outlined below.

1. To find all possible loops in the set of rules for a particular $\Diamond$-rule and generate new resolvents, related to each loop detected before continuing in the main algorithm.

2. To find all possible loops for all $\Diamond$-rules and generate new resolvents for each $\Diamond$-rule before continuing.

3. Detect only one loop each resolution cycle but use a different $\Diamond$-rule each time, until all the $\Diamond$-rules has been processed and then try to find a second loop for each, continuing in this manner until termination occurs.

4. Find one loop (if one exists) for a particular $\Diamond$-rule, continue with the main resolution algorithm, and on the next cycle of temporal resolution, look for another loop for the same $\Diamond$-rule, or if none existed for the next eventuality.

The last alternative is the one which was implemented although all suggestions should produce the same solutions to formulae assuming they all terminate but some may be more efficient than others. It may be interesting to implement all four versions and compare the timings for each version on different formulae (see Future Work 6.1).

The last suggestion was implemented in preference to 1 and 2, because if a loop had been found relatively quickly in the search for loops producing the resolvents that would eventually be used to derive **false**, options 3 and 4 would complete the refutation more quickly than 1 or 2. This is because using option 1 the complete search space must be searched for a particular $\Diamond$-rule, i.e. all loops must be detected until no more are found. Similarly the application of the second strategy would mean the whole search space for each eventuality would have to be searched. Options 3 and 4 however were perceived to have a less obvious difference between them, and gains and losses in efficiency, were thought likely to be related to the particular formula in question rather than a better overall strategy.

4.4.2 Self-Contained Sub-Loops

While considering the extraction of more than one loop from a rule-set, consideration was given to the detection of self-contained loops which form part of a larger loop. This can be best explained by using an example. Given the rule-set

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

this can be represented graphically as in Figure 4.1. There are two obvious loops in this graph

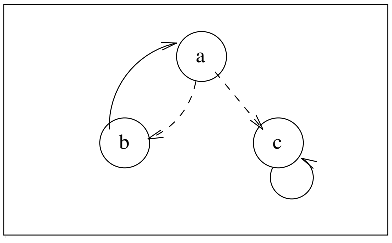

Figure 4.1: Self-Contained Sub-Loops

1. $[(a), [(b), (a)], [(c), (c)]]$, from rules 1–6;
2. $[(c), (c)]$, a sub-loop within the main loop, from rules 3 and 6.

The two new resolvents generated from the first loop are

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

and the new resolvents from the second loop are

$$
\begin{array}{rcl}
\bullet\,\mathbf{true} & \Rightarrow & \neg x \vee \neg c \\
\bigcirc x & \Rightarrow & \neg c\, \mathcal{W} \neg l
\end{array}
$$

After translating the second pair of resolvents into SNF we obtain

$$\begin{array}{rcl}
\text{●}\,\mathbf{false} & \Rightarrow & \neg x \vee \neg c \\
\bigcirc\,\mathbf{true} & \Rightarrow & \neg x \vee \neg c \\
\bigcirc\, x & \Rightarrow & \neg l \vee \neg c \\
\bigcirc\, x & \Rightarrow & \neg l \vee t_1 \\
\bigcirc\, t_1 & \Rightarrow & \neg l \vee \neg c \\
\bigcirc\, t_1 & \Rightarrow & \neg l \vee t_1
\end{array}$$

where t_1 is a new proposition symbol. Translation of the first pair of resolvents into SNF will produce exactly the same six rules in addition to several others. Therefore the larger loop generates more information than from the smaller self-contained sub-loop.

Detecting the "largest" loop in this type of situation would mean solving (i.e. searching for partial loops) all the disjuncts of a disjunctive formula. For example, for a formula

$$\bigcirc a \Rightarrow b_1 \vee b_2 \vee \ldots \vee b_n$$

it is easy to check that each $a, b_1, b_2, \ldots, b_n$ will all imply l, assuming we are resolving with $\mathcal{LP} \Rightarrow \Diamond\neg l$, and this is in fact established before a particular rule is used to generate successors. However it is much more expensive to search for a partial loop from all disjuncts, as possibly b_n might not be part of a partial loop and thus the whole disjunctive rule will fail. Any smaller sub-loops detected earlier in the search will be returned as the solution. In the example illustrated in Figure 4.1, the theorem prover will find the loop $[(c), (c)]$ first and the larger loop next time around the cycle.

4.4.3 Generation of New Resolvents

Having detected a set of rules which together imply a $\Box$-formula, to be resolved with a $\Diamond$-rule, new resolvents must be generated to apply the temporal resolution rule given in Section 2.2.3. Examination of this rule revealed that the reconstruction of the actual rules which had been used to create the loop was not in fact necessary to allow the generation of the new resolvents. Similarly, the structure of the loop detected was also not necessary. The information which was required however was the $\Diamond$-rule and the nodes from the graph which made up the loop.

Example

The two rule-sets below

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\bigcirc a & \Rightarrow & b & \bigcirc a & \Rightarrow & b \vee c \\
\bigcirc b & \Rightarrow & c & \bigcirc b & \Rightarrow & a \\
\bigcirc c & \Rightarrow & a & \bigcirc c & \Rightarrow & a \\
\bigcirc a & \Rightarrow & l & \bigcirc a & \Rightarrow & l \\
\bigcirc b & \Rightarrow & l & \bigcirc b & \Rightarrow & l \\
\bigcirc c & \Rightarrow & l & \bigcirc c & \Rightarrow & l \\
\bigcirc r & \Rightarrow & \Diamond\neg l & \bigcirc r & \Rightarrow & \Diamond\neg l
\end{array}$$

which may be represented graphically as in Figure 4.2 will both produce the same two resolvents.

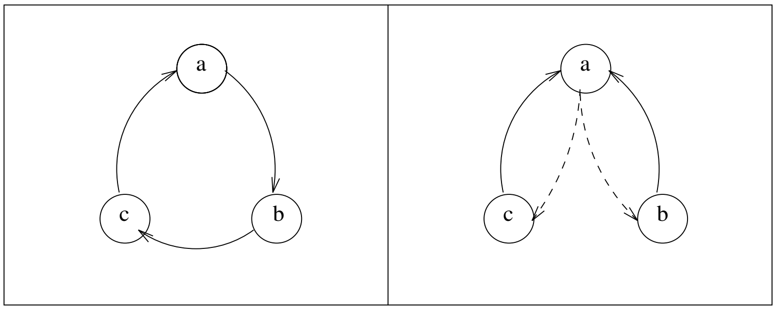

Figure 4.2: Two Loops with Different Structure but the Same Nodes

$$\begin{aligned} \bullet\, \mathbf{true} &\Rightarrow \neg r \vee (\neg a \wedge \neg b \wedge \neg c) \\ \odot\, r &\Rightarrow (\neg a \wedge \neg b \wedge \neg c)\, \mathcal{W}\, \neg l \end{aligned}$$

These resolvents can be generated by knowing the $\Diamond$-rule, in this case $\odot r \Rightarrow \Diamond \neg l$, and the nodes forming the loop, which are (a), (b), and (c) in this example.

4.4.4 Storage of Loops

So as not to generate the same loops, on subsequent applications of the resolution process, the loop and the eventuality it relates to need to be recorded. As loops detected which have the same nodes but may have different structures for a particular $\Diamond$-rule produce the same resolvents (see Section 4.4.3), and that only the nodes and a $\Diamond$-rule are needed to produce these resolvents it would seem sensible to store the loops as a list of nodes related to a particular $\Diamond$-rule. In the example given in Section 4.4.3 if either of these loops had been detected it would have been stored as follows

$$[\odot r \Rightarrow \Diamond \neg l, [(a), (b), (c)]].$$

4.4.5 Termination

Termination of the temporal resolution process will only occur when a new loop is found for an eventuality, or no new loops can be found for any eventuality. Examples may occur (see Example 1b page 51) when initially there may be no loops to be resolved with a particular $\Diamond$-rule, but later in the processing such a loop may be detected by using new resolvents created during previous temporal-resolution cycles.

The full theorem prover will terminate with the formula pronounced as satisfiable only if there are no more new resolvents generated during a complete cycle i.e. no more resolvents from non-temporal resolution and no new loops from temporal resolution for any of the $\Diamond$-rules. If a formula is found to be unsatisfiable, **false**, or one of the three unsatisfiable formulae indicating false, must be generated from non-temporal resolution.

4.4.6 The Detailed Algorithm

The detailed algorithm used for the whole resolution theorem prover is described below. It is assumed the formula is to be shown to be unsatisfiable and must have been negated prior to presentation to the theorem prover if validity is to be shown.

1. Translate the formula to SNF.
2. Separate the $\Diamond$-rules from the $\Box$-rules.
3. Pair each $\Diamond$-rule with an empty list, which will be used to store lists of nodes representing loops found from the temporal resolution cycles, and store the $\Diamond$-rule pairs in a list.
4. Perform non-temporal resolution until **false** is derived or no new resolvents are generated. If **false** has been derived the algorithm terminates.
5. Take the current $\Diamond$-rule and its related loops already found, and perform temporal resolution until
   (a) a new loop is found for this particular $\Diamond$-rule so the temporal resolution processing stops,
   (b) no new loops for this $\Diamond$-rule can be detected, so the next $\Diamond$-rule is used and step 5 is repeated,
   (c) there are no more $\Diamond$-rules left to resolve with, i.e. no new loops have been found for any of the $\Diamond$-rules, so no new resolvents can be generated, and the temporal resolution cycle terminates.
6. Return the path noting the new loop (if one has been found), and the eventuality it is to be resolved with. If no new loops have been detected return an empty path.
7. If a non-empty path has been returned
   (a) store the path as a list of nodes, and save in the $\Diamond$-rule, nodes list pair, initialised in step 3 above.
   (b) use these nodes to generate the new resolvents.
   (c) translate these resolvents onto SNF
8. If an empty path has been returned then the main algorithm terminates with the formula satisfiable, as the last non-temporal processing must have generated no new resolvents, as well as the current temporal resolution having generated no new resolvents, because the non-temporal resolution processing can only be terminated by no new resolvents being derived or **false** being derived, and the derivation of **false** would have terminated the processing before the last temporal resolution had taken place.
9. Repeat steps 4–8 until **false** is derived, i.e. the formula is unsatisfiable, or no new resolvents can be derived, i.e. the formula is satisfiable.

4.5 Examples

In this section we shall give several examples of proofs based on the resolution method described above. The examples are related to those given in Chapter 3, requiring the loops found in the previous examples to be used as part of the refutation.

Example 1a: A Simple Refutation for Simple Loop

To show that the following formula

$$\Box(a \Rightarrow \bigcirc a) \wedge \Diamond\neg l \wedge l \wedge a \wedge \Box(a \Rightarrow \bigcirc l)$$

is unsatisfiable it is rewritten into SNF, as follows

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

The refutation then proceeds as follows

$$\begin{array}{rrcll}
6. & \bullet\,\mathbf{false} & \Rightarrow & \neg a\,\mathcal{W}\,\neg l & [1, 2, 3, \text{Temporal resolution}] \\
7. & \bullet\,\mathbf{false} & \Rightarrow & \neg l \vee \neg a & [6, \text{Rewrite to SNF (initial rule only)}] \\
8. & \bullet\,\mathbf{false} & \Rightarrow & \neg a & [4 + 7, \text{Non-temporal resolution}] \\
9. & \bullet\,\mathbf{false} & \Rightarrow & \mathbf{false} & [5 + 8, \text{Non-temporal resolution}]
\end{array}$$

Example 1b: Simple Loops used to Resolve with Several Eventualities

To show that the formula

$$\Diamond p \wedge \Box(p \Rightarrow \bigcirc p) \Rightarrow \Diamond\Box p$$

is valid, it is negated, giving

$$\Diamond p \wedge \Box(p \Rightarrow \bigcirc p) \wedge \Box\Diamond\neg p$$

and rewritten into SNF, as follows

$$\begin{array}{rrcl}
1. & \bullet\,\mathbf{false} & \Rightarrow & \Diamond p \\
2. & \bullet\,\mathbf{false} & \Rightarrow & \Diamond\neg p \\
3. & \odot p & \Rightarrow & p \\
4. & \odot\,\mathbf{true} & \Rightarrow & \Diamond\neg p
\end{array}$$

The refutation then proceeds as follows

$$\begin{array}{rrcll}
5. & \bullet\,\mathbf{false} & \Rightarrow & \neg p & [2 + 3, \text{Temporal resolution}] \\
6. & \bullet\,\mathbf{true} & \Rightarrow & \neg p & [3 + 4, \text{Temporal resolution}] \\
7. & \boldsymbol{\circ}\,\mathbf{true} & \Rightarrow & \neg p & [6, \text{Rewrite to SNF (global rule)}] \\
8. & \boldsymbol{\circ}\neg p & \Rightarrow & \neg p & [7, \text{Rewrite to (additional) SNF}] \\
9. & \bullet\,\mathbf{false} & \Rightarrow & p & [1 + 8, \text{Temporal resolution}] \\
10. & \bullet\,\mathbf{false} & \Rightarrow & \mathbf{false} & [5 + 9, \text{Non-temporal resolution}]
\end{array}$$

The rewrite to (additional) SNF in rule 8 shows the removal of $\boldsymbol{\circ}\,\mathbf{true}$ from rule 7, that has been described previously in Section 4.2.

Example 2: A Loop From Pure Backwards Search

To show that the following formula

$$\Diamond\neg l \wedge a \wedge l \wedge \Box(a \Rightarrow \bigcirc(b \wedge l)) \wedge \Box(b \Rightarrow \bigcirc(a \wedge l))$$

is unsatisfiable it is rewritten into SNF, as follows

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

The refutation proceeds as follows

$$\begin{array}{rrcll}
8. & \bullet\,\mathbf{false} & \Rightarrow & (\neg b \wedge \neg a)\,\mathcal{W}\,\neg l & [1, 2, 3, 4, 5, \text{Temporal Resolution}] \\
9. & \bullet\,\mathbf{false} & \Rightarrow & \neg l \vee \neg a & [8, \text{Rewrite to SNF (initial rule only)}] \\
10. & \bullet\,\mathbf{false} & \Rightarrow & \neg l & [6 + 9, \text{Non-temporal resolution}] \\
11. & \bullet\,\mathbf{false} & \Rightarrow & \mathbf{false} & [7 + 10, \text{Non-temporal resolution}]
\end{array}$$

Example 3: A Disjunctive Loop with a Prefix

Given the following set of rules in SNF

$$\begin{array}{rrcl}
1. & \boldsymbol{\circ} b & \Rightarrow & c \vee d \\
2. & \boldsymbol{\circ} c & \Rightarrow & b \\
3. & \boldsymbol{\circ} d & \Rightarrow & b \\
4. & \boldsymbol{\circ} b & \Rightarrow & a
\end{array}$$

5.	$\circledbullet a$	$\Rightarrow$	l
6.	$\circledbullet b$	$\Rightarrow$	l
7.	$\circledbullet c$	$\Rightarrow$	l
8.	$\circledbullet d$	$\Rightarrow$	l
9.	$\bullet$**false**	$\Rightarrow$	$\Diamond \neg l$
10.	$\bullet$**false**	$\Rightarrow$	l
11.	$\bullet$**false**	$\Rightarrow$	d

The refutation proceeds as follows

12.	$\bullet$**false**	$\Rightarrow$	$(\neg b \wedge \neg c \wedge \neg d)\, \mathcal{W}\, \neg l$	[1–9, Temporal resolution]
13.	$\bullet$**false**	$\Rightarrow$	$\neg l \vee \neg d$	[12, Rewrite to SNF (initial rule only)]
14.	$\bullet$**false**	$\Rightarrow$	$\neg d$	[10 + 13, Non-temporal resolution]
15.	$\bullet$**false**	$\Rightarrow$	**false**	[11 + 14, Non-temporal resolution]

The loop detected has only used the nodes (b), (c), (d), meaning that the new resolvent produced after temporal resolution is the one shown in rule 12 and is not

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

i.e. it does not include any reference to node (a), as $[(a)]$ was discarded as the prefix to the loop.

Example 4: A Loop Using Combinations of Rules

Given the following set of rules in SNF

1.	$\circledbullet a$	$\Rightarrow$	l
2.	$\circledbullet b$	$\Rightarrow$	l
3.	$\circledbullet c$	$\Rightarrow$	l
4.	$\circledbullet (a \wedge b)$	$\Rightarrow$	a
5.	$\circledbullet (b \wedge c)$	$\Rightarrow$	b
6.	$\circledbullet (a \wedge c)$	$\Rightarrow$	c
7.	$\bullet$**false**	$\Rightarrow$	$\Diamond \neg l$
8.	$\bullet$**false**	$\Rightarrow$	c
9.	$\bullet$**false**	$\Rightarrow$	l
10.	$\bullet$**false**	$\Rightarrow$	b
11.	$\bullet$**false**	$\Rightarrow$	a

the refutation proceeds as follows

12.	$\bullet\,\mathbf{false}$	$\Rightarrow$	$(\neg c \vee \neg a \vee \neg b)\,\mathcal{W}\,\neg l$	[1–7, Temporal resolution]
13.	$\bullet\,\mathbf{false}$	$\Rightarrow$	$\neg l \vee \neg c \vee \neg a \vee \neg b$	[12, Rewrite to SNF (initial rule only)]
14.	$\bullet\,\mathbf{false}$	$\Rightarrow$	$\neg c \vee \neg a \vee \neg b$	[9 + 13, Non-temporal resolution]
15.	$\bullet\,\mathbf{false}$	$\Rightarrow$	$\neg a \vee \neg b$	[8 + 14, Non-temporal resolution]
16.	$\bullet\,\mathbf{false}$	$\Rightarrow$	$\neg b$	[11 + 15, Non-temporal resolution]
17.	$\bullet\,\mathbf{false}$	$\Rightarrow$	$\mathbf{false}$	[10 + 16, Non-temporal resolution]

The output from the resolution theorem prover for all these examples can be found in Appendix B.

Chapter 5

Related Work

In this chapter we will examine briefly some other decision procedures that have been developed for modal and temporal logics, classifying them as those based on resolution, and those based on a tableau approach.

5.1 Methods Based on Resolution

5.1.1 Clausal Methods

Clausal methods require the formula to be checked for unsatisfiability to be translated to a normal form before resolution rules may be applied.

Modal Resolution

Modal resolution based on Robinson's method for classical logic [Rob65], for various modal systems is discussed in [EdC89]. Classical logical operators are extended by the modal operators $\Box$ and $\Diamond$ (or for a collection of modal operators $[k]$ and $< k >$, where for a formula A the formula $< k >$ may be defined as $< k >= \neg[k]\neg A$). The method is clausal, so formulae are translated to a normal form before resolution rules may be applied

A modal formula is said to be in Disjunctive Normal Form (DNF) if it is of the general form

$$\begin{array}{rcl} C & = & L_1 \lor L_2 \lor \ldots \lor L_n \lor \Box D_1 \lor \Box D_2 \lor \ldots \lor \Box D_p \\ & \lor & \Diamond A_1 \lor \Diamond A_2 \lor \ldots \lor \Diamond A_q \end{array}$$

where each L_i is a literal, D_i is in Disjunctive Normal Form, and each A_i is in Conjunctive Normal Form. A modal formula is said to be in Conjunctive Normal Form (CNF), if it is of the form

$$F = C_1 \land C_2 \land \ldots \land C_n$$

where each C_j is in DNF. Formulae must be translated into CNF before any resolution rules may be applied.

The resolution rules are based on the observations that

$$\frac{\Box(p \vee q) \qquad \Diamond\neg p}{\Diamond(\neg p, q)}$$

and

$$\frac{\Box(p \vee q) \qquad \Box\neg p}{\Box q}$$

where (a, b) represents $a \wedge b$. These observations are similar to the resolution rules described in this dissertation as they attempt to resolve $\Box$ and $\Diamond$ formulae or formulae in the context of a $\Box$-operator. It is also noted that resolution rules with only one premise can be applied such as

$$\frac{\Diamond(\neg p, p \vee q)}{\Diamond(\neg p, p \vee q, q)}$$

where resolution is carried out in the context of the $\Diamond$-operator but resolution between two $\Diamond$-operators cannot be applied, for example a rule such as

$$\frac{\Diamond(p \vee q) \qquad \Diamond\neg q}{\Diamond(\neg p, q)}$$

cannot be applied.

The resolution method is based upon rules for computing resolvents, simplification, and inference rules. The resolution rules defined are based upon the three observations above and are adapted for application to different modal systems. The simplification rules are based on standard classical simplification rules and inference rules are used to direct the application of the resolution rules. The procedure as a whole is characterised by requiring formulae to be in a normal form, and that the resolution rules aim to reduce elementary inconsistencies.

The resolution of formulae in the context of $\Box$ and $\Diamond$ operators is exactly the type of resolution rule the Fisher Resolution Method applies, but the problem of identifying hidden $\Box$-formulae, addressed in this dissertation, caused by the interaction between the $\Box$ and $\bigcirc$ operators in the temporal system, does not occur as the modal systems being considered do not include the $\bigcirc$ operator.

Descriptions of other methods for modal resolution may be found in [Min90, Ohl88], for example.

Temporal Resolution

A clausal resolution method for propositional temporal logic is outlined in [CdC84]. The temporal operators defined in the logic include $\bigcirc$, $\Box$, and $\Diamond$ but include no past-time temporal operators. The method described rewrites formulae to a clausal form and then applies a series of temporal resolution rules.

The clausal form described is similar to that outlined in Section 5.1.1, but in this case each C_j, or clause, allows L_i to be a literal preceded by a string of zero or more $\bigcirc$ operators.

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 in this system can be read for example, "the resolvent of $\square x$ and $\diamond y$ is the resolvent of x and y with a '$\diamond$' in front", or "the resolvent of x and $\square y$ is the resolvent of x and $\square\square y$".

Classical operations allow classical style resolution to be performed, for example

$$\Sigma_1(p, \neg p) = \emptyset \quad \text{(where } \emptyset \text{ denotes the empty set or } \textbf{false}\text{)}$$
$$\text{And } (p, \neg p) \text{ is resolvable.}$$

The temporal resolution rules allow resolution between formulae in the context of certain operators, for example

$$\Sigma_2(\square E, \triangle F) = \triangle\Sigma_i(E, F) \quad \text{(provided that } \triangle \text{ is one of } \square, \diamond, \text{ or } \bigcirc\text{)}$$
$$\text{And if } (E, F) \text{ is resolvable then } (\square E, \triangle F) \text{ is resolvable;}$$

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

$$\Sigma_3(\square E, F) = \Sigma_i(\square\square E, F)$$
$$\text{And if } (\square\square E, F) \text{ is resolvable then } (\square E, F) \text{ is resolvable.}$$

Formulae are refuted by translation to clause form and performing resolution on clauses in the context of certain operators outlined in the resolution rules.

The method is similar to the Fisher Method as it uses translation to a clause form and attempts to resolve $\square$ and $\diamond$ rules by using the above rule. The number of temporal operators allowed in formulae though is much more limited as in this resolution method past-time operators are not allowed, and their resolution rule is much more complicated.

5.1.2 Non-Clausal Methods

Non-clausal methods apply resolution rules directly to the formula to be checked for unsatisfiability and avoid the translation of formulae to normal forms which may lead to formulae of greatly increased size. A larger number of resolution rules may need to be defined than are necessary in clausal methods, to enable their application to any formula within the logic. A non-clausal resolution method for classical logics is described in [Mur82].

Non-clausal Resolution

A non-clausal temporal resolution method is described in [AM90]. The method is defined for Propositional Temporal Logic (PTL) with operators $\bigcirc$, $\square$, $\diamond$, $\mathcal{W}$, $\mathcal{P}$, (where $u\mathcal{P}v$ is described as meaning u precedes v, and in the original paper weak until, or unless is denoted as $\mathcal{U}$) and First Order Temporal Logic. An implementation of the described method has been carried out for PTL with the operators $\bigcirc$, $\square$, and $\diamond$.

The resolution systems contain simplification rules, deduction rules, and allows the application of the proof systems recursively. In this system $u < v >$ denotes that the formula v occurs in u and then $u < w >$ denotes the result of replacing exactly one occurrence of v with w in u, and $u < v, v, v >$ indicates the formula v occurs at least three times in u and $u < w >$ denotes replacing exactly three occurrences of v with w in u. So if the formulae $A < u, \ldots, u >$ and $B < u, \ldots, u >$ have a common subformula u then the resolvent $A < \textbf{true} > \vee B < \textbf{false} >$ may be derived. The resolution rule for PTL is of the form

$$A < u, \ldots, u >, B < u, \ldots, u > \longmapsto A < \textbf{true} > \vee B < \textbf{false} >$$

where occurrences of u in A and B are replaced with **true** and **false**, and must all 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 search space is reduced by replacing with **true** at least one positive occurrence of u in A and with **false** at least one negative occurrence of u in B.

The method is non-clausal so does not require a translation to a normal form as the Fisher method. However the system includes many inference rules producing an increased search space.

5.2 Tableaux

DP and DPP

DP and DPP are decision procedures which have been developed and implemented, and are described in detail in [Gou84]. DP is a decision procedure for future-time temporal logics allowing classical connectives and the five future-time temporal operators described in Chapter 2. The decision procedure is based on the semantic tableau method [HC68, Smu68], and is similar to that described in [Wol83].

Using semantic tableau methods a formula f is shown to be valid if it is not possible to construct a model for $\neg f$. The decision procedure DP uses an algorithm to construct such a model. 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), thus decomposing it into present requirements and future commitments.

If a graph, whose nodes represent a state labelled by formulae true in that state, can be built for a formula f and a path traced through the graph, known as a Hintikka structure, then f has a model.

DPP is the past-time version of the decision procedure and includes all the past-time operators defined in Chapter 2.

5.3 Timings

The examples used in Chapter 4, and a selection of other examples were tested on both the resolution theorem prover and DP/DPP. The resolution theorem prover has been implemented in Prolog, while DP/DPP has been written in Pascal. Approximate timings (elapsed time in seconds) for the examples using both the resolution theorem prover and DP/DPP on a Sun SPARC station 2 are given in Table 5.1. Note these timings show only two significant figures. Output from the resolution theorem prover for Examples 1a, 1b, 2, 3, and 4 is given in Appendix B. A list of the examples used to obtain the "Additional Examples" timing is given in Appendix C.

Example	Resolution Prover	DP/DPP
1a	0.1	0.0
1b	0.1	0.1
2	0.2	0.0
3	0.6	0.3
4	1.2	0.0
Additional Examples	3.0	1.5

Table 5.1: Timings of Examples tried on the Resolution Prover and DP/DPP

Some comments on these timings are given in Section 6.2.

Chapter 6

Further Work and Conclusions

In this section we give a brief summary of possible further work that might be attempted as a result of this dissertation.

6.1 Possibilities for Further Work

6.1.1 Further Work Related to the Described Implementation

At several points during the dissertation it has been indicated where decisions between different techniques have been taken, and it has been attempted to show why one process or idea was used in preference to another. Section 4.4.1 for example outlined different ways the temporal resolution could be integrated into the main theorem prover that could be implemented and compared.

The most obvious extension to the work completed is to use a different strategy to represent the rules as graphs and to search them for sets of formulae which together will imply a $\square$-formula. The method that has been adopted in this dissertation creates an ordinary graph, but, assuming the eventuality with which we wish to resolve is $(\neg p)$, avoids building the complete graph by only using rules, or combinations of rules which will generate p. Two alternatives to this would be

1. to build an ordinary graph from the rules combined in SNF_m and search for a strongly connected component;

2. to build a hypergraph directly from the rules in SNF and search for a strongly connected component.

Statistics relating to timings and machine usage, for different formulae, for the three alternatives could then be compared to see whether the currently implemented method was actually better than the other suggestions. More detailed explanations of how to implement 1 and 2 above are given below.

Using "Ordinary" graphs

Given that we want to resolve with $\mathcal{L}Q \Rightarrow \Diamond\neg p$, and the original rule-set has already been translated into SNF, all combinations of the global $\Box$-rules are taken by rewriting to SNF$_m$. Any rules which do not imply p are removed (i.e. do not have p as a consequence on their right hand side). The remaining rules may be represented as a graph having first removed p from all the rules, as each node must automatically include a p as one of its conjuncts and can therefore be ignored. The detection of strongly-connected components (SCC) of this graph will ensure that p is generated at every node and, due to the definition of a strongly-connected component, once entering an SCC, p will continue to be generated. Having represented the rules as a standard graph, strongly-connected components can be found by applying Tarjan's algorithm [AHU74]. The rules that have been used to create the SCC of the graph will be the set which imply $\Box p$ and can be used to apply the Temporal Resolution Rule.

The graph drawn of the rules rewritten to SNF$_m$ will in fact be the tableau for the temporal formula. This method has the advantage of being relatively simple to understand and implement, but will be expensive (exponential) due to the initial translation to SNF$_m$.

Using Hypergraphs

To avoid the need to translate all the rules to SNF$_m$, the rules in SNF can be represented directly as a hypergraph. As the hypergraph will be more complicated than a standard graph, and therefore the notion of a path through the graph also more complex, the search mechanism to detect strongly-connected components is likely to be more expensive than for standard graphs, which has been shown to be linear [AHU74].

Enhancements to the theorem prover which has been developed could also be achieved by analysing in more detail the implementation of the algorithms described, and possible improvements made by use of different data structures, or refinements to sub-processes which had perhaps been noted to be taking longer than expected.

6.1.2 Extensions to the Theorem Prover

On a broader scale, several lines of research could be developed based on the research described.

The development of a first order version of the Resolution Theorem Prover.

The theorem prover has been developed for Propositional Temporal Logic only, and extensions for first order logic could be investigated. A first order translation to SNF exists [Fis92a] and a program to perform the translation has been prototyped. Note, however, that *full* First-Order Temporal Logic is undecidable [Hus87, Aba88]. In spite of this, it may be possible to find subsets of First-Order Temporal Logic to which this form of temporal resolution can be successfully applied.

Efficient translation to normal form.

The development of an efficient translation to normal form could be investigated for PTL. A naive translation of a formula to the normal form required for the resolution method may lead to a formula of greatly increased size. A more refined translation into normal form may reduce the size of formula produced and thus improve the efficiency of the resolution procedure. Research relating to translation to special clausal forms has been investigated, for example in [PG86]. A structure preserving clause form translation for propositional and first-order classical logics is defined which avoids the exponential increase in size that may occur with standard translations. This, other strategies relating to translation to clausal forms, and requirements that any good transformation should have are discussed in [dlTC90]. Such ideas could be adopted and developed for use with temporal logics, and applied before the application of the resolution step.

Parallel Implementation Techniques.

Improvements in efficiency may be obtained through the use of parallel techniques in various aspects of the resolution procedure. An investigation into possible uses of parallel techniques throughout the resolution procedure would identify potential for such improvements. An obvious place to start would be in the translation to SNF where the removal and rewriting of multiple temporal subformulae could be handled in parallel. Similarly, the transformation from SNF to SNF_m could be achieved more efficiently using parallel techniques, thus removing one of the stumbling blocks to the direct application of Tarjan's algorithm on 'ordinary' graphs. Parts of the non-temporal resolution algorithm could be implemented in parallel, for example, attempting resolution between each of the new resolvents from the last cycle round the non-temporal cycle and the rules from previous cycles concurrently. The temporal resolution step could be parallelised in an obvious way when there was more than one $\Diamond$-rule, by searching for a loop for each $\Diamond$-rule concurrently. Finally, the actual graph algorithm described in Sections 3.2 and 3.3 would benefit from the addition of parallelism. For example, when a disjunct is encountered, both branches could be searched simultaneously (possibly with some exchange of searched nodes).

Complexity Analysis.

A theoretical analysis of the complexity of both the translation into normal form and the resolution process itself could be carried out. This would also include looking at restricted versions of temporal resolution together with restrictions on SNF.

Transformations based on renaming, where a sub-formula is replaced by a new proposition symbol and the truth value of the new proposition symbol linked to the sub-formula that has been replaced have been defined for classical logics [PG86]. The renaming transformations aim to obtain a good worst case complexity in size: linear in the propositional case and quadratic in the first order case, although these non-standard transformations may produce a clause form more difficult to refute than the standard one. The translation to SNF uses the renaming technique as part of its transformations [Fis92b].

Applications in program/system verification.

The form of temporal logic formulae represented by SNF is particularly useful for describing state-transition systems since an SNF rule (particularly a global $\square$-rule) corresponds very closely to a transition in such a system. For example, the transitions in a system might be represented by global $\square$-rules, while the initial conditions might be represented by initial $\square$-rules. The $\diamond$-rules then correspond to extra constraints upon the system (such as some form of fairness), or the conditions that the system is being checked against. Given this, the temporal resolution method should be applicable to the verification of programs or systems that can be represented in this way. This is particularly useful if the resolution method can be extended to a reasonable subset of First-Order Temporal Logic.

Application of Resolution to other forms of logic.

The resolution procedure, as described, is only applicable to one particular form of temporal logic. By changing both the resolution rules and the definition of SNF slightly, it should be possible to apply this method to a variety of temporal, and indeed, modal logics. For example, if we remove the discreteness constraint on PTL (thus removing $\bigcirc$, $\bullet$ and $\odot$), then there is no need to search for combinations of $\bigcirc$-formulae representing a hidden $\square$-formula, and the resolution system essentially becomes that described by Mints for modal logics [Min90].

Application to execution.

Just as the resolution method for classical logic has been used to develop the logic programming language PROLOG, so the resolution method described above may be used to develop a *temporal* logic programming language. Such a language promises to be more efficient than similar attempts [AM89] as it would be based on clausal, rather than non-clausal temporal resolution.

The normal form required for the resolution procedure, SNF, is derived from that developed for use in METATEM [BFG+89], which is itself an executable temporal logic. The rewrite rules used to produce the normal form are derived from the work on transformation and determinisation of METATEM programs [FN92].

The Use of Resolution Strategies.

The use of resolution strategies to guide the refutation in classical logics, mentioned previously in Sections 1.1 and 1.4.1, have been fully described in [CL73, WOLB84, Lov78]. The application of these strategies could certainly be implemented for the non-temporal resolution and their development investigated for use to direct the application of the temporal resolution rule.

6.2 Conclusions

The research carried out has shown the application of graph-theoretic techniques to the implementation of a temporal resolution step, which has been incorporated into a resolution theorem prover. This theorem prover has been used to determine the unsatisfiability of temporal formulae.

The method is more widely applicable than the other resolution methods described in Section 5.1, as it deals with both past-time and future-time temporal operators, as well as a wider range of temporal operators. This method defines only one temporal resolution rule whereas the resolution methods described in Section 5.1 tend to define several rules. Only one temporal resolution rule is required due to the translation to clausal form, meaning that all the rules will be in a particular format. The advantage of having only one temporal resolution rule to apply is that the processing required to select and apply the different rules is reduced and so the search space for a refutation is reduced.

When comparing the performance of the resolution theorem prover with that of DP/DPP, by examining the timings presented in Section 5.3, the latter performed better. However, this was expected as DP is the implementation of an efficient algorithm written in Pascal, whereas the resolution theorem prover must be regarded as a prototype and has been implemented in Prolog. The resolution theorem prover performs relatively better on Example 3 which contains a disjunct, but relatively worse on Example 4 where combinations of rules need to be taken to find the loop. Resolution methods tend to perform better on rule-sets containing predominantly disjunctive rules than tableau based methods so the former observation is to be expected. Timings for the resolution theorem prover and DP/DPP are the closest in Example 1b where resolvents from temporal resolution cycles are used to detect new loops and thus more resolvents until false is derived.

The research has shown that the implementation of the resolution method outlined in Chapter 2 is possible. Graph theoretic techniques have been used in the implementation, to both represent the rules, and search for loop in the graphs constructed. The correctness of the algorithm has been shown and an preliminary investigation into its complexity has been carried out. The complexities introduced, particularly on how to represent, and search disjuncts in these graphs, made this much more difficult than had been originally anticipated. Several areas for further work have been outlined both relating to the development of the resolution theorem prover as it has been implemented, and wider areas of related research.

Appendix A

The SNF Translation

The process of transforming formulae into SNF is explained in [Fis91, Fis92b, Fis93]. Some of the transformations, represented as rewrite rules, are shown below where P and F represent arbitrary past-time and future-time formulae respectively and w is a new proposition symbol. The rewrite rules are assumed to take place within the context of a $\Box$ operator and should only be applied after any negations have been 'pushed' down to propositions, analogous to rewriting classical logic formulae into Negation Normal Form (NNF). 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 the future-time operators are as follows:

$$\text{Removal of } \Box: \quad \{P \Rightarrow \Box A\} \longrightarrow \left\{ \begin{array}{lll} P & \Rightarrow & A \wedge w \\ \odot w & \Rightarrow & A \wedge w \end{array} \right\}$$

$$\text{Removal of } \bigcirc: \quad \left\{P \Rightarrow \bigcirc A\right\} \longrightarrow \left\{\odot P \Rightarrow A\right\}$$

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

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

The transformation rules for the removal of some of the past-time operators are as follows:

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

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

Appendix B

Sample Output

The following is output from the resolution theorem prover from the examples which have been developed through the dissertation. The symbols used in the output are explained in Table B.1, where tmp_p(i), for $i \in \mathbf{N}$, are new proposition symbols.

Output	Thesis	Description
Q	$\bullet$	in the last moment in time (weak last)
Y	$\odot$	in the last moment in time (strong last)
F	$\diamond$	sometime
&	$\wedge$	and
|	$\vee$	or
=>	$\Rightarrow$	implies
~	$\neg$	not

Table B.1: Explanation of Symbols used in Program Output

B.1 Example 1a

```
Cycle Number 1
[
  Ya => a
  Ya => l
  Qfalse => l
  Qfalse => a
]
Have generated no new resolvents, current rule-set is :-
[
  Qfalse => a
  Qfalse => l
  Ya => l
```

```
  Ya => a
]
Eventuality rule being considered is Qfalse => F~l
New Loop detected is [[a],[a]]
Loops previously detected are
    [
     [Qfalse => F~l, []]
    ]
New Rules are
[
  Qfalse => (~a W ~l)
]

Cycle Number 1
[
  Qfalse => (~l | ~a)
  Qfalse => (~l | tmp_p(1))
  Ytmp_p(1) => (~l | ~a)
  Ytmp_p(1) => (~l | tmp_p(1))
]
Cycle Number 2
[
  Qfalse => ~l
  Qfalse => ~a
  Qfalse => tmp_p(1)
  Y(tmp_p(1) & a) => ~a
  Y(tmp_p(1) & a) => ~l
  Y(tmp_p(1) & a) => tmp_p(1)
]
Have generated false
Formula is unsatisfiable.
```

B.2 Example 1b

```
Cycle Number 1
[
  Yp => p
]
Have generated no new resolvents, current rule-set is :-
[
  Yp => p
]
Eventuality rule being considered is Qfalse => F~p
```

```
New Loop detected is [[p],[p]]
Loops previously detected are
    [
     [Qfalse => Fp, []]
     [Qfalse => F~p, []]
     [Ytrue => F~p, []]
    ]
New Rules are
[
  Qfalse => ~p
]

Cycle Number 1
[
  Qfalse => ~p
]
Have generated no new resolvents, current rule-set is :-
[
  Qfalse => ~p
  Yp => p
]
Eventuality rule being considered is Ytrue => F~p
New Loop detected is [[p],[p]]
Loops previously detected are
    [
     [Qfalse => Fp, []]
     [Qfalse => F~p, [[[p]]]]
     [Ytrue => F~p, []]
    ]
New Rules are
[
  Qfalse => ~p
  Ytrue => ~p
]

Cycle Number 1
[
  Qfalse => tmp_p(2)
  Y~p => ~p
  Ytmp_p(2) => ~p
]
Cycle Number 2
[
```

```
  Ytrue => (~tmp_p(2) | ~p)
]
Cycle Number 3
[
  Yp => ~tmp_p(2)
]
Have generated no new resolvents, current rule-set is :-
[
  Yp => ~tmp_p(2)
  Ytrue => (~tmp_p(2) | ~p)
  Ytmp_p(2) => ~p
  Y~p => ~p
  Qfalse => tmp_p(2)
  Qfalse => ~p
  Yp => p
]
Eventuality rule being considered is Qfalse => Fp
New Loop detected is [[neg p],[neg p]]
Loops previously detected are
    [
     [Qfalse => Fp, []]
     [Qfalse => F~p, [[[p]]]]
     [Ytrue => F~p, [[[p]]]]
    ]
New Rules are
[
  Qfalse => p
]

Cycle Number 1
[
  Qfalse => p
]
Have generated false
Formula is unsatisfiable.
```

B.3 Example 2

```
Cycle Number 1
[
  Ya => b
  Yb => a
  Ya => l
```

```
  Yb => l
  Qfalse => a
  Qfalse => l
]
Have generated no new resolvents, current rule-set is :-
[
  Qfalse => l
  Qfalse => a
  Yb => l
  Ya => l
  Yb => a
  Ya => b
]
Eventuality rule being considered is Qfalse => F~l
New Loop detected is [[b],[a],[b]]
Loops previously detected are
    [
     [Qfalse => F~l, []]
    ]
New Rules are
[
  Qfalse => ((~b & ~a) W ~l)
]

Cycle Number 1
[
  Qfalse => (~l | ~b)
  Qfalse => (~l | ~a)
  Qfalse => (~l | tmp_p(3))
  Ytmp_p(3) => (~l | ~b)
  Ytmp_p(3) => (~l | ~a)
  Ytmp_p(3) => (~l | tmp_p(3))
]
Cycle Number 2
[
  Qfalse => ~b
  Qfalse => ~a
  Qfalse => ~l
  Qfalse => tmp_p(3)
  Y(tmp_p(3) & b) => ~b
  Y(tmp_p(3) & a) => ~b
  Y(tmp_p(3) & a) => ~l
  Y(tmp_p(3) & b) => ~a
```

```
  Y(tmp_p(3) & a) => ~a
  Y(tmp_p(3) & b) => ~l
  Y(tmp_p(3) & b) => tmp_p(3)
  Y(tmp_p(3) & a) => tmp_p(3)
]
Have generated false
Formula is unsatisfiable.
```

B.4 Example 3

```
Cycle Number 1
[
  Yb => (c | d)
  Yc => b
  Yd => b
  Yb => a
  Ya => l
  Yb => l
  Yc => l
  Yd => l
  Qfalse => d
  Qfalse => l
]
Have generated no new resolvents, current rule-set is :-
[
  Qfalse => l
  Qfalse => d
  Yd => l
  Yc => l
  Yb => l
  Ya => l
  Yb => a
  Yd => b
  Yc => b
  Yb => (c | d)
]
Eventuality rule being considered is Qfalse => F~l
New Loop detected is [[d],[b],[[d]],[[c],[b]]]
Loops previously detected are
    [
     [Qfalse => F~l, []]
    ]
```

```
New Rules are
[
  Qfalse => ((~d & (~b & ~c)) W ~l)
]

Cycle Number 1
[
  Qfalse => (~l | ~d)
  Qfalse => (~l | ~b)
  Qfalse => (~l | ~c)
  Qfalse => (~l | tmp_p(4))
  Ytmp_p(4) => (~l | ~d)
  Ytmp_p(4) => (~l | ~b)
  Ytmp_p(4) => (~l | ~c)
  Ytmp_p(4) => (~l | tmp_p(4))
]
Cycle Number 2
[
  Qfalse => ~d
  Qfalse => ~l
  Qfalse => ~b
  Qfalse => ~c
  Qfalse => tmp_p(4)
  Y(tmp_p(4) & d) => ~d
  Y(tmp_p(4) & c) => ~d
  Y(tmp_p(4) & b) => ~d
  Y(tmp_p(4) & a) => ~d
  Y(tmp_p(4) & b) => (~l | c)
  Y(tmp_p(4) & d) => ~b
  Y(tmp_p(4) & c) => ~b
  Y(tmp_p(4) & b) => ~b
  Y(tmp_p(4) & a) => ~b
  Y(tmp_p(4) & d) => ~l
  Y(tmp_p(4) & c) => ~l
  Y(tmp_p(4) & d) => ~c
  Y(tmp_p(4) & c) => ~c
  Y(tmp_p(4) & b) => ~c
  Y(tmp_p(4) & a) => ~c
  Y(tmp_p(4) & b) => (~l | d)
  Y(tmp_p(4) & d) => tmp_p(4)
  Y(tmp_p(4) & c) => tmp_p(4)
  Y(tmp_p(4) & b) => tmp_p(4)
  Y(tmp_p(4) & a) => tmp_p(4)
```

```
]
Have generated false
Formula is unsatisfiable.
```

B.5 Example 4

```
Cycle Number 1
[
  Ya => l
  Yb => l
  Yc => l
  Y(a & b) => a
  Y(b & c) => b
  Y(c & a) => c
  Qfalse => c
  Qfalse => l
  Qfalse => b
  Qfalse => a
]
Have generated no new resolvents, current rule-set is :-
[
  Qfalse => a
  Qfalse => b
  Qfalse => l
  Qfalse => c
  Y(c & a) => c
  Y(b & c) => b
  Y(a & b) => a
  Yc => l
  Yb => l
  Ya => l
]
Eventuality rule being considered is Qfalse => F~l
New Loop detected is [[c,a,b],[c,a,b]]
Loops previously detected are
    [
     [Qfalse => F~l, []]
    ]
New Rules are
[
  Qfalse => ((~c | (~a | ~b)) W ~l)
]
```

```
Cycle Number 1
[
  Qfalse => (~l | (~c | (~a | ~b)))
  Qfalse => (~l | tmp_p(5))
  Ytmp_p(5) => (~l | (~c | (~a | ~b)))
  Ytmp_p(5) => (~l | tmp_p(5))
]
Cycle Number 2
[
  Qfalse => (~b | (~c | ~l))
  Qfalse => (~a | (~c | ~l))
  Qfalse => (~c | (~a | ~b))
  Qfalse => (~a | (~b | ~l))
  Qfalse => tmp_p(5)
  Y(tmp_p(5) & (c & a)) => (~a | (~b | ~l))
  Y(tmp_p(5) & (b & c)) => (~a | (~c | ~l))
  Y(tmp_p(5) & (a & b)) => (~b | (~c | ~l))
  Y(tmp_p(5) & c) => (~c | (~a | ~b))
  Y(tmp_p(5) & b) => (~c | (~a | ~b))
  Y(tmp_p(5) & a) => (~c | (~a | ~b))
  Y(tmp_p(5) & c) => tmp_p(5)
  Y(tmp_p(5) & b) => tmp_p(5)
  Y(tmp_p(5) & a) => tmp_p(5)
]
Cycle Number 3
[
  Qfalse => (~c | ~l)
  Qfalse => (~c | ~b)
  Qfalse => (~l | ~b)
  Qfalse => (~c | ~a)
  Qfalse => (~l | ~a)
  Qfalse => (~a | ~b)
  Y(tmp_p(5) & (b & (c & a))) => (~l | ~a)
  Y(tmp_p(5) & (b & (c & a))) => (~l | ~b)
  Y(tmp_p(5) & (a & (b & c))) => (~c | ~l)
  Y(tmp_p(5) & (b & c)) => (~a | ~c)
  Y(tmp_p(5) & (c & a)) => (~a | ~b)
  Y(tmp_p(5) & (a & b)) => (~b | ~c)
]
Cycle Number 4
[
  Qfalse => ~c
```

```
  Qfalse => ~l
  Qfalse => ~b
  Qfalse => ~a
  Y(tmp_p(5) & (b & (c & a))) => ~l
  Y(tmp_p(5) & (a & (b & c))) => ~a
  Y(tmp_p(5) & (b & (c & a))) => ~b
  Y(tmp_p(5) & (a & (b & c))) => ~c
]
Have generated false
Formula is unsatisfiable.
```

Appendix C

Formulae used in Additional Timings

The following formulae were tested on both DP/DPP and the resolution theorem prover. The timings for the complete set of formulae are shown in Table 5.1.

$$
\begin{array}{rcl}
\Box\neg w & \Leftrightarrow & \neg\Diamond w \\
\Diamond\neg w & \Leftrightarrow & \neg\Box w \\
\bigcirc\neg w & \Leftrightarrow & \neg\bigcirc w \\
w & \Rightarrow & \Diamond w \\
\Box w & \Rightarrow & w \\
\bigcirc w & \Rightarrow & \Diamond w \\
\Box w & \Rightarrow & \bigcirc w \\
\Box w & \Rightarrow & \Diamond w \\
\Box w & \Rightarrow & \bigcirc\Box w \\
w_1 U w_2 & \Rightarrow & \Diamond w_2 \\
\Box w & \Leftrightarrow & \Box\Box w \\
\Diamond w & \Leftrightarrow & \Diamond\Diamond w \\
\Box\bigcirc w & \Leftrightarrow & \bigcirc\Box w \\
\Diamond\bigcirc w & \Leftrightarrow & \bigcirc\Diamond w \\
\bigcirc(w_1 \wedge w_2) & \Leftrightarrow & (\bigcirc w_1 \wedge \bigcirc w_2) \\
\bigcirc(w_1 \vee w_2) & \Leftrightarrow & (\bigcirc w_1 \vee \bigcirc w_2) \\
\bigcirc(w_1 \Rightarrow w_2) & \Leftrightarrow & (\bigcirc w_1 \Rightarrow \bigcirc w_2)
\end{array}
$$

Bibliography

[Aba88] Martín 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.

[AM89] M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8: 277–295, 1989.

[AM90] M. Abadi and Z. Manna. Nonclausal Deduction in First-Order Temporal Logic. *ACM Journal*, 37(2):279–317, April 1990.

[Bar87] Howard 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.

[BFG+89] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A Framework for Programming in Temporal Logic. In *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Mook, Netherlands, June 1989. (Published in *Lecture Notes in Computer Science*, volume 430, Springer Verlag).

[CdC84] A. Cavali 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*, pages 113–127. LNCS 170, 1984.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[CW91] Mats Carlsson and Johan Widen. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, Kista, Sweden, September 1991.

[dlTC90] 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.

[EdC89] P. Enjalbert and L. Fariñas del Cerro. Modal Resolution in Clausal Form. *Theoretical Computer Science*, 65:1–33, 1989.

[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.

[Fis91] Michael 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] Michael Fisher. A Normal Form for First-Order Temporal Formulae. In *Proceedings of Eleventh International Conference on Automated Deduction (CADE)*, Saratoga Springs, New York, June 1992. (Published in *Lecture Notes in Computer Science*, volume 607, Springer Verlag).

[Fis92b] Michael 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.

[Fis93] Michael Fisher. A Clausal Resolution Method for Propositional Temporal Logic. (Submitted to the Journal of Logic and Computation), 1993.

[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.

[Gab87] Dov 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*, pages 402–450, Altrincham, U.K., 1987. (Published in *Lecture Notes in Computer Science*, volume 398, Springer Verlag).

[GLNP90] G. Gallo, G. Longo, S. Nguyen, and S. Pallottino. Directed hypergraphs and applications. Technical Report 3/90, Departmento di Informatica, Università Degli Studi di Pisa, Pisa, Italy, January 1990.

[Gou84] G. D. Gough. Decision Procedures for Temporal Logic. Master's thesis, Department of Computer Science, University of Manchester, October 1984.

[HC68] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic*. Methuen (UP), 1968.

[HC84] G. E. Hughes and M. J. Cresswell. *A Companion to Modal Logic*. Methuen (UP), 1984.

[Hus87] Wolodymyr Hussak. Decidability in Temporal Presburger Arithmetic. Master's thesis, Department of Computer Science, University of Manchester, February 1987.

[Kow74] R. Kowalski. Predicate Logic as a Programming Language. In *Proc. IFIP*, pages 569–574. North Holland, 1974.

[Kow79] R. Kowalski. *Logic for Problem Solving*. North Holland, 1979.

[Lov78] David W. Loveland. *Automated Theorem Proving: a Logical Basis*. North-Holland, Inc., 1978.

[Min90] G. Mints. Gentzen-Type Systems and Resolution Rules, Part I: Propositional Logic. *Lecture Notes in Computer Science*, 417:198–231, 1990.

[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*. Academic Press, London, 1981.

[Mur82] Neil V. Murray. Completely Non-Clausal Theorem Proving. *Artificial Intelligence*, 18:67–85, 1982.

[Nil82] Nils J. Nilsson. *Principles of Artificial Intelligence*. Springer Verlag, 1982.

[Ohl88] Hans-Jürgen Ohlbach. A Resolution Calculus for Modal Logics. *Lecture Notes in Computer Science*, 310:500–516, May 1988.

[PG86] D. A. Plaisted and S. A. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[Pnu81] Amir Pnueli. The Temporal Semantics of Concurrent Programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pnu86] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. *Lecture Notes in Computer Science*, 224, August 1986.

[RK91] Elaine Rich and Kevin Knight. *Artificial Intelligence*. McGraw-Hill, 1991.

[Rob65] J. A. Robinson. A Machine Based Logic Based on the Resolution Principle. *ACM Journal*, 12(1):23–41, January 1965.

[Smu68] R. M. Smullyan. First order logic. *Ergebnisse der Mathematik*, 43, 1968.

[Wol83] Pierre Wolper. Temporal Logic Can Be More Expressive. *Information and Control*, 56, 1983.

[WOLB84] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning – Introduction and Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.