A Resolution Based Automated Theorem Proving System Using Concurrent Processing Approach

by

SURASH NATARAJAN

Thesis submitted in partial fulfilment of the requirements for the degree of Master of Science

May 1994

Acknowledgment

I wish to express my most sincere gratitude to my supervisor Dr. Nor Adnan Yahya for his continued encouragement and invaluable guidance, without which this dissertation would not have possible. I am also especially grateful to him for having taken so much of his valuable time and for his keen interest throughout the course of this project. I also wish to thank Assoc. Prof. Dr. Zaharin Yusof and Assoc. Prof. Dr. R.K. Subramaniam for their kind advise and guidance. To my family, colleagues and friends many thanks for providing great encouragement and moral support during the whole course of the MSc. programme. Also, I wish to thank the School of Mathematics and Computer Science and the Institute of Postgraduate Studies for providing much assistance.

Table of Contents

List of Figures

Abstrak

Satu Sistem Pembuktian Teorem Automatik Berdasarkan Resolusi Menggunakan Pendekatan Pemprosesan Seiring

Semenjak pembangunan sistem pembuktian teorem automatik berdasarkan resolusi yang pertama di pertengahan 1960an, terdapat penyelidikan yang berterusan di dalam bidang ini untuk mempertingkatkan proses penyelesaian masalah di dalam sistem-sistem pembuktian teorem. Penyelidikan pada masa kini di dalam bidang ini adalah tertumpu kepada penggunaan kaedah-kaedah pengideksan pangkalan data dan pemprosesan selari untuk mempertingkatkan kecekapan sistem-sistem tersebut. Apa yang dimaksudkan tentang kecekapan sistem adalah tertumpu kepada kelajuan perlaksanaan sistem di dalam pembuktian teorem oleh suatu sistem pembuktian teorem automatik.

Penyelidikan yang dilaporkan di dalam tesis ini adalah tertumpu kepada penggunaan kaedah pemprosesan seiring di dalam pembangunan suatu sistem pembuktian teorem automatik berdasarkan resolusi, yang merupakan suatu aplikasi di dalam bidang kecerdasan buatan. Tujuan kami melakukan penyelidikan ini adalah untuk mengkaji keberkesanan pemprosesan seiring di dalam mempertingkatkan proses penyelesaian masalah di dalam suatu sistem pembuktian teorem berdasarkan resolusi. Semasa penyelidikan kami di sini, kami telah mengkaji komponen mana di dalam suatu sistem pembuktian teorem boleh dihuraikan untuk memperkenalkan pemprosesan seiring dan bagaimana caranya ia mesti dilakukan. Tujuan kami di dalam pembinaan sistem ini bukanlah tertumpu kepada penghasilan suatu sistem pembuktian teorem yang berkelajuan tinggi, tetapi adalah untuk membina suatu sistem yang boleh dikatakan sebagai prototaip yang akan mengamkan idea untuk memperkenalkan pemprosesan seiring di dalam pembangunan sistem pembuktian teorem automatik berdasarkan resolusi. Di dalam

Abstract

Ever since the first resolution based automated theorem proving system was developed on a computer in the mid 1960s, there has been constant research in this area on enhancing the problem solving process of the theorem provers. The recent trend in this area is towards exploiting database indexing and parallel processing in increasing the efficiency of these systems, in particular the execution speed of the theorem prover in proving a theorem.

The research reported in this thesis is devoted to the use of concurrent processing for developing a resolution based automated theorem proving system, an application in the area of artificial intelligence. Our purpose in doing this is to study the usefulness of concurrent processing in enhancing the problem solving process in a resolution based automated theorem proving system. During our research here we investigated which component of the theorem prover can be decomposed into introducing concurrent processing and how this should be done. Our main aim in building this theorem prover was not mainly in producing a high performance theorem prover but to build a system that can be considered to be a prototype that would illustrate the idea of introducing concurrent processing in resolution based theorem provers. We believe that concurrent processing is the intermediate step in moving from sequential processing towards parallel processing. Concurrent processing provides the simplicity of sequential system design with efficient processing capabilities of parallel system. In our discussion here we present a novel design of the system and how we propose to implement it.

CHAPTER 1

INTRODUCTION

In this chapter, our aim is to give a background introduction and followed by the definition of our problem of research. This will be followed by an overview of this thesis.

1.1 Background

The field of **Automated Theorem Proving** (**ATP**) began approximately in the late 1950s and despite its pure mathematical origin, developments in the area have been utilized in many important applications in computer science. Most of the work during these three and a half decades has been devoted to the automation of first-order predicate calculus (mainly by the resolution method and its refinements). The importance of ATP within the discipline of computer science, especially with some of the concerns of **Artificial Intelligence** (**AI**) is well known. [Nilsson 80] for example, mentions a range of connections between AI and ATP to support the claim that "... theorem proving is an extremely important topic in the study of AI methods".

The first significant computer program for theorem proving was the Logic Theory Machine [Newell, Shaw & Simon 57], which used *working backwards* (backward chaining) as its basic heuristic algorithm. Since then, automated theorem proving has

world problems . In [Quaife 91] it has been defined that one of the long-range goals of automated theorem proving research is to develop programs intelligent enough to prove theorems that human mathematicians cannot prove.

The *method of resolution* is the most widely used method for automated theorem proving, with systems like AURA (AUtomated Reasoning Assistant) [Smith 88], ITP & LMA (Interactive Theorem Prover & Logic Machine Architecture) [Lusk & Overbeek 84a] [Lusk & Overbeek 84b] and OTTER (Organized Technique for Theorem-proving and Effective Research) [McCune 90] all were developed using this method. Although resolution based ATP systems have shown considerable improvements in power over the years, their performance is still low. Researchers, therefore have began seeking ways to improve the performance of the ATP systems. The two factors that are likely to lead to significant performance improvements, that so far have been identified are *database indexing* and *parallel processing* [Butler, Lusk, McCune & Overbeek 86]. Apart from these two approaches, the more recent approach is the so called "Prolog-technology" theorem-provers[Schuman & Letz 90], which uses the compilation techniques from WAM-based Prolog implementations to achieve extremely high inference rates, at a cost of possibly redundant computations. Although certain small problems can be done very quickly with the Prolog-technology approach, many large problems still remain out of reach of even the best Prolog-technology systems such as SETHEO (SEquantial THEOrem prover) [Letz, Schuman, Bayerl & Bibel 92].

The effect of database indexing in improving the performance of theorem provers is demonstrated by OTTER, which is a sequential first-order logic theorem prover. OTTER's database indexing method has made this program the fastest and the most powerful among all existing resolution based systems. Whereas in the concern of parallelism, [Butler & Karonis 88] developed a scheme based primarily on *domain decomposition*, for parallelizing ATP systems.

The work addressed by this thesis centers around developing a resolution based ATP system using concurrent processing approach based on the scheme developed by [Butler & Karonis 88]. Our aim in constructing a concurrent first-order logic theorem-prover was not mainly in producing an extremely high-performance predicate logic theorem-prover. Rather, it was to create a system that can be considered to be a prototype that would generalize to systems for the predicate calculus in using concurrent processing. The aim of using concurrent processing here was to investigate its usefulness in enhancing the problem solving process in automated theorem proving.

Our aim in this chapter is to introduce the background and define the problems of our research and finally to give an overview of this thesis. We do this by first giving a general introduction as to the background of the problem. This is followed by a brief introduction to the areas of automated theorem proving and concurrent processing concepts that will be used throughout this thesis. Following this is a general look at current implementations of resolution based sequential ATP systems and the problems faced by them. The discussion is then continued by defining the scope and objectives of this thesis and followed by a section on the organization of this thesis. Finally we will give a summary of this chapter.

1.2 Introduction To Automated Theorem Proving

Automated theorem proving involves the programming of computers to perform logical (mathematical) deduction. This should not be confused with numerical calculation in which operations that need to be performed can be exactly specified ahead of time. Rather, theorem provers search for proofs of the truth or falsity of statements given axioms describing the basic assumptions.

the designation "automated reasoning" has been used since 1980 and prior to this, much of the research and many of the applications under this area were discussed in terms of automated theorem proving. The difference between the two fields rests mainly with the way in which the corresponding software is used and with their scope. In automated reasoning, the emphasis is on an active collaboration between the user and the program and on many uses that we would not normally consider to involve "proving theorems". Automated theorem proving is now a part of automated reasoning.

In this section we will give an introduction to the general characteristics of automated theorem proving and ATP systems, followed by a brief look at the method of resolution, which will be the method used in this thesis. As we are using logic as a formal method of reasoning and as such it has its own syntax and logical rules. It is appropriate here that we give the general logical functions or the sentential connectives that will be used throughout this thesis. The logical functions mentioned are AND, OR, NOT, IMPLIES and EQUIVALENT, whose symbol are :

Logical Function	Symbol
AND	$\wedge$
OR	$\vee$
NOT	$\neg$
IMPLIES	$\Rightarrow$
EQUIVALENT	$\Leftrightarrow$

The language of logic is very commonly used as a way of representing facts because it provides a powerful way of deriving new knowledge from old, which is the basis of mathematical deduction. In this formalism, we can conclude that a new statement is true by proving that it follows from the statements that are already known. Let us illustrate some applications of the first-order logic to problem solving by giving an example. Here the approach is first to symbolize the problem by formulas and then to

prove that the formulas are valid or inconsistent. Let us say in our example we have the following statements :

1. All men are mortal.
2. Ali is a man

Our aim is to show that Ali is mortal. From these two statements we derive the following two axioms :

A1 : $(\forall x)(man(x) \Rightarrow mortal(x))$

A2 : man(Ali)

So our aim here is to show that Ali is mortal from A1 and A2. That is, to show that mortal(Ali) is a logical consequence of A1 and A2.

We have :

A1 $\wedge$ A2 : $(\forall x)(man(x) \Rightarrow mortal(x)) \wedge man(Ali)$

If A1 $\wedge$ A2 is true in an interpretation I, then both A1 and A2 are true in I. Since (man(x) $\Rightarrow$ mortal(x)) is true for all x, when x is replaced by "Ali", (man(Ali) $\Rightarrow$ mortal(Ali)) is true in I. That is, $\neg$man(Ali) $\vee$ mortal(Ali) is true in I. However $\neg$man(Ali) is false in I since man(Ali) is true in I. Hence, mortal(Ali) must be true in I. We have therefore shown that mortal(Ali) is true in I whenever (A1 $\wedge$ A2) is true in I. By definition, mortal(Ali) is a logical consequence of A1 and A2.

In our example here, we have shown that the conclusion follows from the given facts. This method, that conclusions follow from axioms is called a *proof*. A procedure for finding proofs is called a *proof procedure*. In our following discussions we will discuss how we can mechanize this proof procedure so that it can be implemented in a computer and in doing so will enable us to use computers in finding proofs.

1.2.1 Basic Concepts Of Automated Theorem Proving

Automated theorem proving rests firmly in the area of artificial intelligence with

basically refers to the study and development of programs that reason logically. Logical reasoning refers to processes that infer new formulas from existing formulas such that the new formulas are always true in any interpretation where the old formulas are true.

One of the most common example of logical reasoning is *modus ponens*. In modus ponens, for example if we have two formulas $p \Rightarrow q$ and p, we can infer the formula q. In basic English, what this formulas mean is that *if p implies q being true and if p is true then q is also true.* There are ways in which this can be proven to be valid, that is any assignment of truth values that makes the two hypotheses true must also assign q to be true, but here we will not touch on this. Besides modus ponens, other examples of logical reasoning are *chain rule* and *resolution.* We will not be discussing about chain rule, whereas resolution will be discussed in the next section. These three examples of logical reasoning may be summarized as follows :

Modus Ponens	**Chain Rule**	**Resolution**
$p \Rightarrow q$	$p \Rightarrow q$	$p \vee \neg q$
p	$q \Rightarrow r$	$q \vee r$
———	———	———
q	$p \Rightarrow r$	$p \vee r$

At this point we can see that to understand automated theorem proving, we must be familiar with logic to some extent. Here we will give a brief introduction to the elements of first-order logic that are most important to the field of automated theorem proving. A first-order logic language consists of a number of different kinds of symbol that are combined to make well-formed formulas(wff). These are variables, constant, function and predicate symbols, the Boolean connectives (AND, OR, NOT, etc.), the quantifiers $\forall$(FORALL) and $\exists$(EXISTS), parentheses and commas. Each function and predicate symbol has an arity or arguments associated with that symbol. The parentheses

functions over the domains of interest and predicate symbols are meant to represent true/false relationship.

The purpose of us explaining the basic structure of first-order logic language here is because the familiarity of it is extremely important not only in ATP but also in AI for several reasons. First, logic offers the only formal approach to reasoning that has a sound theoretical foundation. This is especially important in our attempts to mechanize or automate the theorem proving process in that inferences should be correct and logically sound. Second, the structure of it is flexible enough to permit the accurate representation of natural language reasonably well. This is mainly important in AI systems since most knowledge must originate with and be consumed by humans. To be effective, transformations between natural language and any representation scheme must be natural and easy. Finally it is widely accepted in the AI field as one of the most useful representation method.

Up to now we have only looked at one of the basic purpose of automated theorem proving that is the development of programs that reason logically. Next we are going to look at a basic paradigm for the automation of theorem proving. This paradigm is commonly know as the Argonne paradigm for the automation of theorem proving [Wos 88] and has been the basis for the development of various ATP systems such as ITP & LMA [Lusk & Overbeek 84a] [Lusk & Overbeek 84b], AURA [Smith 88] and OTTER [McCune 90]. This paradigm has 6 components that is :

1. The paradigm relies on the use of a specific language to represent the information, for example facts and relationships to inform the theorem proving program about a given assignment. Each piece of information that is required is represented by using one or more statements (*clauses*) in a language called the *clause language.*

always yield conclusions that follow inevitably from the statements to which the rule is applied.

3. The programs reasoning can be controlled by the use of *strategy*. One type of strategy directs the reasoning by focusing the program's attention on paths conjectured to be greater interest, and another type of strategy restricts the reasoning by prohibiting the program from pursuing paths of certain types.
4. When conclusion is drawn, the program may apply some number of transformation to the conclusion to rewrite it into a *canonical form* or *normal form.*
5. The program may then test the result by comparing it to other retained pieces of information to decide whether the new conclusion is redundant and should therefore be immediately discarded.
6. To "know" when a given assignment has been completed, the program may search the available information to see if two items (clauses) *contradict* each other. To set the stage for employing this test for assignment completion, the information that is given to the reasoning program typically includes a statement or statements (clauses) that correspond to assuming the assignment cannot be completed or that correspond to assuming the desired result is false.

Basically this six components of this paradigm can be summarized into language, inference rule, strategy, a procedure for transforming into a canonical form, procedure for identifying and then discarding redundant information and a test for assignment completion. This six components together make up a basic ATP system. We will be giving a more detailed discussion on these components in chapter 2 of this thesis.

1.2.2 Method of Resolution

The method of resolution was specifically designed for use with computers

that has been a part of AI-problem solving research from the mid-1960s. Resolution is a sound inference rule that when used to produce a contradiction, is also *complete*. What is meant by being complete in this aspect here is for example if we have a set R of inference rules, we can say it is complete if and only if the formula G can be obtained from the formula F with a finite number of applications of members of R whenever F logically implies G. In an important practical application, resolution theorem proving, particularly the resolution contradiction system, has made the current generation of PROLOG interpreters possible [Kowalski 79].

The resolution principle, by using a minimum use of substitution can find contradiction in a database of clauses. Resolution contradiction proves a theorem by negating the statement to be proved and adding this negated goal to the set of axioms that are known to be true. It then uses the resolution rule of inference to show that this leads to a contradiction. If the theorem prover show that the negated goal is inconsistent with the given set of axioms, it follows that the original goal must be consistent. This proves the theorem.

Resolution contradiction proofs involve the following steps :

1. Put the premises or axiom into *clause form.*
2. Add the negation of what is to be proved in clause form to the set of axioms.
3. *Resolve* these clauses together, producing new clauses that logically follow from them.
4. Produce a contradiction by generating the empty clause.

Resolution based proofs by contradiction require that the axioms and the negation of the goal be placed in a normal form called *clause form.* Clause form represents the logical data base as a set of disjunctions of literal. A literal is an atomic expression or the

be unified to make them equivalent. A new clause is then produced consisting of the disjuncts of all the predicates in the two clauses minus the literal and its negative instance.

Based on what we have explained on the method of resolution, here we present a simple example of how resolution can be used to produce a proof. We wish to prove that "Ali will die" from statements that "Ali is a human" and "All humans are mammals" and "All mammals will die". Changing these three premises to predicates and applying modus pones gives :

1. All human are mammals : $\forall(x)(human(x) \Rightarrow mammal(x))$
2. Ali is a human : human(ali)
3. Modus ponens and {ali/x} gives : mammal(ali)
4. All mammals will die $\forall(y)(mammal(y) \Rightarrow die(y))$
5. Modus ponens and {ali/y} gives : die(ali)

(Here the usage of {x/y} is as a substitution operation, where it means that substitute all occurrence of y in that particular clause with x.) Equivalent reasoning by resolution converts these predicates to clause form :

Predicate Form	Clause Form
1. $\forall(x)(human(x) \Rightarrow mammal(x))$	$\neg human(x) \vee mammal(x)$
2. human(ali)	human(ali)
3. $\forall(y)(mammal(y) \Rightarrow die(y))$	$\neg mammal(y) \vee die(y)$

Negate the conclusion that Ali will die :

4. $\neg$die(ali)	$\neg$die(ali)

Compare and clash the clauses having opposite literals, producing new clauses by resolution (Figure 1.1).

The symbol ◊ in figure 1.1 indicates that the empty clause is produced and the contradiction found. The ◊ symbolizes the clashing of a predicate and its negation, that

(unifications) used to make predicates equivalent also gives us the value of the variables under which a goal is true.

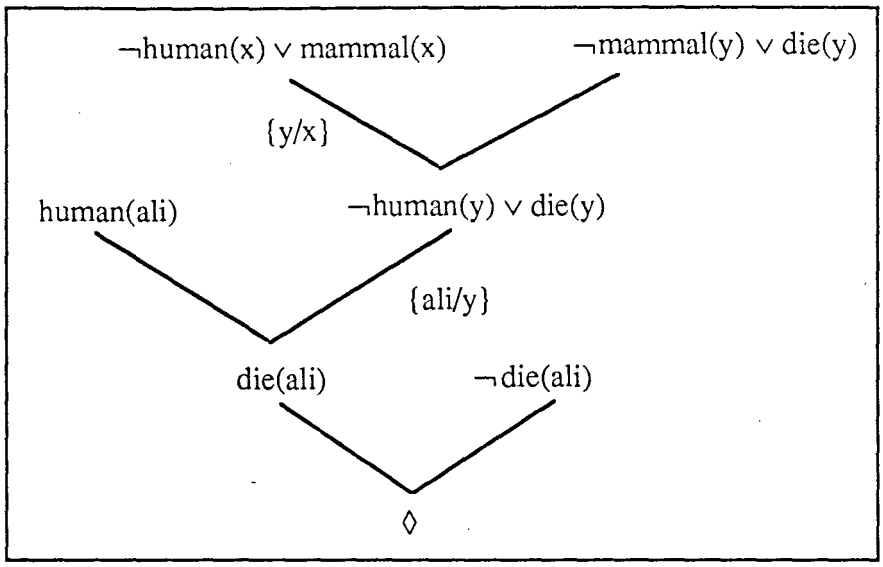


Figure 1.1 : Resolution proof of the "Ali will die" problem

The purpose of us giving this example here is to demostrate the usefulness of the method of resolution in providing the means to mechanically derive conclusions that are valid, hence an extremely useful method for theorem proving using computers.

1.3 Introduction To Concurrent Processing Concepts

Up to this point we have mentioned that our aim in this thesis is to introduce concurrent processing in developing an ATP system. If we observe here carefully, we have been using the term *concurrency* instead of *parallelism*. Here we would like to stress that concurrency and parallelism refer to two different meanings. In [Tick 91] it has been mentioned that parallelism and concurrency are different although parallel and concurrent processing are often confused. Parallel processing refers to processing that can be executed in parallel, i.e., parallelism is the ability to gain speedup by executing actions simultaneously, whereas concurrent processing refers to processing that can express

without requiring multiple physical processors for achieving this task. Parallelism does not imply concurrency, nor does concurrency imply parallelism.

Here we also would like to introduce the difference between concurrent processing and sequential processing. A *sequential program* consists of data declarations and executable instructions in a programming language. The instructions are executed sequentially on a computer which also allocates memory to hold data. A *concurrent program* is a set of ordinary sequential programs which are executed in *abstract parallelism* [Ben-Ari 90]. The parallelism is abstract because we do not require that a separate physical "processor" be used to execute each process, unlike in parallel processing. What is being mentioned here is that a sequential program specifies sequential execution of a list of statements and a concurrent program specifies two or more sequential programs that may be executed concurrently.

In the following two sections, we will be giving a brief discussion about the various programming methods in concurrent programming followed by a brief discussion on the environment and its characteristics of our concurrent programming environment.

1.3.1 Concurrent Programming Methods

A concurrent program can be executed either by allowing processes to share one or more processors or by running each process on its own processor. The first approach is referred to as *multiprogramming*, it is supported by an operating system kernel [Dijkstra 68] that multiplexes the processes on the processor. The second approach is referred to as *multiprocessing* if the processors share a common memory or as *distributed processing* if the processors are connected by a communications network. Hybrid approaches also exist, for example, processors in a distributed system are often multiprogrammed [Jones & Schwarz 80].

In order to cooperate, concurrently executing processes must communicate and synchronize. Communication allows execution of one process to influence execution of another. Interprocess communication is based on the use of *shared memory* or on *message passing* [Andrews & Schneider 83]. Whereas synchronization is often necessary when processes communicate because processes are executed with unpredictable speeds.

In message passing, we create many concurrent processes and each process communicates by exchanging messages. In message passing method, no data objects are shared among processes. Each process has its own local set of private data objects. In order to communicate, processes must send data objects from one local process to another (Figure 1.2).

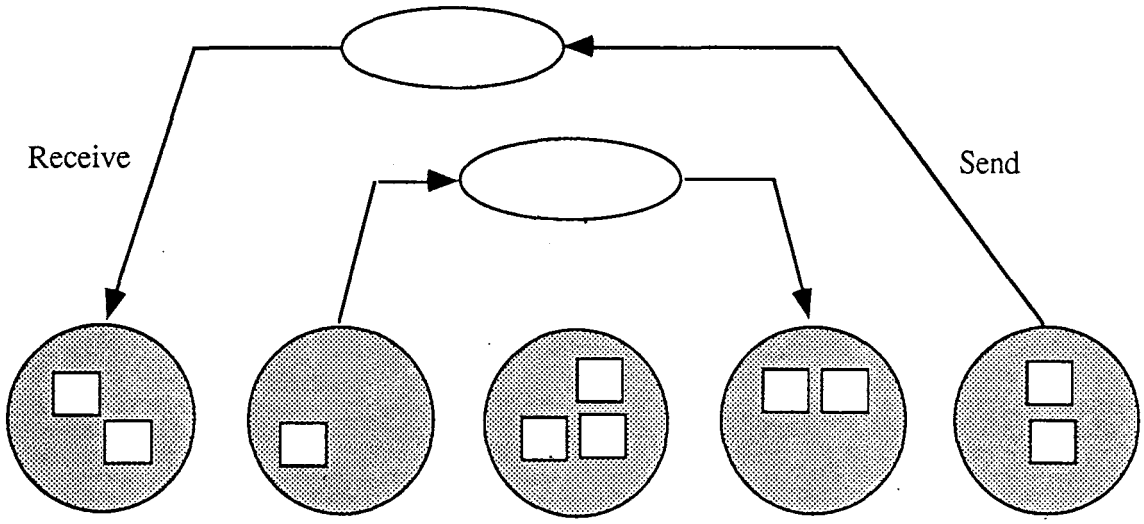


Figure 1.2 : Message Passing - A collection of concurrent processes communicate by exchanging messages (Processes are round, data objects are square, messages oval).

Shared memory operations allow two or more processes to share a segment of physical memory but data areas of the processes that communicate are entirely separate. In this method, processes communicate and coordinate by leaving data in the shared memory (Figure 1.3).

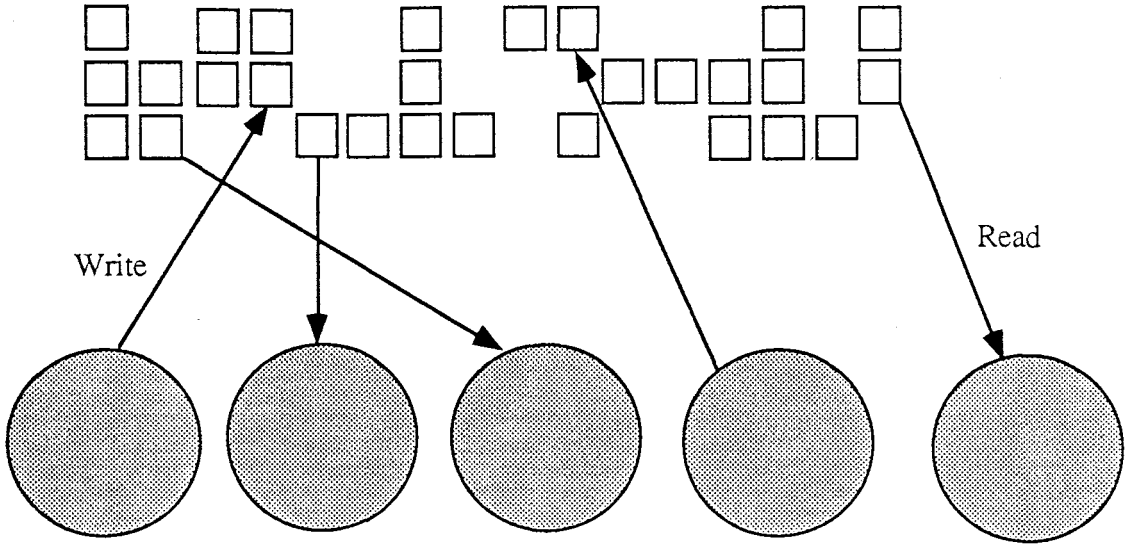
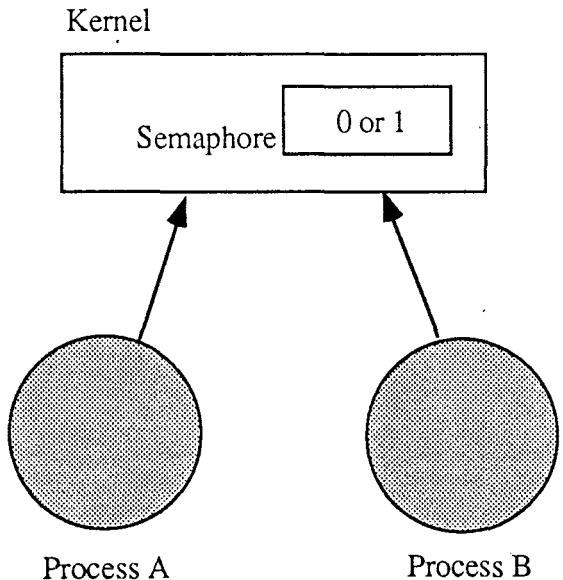


Figure 1.3 : Shared Memory - Concurrent Processes and shared memory figure as autonomous parts of the program structure. Processes communicate by reading and writing in shared memory.

In process synchronization, semaphores are used as a synchronization primitive. One of the main uses of semaphores is to synchronize the access to shared memory segments. Semaphores are basically integer valued variable, that is a resource counter. The value of the variable at any point in time is the number of resource units available. If we have one resource, say a shared memory segment, then the valid semaphore values are zero and one (Figure 1.4).



1.3.2 Concurrent Programming Environment

In this section we will discuss briefly about our concurrent programming environment. The reason for this is, our design which will be discussed in chapter 3 and chapter 4 is influenced by this environment and we feel it is appropriate that we give some basic introduction to it at this point. A more detailed discussion will be given in chapter 2.

Our concurrent programming environment is based on the Unix InterProcess Communication (IPC). IPC involves sharing data between process and when necessary, coordinating access to the shared data. There are basically three main types of IPC :-

1. Message Passing - The message passing facility allows a process to send and receive messages. A message being in essence an arbitrary sequence of bytes or characters.
2. Semaphores - Semaphores provide a low-level means for process synchronization. It is not suited to the transmission of large amounts of information.
3. Shared Memory - Shared memory allows two or more processes to share the data contained in specific memory segments.

These three mechanisms dominate the Unix IPC package with message passing allowing processes to send formatted data streams to arbitrary processes, shared memory allowing processes to share parts of their virtual address space and semaphores allowing processes to synchronize execution.

With these IPC mechanisms we do have the necessary facilities to introduce concurrent processing into automated theorem proving. As we have mentioned earlier, the two components in concurrent processing is interprocess communication and

which provides facilities for process synchronization. In introducing concurrent processing into any problem solving process, the tasks involved are mainly in breaking up the problem solving process into various concurrently executing tasks which collectively aim to solve the problem in focus. These concurrently executing processes need some form of interprocess communication to help establish links that allow these various processes to communicate in their problem solving process and some form of synchronizing mechanism to coordinate their tasks. This is where the IPC mechanisms play their roles. Earlier in figure 1.1 we have shown the resolution proof of the "Ali will die" problem, which is done sequentially. Now let us show how this same problem can be solved concurrently. In this problem we have four clauses and our aim is to find a contradiction. Let us divide this clause space into two groups and these two groups of clauses will be resolved concurrently and both the results will then be combined to obtain a contradiction. Figure 1.5 shows how this is done.

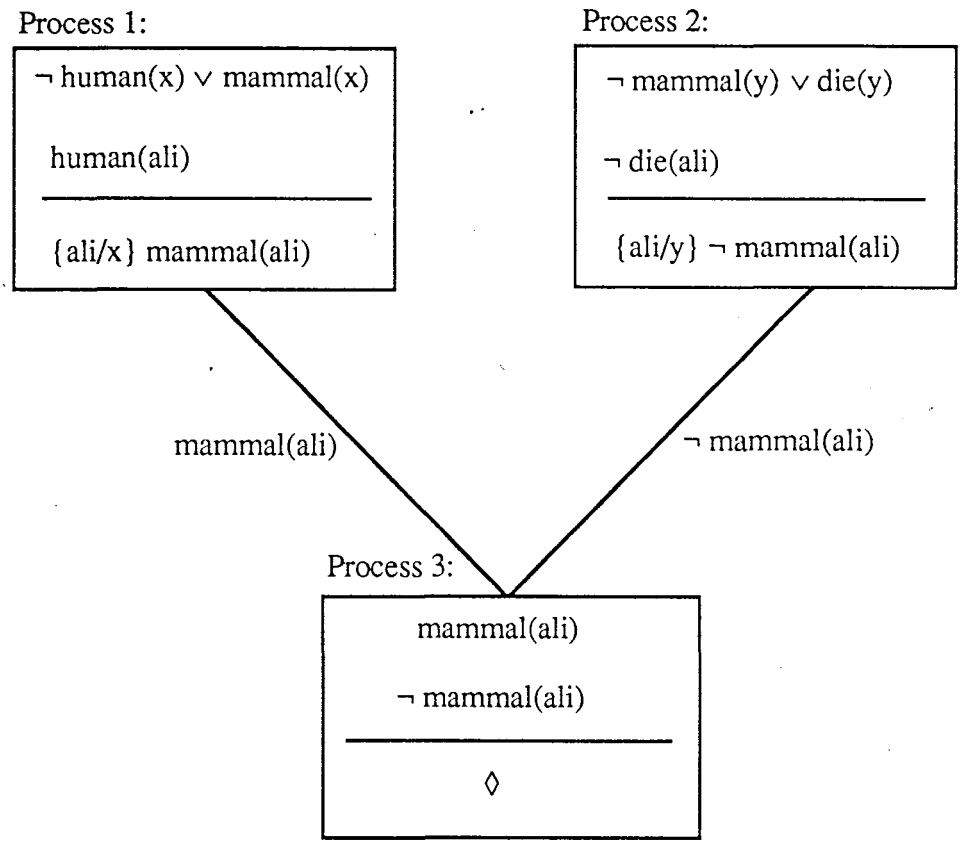


Figure 1.5 : A Concurrent Simulated Resolution Proof of the "Ali will die" Problem

So based on this example we can see the two components of concurrent processing in actions. That is, firstly for interprocess communication involves the passing of clauses between process 1 and process 3 and also process 2 and process 3. Whereas for synchronization is basically involved in process 3 where it has to make sure that both process 1 and process 2 have passed whatever necessary clause to it before it can proceed with its own execution. This simple example here basically highlights how concurrent processing can be introduced into automated theorem proving.

1.4 Current Implementations of Automated Theorem Proving Systems

As theorem proving is the most optimum strategy for applying logic in such areas as database/query systems and logic-based languages, a great deal of AI research has been devoted to the subject. As a result of this, a number of theorem proving systems have emerged from this research.

Basically the theorem provers that have been developed in the area of classical logic, can be divided into *resolution based* and *nonresolution based.* Up till now, we have only been discussing about resolution based system. The nonresolution based theorem provers are designed to emulate the reasoning of human theorem provers. For this reason it is also known as *natural deduction* theorem prover. Nonresolution theorem provers generally use a backward-chaining technique involving substitution in an attempt to transform the theorem to be proven into a form consistent with the original axioms. Among the more successful nonresolution theorem provers that have been developed are IMPLY [Bledsoe 77] and BMTP (Boyer-Moore Theorem Prover)[Boyer & Moore 79]. As we are more interested in resolution based theorem proving, we would not be discussing further in the area of non resolution theorem proving.

As mentioned earlier in the chapter, there has been various resolution based

sequential processing. In the area of parallel processing, there were also various systems developed, such as ROO (Radical OTTER Optimization) [Lusk, McCune & Slaney 91] which used the shared memory multiprocessor concept to parallelize the ATP algorithm and PARROT-II (PARRallel OTter - II) [Jindal, Overbeek & Kabat 92] which used the distributed processors concept to parallelize the ATP algorithm.

Here we can see that there has been numerous ATP systems developed over the years not only using sequential processing but also using parallel processing. The development of these systems have been initiated by the various problems that exists in automated theorem proving. In [Wos 88] some of these problems have been identified, among them are :

1. Clause Retention - The ATP program keeps too many deduced clauses (too many conclusions) in its database of information.
2. Inadequate Focus - The ATP program gets lost too easily.
3. Redundant Information - The ATP program generates the same clauses (or proper instances of clauses already retained) over and over again.
4. Clause Generation - The ATP program draws far too many conclusions, many of which are redundant and many of which are irrelevant even though they are not redundant.
5. Size of the Deduction Steps - The inference rules do not take deduction steps of appropriate size.
6. Choice of Transformation for Canonicalization - The ATP program does not always use the most effective transformation to rewrite information into a canonical form.
7. Metarules - No adequate guidelines exists for selecting the appropriate representation, inference rule, strategy, transformation for canonicalization and type information discarding procedure.

8. Database Indexing - The ATP program requires too much time to find the appropriate information in its database.

A look at this list, reveals the wide range of interesting research offered by automated theorem proving. This is one of the reason why research in this area has attracted numerous researches with various system developed using various techniques and concepts to solve these numerous problems that exists.

1.5 Scope and Objectives

In this chapter we have mentioned that we intend to introduce concurrent processing concept in automated theorem proving. Our aim in doing this is to enhance one of the main components of an ATP system, that is the clause generation component. We intend to decompose this component into multiple processes with the understanding that with multiple processes generating clauses, an ATP systems goal in theorem proving can be archived faster.

As we know that the aim of theorem provers is to search for proofs of the truth or falsity of statements given axioms describing the basic assumptions. For example in the case of resolution based automated theorem provers, using the existing clauses, new clauses are resolved and added to the clause space until a clause that contradicts with any existing clause in the clause space can be found. So what we have here is a search process for the contradicting clause. If we observe our earlier example in figure 1.1 which is based on the sequential technique, we can see that in resolving new clauses from existing clauses until a contradicting clause can be found, the search at any particular time is only focusing on one clause at a time to resolve it against all possible clauses to generate new clauses. With a closer look at this problem we can see that while focusing

and this is highlighted in our example shown in figure 1.5. So if we compare these two techniques, we can see that the sequential method takes one step at a time in finding the contradicting clause, whereas the concurrent method takes multiple steps at a time in its search for the contradicting clause. So basically we believe that by focusing on multiple clauses simultaneously rather than a single clause at any time, the search for the contradicting clause can be further enhanced.

Our main objective in this thesis is to show how we will convert a sequential ATP algorithm into a concurrent ATP algorithm and explain how we intend to implement this algorithm. We will be doing this by first studying a sequential ATP algorithm and identify the appropriate section of the algorithm to be decomposed into concurrent processing, hence introducing a concurrent ATP algorithm. This will be followed by an explanation of how we intend to implement this algorithm. The scope and objectives of this thesis are therefore :

1. To study an sequential ATP algorithm and identify the components that can be decomposed into concurrent processing with the aim of enhancing the clause generating component of the algorithm.
2. To give a novel design and how we intend to implement this algorithm.

1.6 Organization Of The Thesis

Having mentioned the scope and objectives of our thesis, in this section we would like to give a brief description about the organization of this thesis. The thesis will be organized in the following manner :

Chapter 1 : This chapter relates to the background and the definition of the problem of our research. Here we give a general introduction to our problem followed by the scope and objectives of this thesis.

Chapter 2 : This chapter provides a general intro[illegible]

to better understand the problem and the problem-solving discussed in this thesis.

Chapter 3 : This chapter discusses about the conversion of a sequential automated theorem proving algorithm into a concurrent algorithm. Here we attempt to explain the component of the sequential algorithm that we chose to convert into a concurrent version and how the conversion is done.

Chapter 4 : This chapter provides a discussion on how we intend to implement the algorithm that we have discussed about in chapter 3. Here we will provide the basic structures of the proposed implementation of the algorithm.

Chapter 5 : Having discussed about the conversion of a sequential automated theorem proving algorithm into a concurrent version and its proposed implementation, in this chapter we discuss about the various aspects of our work in this thesis and how the proposed algorithm can be further enhanced in future work.

Concluding remarks on the various issues touched in this thesis are given in the conclusion.

1.7 Summary of Chapter

In this chapter our aim was to give an introduction to the background of our research. We did this by first introducing our problem of research. This was followed by a background look at our field of research which is in the area of automated theorem proving and the concepts of concurrent processing, with the aim of better understanding the problem. Next we gave the scope and objectives of this thesis as to make our work more clearer. Finally we presented the organization of this thesis for better understanding

Chapter 2

Automated Theorem Proving and Concepts for Concurrent Processing

In this chapter our aim is to provide a general introduction to the area of automated theorem proving and concepts for concurrent processing with the aim to better understand the problem and the problem-solving discussed in this thesis.

2.1 Introduction

As our research in this thesis centers around introducing concurrent processing into ATP systems, we turn to a brief review of the pertinent aspects of automated theorem proving and concurrent processing in this chapter. Our discussion here will be divided into two main sections. In the first section, we will discuss about the concepts of theorem proving and the basic elements of automated theorem proving and this will be followed by the second section, where we will discuss further the concepts of concurrent processing based on our earlier discussion in chapter 1.

The discussion on automated theorem proving will be divided it into two parts. In the first part we will give the basic concepts of theorem proving and this will be followed by a discussion on the elements of automated theorem proving. Whereas our discussion on concurrent processing will also be divided into two parts. The first part will be on the

be based on the Unix IPC system because this will be our proposed platform for implementation and it influences the conversion and design of our system.

2.2 Concept of Theorem Proving

In theorem proving, what we attempt to show is that, a particular well-formed formula (wff), B, is a logical consequence of a set $S = \{A_1,.....,A_k\}$ of wff's, collectively called the *axioms* of the problem. A rule of inference is a rule by which new expressions can be derived from previously established ones. For instance, if A_i and A_j are previously established wff's,

$$f_q(A_i,A_j) \Rightarrow A_k$$

indicates that Ak can be derived from A_i and A_j using inference rule f_q.

In order to understand theorem proving more formally, a notation for the first-order predicate calculus is needed. In our discussion here the universe of discourse will contain a set of elementary symbols, a,b,c,..., which serve as *constants*, a number of *variable* symbols, written from the "end" of the alphabet, s,t,u,v,w,x,y,z and *functions*. Functions are mappings, a function of n arguments maps from the elements of the set D^n (all possible ordered sets of n terms) into the set D. For example, the function + is a binary function, which maps a pair of real number into a single real number. The letters f,g,h will be used for functions. Finally, capital letters (usually P,Q) will indicate *relations* or *predicates*. A predicate of n arguments maps from D^n into the set {T,F}. That is, any n-ary relationship between terms is either true or false.

Now let us consider the structure of a well-formed formula. A *term* is either a variable, a constant or a function. An n-ary function must have n terms for its arguments. Thus the following are terms :

a, b, c, f(a), g(f(x),y), h(g(a,w),f(x))

R(f(x),a), P(a,y), Q(g(a,b),f(x))

A *literal* is an atomic formula or its negation. When the structure of an atomic formula is not relevant, we shall write atomic formulas as capital letters, A,B, etc., with negations indicated as ¬A, ¬B.

A *clause* is a disjunction of literals and a set of clauses S is interpreted as a single statement that is the conjunction of all of its clauses. Since atomic formulas are predicates that map into the set {T,F}, the truth values of a set of atoms determine the truth values of the clauses and set of clauses that can be constructed from them.

These ideas are illustrated in Figure 2.1, which shows some simple statements written in this notation. The constants of this example {a,b} should be interpreted as any two real numbers, whereas the functions should be interpreted as single-valued functions on real numbers. The predicates are the three possible orderings (greater than, less than, equal to) that can hold between any pair of real numbers. Obviously a statement of a particular ordering for a given pair is either true or false, indicating clauses C1, C2 and C3 in S states that they are simultaneously true.